МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ

ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ
РАДИОЭЛЕКТРОНИКИ

СЕМЕНЕЦ В.В.
ХАХАНОВА И.В.
ХАХАНОВ В.И.

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ СИСТЕМ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА VHDL

Рекомендовано Министерством образования и науки Украины в качестве учебного пособия для студентов специальностей: "Компьютерные системы и сети", "Специализированные компьютерные системы"

Харьков 2003

УДК 681.32:519.713

Проектування цифрових систем з використанням мови VHDL: Навч. посібник/ В.В. Семенець, І.В. Хаханова, В.І. Хаханов.– Харків: ХНУРЕ, 2003.– 492 с.– Рос. мовою.

Запропоновано практично орієтовані технології проектування цифрових пристроїв та систем на базі сучасних комплектуючих виробів у вигляді кристалів програмованої логіки, що випускаються провідними фірмами-виробниками. Описано фізичні й логічні основи функціонування елементів, вузлів та пристроїв програмованої логіки. Виконано порівняльний аналіз номенклатури комплектуючих виробів з позиції швидкодії – вартості проектованого цифрового виробу. Наведено моделі, методи та засоби описання, вводу, моделювання, синтезу, імплементації, оптимізації й верифікації, що використовуються в програмно-апаратних комплексах автоматизованого проектування цифрових функціональних блоків, вузлів комп'ютерних систем на основі використання мови VHDL, які доведені до практичних рекомендацій з наданням лістингів описання поведінки пристроїв, що робить дану книгу унікальним та корисним виданням.

Навчальний посібник призначений для студентів та аспірантів спеціальностей комп'ютерної інженерії, а також для широкого кола читачів, бажаючих оволодіти повним циклом сучасної технології автоматизованого проектування систем на кристалі – Systems on Chip.

Іл. 459. Табл.44. Бібліогр.: 39 назв.

Представлены практически ориентированные технологии проектирования цифровых устройств и систем на основе современных комплектующих изделий в виде кристаллов программируемой логики, выпускаемых ведущими фирмами-производителями. Описаны физические и логические основы функционирования элементов, узлов и устройств программируемой логики. Выполнен сравнительный анализ номенклатуры комплектующих изделий с позиции быстродействия-стоимости проектируемого цифрового изделия. Представлены модели, методы и средства описания, ввода, моделирования, синтеза, имплементации, оптимизации и верификации, используемые в программно-аппаратных комплексах автоматизированного проектирования. Предложены современные технологии проектирования цифровых функциональных блоков, узлов компьютерных систем на основе использования языка VHDL, которые доведены до практических рекомендаций с представлением листингов описания поведения устройств, что делает данную книгу уникальным и полезным изданием.

Учебное пособие предназначено для студентов и аспирантов специальностей компьютерной инженерии, а также для широкого круга читателей, желающих овладеть полным циклом современной технологии автоматизированного проектирования систем на кристалле – Systems on Chip.

Ил. 459. Табл.44. Библиогр.: 39 назв.

Рецензенты:

Г.И. Загарий, д-р техн. наук, проф. Харьковской государственной академии железнодорожного транспорта

Л.В. Дербунович, д-р техн. наук, проф. Национального технического университета "Харьковский политехнический институт"

ISBN 966-659-052-2

ВВЕДЕНИЕ

Наметилась тенденция к освоению новых технологий проектирования цифровой аппаратуры украинскими университетами, академическими и научно-исследовательскими институтами, а также производственными компаниями. Это связано не только с миссионерской деятельностью ведущих зарубежных фирм, таких как Aldec Inc. (США), но и с возрастающей потребностью украинского рынка электронных технологий в выпуске конкурентоспособных специализированных вычислительных устройств собственного изготовления, защищенных от несанкционированного доступа и управления. Кроме того, американский и западно-европейский рынки электронных технологий нуждаются как в расширении своего позитивного влияния на Восточную Европу, так и в новых интеллектуальных ресурсах, способных не только использовать средства автоматизированного проектирования цифровых систем, но и усовершенствовать последние. Поэтому для отечественных разработчиков патентно чистой и легко сертифицируемой цифровой аппаратуры решение проблемы реструктуризации электронной промышленности обусловлено:

1. Необходимостью перехода всей электронной промышленности, занимающейся разработкой и производством специализированных вычислительных устройств и систем, на современную элементную базу в виде кристаллов программируемой логики, обладающих преимуществами: надежность, структурно-функциональная прозрачность элементной базы, сертифицируемость, низкая стоимость, высокое быстродействие, минимальные недельные сроки создания проекта. Это обеспечит снижение эксплуатационных расходов, энергосбережение в таких областях народного хозяйства, как приборостроение, бытовая техника, транспорт, связь, энергетика.

2. Внедрением новых технологий автоматизированного проектирования сложных цифровых систем Hardware-Software Co-operation Design с применением современных средств ввода, синтеза и имплементации (Active-HDL, MAX+PLUS II, Foundation, FPGA Express Synthesis, ActelDeskTop, Synplify, Aliance, FPGA-Compiler II) ведущих фирм мира (Aldec, Cadence, Altera, Xilinx, Synopsys, Mentor Graphics, Asset), использующих языки описания аппаратуры (VHDl, Verilog, System C, EDIF).

3. Необходимостью разработки новых средств диагностического обслуживания проектируемых на кристаллах программируемой логики сверхсложных цифровых систем в целях их сертификации, верификации и тестирования на стадиях проектирования и производства. Упомянутые средства должны обеспечивать: тестопригодное проектирование цифровых систем на основе технологий Boundary Scan и BIST, синтез псевдослучайных и детерминированных тестов, проверяющих исправное поведение или дефекты заданного класса, моделирование неисправностей для определения качества и валидности сгенерированных тестов, поиск дефектов в проектируемых или работающих устройствах с заданной глубиной диагностирования.

На мировом рынке электронных технологий многообразие предлагаемых схемотехнических решений делится на три основных типа: микропроцессоры, заказные БИС и программируемая логика. Первые два, наряду с высоким быстродействием, имеют недостатки, связанные со значительными временными и финансовыми затратами проектирования вычислительных устройств на их основе, а также с возможностью имплементации в кристаллы деструктивных модулей, способных разрушать или модифицировать информацию. Что касается третьего типа, то современный уровень развития субмикронных технологий микроэлектроники позволяет создавать на одном кристалле программируемой логической интегральной схемы (ПЛИС) сверхсложные цифровые системы, содержащие 3-5 миллионов эквивалентных вентилей. Это дает возможность проектировать сложные специализированные вычислительные устройства в течение нескольких недель с помощью средств автоматизированного проектирования известных фирм, таких как: Aldec, Cadence, Altera, Xilinx, Synopsys. При этом возрастают требования к языкам описания аппаратуры.

Актуальность их использования связана с новыми технологиями создания проектов. Автоматизированное проектирование цифровых систем в последние годы имеет тенденцию к применению языков описания аппаратуры высокого уровня. Существует определенный прогресс в переходе от традиционных представлений (VHDl, Verilog, Abel, EDIF, TDF, XNF) к новым интегрированным языкам типа System C, который сочетает преимущества параллелизма VHDL с семантическими возможностями языка программирования C++. Прогресс в средствах описания цифровых систем связан с возрастанием рыночного спроса на мощные компиляторы и симуляторы, способные решать задачи ввода и верификации проектов, содержащих сотни и тысячи строк исходных описаний. К тому же появление новых субмикронных микроэлектронных технологий привело к созданию мощной элементной базы в виде программируемых логических интегральных схем: Field Programable Gate Array (FPGA), Complex Programable Logic Device (CPLD), на которых реализуются Systems on Chip (SoC). Поэтому в настоящее время пользователь имеет высокий технический потенциал для проектирования любых цифровых систем: кристаллы ПЛИС со степенью интеграции до 10 млн вентилей; быстродействие 500 МГц; стоимость чипа в несколько долларов; мощные компиляторы для всех языков описания аппаратуры высокого уровня (фирм Aldec, Cadence, Altera, Xilinx, Synopsys); достаточное количество синтезаторов (MAX+PLUS II – Multiple Array MatriX Programmable Logic User System – Altera; Foundation 2.1 – Xilinx, Aldec, Synopsys; FPGA Express Synthesis – Synopsys; ActelDeskTop – Actel, VeriBest, Synplicity; Synplify – Synplicity) для преобразования описания цифровой системы в конструкции ПЛИС.

По оценкам ведущих фирм мира (Aldec, Cadence, Altera, Xilinx, Synopsys, Mentor Graphics, Asset) в Украине через 2 года все предприятия электронной промышленности перейдут на новые технологии проектирования на основе языков описания аппаратуры высокого уровня с использованием компиляторов и синтезаторов, способных создавать проекты, имплементируемые в ПЛИС. Отсюда возникает потребность в ежегодной подготовке только для украинского рынка не менее 5000 специалистов в области Hardware Software Co-operation design, способных обслуживать системы автоматизированного проектирования, синтеза, имплементации и тестирования цифровых изделий.

Предлагаемая книга будет способствовать получению знаний в области новых технологий проектирования, ориентированных на реализацию проектов в виде System on Chip в кристаллах программируемой логики на основе использования лицензионных программных продуктов ведущих фирм мира.

Предназначена для студентов и аспирантов специальностей компьютерной инженерии: 7.091501 – “Компьютерные и интеллектуальные системы и сети”; 7.091502 – “Системное программирование”; 7.091503 – “Специализированные компьютерные системы”; 7.091504 – “Защита информации с ограниченным доступом и автоматизиция ее обработки”; 7.091401 – “Системы управления и автоматики”; 7.091403 – “Гибкие компьютерные системы робототехники”.

ГЛАВА 1
СОВРЕМЕННЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ ЦИФРОВЫХ СИСТЕМ

1.1. Модели цифровых систем

Создание цифрового проекта связано с математической формализацией рассматриваемых процессов и явлений, превращаемых в общепринятые понятия моделей и методов. По мнению В.М. Глушкова формализация процесса разработки сложной цифровой системы оперирует развивающейся во времени информационной моделью, которая содержит и исчисленческую часть. Последняя – исчисление, определяемое как совокупность правил вывода, – позволяет получать дедуктивным путем следствия из конечного множества аксиом.

По мере создания проекта меняется соотношение между информационной и исчисленческой моделями – повышается удельный вес последней – происходит математизация проекта в целях его формального описания на языках программирования или описания аппаратуры. Это способствует уменьшению временных и материальных затрат при решении проблемы проектирования.

Математизация – это возможность использования воздвигнутого на основе математического языка огромного и стройного здания дедуктивных построений, как результата усилий многих поколений математиков. Поэтому формализация решаемой задачи в любой области науки дает исследователю возможность пользоваться определенной частью здания в виде соответствующего математического аппарата, благодаря чему удается сэкономить время на дедуктивные построения в целях получения необходимых выводов. В связи с этим задача определения той части математического здания, которое наиболее адекватно описывает рассматриваемую проблему и с минимальными затратами формализует процесс ее решения, представляется весьма актуальной. Для решения задач проектирования полезно использовать не только новые методы и средства синтеза устройств, опирающиеся на современную элементную базу в виде программируемой логики, но и современные языки описания аппаратуры, ориентированные на эффективное решение задач, связанных с созданием сложных цифровых систем.

В основу современных методов проектирования цифровых устройств положено понятие модели как такого математического описания системы, в котором сохранены только необходимые свойства, существенные для синтеза, верификации и имплементации. Любая система может быть представлена несколькими различными по форме, адекватности и степени подробности моделями.

Термин "система", в зависимости от контекста, может быть применен как к микросхеме (чипу), так и к суперкомпьютеру. Будем считать, что цифровая система – это устройство, преобразующее или сохраняющее информацию. Системой является как целое устройство, реализованное на плате, так и элементы – микросхемы, из которых оно состоит.

Для проектирования современных устройств, которые могут иметь тысячи и даже миллионы компонентов, используется системное проектирование – systematic metodology. Оно заключается в том, что на первом этапе создается абстрактная структура, удовлетворяющая требуемым условиям – спецификации. Затем она разбивается на подсистемы, которые вместе реализуют ту же функцию, что и модель верхнего уровня. На следующем шаге каждая подсистема может быть разбита на составляющие более низкого уровня. Такие итерации выполняются до тех пор, пока на

самом нижнем уровне модель устройства не будет состоять только из примитивных, не разбиваемых более компонентов.

Преимущество данного метода состоит в том, что каждая подсистема может быть спроектирована отдельно, независимо от других. Таким образом, проектировщику не нужно работать со всем объемом информации, а только с тем, который необходим для создания конкретной части проекта.

Существует несколько практических причин применения моделей:

1. Формальная модель используется для создания технического задания.

2. Модель необходима при кооперации труда проектировщика и пользователя. Вместо длинного и не всегда полного описания поведения системы в разных режимах проектировщик может предоставить формальную модель системы.

3. Возможность тестирования и верификации проекта через создание его моделей. Вначале создается общая модель, описывающая функционирование всего устройства. Для ее верификации строится тест, полученные выходные реакции принимаются в качестве эталонных. Затем, следуя концепции нисходящей методики проектирования, эта модель разбивается на подсистемы. Полученный ранее тест может быть использован для проверки новой, более детализированной модели. Проект требует доработки, если полученные на данном шаге реакции не совпадают с эталонными. Иначе, детализация модели проекта будет продолжаться до тех пор, пока составляющие проект компоненты не будут соответствовать реальным устройствам.

Полученный тест может быть использован и на стадии производства для верификации реального устройства. Подразумевается, что тест проверяет все режимы, в которых созданная цифровая система будет эксплуатироваться.

4. Возможность формальной верификации схемы. Формальная реализация требует математического описания функций системы. Оно может быть выполнено в терминах формальной логики, включающей параметры времени. Формальная верификация также требует математического определения конструкций языка моделирования или обозначений, используемых для описания схем. Хотя формальная верификация в настоящее время еще широко не применяется, ведутся активные исследования в данном направлении. Уже существуют демонстрации методов формальной верификации при проектировании реальной схемы.

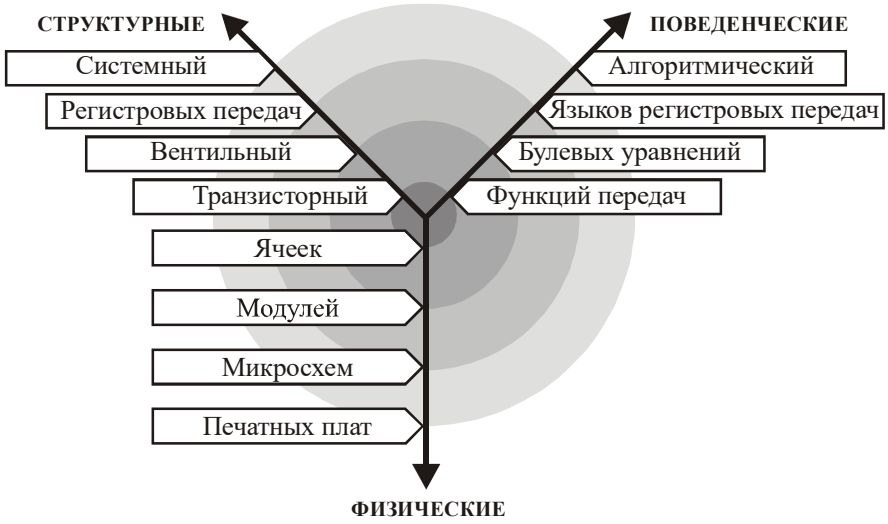
5. Возможность автоматического синтеза схем. Если возможно формально описать функции системы, теоретически можно трансформировать это описание в схему, реализующую данные функции. Это позволяет сократить долю ручного труда в проектировании и уменьшить вероятность внесения ошибок в проект. Если использовалась автоматическая трансляция описания в схему, можно утверждать, что последняя соответствует исходной спецификации.

1.1.1. Классификация моделей

Модели можно разделить на три типа: 1) поведенческие или функциональные, 2) структурные, 3) физические или геометрические (рисунок 1.1). Первые описывают закон функционирования устройства безотносительно к его схемотехнической реализации. Поэтому формой представления объекта являются языки: регистровых передач, булевых уравнений, описания алгоритмов, функций. Структурные отличаются от первых введением в поведенческую модель отношения иерархии, оперирующего понятием структуры для некоторой функции. Это дает возможность несколько углубить реализацию поведенческой модели до уровня вентилей или транзисторов, что является шагом к более адекватному описанию объекта, ориентированному на его имплементацию в кристаллы программируемой логики или заказных микросхем. Физические модели максимально ориентированы на представление объекта в конкретной схемотехнической базе – возможность изначального описания поведения объекта на физическом уровне наиболее предпочтительна для получения максимального быстродействия и

оптимальной аппаратурной реализации. Платой за такие преимущества являются практически ручное проектирование и связанные с этим значительные временные затраты.

Рисунок 1.1. Типы и уровни абстракции моделей



Приведем примеры моделей, соответствующих различным уровням классификации. Рассмотрим микросхему контроллера, используемого в качестве устройства управления для измерительных инструментов. Его функция заключается в получении данных и отображении их в удобной для человека форме на мониторе.

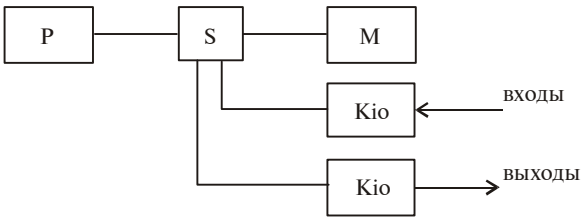
На самом высоком уровне функционирование микроконтроллера может быть представлено в виде граф-схемы алгоритма, подобной макроописанию компьютерной программы. Такой уровень функционального описания соответствует поведенческой модели. Реализация алгоритма функционирования контроллера представлена на рисунке 1.2. Эта модель описывает сканирование входов данных контроллером и отображение их в определенном масштабе.

Рисунок 1.2. Алгоритм для контроллера измерительного инструмента

```
цикл
   для каждого входа данных цикл
      Считать значение с входа;
      откорректировать данные, используя текущую шкалу для данного
входа;
      преобразовать масштабированные данные в десятичное число;
      вывести данные на монитор, соответствующий данному входу;
конец цикла;
ждать 10 ms;
конец цикла;
```

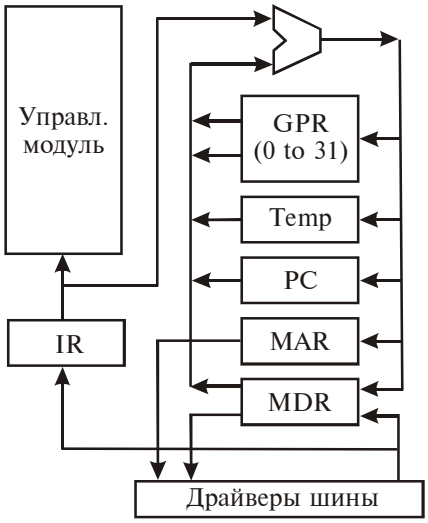
На этом уровне абстракции модель представляет собой соединение таких элементов: процессор, память, входные/выходные устройства. Такой уровень часто называют системным, типа процессор-память (Processor Memory Switch (PMS)). На рисунке 1.3 представлена структурная модель контроллера измерительного инструмента PMS уровня. Она включает процессор (P), память (M), переключатель (S) и два входных/выходных контроллера (Kio).

Рисунок 1.3. PMS-модель микроконтроллера



Модели следующего уровня абстракции (см. рисунок 1.1) описывают систему в виде модулей, хранящих и трансформирующих данные. Для структурного типа его часто называют уровнем регистровых передач. Такие модели состоят из путей данных и управляющего модуля. Пути данных – это регистры для хранения данных и элементы, преобразующие данные при передаче между регистрами. Модель такого уровня представлена на рисунке 1.4. Она включает регистры: общего назначения (GPR), счетчика команд (PC), адреса памяти (MAR), данных памяти (MDR), временных значений (Temp), командный (IR), арифметический модуль, драйверы шины, управляющий модуль.

Рисунок 1.4. Структурная модель уровня регистровых передач микроконтроллера



Функциональные модели языков регистровых передач (*register-transfer language* – RTL) часто используют для описания поведения системы на этом уровне. Сохранение данных описывается с помощью регистровых переменных, а их преобразование – с помощью арифметических и логических операторов. Например, RTL-модель для рассматриваемого микроконтроллера измерительного прибора может иметь следующий вид:

```
MAR <- PC, memory_read <- 1
PC <- PC + 1
wait until ready = 1
IR <- memory_data
memory_read <- 0
```

Модель описывает работу контроллера, определяемую командой, полученной из памяти. Содержание PC регистра передается в регистр адреса памяти и устанавливается сигнал **memory_read.** Затем данные из PC регистра преобразуются (в этом случае увеличиваются на 1) и передаются обратно в PC регистр. Когда вход **ready** для памяти устанавливается в 1, данные из памяти передаются в командный регистр. Затем сигнал **memory_read** устанавливается в 0.

Третий уровень абстракции – это традиционный логический уровень, на котором структурная модель состоит из вентилей, а функциональная описывается с помощью булевых уравнений и таблиц истинности.

На самом нижнем уровне абстракции структурную модель можно описывать с помощью транзисторов, а функциональную – используя дифференциальные уравнения для тока и напряжения в цепи. Обычно проектировщики не работают на этом уровне, поскольку существуют средства проектирования для автоматического преобразования из моделей высокого уровня. Модели физического типа зависят от микроэлектронной или схемотехнической среды. Здесь они рассматриваться не будут и данный тип моделей приведен только для полноты классификации.

1.2. Языки описания аппаратуры

В связи с постоянным усовершенствованием технологий производства интегральных схем появляется возможность размещать большее число элементов в одной микросхеме. Цифровые системы все более усложняются. Детальное проектирование таких устройств на вентильном (логическом) уровне становится практически невозможным. По этой причине возрастает значимость языков описания аппаратуры. Они позволяют разрабатывать и верифицировать цифровые устройства на верхних уровнях до преобразования в логический. Этот подход подобен написанию программных средств на языках программирования (Си) и использованию компилятора для перевода такой программы в машинный код. Два наиболее популярных языка описания аппаратуры – это VHDL и Verilog.

VHDL – это аббревиатура от английского выражения VHSIC Hardware Description Language. В свою очередь, VHSIC происходит от названия программы Very High Speed Integrated Circuit (высокоскоростные интегральные схемы). Эта программа, финансированная Министерством Обороны США, ставила своей целью развитие нового поколения высокоскоростных интегральных схем. Первая версия языка была представлена в 1985 г. Впоследствии он был передан IEEE для стандартизации. В 1987 г. язык был утвержден как стандарт IEEE 1076–1987. Через пять лет он был рассмотрен повторно, в результате чего новая версия 1076-93 содержит ряд дополнительных возможностей.

С тех пор VHDL пользуется постоянно возрастающей популярностью среди специалистов по автоматизированному проектированию (CAD) электронных систем. Первые VHDL-приложения появились в начале 90-х годов. Для синтеза был разработан пакет IEEE 1164. На практике каждый крупный производитель систем автоматизированного проектирования поддерживает VHDL.

VHDL позволяет описывать системы на различных уровнях. Он реализует методологию нисходящего проектирования, в которой система сначала описывается на высоком уровне и тестируется с помощью средств моделирования, после чего поэтапно приводится к структурному описанию, тесно связанному с фактической аппаратной реализацией. VHDL был разработан независимым для технологии реализации. Если какой-то проект, описанный в VHDL, реализован для конкретной электронной технологии, то это же VHDL-описание может быть использовано в качестве исходного для реализации устройства в новой технологии.

Verilog, или Verilog HDL – язык описания аппаратуры, разработанный в 1985 г. Филиппом Мурби (Philip Moorby), нуждавшемся в простом, наглядном и эффективном способе для описания цифровых схем, моделирования и анализа их функционирования. Язык становится собственностью Gateway Design Automatiion, затем его приобретает Cadence Design Systems. В 1990 г. Cadence открывает язык, что приводит его к стандартизации IEEE в 1995 г.

Что же лучше – VHDL или Verilog? Verilog легче в понимании и использовании. Он применялся и применяется в промышленности , где требуется моделирование и синтез. Но он имеет недостаток в конструкциях для описания уровней системы. Verilog получил признание в проектировании ASIC схем, особенно для проектов низкого уровня (регистровых передач и ниже). Наиболее популярен в Северной Америке и Азии, особенно в Японии. Непопулярен в Европе.

VHDL более сложный язык, его труднее изучать и использовать. Тем не менее он обладает большей гибкостью, что является его преимуществом и недостатком, из-за богатства допустимых стилей кода. Поскольку VHDL лучше подходит для работы с очень сложными проектами, в настоящее время его популярность возрастает. Особенно это относится к Европе, США и Канаде. Язык не пользуется успехом в Японии. Однако в мире всё больше и больше предпочитают язык VHDL.

1.3. Этапы проектирования цифровых устройств

Проектирование цифровой системы начинается с разработки технического задания, на основе которого затем строится функциональная схема, последовательно преобразуемая в реальное устройство. Моделирование и синтез являются дополняющими друг друга процедурами, используемыми в процессе проектирования, хотя их точное взаимоотношение зависит от планируемой реализации. Рассмотрим этапы проектирования для специализированных интегральных схем (ASIC) (рисунок 1.5). Такие заказные микросхемы являются узкоспециализированными, имеют высокую производительность при выполнении любых вычислений, но они самые дорогие.

Первый этап – создание технического задания, которое обычно включает требование к рабочим характеристикам, описание интерфейса, стоимостные ограничения и другие физические требования, такие, как метрические характеристики системы и рассеивание мощности. На основе технического задания создается предварительная высокоуровневая функциональная схема. Моделирование на этом этапе используется для приведения этой схемы в соответствие с техническим заданием. Затем она преобразуется в модель уровня регистровых передач (RTL – register transfer level), описывающую регистры, модули памяти, операционные и управляющие автоматы. Следующий этап – создание логических схем для каждого компонента. На уровне регистровых передач и на логическом может быть использовано моделирование для верификации проекта. Моделирование неисправностей позволяет получать выходные реакции схемы для заданного класса дефектов, возникающих на этапах производства и эксплуатации. Если существующие статистические данные указывают на достаточно высокий процент этих дефектов, проект следует модифицировать с целью обеспечить тестопригодность и восстанавливаемость. Наконец, описание логического уровня преобразуется в схемное, а затем проектируется топология кристалла и вычисляются реальные физические свойства проекта, такие как площадь кристалла и рассеивание мощности. Проверяются проектные нормы, определяются параметры схемы, после чего на этом уровне может быть выполнена верификация проекта.

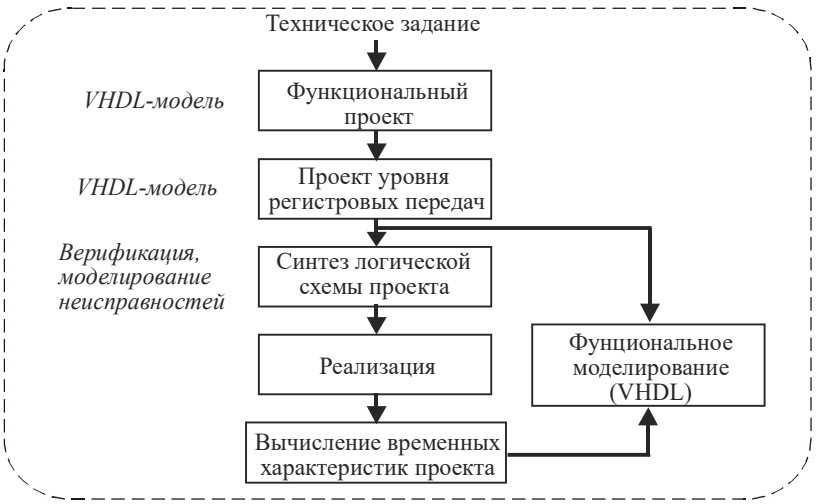
На каждом шаге иерархического проектирования для описания структуры используются компоненты. На самом верхнем абстрактном уровне – это небольшое количество сложных элементов, таких как сумматоры или память; на нижнем – огромное число простых элементарных компонентов, например, вентилей или транзисторов. Каждый уровень иерархии проекта соответствует некоторой абстракции, имеет собственное множество операций и поддерживающих их инструментов проектирования, часть из которых приведена на рисунке 1.5. По любому уровню абстрактного представления можно предсказать поведение объекта. Но физические свойства и возможности схемы желательно рассматривать, переходя к нижним уровням, хотя для них требуется более продолжительное время анализа и проектирования.

Исправление ошибок проектирования, обнаруженных на заключительных этапах разработки устройства, является очень дорогим процессом. Это может привести к увеличению временных затрат на создание проекта, а следовательно, к возрастанию стоимости готового изделия, не говоря уже о потерях прибыли из-за опоздания на рынок. Благодаря возможности моделирования поведения объекта на разных уровнях абстракции, неисправности можно обнаруживать и исправлять значительно раньше. Языки описания аппаратуры предназначены для их использования на всех этапах проектирования и тестирования, они обеспечивают совместимость и соответствие между различными уровнями, а также возможности диагностирования ошибок проектирования при наличии тестов проверки неисправностей, относящихся к модельным или физическим дефектам.

Рисунок 1.5. Основные этапы нисходящего проектирования цифровых систем

Типичная процедура автоматического синтеза зависит от выбранной для имплементации схемотехники (программируемой логики). Рассмотрим этапы проектирования устройства для реализации на программируемых пользователем логических матрицах (Field Programmable Gate Array – FPGA). В данном случае FPGA можно рассматривать как электронное устройство с большим числом вентилей и триггеров, связи между которыми можно запрограммировать как автоматически, так и ручным способом. На рисунке 1.6 показана измененная схема этапов проектирования. В начале проектирования на основе технического задания создается функциональная схема на языке VHDL и выполняется ее верификация. Затем она преобразуется к уровню регистровых передач, определяются потоки данных и основные аппаратные функциональные модули. После получения VHDL-модели уровня регистровых передач с помощью синтезирующего компилятора создается вентильный эквивалент объекта. На этом этапе логическое моделирование может быть использовано для предварительной оценки возможностей спроектированного устройства. Объект на логическом уровне представлен из логических элементов и триггеров, входящих в состав FPGA. Соответствующие вентили и триггеры соединяются с помощью конфигурирующих сигналов. Такой процесс называется размещением и маршрутизацией, по окончании которого можно получить точные оценки задержек на линиях и элементах. После этого выполняется моделирование реальных временных характеристик.

Рисунок 1.6. Этапы разработки проекта для FPGA



ГЛАВА 2
ВВЕДЕНИЕ В VHDL

Описаны базисные свойства VHDL, проиллюстрированы способы описания простых комбинационных и последовательностных схем с помощью этого языка. В последующих главах покажем использованиеVHDL для разработки цифровых систем.

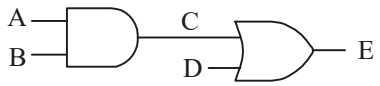
2.1. VHDL-модели для комбинационных схем

Начнем с описания простой вентильной схемы, представленной на рисунке 2.1. Если каждый логический элемент схемы имеет задержку 5ns, она может быть задана следующим образом:

```
C <= A and B after 5 ns;
E <= C or  D after 5 ns;
```

где A, B, C, D и E – сигналы (signal). Сигнал в VHDL обычно соответствует линии в реальном устройстве. Символ "<=" обозначает оператор назначения сигнала, указывающего на то, что вычисляемое значение с правой стороны присваивается сигналу, представленному слева. Во время моделирования первый оператор будет вычисляться в любой момент при изменении A или B, а второй оператор – при изменении C или D. Пусть первоначально A=1, а B=C=D=E=0. Если B становится 1 в нулевой момент времени, то C изменится в 1 в момент времени t = 5 ns. Сигнал E изменится в 1 в момент времени t = 10 ns.

Рисунок 2.1. Вентильная схема



Описанные выше VHDL-операторы назначения сигнала называются параллельными, если они не помещены в VHDL-процесс или блок. VHDL -симулятор проверяет правую часть параллельного оператора. При каждом изменении сигнала выражение с правой стороны вычисляется заново и после соответствующей задержки сигналу с левой стороны присваивается новое значение.

Описывая схему, можно не указывать задержки. Например, в выражении

```
C <= A and B;
E <= C or D;
```

задержки равны нулю. В этом случае симулятор будет использовать бесконечно малую задержку, называемую Δ (дельта). Если *B* изменится в 1 в момент времени t = 0, то *C* изменит свое значение в момент $t = 0 + \Delta$, а *E* – при $t = 0 + 2\Delta$.

В отличие от последовательной программы порядок следования параллельных команд не существенен. Если записать:

```
E <= C or D;
C <= A and B;
```

то результаты моделирования не будут отличаться от предыдущих. Даже если VHDL-программа не имеет явных циклов, параллельные операторы могут выполняться многократно, как если бы они были в цикле. VHDL выражение:

```
CLK <= not CLK after 10 ns;
```

сгенерирует сигнал синхронизации с полупериодом 10 ns. Если CLK первоначально равно '0', то его значение изменится в '1' через 10 ns. Когда CLK изменится в '1', команда

будет выполнена снова, и CLK станет равным '0' через следующие 10 ns. Этот процесс бесконечен. С другой стороны, параллельный оператор

```
CLK <= not CLK;
```

вызывает ошибку времени выполнения (run-time error) при моделировании. Поскольку при нулевой задержке значение сигнала CLK будет изменяться с интервалом $0 + \Delta$, $0 + 2\Delta$, $0 + 3\Delta$ ..., то реальное время никогда не будет достигнуто.

VHDL нечувствителен к регистрам – прописные и строчные буквы обрабатываются одинаково компилятором и симумятором. Таким образом, команды

```
     CLK <= NOT clk After 10 ns;
and  CLK <= not Clk after 10 ns;
```

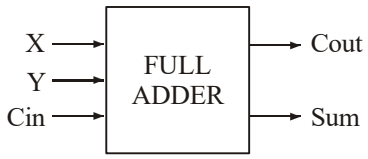
обрабатываются одинаково. Имена сигналов и другие VHDL-идентификаторы могут содержать буквы, числа и символ подчеркивания "_". Идентификатор должен начинаться с буквы и не может заканчиваться подчеркиванием. Таким образом, C123 и ab_23 – корректные идентификаторы, а 1ABC и ABC_ – нет. Каждое выражение в VHDL должно заканчиваться точкой с запятой.

2.1.1. Пары "Entity-Architecture"

Чтобы написать полную VHDL-программу, нужно описать все входные и выходные сигналы и определить тип каждого сигнала. В качестве примера опишем полный сумматор, представленный на рисунке 2.2. Полное описание должно включать интерфейс (entity) и архитектуру. Интерфейс определяет входы и выходы сумматора:

```
entity FullAdder is
   port (X, Y, Cin: in bit;         -- Входы(*
         Cout, Sum: out bit);       -- Выходы
end FullAdder;
(*Большинство современных средств проектирования не руссифицированы
и не позволяют использовать кириллицу для комментариев
```

Рисунок 2.2. Полный сумматор



Cout=(X'YCin+XYCin)+(XY'Cin+XYCin)+(XYCin'+XYCin)= YCin + XCin + XY
Sum= X'(Y'Cin +YCin')+X(Y'Cin'+ YCin)
=X'(Y⊕Cin)+X(Y⊕Cin)'=X⊕Y⊕Cin

Entity, is, port, in, out и **end** – зарезервированные или ключевые слова, имеющие специальное значение для VHDL-компилятора. Здесь и далее все зарезервированные слова выделены жирным шрифтом. Все, что следует за двойной чертой (--) – комментарий. Декларация порта (port) задает входные сигналы X, Y и Cin типа "бит" (bit), и выходные сигналы Cout и Sum типа "бит". Каждый сигнал в этом примере имеет тип "бит", что означает, что он может принимать только значения '0' или '1'.

Функционирование полного сумматора описывается в архитектуре:

```
architecture Equations of FullAdder is
begin        -- Параллельные операторы назначения
   Sum <= X xor Y xor Cin after 10 ns;
   Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
```

В этом примере название архитектуры (Equations) выбрано произвольно, но название объекта (FullAdder) должно соответствовать имени интерфейса (entity). VHDL – операторы присваивания Sum и Cout представляют логические уравнения для полного сумматора (см. рисунок 2.2). Также можно использовать другие архитектурные описания, например, задающие функцию в виде таблицы истинности или логических элементов и связей между ними. В уравнении Cout требуются круглые скобки вокруг (X **and** Y), поскольку в VHDL не определено старшинство операций для логических операторов.

При описании цифровой системы в VHDL нужно задать интерфейс и архитектуру верхнего уровня, а также определить объект и архитектуру для каждого из модулей компонентов, которые являются частью системы (рисунок 2.3). Интерфейс может содержать список сигналов, используемых для соединения с другими модулями или с внешней средой. Синтаксис декларации интерфейса имеет следующую форму:

```
entity entity-name is
   [port(interface-signal-declaration);
end [entity] [entity-name];
```

Элементы, заключенные в квадратные скобки, являются необязательными. Описание сигнала в интерфейсе обычно имеет следующую форму:

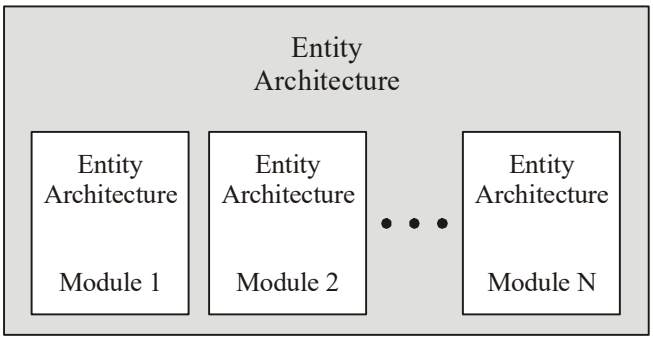
```
Список сигналов интерфейса: тип режима [: = начальное значение]{ ;
Список сигналов интерфейса: тип режима [: = начальное значение]}
```

Фигурные скобки указывают нулевое или некоторое количество повторений данного предложения. Входные сигналы имеют режим **in**, выходные – **out**, двунаправленные – **inout**. В описанных примерах использовался только тип **bit**; другие типы будут описаны в подразд. 2.6 и 2.7. Необязательное начальное значение применяется для инициализации сигналов в связанном списке; в противном случае используется заданное по умолчанию начальное значение для указанного типа. Например, объявление порта:

```
port(A, B: in integer := 2; C, D: out bit);
```

указывает, что A и B – входные сигналы целого типа (integer), которые принудительно получают начальное значение 2, а C и D – выходные, типа bit, которые по умолчанию инициализируются нулем.

Рисунок 2.3. Структура VHDL – программы



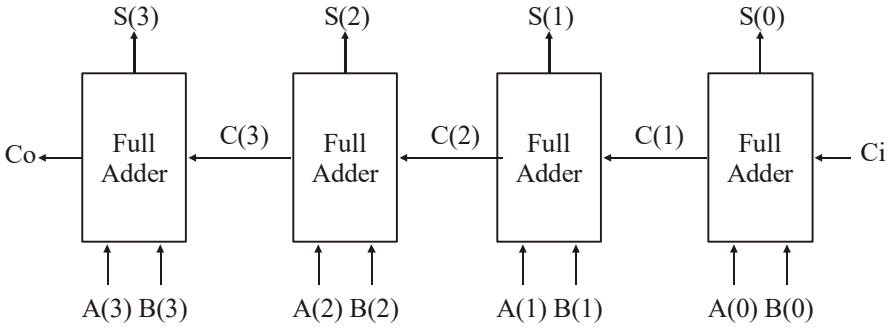
С каждым интерфейсом можно связать одну или несколько деклараций архитектуры вида:

```
architecture architecture-name of entity-name is
          [declarations]
begin
     architecture body
end [architecture] [architecture-name] ;
```

2.1.2. Четырехразрядный полный сумматор

Используем определенный ранее модуль FullAdder, как компонент в системе, состоящей из четырех полных сумматоров, образующих 4-разрядный двоичный сумматор (рисунок 2.4). Определим интерфейс 4-разрядного сумматора (рисунок 2.5). Так как входы и sum-выход имеют размерность 4 бита, то целесообразно объявить их как bit_vectors, с убывающей индексацией 3 **downto** 0. Вместо этого можно также использовать диапазон 1 **to** 4.

Рисунок 2.4. Четырехразрядный двоичный сумматор



Задаем FullAdder как компонент (**component**) внутри архитектуры Adder4 (см. рис. 2.5). Описание компонента аналогично заданию интерфейса полного сумматора, а входные и выходные порты соответствуют сигналам, объявленным для полного сумматора. После оператора компонента объявлен 3-битный внутренний сигнал переноса C.

В архитектуре создаются четыре копии компонента FullAdder. Каждая из них имеет имя (типа FA0) и карту портов (**port map**). Сигналы, следующие после **port map**, соответствуют портам, описанным в декларации компонента. Таким образом, A(0), B(0) и Ci соответствуют входам X, Y и Cin, а C(1) и S(0) – выходам Cout и Sum.

Рисунок 2.5. Структурное описание черырехразрядного сумматора

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;   -- Входы
        S: out bit_vector(3 downto 0); Co: out bit);   -- Выходы
end Adder4 ;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit;          -- Входы
        Cout, Sum: out bit);        -- Выходы
end component ;
signal C: bit_vector(3 downto 1);
begin          -- Реализация четырех копий FullAdder
     FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
     FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
     FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
     FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;
```

При подготовке к моделированию интерфейсы и архитектуры FullAdder и Adder4 можно размещать вместе в одном файле и компилировать. Можно также откомпилировать FullAdder отдельно и разместить результирующий код в библиотеке, которая подключается к Adder4 при компиляции.

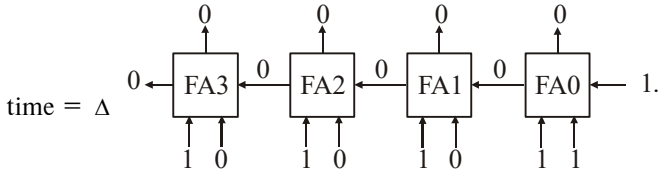
Все примеры моделирования в этом тексте используют симулятор из пакета Active-HDL. Другие VHDL-симуляторы применяют подобные командные файлы и могут производить вывод данных в аналогичном формате. Для тестирования работы Adder4 применяются следующие макрокоманды:

```
asim Adder4
list A B Co C Ci S
-- put these signals on the output list
force A 1111
-- set the A inputs to 1111
force B 0001
-- set the B inputs to 0001
force Ci 1
-- set Ci to 1
run 50 ns
-- run the simulation for 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

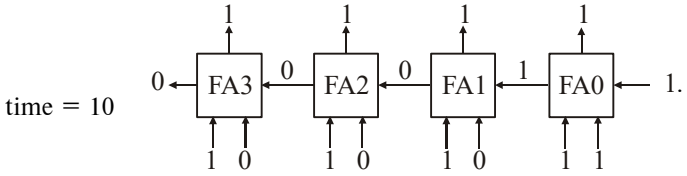
Время моделирования одной операции сложения равно 50 ns. Этого достаточно для передачи сигнала переноса через все сумматоры. Результат моделирования:

ns	delta	A	B	Co	C	Ci	S
0.000	0	0000	0000	0	000	0	0000
0.000	1	1111	0001	0	000	1	0000
10.000	0	1111	0001	0	001	1	1111
20.000	0	1111	0001	0	011	1	1101
30.000	0	1111	0001	0	111	1	1001
40.000	0	1111	0001	1	111	1	0001
50.000	1	0101	1110	1	111	0	0001
60.000	0	0101	1110	1	110	0	0101
70.000	0	0101	1110	1	100	0	0111
80.000	0	0101	1110	1	100	0	0011

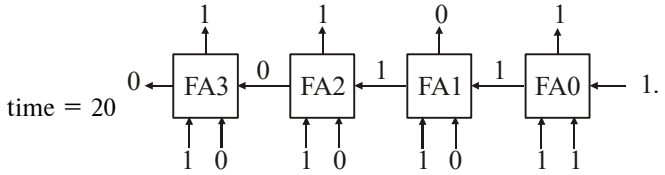
Листинг показывает, как перенос проходит одну позицию каждые 10 ns. Входы полных сумматоров изменяются в момент времени Δ:



Сумма и перенос вычисляются каждым модулем FA и появляются на выходах FA через 10 ns:



Поскольку значения входов в FA1 изменились, следовательно, значения выходов меняются через 10 ns:



Окончательные результаты моделирования:

1111 + 0001 + 1 = 0001 с переносом 1 (в момент времени = 40 ns) и

0101 + 1110 + 0 = 0011 с переносом 1 (в момент времени = 80 ns).

Моделирование останавливается в момент времени t = 80 ns, поскольку после этого не происходит никаких дальнейших изменений.

2.2. Создание VHDL-моделей триггеров

Для создания моделей последовательностных схем в VHDL обычно используется *процесс*. Он может иметь форму:

```
process(sensitivity-list)
begin
   sequential-statements
end process;
```

Всякий раз, когда один из сигналов в списке чувствительности (sensitivity-list) изменяется, инструкции в теле процесса последовательно выполняются один раз.

Следующий пример иллюстрирует различия при выполнении последовательных и параллельных операторов. Программа VHDL имеет сигналы A, B, C и D типа integer, которым присвоены начальные значения: A = 1, B = 2, C = 3 и D = 0. Программа содержит следующие параллельные операторы:

```
A <= B;   -- оператор 1
B <= C;   -- оператор 2
C <= D;   -- оператор 3
```

Предположим, что D изменяется на 4 в момент времени t = 10. Происходит следующая последовательность событий: поскольку сигнал D изменился, выполняется оператор 3, а C изменяется на 4 в момент времени $10 + \Delta$. Затем изменился сигнал C и выполняется оператор 2, а B получает новое значение в момент времени $10 + 2\Delta$. Затем изменение B запускает выполнение оператора1 и A модифицируется в момент $10 + 3\Delta$. Поскольку сигнала A нет в правой части команд, то выполнение программы закончено:

```
time delta   A  B  C  D
 0     +0    1  2  3  0
 10    +0    1  2  3  4   (оператор3 выполняется первым)
 10    +1    1  2  4  4   (затем выполняется оператор 2)
 10    +2    1  4  4  4   (затем выполняется оператор 1)
 10    +3    4  4  4  4   (отсутствие выполняемых команд)
```

Рассмотрим программу с теми же инструкциями, помещенными в процесс:

```
process (B, C, D)
begin
   A <= B;   -- оператор 1
   B <= C;   -- оператор 2
   C <= D;   -- оператор 3
end process;
```

Предположим, что A, B, C и D инициализированы как прежде и D изменяется на 4 в момент времени t =10. Поскольку D находится в списке чувствительности, как только он меняется, процесс начинает выполняться. Последовательно реализуются инструкции 1, 2 и 3; затем процесс возвращается к вершине и ждет до тех пор, пока не будет обнаружено изменение сигнала в списке чувствительности. Выполнение этих трех операторов происходит мгновенно в момент времени t =10; для этого даже не требуется дельта–задержка. Поскольку A, B и C – сигналы, их значения не модифицируются до времени $10 + \Delta$. Поэтому при выполнении операторов используются старые значения B, C и D. Сигналы A, B и C изменят свои значения в момент времени $10 + \Delta$, поэтому выполнение процесса начнется снова. A и B изменятся в момент времени $10 + 2\Delta$ и процесс выполнится в третий раз. В итоге будет получена такая последовательность событий:

```
time delta  A  B  C  D
 0    +0    1  2  3  0
 10   +0    1  2  3  4 (выполняются операторы 1,2,3;
                        затем обновляются A,B,C)
 10   +1    2  3  4  4 (выполняются операторы 1,2,3;
                        затем обновляются A,B,C)
 10   +2    3  4  4  4 (выполняются операторы 1,2,3;
                        затем обновляются A,B,C)
 10   +3    4  4  4  4 (никаких действий не происходит)
```

Этот пример показывает, как операторы назначения сигнала могут использоваться в качестве последовательных команд в процессе. Другая часто используемая последовательная команда – условный оператор **if**. Он имеет форму:

```
if condition then
       sequential statementsl
  else sequential statements2
end if;
```

Condition – булево выражение, принимающее значения TRUE или FALSE. Если оно равно TRUE, выполняется последовательный оператор statementsl; иначе – statements2.

Далее VHDL-процесс используется для создания модели простого D-триггера, который изменяет свое состояние по переднему фронту синхроимпульса. Сигнал QN представляет Q' выход триггера. В описании интерфейса (рисунок 2.6) используется явная инициализация QN в '1', поскольку он должен быть дополнением к Q, а сигналы типа bit инициализируются значением '0' по умолчанию. Синтаксис VHDL требует, чтобы битовые значения типа '0' и '1' были включены в одиночные кавычки. Название архитектуры SIMPLE выбрано произвольно. Поскольку триггер может изменять состояние только при модификации синхровхода, определен процесс, который выполняется лишь при изменении CLK. Таким образом, CLK – единственный сигнал в списке чувствительности. Значение синхросигнала также проверяется внутри процесса. Если CLK = '1', это означает, что на CLK пришел передний фронт. В этом случае Q становится равным D, а QN – дополнением к D. Задержка 10 ns представляет время распространения между изменением синхронизации и выходов триггера.

Рисунок 2.6. Модель D-триггера

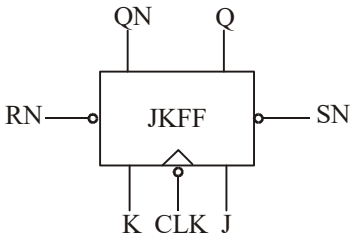
```
entity DFF is
  port (D, CLK: in bit;
        Q: out bit; QN: out bit := '1' ) ;
     -- инициализация QN в '1',
     -- так как bit сигналы инициализируются в '0', по умолчанию
end DFF;

architecture SIMPLE of DFF is
begin
  process (CLK)           -- процесс выполняется при изменении CLK

  begin
     if CLK = '1' then       -- передний фронт синхроимпульса
        Q <= D after 10 ns;
        QN <= not D after 10 ns;
     end if;
  end process;
end SIMPLE;
```

Разработать модель JK-триггера (рисунок 2.7), который имеет входы установки (SN) и сброса (RN) (активный уровень 0 для обоих входов) и меняет свое состояние по заднему фронту синхроимпульса. Далее используется суффикс N для указания активного низкого уровня (0) сигнала. Для простоты предположим, что возникновение

состояния триггера, определяемого как SN = RN = 0, невозможно. Далее будет рассмотрена более совершенная модель, учитывающая этот случай. VHDL-код для JK-триггера приведен на рисунке 2.8. Уравнения для определения состояния триггера есть его характеристические уравнения:

$Q^+ = JQ' + K'Q.$

Рисунок 2.7. JK-триггер



Поскольку процесс выполняется всякий раз, когда изменяются сигналы SN, RN или CLK, следует определить задний фронт CLK. Простая проверка посредством присваивания CLK = '0' не сработает, если процесс будет активизирован изменением SN или RN. Поэтому вместо упомянутого оператора используется конструкция (**elsif** CLK ='0'**and** CLK'event). CLK'event принимает значение TRUE, если CLK только что изменил свое состояние. Таким образом, (**elsif** CLK = '0' **and** CLK'event) равно TRUE, если и только если на CLK пришел задний фронт. CLK'event – пример атрибутов сигнала, которые будут подробно обсуждены в подразд. 9.1.

Рисунок 2.8. Модель JK-триггера

```
entity JKFF is
  port (SN, RN, J, K, CLK: in bit;           -- входы
        Q: inout bit; QN: out bit := '1');-- см. Примечание 1
end JKPF;

architecture JKFF1 of JKFF is begin
  process (SN, RN, CLK)                    -- см. Примечание 2
  begin
     if RN = '0' then Q<= '0' after 10 ns;
                                  -- RN=0, триггер установлен в 0
     elsif SN = '0' then Q<= '1' after 10 ns;
                                  -- SN=0, триггер установлен в 1
     elsif CLK = '0' and CLK'event then      -- см. Примечание 3
        Q <= (J and not Q) or (not K and Q) after 10 ns;
                                             -- см. Примечание 4
     end if;
  end process ;
QN <= not Q;                                 -- см. Примечание 5 :
end JKFF1 ;
```

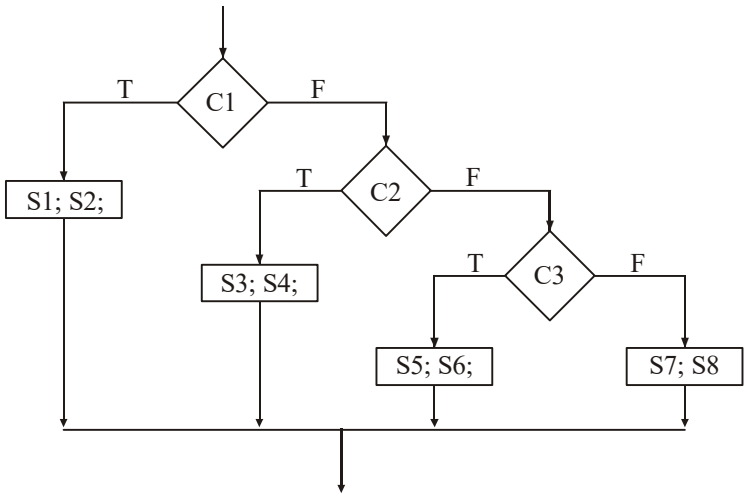
Примечание 1: Q объявлен как **inout**, потому что Q в архитектуре появляется в левой и правой частях оператора назначения сигнала.
Примечание 2: Триггер может изменять состояние в ответ на изменения в SN, RN и CLK, поэтому эти 3 сигнала включены в список чувствительности.
Примечание 3: Условие (CLK = '0' **and** CLK'event) ИСТИННО, если CLK был только что изменен с '1' на '0'.
Примечание 4: Характеристическое уравнение, описывающее поведение JK-триггера.
Примечание 5: Каждый раз, когда Q изменяется, обновляется значение QN. Если бы эта инструкция была помещена в процесс, вместо нового использовалось бы старое значение Q.

Предыдущий пример показывает использование оператора **elsif**, который является альтернативным путем записи вложенных условных операторов. Его общая форма имеет следующий вид:

```
if condition then
  sequential statements
{elsif condition then sequential statements }
  -- 0 or more elsif clauses may be included
[else sequential statements]
end if;
```

Фигурные скобки указывают, что может быть включено любое число операторов **elsif**, а квадратные – что предложение **else** необязательно. Пример на рисуноке 2.9 показывает, как может быть представлена блок-схема с помощью вложенных операторов **ifs** или эквивалентных **elsifs**. В этом примере Cl, C2 и C3 представляют условия, которые могут быть истинны или ложны, а S1, S2, ..., S8 задают последовательные ореаторы. Каждый **if** требует соответствующего **end if**, а **elsif** – нет.

Рисунок 2.9. Эквивалентные описания блок-схемы с использованием вложенных Ifs и Elsifs




```
if (Cl) then SI; S2;
   else if (C2) then S3; S4;
      else if (C3) then S5; S6;
         else S7; S8;
      end if;
   end if;
end if;
```

```
if (Cl) then SI; S2 ;
   elsif (C2) then S3; S4;
   elsif (C3) then S5; S6;
else S7; S8;
end if;
```

2.3. VHDL-модель мультиплексора

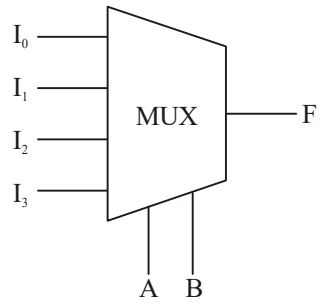
На рисунке 2.10 изображен мультиплексор (MUX) "из 4 в 1" с четырьмя входами данных и двумя управляющими, A и B. Последние формируют адрес входа данных, которые будут поступать на выход. Логическое уравнение для MUX "из 4 в 1" имеет следующий вид:

$$F = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3 .$$

Таким образом, один из способов описания модели мультиплексора с помощью операторов VHDL имеет вид:

```
F <= (not A and not B and I0) or (not A and B and II) or (A and not
B and I2) or (A and B and I3);
```

Рисунок 2.10. Мультиплексор "из 4 в 1"



Для создания поведенческой модели мультиплексора можно использовать условный оператор назначения сигнала (*conditional signal assignment statement*). Этот оператор имеет форму:

```
signal_name <= expression1 when condition1
     else expression2 when condition2
. . .
     [else expressionN];
```

Параллельный оператор соответствует последовательному оператору if. Он выполняется всякий раз, когда один из сигналов в выражениях или условиях изменяет значение. Если condition1 истинно, signal_name устанавливается равным значению expression1, иначе, если condition2 истинно, signal_name устанавливается равным значению expression2, ...

Модель мультиплексора имеет вид:

```
F <= 10 when Sel = 0
     else I1 when Sel = 1
     else I2 when Sel = 2
     else I3;
```

где Sel – целочисленный эквивалент 2-разрядного двоичного числа с битами A и B.

Также модель мультиплексора можно создать, используя селективный оператор назначения сигнала (*selected signal assignment statement*):

```
with expression select
   signal_name  <= waveform1 when choices1,
                waveform2 when choices2,
                . . .
                [waveformN when choicesN,]
                [waveformN+1 when others];
```

Селективный оператор назначения сигнала во многом подобен условному оператору. Он выполняется, если происходит изменение сигнала в expression или waveforms. Сначала вычисляется expression для определения варианта выбора. Если его значение соответствует choices1, сигнал signal_name получает значение waveform1; если – choices2, сигнал signal_name получает значение waveform2. Оператор должен включать все возможные варианты выбора, иначе используется оператор **others**. Варианты выбора могут представлять собой диапазон или перечисление параметров. Не разрешается перекрывать (пересекать условия) значения в операторе выбора.

Модель мультиплексора, созданная с использованием селективного оператора назначения сигнала:

```
with Sel select
   F  <= I0 when 0,
         I1 when 1,
         I2 when 2,
```

```
        I3 when 3;
    end case;
```

Если для создания модели мультиплексора используется оператор процесса, параллельный оператор не может применяться. В таком случае можно использовать последовательный оператор выбора **case**:

```
case Sel is
  when 0 => F <= I0;
  when 1 => F <= I1;
  when 2 => F <= I2;
  when 3 => F <= I3;
end case;
```

Оператор **case** обычно записывается в форме:

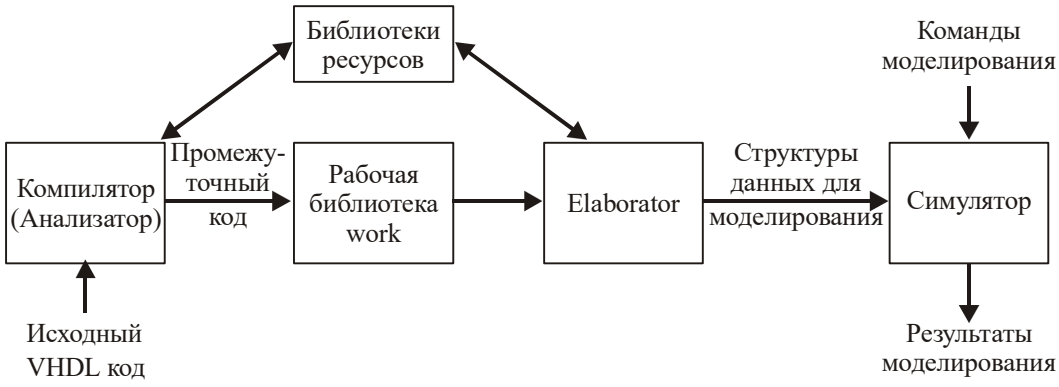
```
case expression is
     when choice1 => sequential statements1
     when choice2 => sequential statements2
     . . .
     [when others => sequential statements]
end case ;
```

Прежде всего вычисляется выражение expression. Если оно равно choice1, то выполняется последовательный оператор statements1; если – choice2, то выполняется последовательный оператор statements2. Все возможные значения выражения должны быть включены в поля выбора. Если они явно не задаются, в **case** нужно использовать идентификатор **when others**.

2.4. Компиляция и моделирование VHDL-кода

Необходимость моделирования после описания проектируемой системы на VHDL определяют следующие две причины. Во-первых, следует проверить, правильно ли VHDL-код реализует проект; во-вторых, соответствует ли последний техническому заданию. Перед началом моделирования VHDL-код должен быть откомпилирован (рисунок 2.11). VHDL-компилятор (или анализатор) проверяет исходный код VHDL на соответствие синтаксическим и семантическим правилам VHDL. Компилятор выводит сообщение, если обнаруживает синтаксическую ошибку (например, отсутствует точка с запятой) или семантическую (например, выполняется сложение двух сигналов несовместимых типов). Он также проверяет правильность ссылок к библиотекам. Если VHDL-код соответствует всем правилам, компилятор генерирует промежуточный код, который может использоваться программой моделирования или синтеза.

Рисунок 2.11. Компиляция, разработка и моделирование VHDL - кода



В ходе подготовки к моделированию промежуточный VHDL-код должен быть преобразован в форму, приемлемую для программы моделирования. Этот шаг называется *разработкой*. В процессе ее создаются порты для каждой реализации компо-

нента, выделяется память для требуемых сигналов, определяются взаимосвязи между сигналами портов и в надлежащей последовательности устанавливается механизм для запуска процессов VHDL. Результирующие структуры данных представляют собой цифровую систему для моделирования. После стадии инициализации программа моделирования переходит в стадию выполнения. Она принимает команды моделирования, которые управляют анализом цифровой системы, и выводит результаты.

В качестве примера рассмотрим шаги моделирования VHDL- кода, представленного на рисунке 2.12. Ключевое слово **transport** определяет тип задержки (типы задержек будут обсуждаться в подразд. 9.2). На этапе разработки для каждого сигнала создется драйвер. Он содержит текущее значение сигнала и очередь будущих. Каждый раз, когда предполагается, что сигнал изменится в будущем, новое значение помещается в очередь с указанием времени, в котором намечено изменение этого сигнала.

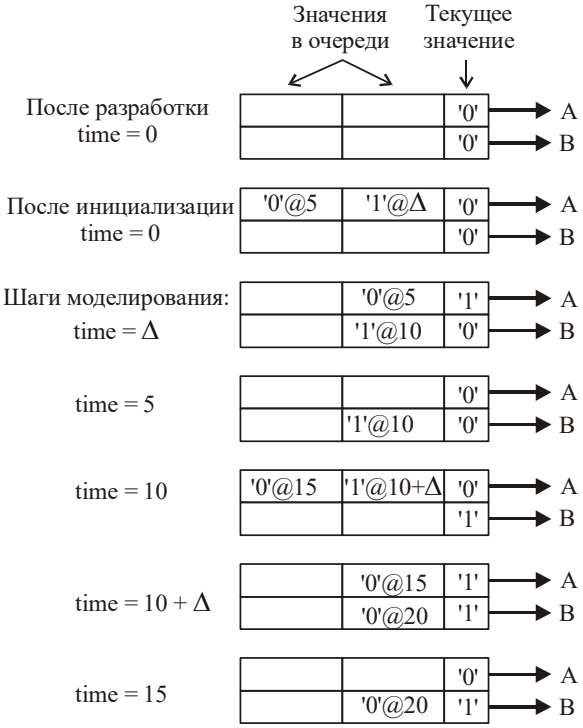
Рисунок 2.12. VHDL-код для примера моделирования

```
entity simulation_example is
end simulation_example;
architecture testi of simulation_example is
   signal A,B: bit;
begin
   P1: process(B)
   begin
      A <= '1';
      A <= transport '0' after 5 ns;
   end process P1;
   P2: process(A)
   begin
      if A = '1' then B <= not B after 10 ns; end if;
   end process P2 ;
end testi;
```

На рисунке 2.13 представлены драйверы для сигналов A и B. После того, как этап разработки закончен, каждый драйвер содержит '0'.Это задаваемое по умолчанию начальное значение для элементов типа bit. В начале моделирования происходит инициализация. Оба процесса выполняются одновременно один раз и приостанавливаются в ожидании изменения значения сигнала из списка чувствительности. При выполнении процесса P1 в момент времени t = 0 два изменения A заносятся в очередь будущих значений (A устанавливается в '1' при t = Δ и возвращается в '0' в момент t =5 ns). В то же время выполняется процесс P2, но B не изменяется, поскольку A все еще равно '0' при t = 0 ns. В момент Δ A изменяется на '1'. Изменение в A вызывает выполнение процесса P2. И поскольку A = '1', изменение B на '1' в момент времени t =10 ns заносится в очередь будущих значений. Следующее ожидаемое изменение произойдет при t = 5 ns, когда A изменится на '0'. Это запустит на выполнение процесс P2, но B не изменится. Сигнал B становится равным '1' в момент t = 10 ns. Изменение B вызывает выполнение процесса P1.При этом в очередь для сигнала A будут занесены два изменения. Когда A станет равным '1' в момент времени 10 ns + Δ , выполнится процесс P2 и B изменится в момент 20 ns. Затем A изменится при t = 15 ns и моделирование будет продолжаться до тех пор, пока не будет достигнут предел времени выполнения.

VHDL-симуляторы используют событийное моделирование (event-driven simulation), иллюстрируемое предыдущим примером. Изменение сигнала является *событием (event)*. Каждый раз, когда происходит событие, все процессы, ожидающие его, выполняются мгновенно, а результирующие изменения сигнала заносятся в очередь, чтобы инициировать процесс в будущем. После завершения обработки всех активных процессов программа моделирования переходит к следующему по времени событию из списков будущих значений сигналов. Это продолжается до тех пор, пока не будут обработаны все события или будет исчерпан лимит времени выполнения.

Рисунок 2.13. Драйверы сигнала для примера моделирования



2.5. Проектирование последовательностных схем

2.5.1. Цифровые автоматы

Рассмотрим несколько способов создания VHDL-моделей последовательностных автоматов. В качестве примера спроектируем автомат Мили (рисунок 2.14), который принимает число в двоично-десятичном коде 8-4-2-1 и выдает его увеличенным на 3. После этого автомат должен переходить в начальное состояние и ожидать поступления следующего входного кода.

Рисунок 2.14. Общая модель автомата Мили

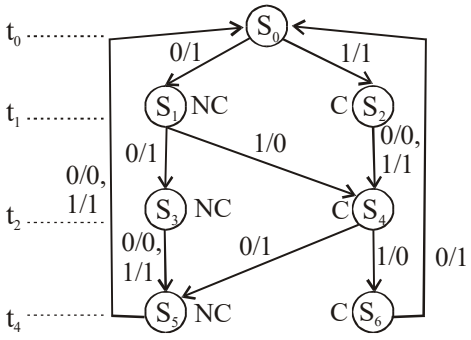
Таблица 2.1. Функционирование автомата

Вход X				Выход Z			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Используя таблицу функционирования автомата (таблица 2.1), построим граф состояний. В момент времени t_0 добавим единицу к младшему разряду числа. Если X=0, то Z=1(без переноса), а если X=1, то Z=0 (с переносом в следующий разряд). S0 – это начальное состояние, S1 – означает отсутствие переноса после первой операции сложения, S2 – перенос 1 в следующий разряд.

В момент времени t_1 добавим единицу к следующему биту, поступившему на вход. При отсутствии переноса из младшего разряда (автомат находится в состоянии S1) X=0 дает Z=0+1+0=1 без переноса в старший разряд (состояние S3), X=1 дает Z=1+1+0=0 с переносом в старший разряд (состояние S4). При наличии переноса из младшего разряда (состояние S2) X=0 дает Z=0+1+1=0 и перенос в старший разряд (состояние S4), а X=1 дает Z=1+1+1=1 и перенос (состояние S4).В момент времени t_2 полученный разряд складывается с 0, а состояния S5 и S6 определяются аналогичным образом. В момент времени t_3 выполняется сложение последнего разряда с 0 и автомат переходит в состояние S0. На рисунке 2.15 приведен граф состояний и соответствующая ему таблица переходов автомата.

Рисунок 2.15. Граф состояний и таблица переходов и выходов автомата Мили



NC – без переноса, C – с переносом

а

Q	Q+		Z	
	X=0	X=1	X=0	X=1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S6	S6	1	0
S5	S0	S0	0	1
S6	S0	-	1	-

б

2.5.2. Минимизация автоматов

Граф автомата, изображенный на рисунке 2.15, является минимальным. Если же разработанный автомат необходимо минимизировать, для этого используют приведенную ниже стратегию.

Алгоритм минимизации состояний автомата.

Пусть имеем следующую таблицу переходов автомата:

Q	Q+		Z	
	X=0	X=1	X=0	X=1
a	c	f	0	0
b	d	e	0	0
c	h	g	0	0
d	b	g	0	0
e	c	b	0	1
f	f	a	0	1
g	c	g	0	1
h	c	f	0	0

Рассмотрим состояния a и h.Они эквивалентны, поэтому строку h можно удалить из таблицы, а все символы h заменить на a. Для того чтобы установить, есть ли еще эквивалентные символы в таблице, дадим определение эквивалентности. По таблице состояния a и b эквивалентны, $a \equiv b$, если и только если $c \equiv d$ и $e \equiv f$. Другими словами, c-d и e-f – связанные пары для a-b. Для того чтобы определить все связанные пары, необходимо построить диаграмму. На пересечении строки b и столбца a записываем

c-d и e-f. Если два состояния имеют различные значения на выходах, то они считаются не эквивалентными, например, e≠d и соответствующий им квадрат зачеркивается крестом. В следующий проход зачеркиваются квадраты, содержащие связанные пары, для которых соответствующий им квадрат зачеркнут, – необходимо зачеркнуть все квадраты, содержащие связанные пары e-f и f-g. Последнюю итерацию следует продолжать до тех пор, пока нельзя будет зачеркнуть ни одного квадрата. Незачеркнутыми в таблице останутся только эквивалентные состояния, в данном случае это a ≡ b, c ≡ d и e ≡ f. Диаграммы связанных состояний и полученная минимальная таблица переходов приведены на рисунках 2.16 и 2.17.

Минимизация автомата Мура выполняется аналогичным образом.

Рисунок 2.16. Диаграммы связанных состояний

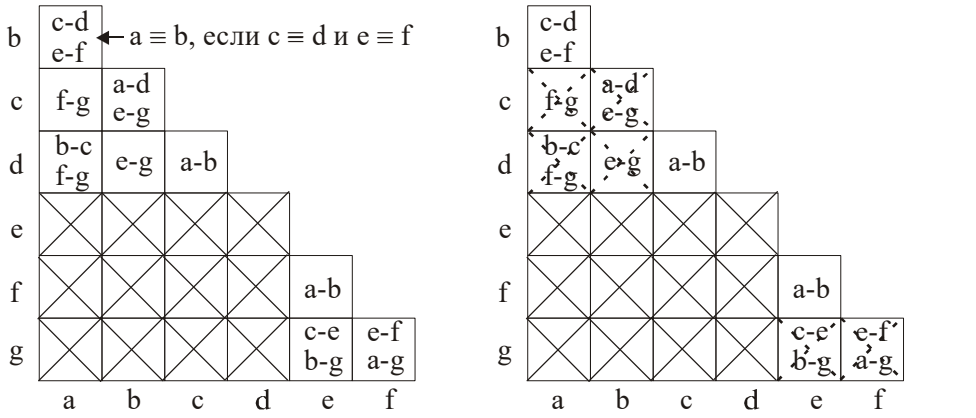


Рисунок 2.17. Минимизированная таблица переходов

Q	Q^+		Z	
	X=0	X=1	X=0	X=1
a	c	e	0	0
c	a	g	0	0
e	c	a	0	1
g	c	g	0	1

2.5.3. Создание VHDL-моделей для автоматов

Сначала создадим поведенческую модель для последовательностной схемы Мили, основанной на таблице переходов (см. рисунок 2.15, б). Как показано на рисунке 2.14, автомат Мили состоит из комбинационной схемы и регистра состояний. Для их представления в VHDL-модели используются два процесса (рисунок 2.18). На поведенческом уровне состояния автомата в моменты времени t и t+1 описаны сигналами типа "integer", устанавливаемыми в нуль в начальный момент времени. Первый процесс представляет комбинационную схему. Учитывая, что выходы Z и Nextstate могут изменяться, когда модифицируется состояние State или входы X, список чувствительности содержит сигналы State и X. Оператор case проверяет состояние State, после чего Z и Nextstate получают новые значения, в зависимости от сигнала X. Второй процесс соответствует регистру состояния. Всякий раз, когда на вход синхронизации приходит передний фронт, State получает значение сигнала Nextstate, поэтому CLK находится в списке чувствительности.

На рисунке 2.18 State имеет тип integer. Поскольку только семь значений данного типа используется в качестве вариантов для выбора в операторе **case**, для остальных

значений включена инструкция **when others => null**. Оператор **null** означает отсутствие какого-либо действия.

Рисунок 2.18. Поведенческая модель автомата Мили

```
-- Это поведенческая модель устройства Мили (рисунок 2.15),
-- основанная на таблице переходов. Вычисление значения на выходе
-- (Z) и следующее состояние происходит до поступления активного
-- синхросигнала. Переход в следующее состояние выполняется по
-- переднему фронту синхроимпульса.
entity SM1_2 is
  port(X, CLK: in bit;
       Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
  signal State, Nextstate: integer := 0;
begin
  process(State,X)              -- Комбинационная часть устройства
  begin
     case State is
        when 0 =>
           if X='0' then Z<='1'; Nextstate<=1; end if;
           if X='1' then Z<='0'; Nextstate<=2; end if;
        when 1 =>
           if X='0' then Z<='1'; Nextstate<=3; end if;
           if X='1' then Z<='0'; Nextstate<=4; end if;
        when 2 =>
           if X='0' then Z<='0'; Nextstate<=4; end if;
           if X='1' then Z<='1'; Nextstate<=4; end if;
        when 3 =>
           if X='0' then Z<='0'; Nextstate<=5; end if;
           if X='1' then Z<='1'; Nextstate<=5; end if;
        when 4 =>
           if X='0' then Z<='1'; Nextstate<=5; end if;
           if X='1' then Z<='0'; Nextstate<=6; end if;
        when 5 =>
           if X='0' then Z<='0'; Nextstate<=0; end if;
           if X='1' then Z<='1'; Nextstate<=0; end if;
        when 6 =>
           if X='0' then Z<='1'; Nextstate<=0; end if;
        when others => null;            -- отсутствие действия
     end case;
  end process;

  process(CLK)                     -- регистр состояния
  begin
       if CLK='1' then          -- передний фронт синхроимпульса
       State <= Nextstate;
     end if;
  end process;
end Table;
```

Ниже представлены команды программы моделирования, которые могут быть использованы для тестирования разработанной модели (см. рисунок 2.16).

```
asim SM1_2
wave CLK X State NextState Z
force -repeat 200 ns CLK 0,1 100 ns
force X 0 0 ns, 1 350 ns, 0 550 ns, 1 750 ns, 0 950 ns, 1 1350 ns
run 1600 ns
```

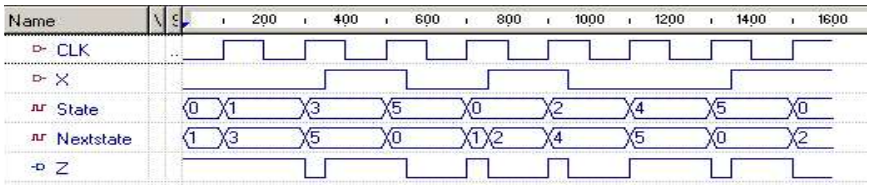
Первая команда инициализирует моделирование устройства SM1_2. Вторая указывает сигналы, которые должны быть включены в окно waveform для наблюдения.

Следующая команда описывает синхросигнал с периодом 200 ns. CLK = '0' в момент времени 0 ns, CLK = '1' – в момент времени 100 ns.Такие изменения повторяются через каждые 200 ns. В команде вида:

```
force signal_name vl t1, v2 t2, ...
```

signal_name получает значение vl в момент времени t1, v2 – в момент t2,... . Сигнал X, равный '0' в момент времени 0 ns, изменяется на '1' в момент 350 ns, затем модифицируется на '0' в момент времени 550 ns,... . Вход X соответствует последовательности 0010 1001 тогда и только тогда, когда определены изменения на X. При выполнении командного файла получаются временные диаграммы (waveforms), показанные на рисунке 2.19.

Рисунок 2.19. Waveforms для автомата Мили



Функциональная VHDL-модель автомата уровня регистровых передач (рисунок 2.20) основана на уравнениях состояний автомата в момент времени t+1 и выходов, которые получены в результате синтеза:

$D_1=Q_1^+=Q_2'$ $D_1=Q_2^+=Q_1$ $D_3=Q_3+=Q_1Q_2Q_3+XQ_1Q_3'+XQ_1'Q_2'$
$Z=XQ_3'+XQ_3$.

Триггеры Q1, Q2 и Q3 модифицируются в процессе с синхросигналом CLK в списке чувствительности и получают новые значения по переднему фронту синхроимпульса. Для моделирования промежутка времени между появлением активного фронта синхросигнала и изменением на выходах триггеров используется задержка в 10 ns. Несмотря на то, что все операции назначения сигнала выполняются в процессе последовательно, Q1, Q2 и Q3 будут модифицированы одновременно, в момент времени $T + \Delta$, где T – время появления синхроимпульса. Таким образом, старое значение Q1 используется для вычисления $Q2^+$, а старые значения Q1, Q2 и Q3 – для определения $Q3^+$. Параллельный оператор назначения сигнала для Z обновляет его всякий раз при изменении X или Q3. Величина 20 ns есть задержка двух вентилей.

Рисунок 2.20. Модель последовательного автомата, использующая уравнения

```
-- Описание автомата с использованием уравнений состояний автомата
-- и уравнения выхода.
-- Состояния автомата кодируются следующим образом:
-- S0-->0; S1-->4; S2-->5; S3-->7; S4-->6; S5-->3; S6-->2

entity SM1_2 is
  port(X,CLK: in bit;
       Z: out bit) ;
end SM1_2 ;

architecture Equations1_4 of SM1_2 is
  signal Q1,Q2,Q3: bit;
begin
  process(CLK)
  begin
    if CLK='1' then                  -- передний фронт синхросигнала
      Q1<=not Q2 after 10 ns;
      Q2<=Q1 after 10 ns;
      Q3<=(Q1 and Q2 and Q3) or ((not X) and Q1 and (not Q3)) or
         (X and (not Q1) and (not Q2)) after 10 ns;
```

```
    end if;
  end process;
  Z<=((not X) and (not Q3)) or (X and Q3) after 20 ns;
end Equations1_4;
```

На рисунке 2.21 представлена схема, которая получена при реализации автомата Мили (см. рисунок 2.15). Структурная VHDL-модель, соответствующая данной схеме, приведена на рисунке 2.22. В ней используется семь вентилей И-НЕ(NAND), три D-триггера и один инвертор. Все эти элементы определены в библиотеке BITLIB. Их описания содержатся в пакете, называемом BitPackage (листинг BitPackage).

2.5.4. Листинг пакета BitPackage

```
-- Пакет для реализации операций над элементами типа bit и bit_vector
-- Декларативная часть пакета
package BitPackage is
  function add4 (regl,reg2: bit_vector(3 downto 0);carry: bit)
    return bit_vector;
  function falling_edge(signal clock:bit)
    return Boolean;
  function rising_edge(signal clock:bit)
    return Boolean;
  function vec2int(vecl: bit_vector)
    return integer;
  function int2vec(int1,NBits: integer)
    return bit_vector;
  procedure Addvec    (Add1,Add2: in bit_vector; Cin: in bit;
                       signal Sum: out bit_vector;
                       signal Cout: out bit;   n: in natural);
  component jkff
    generic(DELAY:time := 10 ns);
    port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
  end component;
  component dff
    generic(DELAY:time := 10 ns);
    port (D, CLK: in bit; Q: out bit; QN: out bit :=  '1');
  end component;
  component and2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
  end component;
  component and3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
  end component;
  component and4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
  end component;

  component or2
    generic(DELAY:time := 10 ns);
    port(A1, A2: in bit; Z: out bit);
  end component;
  component or3
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3: in bit; Z: out bit);
  end component;
  component or4
    generic(DELAY:time := 10 ns);
    port(A1, A2, A3, A4: in bit; Z: out bit);
  end component;
  component nand2
```

```
      generic(DELAY:time := 10 ns);
      port(Al, A2: in bit; Z: out bit);
   end component;
   component nand3
      generic(DELAY:time := 10 ns);
      port(Al, A2, A3: in bit; Z: out bit);
   end component;
   component nand4
      generic(DELAY:time := 10 ns);
      port(Al, A2, A3, A4: in bit; Z: out bit);
   end component;

   component nor2
      generic(DELAY:time := 10 ns);
      port(Al, A2: in bit; Z: out bit);
   end component;
   component nor3
      generic(DELAY:time := 10 ns);
      port(Al, A2, A3: in bit; Z: out bit);
   end component;
   component nor4
      generic(DELAY:time := 10 ns);
      port(Al, A2, A3, A4: in bit; Z: out bit);
   end component;
   component inverter
      generic(DELAY:time := 10 ns);
      port(A : in bit; Z: out bit);
   end component;
   component xor2
      generic(DELAY:time := 10 ns);
      port(Al, A2: in bit; Z: out bit);
   end component;
   component c74163
      port(LdN, CIrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
           Cout: out bit; Q: inout bit_vector(3 downto 0) );
   end component;
end BitPackage;
package body BitPackage is

-- Функция выполняет сложение двух чисел, представленных 4-битными
-- векторами, и возвращает 5-битовую сумму
function add4 (regl,reg2: bit_vector(3 downto 0);carry: bit)
   return bit_vector is
variable cout: bit:= '0';
variable cin: bit:=carry;
variable retval: bit_vector(4 downto 0):="00000";
begin
   Ipl: for i in 0 to 3 loop
      cout:=(reg1(i) and reg2(i)) or ( regl(i) and cin) or (reg2(i)
and cin);
      retval(i):= regl(i) xor reg2(i) xor cin;
      cin := cout;
   end loop Ipl;
retval(4):=cout;
return retval;
end add4;

-- Функция определяет задний фронт синхросигнала
function falling_edge(signal clock:bit)
   return Boolean is
begin
   return clock'event and clock = '0';
end falling_edge;
```

```
-- Функция определяет передний фронт синхросигнала
function rising_edge(signal clock:bit)
  return Boolean is
begin
  return clock'event and clock = '1';
end rising_edge;

-- Функция vec2int преобразует bit_vector в integer
function vec2int(vecl: bit_vector)
  return integer is
variable retval: integer:=0;
alias vec: bit_vector(vecl'length-1 downto 0) is vecl;
begin
  for i in vec'high downto 1 loop
    if (vec(i) = '1' ) then retval:=(retval+1)*2;
                       else retval:=retval*2;
    end if;
  end loop;
  if vec(0)= '1' then retval:=retval + 1; end if;
  return retval;
end vec2int;

-- Функция int2vec преобразует положительное целое число в
bit_vector
function int2vec(int1,NBits: integer)
  return bit_vector is
variable N1: integer;
variable retval: bit_vector(NBits-1 downto 0);
begin
  assert int1 >= 0;
    report "Function int2vec: input integer cannot be negative"
    severity error;
  N1:=int1;
  for i in retval 'Reverse_Range loop
    if (N1 mod 2)=1 then retval(i):= '1';
                    else retval(i):='0';
    end if;
    N1:=N1/2;
  end loop;
  return retval;
end int2vec;

-- Процедура выполняет сложение двух n-битовых векторов и переноса,
-- возвращает n-битовую сумму и перенос. Предполагается, что
-- слагаемые вектора Add1 и  Add2 имеют одинаковую длину и
-- направление n-1 downto 0.
procedure Addvec
     (Add1,Add2: in bit_vector; Cin: in bit;
     signal Sum: out bit_vector;
     signal Cout: out bit;
     n:in natural) is
  variable C: bit;
begin
  C := Cin;
  for i in 0 to n-1 loop
    Sum(i) <= Add1(i) xor Add2(i) xor C;
    C:=(Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
  end loop;
  Cout <= C;
end Addvec;

end BitPackage;
```

```
-- 2-входовой вентиль И
entity And2 is
   generic(DELAY:time);
   port (A1,A2: in bit;
         Z: out bit);
end And2;
architecture concur of And2 is
begin
   Z <= A1 and A2 after DELAY;
end;
-- 3-входовой вентиль И
entity And3 is
   generic(DELAY:time);
   port (A1,A2, A3:in bit;
         Z: out bit);
end And3;
architecture concur of And3 is
begin
Z <= A1 and A2 and A3 after DELAY;
end;

--4-входовой вентиль И
entity And4 is
   generic(DELAY:time);
   port (A1,A2,A3,A4: in bit;
         Z: out bit) ;
end And4 ;
architecture concur of And4 is
begin
   Z <= A1 and A2 and A3 and A4 after DELAY;
end;

--2-входовой вентиль ИЛИ
entity Or2 is
   generic(DELAY:time);
   port (A1,A2: in bit;
         Z: out bit);
end Or2 ;
architecture concur of Or2 is
begin
   Z <= A1 or A2 after DELAY;
end;

--3-входовой вентиль ИЛИ
entity Or3 is
   generic(DELAY:time);
   port (A1,A2,A3: in bit;
         Z: out bit) ;
end Or3;
architecture concur of Or3 is
begin
   Z <= A1 or A2 or A3 after DELAY;
end;

--4-входовой вентиль ИЛИ
entity Or4 is
   generic(DELAY:time) ;
   port (A1,A2,A3,A4: in bit;
         Z: out bit) ;
end Or4;
architecture concur of Or4 is
begin
   Z <= A1 or A2 or A3 or A4 after DELAY;
end;
```

```
--2-входовой вентиль И-НЕ
entity Nand2 is
   generic(DELAY:time);
   port (A1,A2: in bit;
         Z: out bit);
end Nand2;
architecture concur of Nand2 is
begin
   Z <= not (A1 and A2) after DELAY;
end;
--3-входовой вентиль И-НЕ
entity Nand3 is
   generic(DELAY:time);
   port (A1,A2, A3: in bit;
         Z: out bit);
end Nand3;
architecture concur of Nand3 is
begin
   Z <= not (A1 and A2 and A3) after DELAY;
end;
--4-входовой вентиль И-НЕ
entity Nand4 is
   generic(DELAY:time);
   port (A1,A2,A3,A4: in bit;
         Z: out bit);
end Nand4;
architecture concur of Nand4 is
begin
   Z <= not (A1 and A2 and A3 and A4) after DELAY;
end;
--2-входовой вентиль ИЛИ-НЕ
entity Nor2 is
   generic(DELAY:time);
   port (A1,A2: in bit;
         Z: out bit);
end Nor2;
architecture concur of Nor2 is
begin
   Z <= not (A1 or A2) after DELAY;
end;
--3-входовой вентиль ИЛИ-НЕ
entity Nor3 is
   generic(DELAY:time);
   port (A1,A2,A3: in bit;
         Z: out bit);
end Nor3;
architecture concur of Nor3 is
begin
   Z <= not (A1 or A2 or A3) after DELAY;
end;
--4-входовой вентиль ИЛИ-НЕ
entity Nor4 is
   generic(DELAY:time);
   port (A1,A2,A3,A4: in bit;
         Z: out bit);
end Nor4;
architecture concur of Nor4 is
begin
   Z <= not (A1 or A2 or A3 or A4) after DELAY;
end;
--Инвертор
entity Inverter is
   generic(DELAY:time) ;
   port (A: in bit;
```

```
        Z: out bit) ;
end Inverter;
architecture concur of Inverter is
begin
   Z <= not A after DELAY;
end;
--2-входовой вентиль сложения по модулю 2
entity XOR2 is
   generic(DELAY:time);
   port (A1,A2: in bit;
        Z: out bit) ;
end XOR2 ;
architecture concur of XOR2 is
begin
   Z <= A1 xor A2 after DELAY;
end;
--JK-триггер
entity JKFF is
   generic(DELAY:time);
   port(SN, RN, J, K, CLK: in bit;
        Q, QN: inout bit);
end JKFF;
use work.BitPackage.all;
architecture JKFF1 of JKFF is
begin
   process(CLK, SN, RN)
   begin
      if RN= '0' then Q <= '0' after DELAY;
      elsif SN= '0' then Q<= '1' after DELAY;
      elsif falling_edge(CLK) then
         Q <= (J and not Q) or (not K and Q) after DELAY;
      end if;
   end process;
   QN <= not Q;
end JKFF1;
--D-триггер
entity DFF is
   generic(DELAY:time) ;
   port (D, CLK: in bit;
        Q: out bit; QN: out bit := '1');
-- инициализация QN в '1', поскольку по умолчанию bit тип
-- инициализируется в '0'
end DFF;
architecture SIMPLE of DFF is
begin
   process(CLK)
   begin
      if CLK = '1' then            -- передний фронт синхросигнала
         Q <= D after DELAY;
         QN <= not D after DELAY;
      end if;
   end process;
end SIMPLE;
--74163 счетчик
entity c74163 is
   port(LdN, CIrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
        Cout: out bit; Q: inout bit_vector(3 downto 0));
end c74163;
use work.BitPackage.all;
architecture b74163 of c74163 is
begin
   Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
   process
   begin
      -- изменение состояния по переднему фронту синхросигнала
```

```
      wait until CK = '1';
      if CirN = '0' then Q <= "0000";
      elsif LdN = '0' then Q <= D;
      elsif (P and T) = '1' then
         Q <= int2vec(vec2int(Q)+1, 4);
      end if;
   end process;
end b74163;
```

Операторы **library** и **use** описаны в подразд. 2.11. Поскольку Q1, Q2 и Q3 после инициализации устанавливаются в '0', дополнительные выходы триггера (Q1N, Q2N и Q3N) инициализируются значением '1'; G1 – вентиль 3И-НЕ – с входами Q1, Q2, Q3 и выходом A1; FF1 – D-триггер (см. рисунок. 2.5) со входом D, связанным с Q2N. Все вентили и триггер в BitPackage по умолчанию имеют задержку 10 ns. При выполнении приведенных ниже макро-команд получены результаты моделирования (рисунок 2.23), которые аналогичны результатам, полученным для первой модели:

```
asim SM1_2
wave CLK X Ql Q2 Q3 Z
force -repeat 200 ns CLK 0,1 100 ns
force X 0 0 ns, 1 350 ns, 0 550 ns, 1 750 ns, 0 950 ns, 1 1350 ns.
run 1600 ns
```

Рисунок 2.21. Пример реализации вентильной схемы автомата Мили

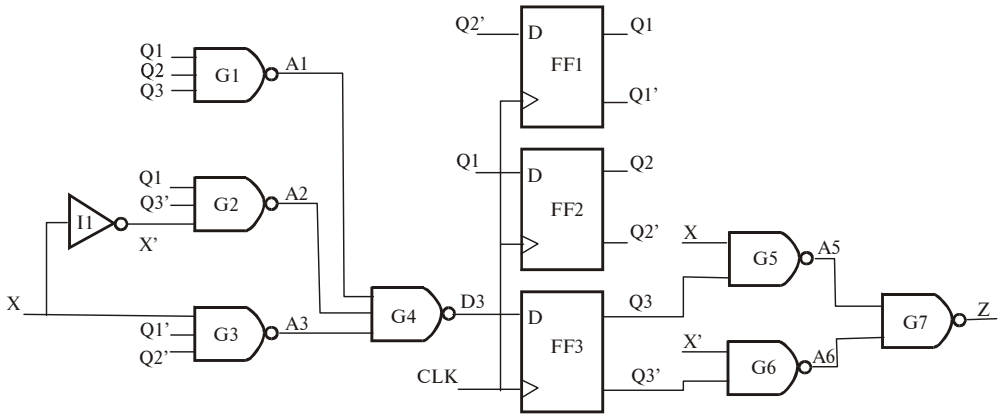
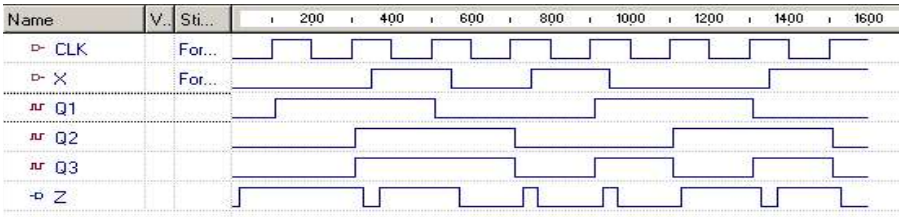


Рисунок 2.22. Структурная VHDL-модель автомата

```
-- Структурное VHDL-описание схемы

library BITLIB;
use BITLIB.BitPackage.all;

entity SM1_2 is
   port(X,CLK: in bit;
            Z: out bit);
end SM1_2;

architecture Structure of SM1_2 is
   signal A1,A2,A3,A5,A6, D3 : bit:='0';
   signal Q1,Q2,Q3: bit:='0';
   signal Q1N,Q2N,Q3N, XN: bit:='1';
begin
   I1: Inverter port map (X,XN);
   G1: Nand3 port map (Q1,Q2,Q3,Al);
   G2: Nand3 port map (Q1,Q3N,XN,A2);
   G3: Nand3 port map (X,Q1N,Q2N,A3);
   G4: Nand3 port map (A1,A2,A3,D3);
   FF1: DFF port map (Q2N,CLK,Q1,Q1N);
   FF2: DFF port map (Q1,CLK,Q2,Q2N);
```

```
   FF3: DFF port map (D3,CLK,Q3,Q3N);
   G5: Nand2 port map (X,Q3,A5);
   G6: Nand2 port map (XN,Q3N,A6);
   G7: Nand2 port map (A5,A6,Z);
end Structure;
```

Рисунок 2.23. Waveforms для структурного описания схемы



Другой формой описания является процесс с оператором **wait** вместо списка чувствительности. В процессе не может одновременно использоваться оператор **wait** и список чувствительности. Процесс с оператором **wait** имеет форму:

```
process
begin
   sequential-statements
   wait-statement
   sequential-statements
   wait-statement
end process;
```

Последовательные операторы в процессе будут выполняться по порядку до оператора **wait**. После этого процесс приостанавливается и ожидает выполнения условия, указанного в операторе **wait**. Затем выполнение процесса продолжается до следующего оператора **wait** или пока не будет достигнут его конец. Тогда выполнение начнется снова с начальной точки процесса.

Операторы **wait** могут иметь три следующие формы:

```
wait on sensitivity-list;
wait for time-expression;
wait until boolean-expression;
```

В первом случае ожидается изменение сигналов в списке чувствительности. Вторая форма ждет, пока время, указанное в выражении, не будет достигнуто. Если используется **wait for** 5 ns, процесс приостанавливается на 5 ns. Если используется **wait for** 0 ns, то задержка равна Δ. Для третьей формы булево выражение вычисляется всякий раз, когда меняется один из сигналов в выражении. Процесс продолжается, пока выражение в операторе **wait until** равно TRUE. Например,

```
wait until A = B;
```

будет ожидать изменения A или B. Затем вычисляется выражение A=B, и если результат TRUE, выполнение процесса возобновляется; иначе процесс будет ожидать очередного изменения A или B, чтобы снова проверить истинность выражения A= B.

Следующий пример (рисунок 2.24) представляет поведенческую модель устройства Мили, состоящую из одного процесса с оператором **wait**. Оператор **case** здесь такой же, как и на рисунке 2.18. После выполнения оператора **case** процесс приостанавливается, ожидая события на синхровходе или на входе X. По переднему фронту синхроимпульса состояние автомата изменяется. Второй оператор **wait for** 0 ns гарантирует, что состояние автомата будет модифицировано до следующего выполнения оператора.

На рисунке 2.24 показано, что если X изменяется в тот же момент времени, в который на синхронизацию приходит передний фронт, для вычисления значений

Nextstate и Z будет использоваться новое значение X, и временные диаграммы будут корректны. Если X меняется после прихода переднего фронта синхронизации, Nextstate и Z будут уже вычислены с использованием старого значения X. Изменение в X приведет к очередному вычислению Nextstate и Z, и временные диаграммы будут также корректны. Для описания конечного автомата более предпочтительна двухпроцессная модель, чем модель с одним процессом, так как она точнее моделирует реальные аппаратные средства. В случае использования CAD-систем для автоматического синтеза инструкции, подобные **wait for** 0 ns, не разрешены.

Для моделирования задержки распространения на регистре состояний можно модифицировать VHDL-код (см. рисунок 2.20), внеся следующие изменения в оператор **if**:

```
if rising_edge(CLK) then
     state <= Nextstate after delayl;
     wait for delayl;
end if;
```

Тогда процесс не реагирует на изменение X, которое произошло в момент ожидания delayl. При этом устройство при моделировании допускает сбои, которые происходят на выходе waveform. Это несущественно, если поведенческая модель последовательностной схемы используется для проверки выходной последовательности. Если модель используется как часть большой системы, правильность синхронизации может быть существенной.

Рисунок 2.24. Поведенческая модель с одним процессом для автомата Мили

```
-- Это поведенческая модель автомата Мили, основанная на его таблице
-- переходов. Выход (Z) и следующее состояние вычисляется по
-- переднему фронту синхросигнала или при изменении на входе (X).
-- Изменение состояния происходит по переднему фронту синхросигнала.

library BITLIB;
use BITLIB.BitPackage.all;

entity SM1_2 is
  port(X, CLK: in bit;
           Z: out bit) ;
end SM1_2;

architecture Table of SM1_2 is
  signal State, Nextstate: integer := 0;
begin
  process
  begin
  case State is
     when 0 =>
          if X='0' then Z<='1';  Nextstate<=1;  end if;
          if X='1' then Z<='0';  Nextstate<=2;  end if;
     when 1 =>
          if X='0' then Z<='1';  Nextstate<=3;  end if;
          if X='1' then Z<='0';  Nextstate<=4;  end if;
     when 2 =>
          if X='0' then Z<='0';  Nextstate<=4;  end if;
          if X='1' then Z<='1';  Nextstate<=4;  end if;
     when 3 =>
          if X='0' then Z<='0';  Nextstate<=5; end if;
          if X='1' then Z<='1';  Nextstate<=5; end if;
     when 4 =>
          if X='0' then Z<='1';  Nextstate<=5; end if;
          if X='1' then Z<='0';  Nextstate<=6; end if;
     when 5 =>
          if X='0' then Z<='0';  Nextstate<=0; end if;
          if X='1' then Z<='1';  Nextstate<=0; end if;
```

```
        when 6 =>
             if X='0' then Z<='1';  Nextstate<=0; end if;
           when others => null;          -- отсутствие действия
        end case;
        wait on CLK, X;
                        -- функция rising_edge из библиотеки BITLIB
        if (rising_edge(CLK)) then
             State <= Nextstate;
                        --оператор wait для обновления состояния State
             wait for 0 ns;
        end if;
      end process ;
end table;
```

2.6. Переменные, сигналы и константы

До этого момента в процессах использовались только сигналы без переменных. Они могут применяться для локального хранения данных в процессах, процедурах и функциях. Декларация переменной имеет форму

```
variable list_of_variable_names : type_name [ := initial_value];
```

Переменные должны быть объявлены в процессе, в котором они используются и являются локальными для него. Исключение из этого правила составляют разделенные (shared) переменные. Сигналы, напротив, должны быть объявлены вне процесса. Они предназначены для связи между процессами. Сигналы, объявленные в начале архитектуры, могут использоваться в любом месте в пределах этой архитектуры. Объявление сигнала имеет форму:

```
signal list_of_signal_names : type_name [ := initial_value ];
```

Обычная форма описания константы:

```
constant constant_name : type_name := constant_value;
```

Постоянная задержка *delay1* типа time, имеющая значение 5 ns, может быть определена как:

```
constant delayl: time: =5 ns;
```

Константы, объявленные в начале архитектуры, могут использоваться в любом ее месте, но константы, объявленные в пределах процесса, являются локальными для него. Переменные модифицируются, используя оператор присваивания вида:

```
Variable_name: = expression;
```

При выполнении этого оператора переменная изменяется мгновенно, без задержки. Не используется даже дельта-задержка. Для контраста рассмотрим присваивание сигнала вида

```
Signal_name < = expression [after delay];
```

В данном случае выражение expression программирует сигнал на изменение после задержки, при отсутствии последней сигнал будет запрограммирован на изменение после дельта-задержки.

Примеры на рисунках 2.25 и 2.26 иллюстрируют различие между использованием переменных и сигналов в процессе. Первые должны быть объявлены и инициализированы в процессе, тогда как сигналы следует объявить и инициализировать вне процесса. Из рисунка 2.25 видно, что если сигнал trigger изменяется в момент времени

t =10, то в тот же момент переменные Varl, Var2 и Var3 будут последовательно вычислены и модифицированы и эти новые значения будут использованы при вычислении Sum. Допустим, что переменные равны Varl =2+3=5, Var2 = 5, Var3 = 5. Тогда Sum = 5 + 5 + 5. Поскольку Sum – это сигнал, то он модифицируется с задержкой на время Δ, поэтому Sum = 15 в момент времени 10 + Δ. Если сигнал trigger изменяется в момент времени t = 10 (см. рисунок 2.26), то сигналы Sigl, Sig2, Sig3 и Sum будут все вычислены в момент времени t=10, но они не будут модифицированы до времени t =10 + Δ. Старые значения Sigl и Sig2 используются для вычисления Sig2 и Sig3. Поэтому в момент времени 10 + Δ Sigl = 5, Sig2 = 1, Sig3= 2 и Sum = 6.

Рисунок 2.25. Процесс, использующий переменные

```
entity dummy is
end dummy ;

architecture var of dummy is
   signal trigger, sum: integer:=0;
begin
   process
   variable var1: integer:=1;
   variable var2: integer:=2;
   variable var3: integer:=3;
      begin
         wait on trigger;
         var1 := var2 + var3;
         var2 := var1 ;
         var3 := var2 ;
         sum <= var1 + var2 + var3;
   end process;
end var;
```

Рисунок 2.26. Процесс, использующий сигналы

```
entity dummy is
end dummy;

architecture sig of dummy is
   signal trigger, sum: integer:=0;
   signal sig1: integer:=1;
   signal sig2: integer:=2;
   signal sig3: integer:=3;
begin
   process
   begin
      wait on trigger;
      sig1 <= sig2 + sig3;
      sig2 <= sig1;
      sig3 <= sig2;
      sum <= sig1 + sig2 + sig3;
   end process;
end sig;
```

2.7. Типы данных в VHDL

Переменные, сигналы и константы могут иметь любой из предопределенных типов VHDL или определяемый пользователем тип. Тип задает возможные значения объекта и операции, которые могут быть выполнены над ним. В VHDL существует 4 класса типов:

скалярные (scalar types) – не содержат подэлементов;

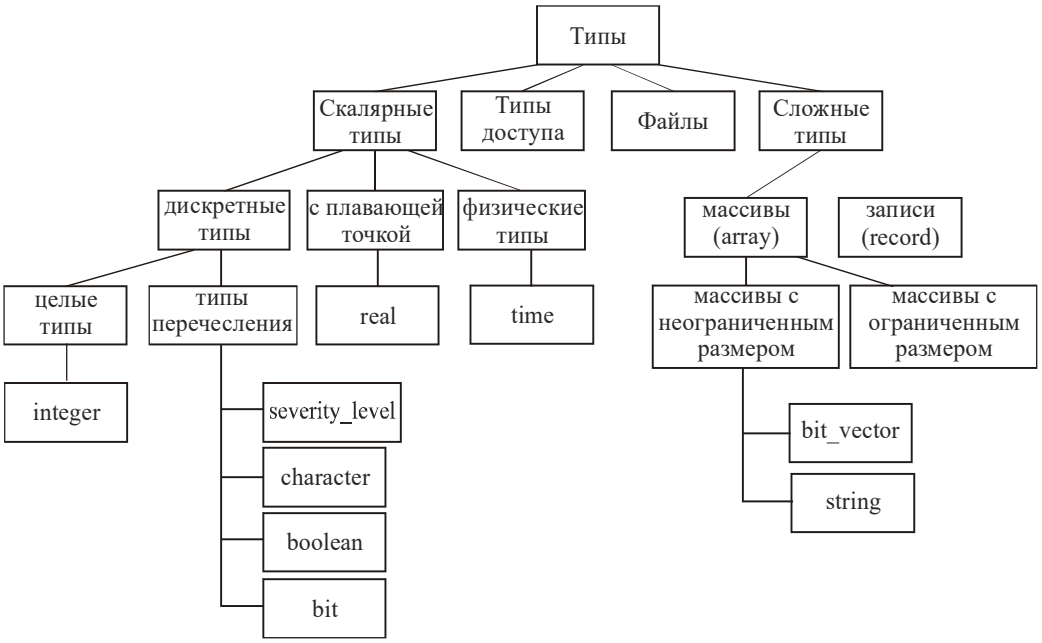
сложные (composite types) – состоят из подэлементов;

доступа (access types) – обеспечивают доступ к объектам заданного типа (указатели);

файлы (files) – обеспечивают доступ к объектам , содержащим последовательность заданного типа.

Классификация типов в VHDL приведена на рисунке 2.27.

Рисунок 2.27. Классификация типов



Определение типа

Для задания нового типа в VHDL используют декларацию типа. Синтаксис его определения имеет следующий вид:

type identifier **is** type_definition;

где, identifier – имя типа, type_definition – описание типа.

Следует отметить важную особенность: если два типа объявлены раздельно – это разные типы. Объекты данных типов нельзя сравнивать, нельзя использовать в операциях, требующих данных одинаковых типов. Например, при определении

type apples **is range** 0 **to** 100;

type oranges **is range** 0 **to** 100;

нельзя присвоить значения типа apples переменной oranges, так как это разные типы.

Подтипы

Часто модель содержит объекты, которые имеют ограниченный диапазон целого множества значений. Их можно описывать, объявляя как подтип (subtype). Он содержит сокращенное множество значений базового типа.

Описать подтип можно с помощью следующих синтаксических правил:

```
subtype identifier is name [range simple_expression (to||downto)
simple_expression];
```

где identifier – имя подтипа, name – имя базового типа.

Например, зададим определение подтипа целого типа:

```
subtype small_in is integer range -128 to 127;
```

Значения типа small_in ограничены диапазоном 128 – 127. Используя типы small_int и integer, можно объявить переменные и использовать их в одном выражении:

```
variable deviation: small_int;
variable adjusment: integer;
deviation:=deviation+adjustment;
```

В данном случае можно смешивать значения подтипа и базового типа. Нельзя переменной подтипа присваивать значения базового типа, если они выходят за его диапазон.

Скалярные типы

Значения скалярного типа не имеют подэлементов.

Integer types

Integer соответствует целым числам. Стандарт VHDL определяет диапазон целых чисел от - 2 147 483 647 до 2 147 483 647 ($-2^{31}+1 – +2^{31}-1$).

Можно задать новый тип integer, задав его диапазон следующим образом:

```
range simple_expression (to || downto) simple_expression
```

Выражения simple_expression должны иметь целые значения. Например, даны определения двух целых типов:

```
type day_of_month is range 0 to 31;
type year is range 0 to 2100;
```

Это два различных типа, даже если они имеют одно и то же значение. Используя полученные типы, определим переменные:

```
variable today: day_of_month:=9;
variable start_year:=1987;
```

Поскольку типы day_of_month и year являются различными, нельзя выполнить следующее присваивание:

```
start_year:=today;
```

Если для описания границ диапазона используются выражения, то значения, применяемые в нем, должны быть локально-статическими (locally static). Они должны быть известны в момент анализа модели. Например, это могут быть константы:

```
constant number_of_bits: integer:=32;
type bit_index is range 0 to number_of_bits-1;
```

По умолчанию переменные целого типа при инициализации получают значение, равное крайнему левому из диапазона.

VHDL-стандарт включает два предопределенных подтипа целого типа:

```
sybtype natural is integer range 0 to highest integer;
subtype positive is integer range 1 to highest integer;
```

Когда из проекта видно, что число не может иметь отрицательного значения, считается хорошим стилем использовать один из приведенных подтипов, вместо целого типа.

Floating-point types

Типы с плавающей точкой применяются для представления вещественных чисел. В VHDL предопределен тип с плавающей точкой – **real**, который может иметь диапазон от -1.0E+38 до +1.0E+38 с точносью до 6 десятичных чисел. Это соответствует IEEE стандарту 32-го представления чисел с плавающей точкой.

Можно определить новый тип с плавающей точкой, указывая границы диапазона типа:

```
range simple_expresion (to||downto) simple_expression;
```

Это определение аналогично заданию интервала целого типа, за исключением границ диапазона, которые в данном случае должны быть числами с плавающей точкой:

```
type input_level is range -10.0 to +10.0;
type probability is range 0.0 to 1.0;
```

По умолчанию переменные типа с плавающей точкой инициализируются крайним левым значением диапазона. Например, следующая переменная после инициализации будет равна -10.0:

```
variable input_A: input_level;
```

Phisical types

Последний численный тип в VHDL – это физический тип. Он используется для определения реальных физических величин, таких как длина, масса, время и электрический ток. Имеется два типа определения единиц измерения в описании физического типа: первичные и вторичные, которые определяются в терминах первичных единиц. Синтаксис физического типа

```
range simple_expression (to||downto) simple_expression
  units
     identifier;
     { identifier=physical_literal;}
  end units [identifier]
```

Самый первый оператор после ключевого слова **units** определяет наименьшие единицы измерения. Ниже приведено описание физического типа, представляющего электрическое сопротивление:

```
type resistance is range 0 to 1E9
  units
     ohm;
  end units resistance;
```

Примеры численных значений переменных данного типа: 5 ohm, 22 ohm. Необходимо оставлять пробел перед единицами измерения. Численное значение 1 можно опускать, записи *ohm* и *1 ohm* – равнозначны.

Пример описания вторичных единиц измерения:

```
type resistance is range 0 to 1E9
  units
     ohm;
     kohm=1000 ohm;
     Mohm=1000 kohm;
  end units resistance;
```

Несмотря на наглядность и привлекательность, физические типы не синтезируются.

В VHDL существует предопределенный физический тип – время (time). Он очень важен в VHDL, используется для описания задержек и имеет следующее определение:

```
type time is range implementetion defined
  units
     fs;
     ps=1000 fs;
     ns=1000 ps;
     us=1000 ns;
     ms=1000 us;
     sec=1000 ms;
     min=60 sec;
     hr=60 min;
  end units;
```

По умолчанию первичная единица измерения fs является пределом разрешения, который может быть использован при создании моделей. Промежуток времени, меньше предела разрешения, округляется до нуля.

Типы перечисления (Enumeration types)

Часто при создании моделей на абстрактных уровнях полезно использовать множества имен, обозначающих некоторые сигналы. Это можно сделать с помощью типа перечисления (Enumeration type). Например, при моделировании процессора можно определить имена функций для АЛУ:

```
type alu_function is (disable, pass, add, substract, multiply,
divide);
```

Синтаксис определения типа перечисления

```
(( identifier|character_literal) {,...})
```

В списке должен находиться хотя бы один элемент-идентификатор или символ. Примеры определения и использования типов перечисления:

```
type octal_digit is ('0', '1', '2', '3', '4', '5', '6', '7');

variable alu_op:alu_fuction;
variable last_digit:octal_digit:='0';

alu_op:=subtract;
last_digit:='7';
```

Разные типы перечисления могут включать одинаковые идентификаторы и только из контекста можно узнать, к какому типу они принадлежат. При инициализации переменных типов перечисления (по умолчанию) они получают значения крайнего левого элемента из списка.

Существует несколько предопределенных типов перечисления:

```
type severiti_level is (note, warring, error, failure);
type file_open_status is (open_ok, status_error, name_error,
mode_error);
type file_open_kind is (read_mode, write_mode, append_mode);
```

Character

CHARACTER – тип перечисления, включающий три группы ASCII символов:

– стандартное множество управляющих символов, представленных буквенным обозначением;

– стандартное множество символов;

– расширенное множество, представленное символами или обозначениями, состоящими из первой буквы C и трех десятичных чисел, соответствующих ASCII коду символа (например, C128, C147):

```
type CHARACTER is (
nul, soh, stx, etx, eot, enq, ack, bel,
bs, ht, lf, vt, ff, cr, so, si,
dle, dc1, dc2, dc3, dc4, nak, syn, etb,
can, em, sub, esc, fsp, gsp, rsp, usp,
' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', del,
c128, c129, c130, c131, c132, c133, c134, c135,
c136, c137, c138, c139, c140, c141, c142, c143,
c144, c145, c146, c147, c148, c149, c150, c151,
c152, c153, c154, c155, c156, c157, c158, c159,
' ', '¡', '¢', '£', '¤', '¥', '¦', '§',
'¨', '©', 'ª', '"', '¬', '-', '®', '¯',
'°', '±', '²', '³', '´', 'µ', '¶', '·',
'¸', '¹', 'º', '"', '¼', '½', '¾', '¿',
'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç',
'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï',
'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×',
'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß',
'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç',
'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï',
'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷',
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ' );
```

Пример использования переменных типа character:

```
variable cmd_char, terminator: character;
cmd_char:='P';
terminator:=cr;
```

SEVERITY_LEVEL_Type

SEVERITY_LEVEL – тип перечисления, содержащий четыре значения: NOTE, WARNING, ERROR и FAILURE и представляющий уровень серьезности ошибки при использовании операторов контроля.

Описание:

```
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

Boolean

Тип boolean имеет следующее описание:

```
type boolean is (false, true);
```

Результатом операций отношения будут значения типа boolean. Операции равно "=" и не равно"/=" могут быть применены к операндам различных типов (кроме типа файл),

включая и сложные типы, но оба операнда должны быть одного типа. Например, результат операций

123=123 'A'='A' 7 ns=7 ns

будет иметь значение true, а

123=256 'A'='z' 7 ns=2 ns – значение false.

Операции "<", "<=", ">", "=>" могут быть применены к типам, имеющим порядок, включая все скалярные типы.

Логические операторы and, or, nand, nor, xor, xnor и not применимы к операндам булевого типа и производят результат булевого типа.

Bits

Поскольку VHDL используется для моделирования цифровых систем, полезно иметь тип данных для представления битовых значений:

```
type bit is ('0','1');
```

Логические операторы могут быть применены к значениям типа бит, результат операций также имеет тип бит. Например, '0' **and** '1' = '0', '1' **xor** '1' ='0'. Тем не менее смешение типов недопустимо. Выражение '0' **and** true будет ошибочным.

Разница между типами **bit** и **boolean** в том, что boolean используется для описания абстрактных моделей, а тип bit – для моделей аппаратно-логического уровня.

Standart logic

Предопределенный тип bit используется для описания более абстрактных моделей, когда не концентрируется внимание на более подробном описании. Однако, когда надо повысить детализацию проекта, необходимо учитывать электрические свойства используемых сигналов. Для этого существует много способов, но один из них – это IEEE стандартный пакет str_logic_1164. Std_ulogic – один из типов, описанных в пакете:

```
type std_ulogic is ('U',  -- Неинициализированный
        'X', -- Сильное неизвестное
        '0', -- Сильный 0
        '1', -- Сильный 1
        'Z', -- Высокий импеданс
        'W', -- Слабое неизвестное
        'L', -- Слабый 0
        'H', -- Слабая1
        '-', -- Don't Care
```

Type Qualification

Когда из контекста неясно, какого типа отдельные значения, тогда он указывается перед данными, которые берутся в скобки. Например,

```
type logic_level is (unknow, low, undriven, high);
type system_state is (unknown, ready, busy);


logic_level'(unknown), system_state'(unknown)
```

Преобразования типов

VHDL – язык со строгим контролем за использованием типов. В арифметических операторах применяются операнды одного типа. Это предотвращает смешивание операторов в арифметических выражениях. В случаях, когда это необходимо, для

преобразования значений между типами integer и floating point можно использовать их преобразование. Например,

```
real(123)  integer(3.6)
```

При преобразовании real в integer выполняется округление числа в сторону ближайшего целого.

2.8. Массивы

Для использования массива в VHDL необходимо сначала объявить его тип, а затем – его элементы. Например, следующее объявление определяет тип одномерного массива по имени SHORT_WORD:

```
type SHORT_WORD is array (15 downto 0) of bit;
```

Массив этого типа имеет целочисленный индекс с диапазоном от 15 до 0 и каждый элемент представлен типом "бит".

Для объявления элементов типом SHORT_WORD необходимо определить:

```
signal      DATA_WORD:   SHORT_WORD;
variable    ALT_WORD:    SHORT_WORD := "0101010101010101";
constant    ONE_WORD:    SHORTS-WORD := (others => '1');
```

DATA_WORD – сигнал, имеющий тип – 16-битный массив, с индексацией от 15 до 0, который по умолчанию имеет начальные значения всех битов, равные '0'. ALT_WORD – переменная, имеющая тип – 16-битный массив, начальными значениями которого являются чередующиеся нули и единицы. ONE_WORD – константа, имеющая тип – 16-битный массив, все биты которого установлены в "1" (**others** = > '1'). Используя индексы, можно обращаться к отдельным элементам массива. Например, ALT_WORD(0) относится к самому младшему биту массива ALT_WORD. Можно также обратиться к сегменту массива для определения индексного диапазона: ALT_WORD (5 downto 0) соответствует шести младшим битам массива ALT_WORD, которые имеют по умолчанию значения 010101.

Тип и элемент массива имеют здесь форму:

```
type array_type_name is array index_range of element_type;
signal array_name: array_type_name [ := initial_values ];
```

В приведенном объявлении сигнал может быть заменен переменной или константой.

Можно определять типы многомерных массивов, имеющих два или больше измерений. Следующий пример определяет переменную двухмерного массива, который является матрицей целых чисел с четырьмя строками и тремя столбцами:

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 := ((1, 2, 3), (4, 5, 6), (7, 8, 9),
(10, 11, 12));
```

Переменная matrixA будет иметь вид:

```
1  2  3
4  5  6
7  8  9
10 11 12
```

Элемент массива matrixA (3,2) есть пересечение третьей строки и второго столбца, которое имеет значение 8.

При объявлении типа массива его размер можно оставить неопределенным. Такой массив имеет неопределенный (или неограниченный) тип. Например,

```
type intvec is array (natural range <>) of integer;
```

описывает тип intvec как одномерный массив целых чисел с неограниченным индексным диапазоном натуральных чисел. По умолчанию для индексов массива используется тип integer, но может быть задан и другой. Поскольку диапазон индексов не определен в массиве неограниченного типа, его необходимо указать при объявлении объекта данного типа. Например,

```
signal intvec5: intvec(1 to 5) := (3,2,6,8,1)
```

определяет массив сигнала, названный intvec5 с индексным диапазоном от 1 до 5, имеющий начальные значения 3, 2, 6, 8, 1. Следующее объявление определяет двумерную матрицу с неограничеными размерами строки и столбца:

```
type matrix is array (natural range <>, natural range <>) of
integer;
```

VHDL-код на рисунке 2.28 представляет поведенческую модель для последовательностной логической схемы, изображенной на рисунке 2.15, использующую массивы для представления таблиц состояний и выходов. Определено два двумерных массива: integer индексация – для таблицы состояний и bit индексация – для таблицы выходов. В обоих случаях первый и второй индекс – целое число и бит из неограниченного диапазона. Когда фактические таблицы состояния и выходов объявляются как константы, задаются фактические индексные диапазоны: для строк – от 0 до 6, для столбцов – от '0' до 1'. Каждый раз, когда изменяется X или State, таблицы состояния и выходов читаются параллельными инструкциями для определения значений NextState и Z. State модифицируется по переднему фронту синхроимпульса.

Рисунок 2.28. Модель последовательностной цифровой схемы на основе таблиц состояний и выходов

```
entity SM1_2 is
  port (X, CLK: in bit;
        Z: out bit);
end SM1_2 ;

architecture Table of SM1_2 is
  type StateTable is array (integer range <>, bit range <>) of
integer;
  type OutTable is array (integer range <>, bit range <>) of bit;
  signal State, NextState: integer := 0;
  constant ST: StateTable (0 to 6, '0' to '1') :=
     ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
  constant OT: OutTable (0 to 6, '0' to '1') :=
       (('1','0'),('1','0'), ('0','1'), ('0','1'), ('1', '0'),
       ('0','1'), ('1','0'));
begin                              -- параллельные операторы
  NextState <= ST(State,X);
          -- читается следующее состояние из таблицы состояний
  Z <= OT(State,X);  -- читается значение на выходе из таблицы
выходов
  process(CLK)
     begin
        if CLK = '1' then          -- передний фронт на CLK
           State <= NextState;
     end if;
  end process;
end Table;
```

В следующем примере для индексов массива используется тип перечисления:

```
type controller_state is (initial, idle, active, error);
type state_counts is array (idle to error) of natural;
```

Такое описание для индексации можно применять, если из контекста ясно, какой тип используется. Если в модели существует несколько типов перечисления, использующих одинаковые элементы, то применяется другой способ описания индексов, в котором явно указывается тип. Предыдущий пример может выглядеть следующим образом:

```
type state_counts is array (controller_state range idle to error) of
natural;
```

Если описываемый массив будет индексироваться всем диапазоном типа индексов, то такой интервал указывать необязательно:

```
subtype coeff_ram_address is integer range 0 to 63;
type coeff_array is array (coeff_ram_address) of real;
```

VHDL имеет следующие предопределенные типы массивов с неограниченным размером:

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;
```

Символы в строковой константе должны быть заключены в двойные кавычки. Например, "This is string." является строковой константой. Следующий пример объявляет константу stringi типа string:

```
constant stringi: string(1 to 29) := "This string is 29 characters."
```

Константа bit_vector может быть описана или как список битов, отделенных запятыми, или как строка. Например, записи ('1', '0', '1', '1', '0') и "10110" – эквивалентны. Следующий пример объявляет константу A, которая имеет тип bit_vector с диапазоном от 0 до 5:

```
constant A : bit_vector(0 to 5) := "101011";
```

После объявления типа можно определить связанный с ним подтип, включающий подмножество значений, определенных базовым типом. Например, SHORT_WORD, который был определен ранее, можно определить как подтип bit_vector:

```
subtype SHORT_WORD is bit_vector (15 downto 0);
```

Битовые строки могут быть записаны в двоичном, восьмеричном и шестнадцатеричном виде. Буква перед битовой строкой определяет систему счисления:

B – двоичная,
O – восьмеричная,
X – шестнадцатеричная.

Базис можно обозначать большой и малой буквой:

```
B"0100011"   B"10"   b"111_0010_0001"   B""
```

В битовую строку можно включать символ подчеркивания. Он разделяет соседние элементы для удобства чтения и не имеет никакого смыслового значения:

```
O"372"=B"011_111_010"   o"00"=B"000_000"
X"FA"=B"1111_1010" x"0d"=B"0000_1101"
```

2.9. Операторы VHDL

Предопределенные операторы VHDL могут быть разбиты на семь классов:

1. Двоичные логические операторы: **and, or, nand, nor, xor, xnor.**

2. Операторы отношения: **=, /=, <, <=, >, >=.**

3. Операторы сдвига: **sll, sri, sla, sra, rol, ror.**

4. Операторы сложения: **+ , - , & ,** (**конкатенация**).

5. Одинарные знаковые операторы: **+ , -.**

6. Операторы умножения: ***, / , mod, rem.**

7. Смешанные операторы: **not, abs,** **.

Когда не используются круглые скобки, операторы из 7-го класса имеют самый высокий приоритет и обрабатываются первыми, затем идут операторы 6-го класса, 5-го... . Операторы 1-го класса имеют самый низкий приоритет и обрабатываются последними. Операторы одного класса имеют одинаковый приоритет и обрабатываются в выражении по порядку слева направо. Порядок старшинства может быть изменен, если используются круглые скобки. В следующем выражении A, B, C и D имеют тип bit_vector:

```
(A & not B or C ror 2 and D) = "110010"
```

Операторы обрабатываются в следующем порядке:

```
not, &, ror, or, and, =
```

Например, если A = "110", B = "111", C = "011000" и D = "111011", вычисление будет осуществляться следующим образом:

```
not B = "000" (побитовое дополнение)
A &not B = " 110000" (конкатенация)
C ror 2 = "000110" (сдвиг вправо на 2 разряда)
(A & not B) or (C ror 2) = " 110110 (побитовое "или")
(A &not B or C ror 2) and D = " 110010" (побитовое "и")
[(A &not B or C ror 2 and D) = " 110010"] = TRUE (с помощью скобок
производится проверка на равенство левой и правой частей и результат
– ИСТИНА).
```

Двоичные логические операторы (класс 1), так же как и оператор **not,** могут применяться к битам, булевым переменным, бит-векторам и булевым векторам. Операторы класса 1 требуют двух операндов одинакового типа и результат имеет тот же тип, что и операнды.

Результат использования операторов отношения (класс 2) всегда имеет булево значение (FALSE или TRUE). Знак равенства (=) и неравенства (/ =) может применяться почти к любому типу. Другие операторы отношения могут применяться к любому численному или перечисляемому типу, так же как и к некоторым типам массивов. Например, если A = 5, B=4 и C = 3, выражение (A > = B) and (B < = C) будет ложным (FALSE).

Операторы сдвига могут быть применены к любым битовым и булевым векторам. В следующем примере bit_vector A равен "10010101":

A **sll** 2 – "01010100" (логический сдвиг влево, с заполнением '0')
A **srl** 3 – "00010010" (логический сдвиг вправо, с заполнением '0')
A **sla** 3 – "10101111" (арифметический сдвиг влево, с заполнением '1')
A **sra** 2– "11100101" (арифметический сдвиг вправо, с заполнением '1')
A **rol** 3 – "10101100" (циклический сдвиг влево)
A **ror** 5 – "10101100" (циклический сдвиг вправо)

Операторы + и - могут использоваться с целыми и вещественными операндами. Оператор конкатенации & может применяться при объединении двух векторов (или элемента и вектора, или двух элементов) для формирования одного длинного вектора. Например, "010" & '1' образуют "0101", а "ABC" & "DEF" – "ABCDEF".

Операторы умножения * и деления / выполняют умножение и деление целых и вещественных операндов. Операторы **rem** (остаток) и **mod** (по модулю)применяются к операндам целого типа. Оператор вида *A* **rem** *B* находит остаток от деления *A* на *B*. Результат операции *A* **mod** *B* имеет тот же знак, что и *B*, а абсолютное значение – меньше чем *B*. Например,

```
5 rem 3 = 2,  (-5) rem 3 = -2,   5 rem (-3) = 2,  (-5) rem (-3) = -2
5 mod 3=2,    (-5) mod 3=1,    5 mod (-3) = -1,   (-5) mod (-3) = -2
```

Возводить в степень (**) можно целые и вещественные операнды, степень описывается целым значением. Оператор **abs** находит абсолютное значение численного операнда.

2.10. Функции VHDL

Функция выполняет последовательный алгоритм и возвращает единичное значение в вызвавшую основную программу. Результатом работы следующей функции будет bit_vector, эквивалентный исходному, только циклически сдвинутый на одну позицию вправо:

```
function rotate_right (reg: bit_vector)
  return bit_vector is
begin
  return reg ror 1 ;
end rotate_right;
```

Вызов функции может быть использован в любом месте, где записывается выражение. Например, если A = "10010101", то

```
B <= rotate_right(A);
```

назначит переменной B значение "11001010", а *A* не изменится. Общая форма декларации функции имеет вид:

```
function function-name (formal-parameter-list)
  return return-type is
     [declarations]
begin
  sequential statements – must include return return-value;
end function-name;
```

Общее правило вызова функции:

```
function_name (actual-parameter-iist)
```

Число и тип параметров в actual-parameter-list должны соответствовать списку параметров, указанных при декларации функции. Они рассматриваются как входные значения и никогда не изменяются во время выполнения функции.

Функция, представленная на рисунке 2.29, использует оператор цикла **for**. Его общая форма имеет следующий вид:

```
[loop-label:] for loop-index in range loop
  sequential statements
end loop [loop-label];
```

Индекс цикла автоматически описывается при использовании оператора цикла и его не следует явно декларировать. Он инициализируется первым значением из указанного диапазона, после чего выполняются последовательные операторы (sequential statements). Индекс цикла может быть использован в последовательных операторах, но не может быть изменен в них. Когда конец цикла достигнут, индекс получает следующее значение из диапазона и последовательность операторов выполняется снова. Этот процесс выполняется до тех пор, пока индекс цикла не переберет все значения из диапазона, после чего выполнение оператора цикла заканчивается и индекс становится недоступным. На рисунке 2.29 индекс (i) в начале выполнения цикла будет инициализирован значением 0 и последовательность операторов в цикле будет выполнена. Вычисление операторов в цикле повторится для i = 1, i = 2 и i = 3, после чего выполнение цикла закончится.

Для окончания цикла может использоваться оператор **exit,** имеющий форму:

```
exit; или exit when condition;
```

Выполнение цикла прерывается, если будет выполнен оператор **exit** и если значение выражения condition равно TRUE (для второго случая).

Рисунок 2.29. Функция Add, выполняющая сложение двух битовых векторов

```
-- Эта функция выполняет сложения двух четырехбитных векторов и переноса
-- возвращает 5-битную сумму

function add4 (A,B: bit_vector(3 downto 0); carry: bit)
  return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable sum: bit_vector(4 downto 0):="00000";
begin
loopi: for i in 0 to 3 loop
  cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
  sum(i) := A(i) xor B(i) xor cin;
  cin := cout;
end loop loopi;
sum(4):= cout;
return sum;
end add4;
```

Если A, B и C имеют тип integer, выражение C <= A + B присвоит переменной C значение, равное сумме A и B. Тем не менее, если A, B и C имеют тип bit_vector, оператор "+" нельзя использовать, поскольку он не определен для типа bit_vector. В таком случае можно написать функцию, выполняющую сложение двух векторов. Фунция, приведенная на рисунке 2.29, выполняет сложение двух четырехбитных векторов и перенос из младшего разряда, а также возвращает сумму в форме пятибитного вектора. Функция имеет имя add4, формальные параметры A, B, carry и тип результата bit_vector. Для хранения промежуточных значений во время вычисления определены переменные cout и cin. Для хранения значения, которое будет возвращаться в программу, используется переменная sum. При вызове функции переменная cin будет инициализироваться значением параметра carry. Цикл **for** будет последовательно складывать биты векторов A и B аналогично последовательному сумматору. При первом выполнении цикла для вычисления переменных cout и sum(0) используются A(0), B(0) и cin. Затем переменная cin получает значение cout и цикл выполняется снова. При втором выполнении цикла переменные cout и sum(1) вычисляются с использованием A(1), B(1) и обновленного значения cin. После четвертого выполнения операторов в цикле все значения sum(i) будут вычислены и сумма векторов sum возвращается в программу, которая вызвала функцию. Общее время моделирования,

необходимое для выполнения функции add4, равно нулю. Не требуется даже дельта-задержка, поскольку все вычисления выполняются с использовнием переменных.

Вызов функции сложения add4 имеет вид:

```
add4( A, B, carry )
```

A, *B* и *carry* могут быть заменены любыми выражениями, результат вычисления которых будет иметь тип bit_vector размерностью 3 **downto** 0 или bit соответственно. Например, выражение

```
Z <= add4(X, not Y, '1');
```

вызывает функцию add4. Параметры A, B и carry устанавливаются равными значениям X, notY и '1' соответственно. X и Y должны быть бит-векторами размерностью 3 **downto** 0. Функция вычисляет

```
sum = A + B + carry =X+ not Y + '1'
```

и возвращает это значение. Так как sum – переменная, её вычисление не требует времени. После дельта-задержки Z устанавливается равным возвращенному значению sum. Поскольку **not** Y + '1' равно дополнительному коду к Y, вычисление эквивалентно вычитанию путем прибавления дополнительного кода. Если опустить перенос в старший разряд, сохраненный в Z(4), результат будет равен Z (3 downto 0) = X -Y.

Функции часто используются для преобразования типов. Функция vec2int(bitvec) принимает бит-вектор как входное значение и возвращает соответствующее значение типа integer. Входными параметрами функции int2vec (int, N) являются два положительных целых числа, результат – значение int, преобразованное в bit_vector длиной N. Векторы, являющиеся входными параметрами в функциях, могут иметь любую длину. Все описанные в параграфе функции принадлежат пакету BitPackage из библиотеки BITLIB.

2.11. Процедуры VHDL

Процедуры позволяют выполнить декомпозицию VHDL-кода на модули. В отличие от функций, которые возвращают только одно значение, процедуры могут возвращать любое число значений, используя выходные параметры. Синтаксическая форма описания процедуры:

```
procedure procedure_name (formal-parameter-list) is
     [declarations]
begin
     sequential statements
end procedure-name;
```

Formal-parameter-list (формальный список параметров) определяет входы и выходы процедуры и их типы. Вызов процедуры может быть последовательным или параллельным оператором вида:

```
procedure_name (actual-parameter-list);
```

Реализуем процедуру Addvec, которая складывает два N-разрядных вектора и перенос и возвращает N-разрядную сумму и перенос. Вызов данной процедуры будет иметь форму

```
Addvec ( A, B, Cin, Sum, Cout, N) ;
```

где A, B и Sum – N-разрядные векторы, Cin и Cout – биты, N – целое число.

На рисунке 2.30 представлен VHDL-код процедуры сложения, которая имеет входные параметры: Addl, Add2, Cin и выходные – Sum и Cout. Целое положительное число N определяет число битов в бит-векторах. В процедуре используется тот же алгоритм сложения, что и в функции add4. *C* – переменная, которая после каждой итерации цикла должна обновляться. Тем не менее, Sum может быть сигналом, потому что не используется в цикле. После N итераций цикла все значения сигнала Sum будут вычислены, но он получит новое значение только после дельта-задержки.

Рисунок 2.30. Процедура сложения бит-векторов

```
-- Эта процедура складывает два n-разрядных бит-вектора и перенос,
-- возвращает n-разрядную сумму и перенос. Addl и Add2 установлены
-- одинаковой длины и размерностью n-1 downto 0.
procedure Addvec
   (Addl,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit;
   n:in positive) is
   variable C: bit;
begin
   C := Cin;
   for i in 0 to n-1 loop
      Sum(i)<=Addl(i) xor Add2(i) xor C;
      C:=(Addl(i) and Add2(i)) or (Addl(i) and C) or (Add2(i) and C);
   end loop;
   Cout<=C;
end Addvec;
```

При объявлении процедуры в списке формальных параметров должны быть описаны: класс, режим и тип каждого параметра. Классом может быть **signal, variable** или **constant**. Если он опущен, по умолчанию используется **constant**. Если класс – **signal**, то фактический параметр в вызове процедуры должен быть сигналом того же самого типа. Аналогично для формального параметра класса **variable** фактический параметр должен быть переменной того же типа. Однако для формального параметра **constant** фактический параметр может быть любым выражением, значение которого имеет тот же самый тип, что и константа. Значение константы используется внутри процедуры и не может быть изменено. Таким образом, формальный параметр **constant** всегда имеет режим **in**. Сигналы и переменные могут иметь режимы **in**, **out** или **inout**. Параметры, имеющие режим **out** и **inout,** могут быть изменены в процедуре, поэтому они используются для возвращения значений в программу.

В процедуре Addvec параметры Addl, Add2 и Cin есть по умолчанию константы. Поэтому при вызове процедуры они могут быть заменены любыми выражениями, значения которых соответствуют типу и размеру этих констант.Поскольку Sum и Cout изменяются в пределах процедуры и используются для возвращения значений, они объявляются как сигналы. Таким образом, при вызове процедуры Sum и Cout могут быть заменены только сигналами соответствующего типа и размерности.

Список формальных параметров в декларации функции подобен их списку в объявлении процедуры, за исключением параметров класса **variable,** которые не разрешены для функций. Кроме того, все параметры должны иметь режим **in**, который является заданным по умолчанию. Параметры режимов **out** или **inout** не разрешены, так как функция возвращает только одиночное значение, которое не может быть возвращено через параметр. В табл. 2.2 представлены режимы и классы, которые могут использоваться для параметров процедур и функций.

Таблица 2.2. Параметры для вызовов подпрограмм

		Фактические параметры	
Режим	Класс	Вызов процедуры	Вызов функции
in[1]	constant[2]	expression	expression
	signal	signal	signal
	variable	variable	n/a
out/inout	signal	signal	n/a
	variable[3]	variable	n/a

[1] режим по умолчанию для функций

[2] по умолчанию, in режим

[3] по умолчанию, out/inout режим

2.12. Пакеты и библиотеки

Пакеты и библиотеки обеспечивают удобный способ ссылок на часто используемые функции и компоненты. Пакет состоит из декларации и необязательного тела. Декларация содержит набор определений, которые могут быть использованы несколькими модулями проекта. Например, в ней могут содержаться описания типов, сигналов, компонентов, декларации функций и процедур. Тело пакета обычно содержит тела процедур и функций. Пакет и связанные с ним, откомпилированные VHDL-модели могут быть помещены в библиотеку, следовательно, они будут доступны различным VHDL-проектам. Объявление пакета имеет форму

```
package package-name is
     package declarations
end [package][package-name] ;
```

тело пакета имеет форму

```
package body package-name is
  package body declarations
end [package body][package name]
```

В большинстве примеров данной книги используется пакет BitPackage. Он содержит обычно используемые компоненты и функции, которые применяют сигналы типа bit и bit_vector. Большинство компонентов в этом пакете имеет заданную по умолчанию задержку 10 ns, но она может быть изменена благодаря использованию константы generics. Откомпилированный пакет находится в библиотеке BITLIB.

Одним из компонентов библиотеки является логический элемент 2И-НЕ, названный Nand2 и имеющий заданную по умолчанию задержку 10 ns. Декларация пакета BitPackage включает декларацию компонента:

```
component Nand2
  generic (DELAY : time := 10 ns)
  port (A1, A2 : in bit; Z : out bit);
end component;
```

В модели NAND-вентиля используется параллельный оператор. Описание компонента имеет следующий вид:

```
-- 2-input NAND gate
entity Nand2 is
  generic (DELAY : time)
  port (A1, A2 : in bit; Z : out bit);
end Nand2;
architecture concur of Nand2 is
begin
```

```
   Z <= not(A1 and A2) after DELAY;
end;
```

Для получения доступа к компонентам и функциям в пределах пакета требуется применение операторов **library** и **use.** Оператор

```
library BITLIB;
```

позволяет проекту получить доступ к BITLIB. Оператор

```
use BITLIB.BitPackage.all;
```

позволяет проекту использовать полный пакет BitPackage. Оператор формы

```
use BITLIB.BitPackage.Nand2;
```

может использоваться, если необходимо применять отдельный компонент или функцию из пакета.

2.13. VHDL-модель счетчика 74163

Устройство 74163 (рисунок 2.31) – 4-разрядный полный синхронный двоичный счетчик, который существует и в ТТЛ-, и в CMOS-семействах элементов. Кроме счета он может выполнять операции сброса и параллельной загрузки. Таблица 2.3 описывает работу счетчика.

Таблица 2.3. Описание поведения счетчика

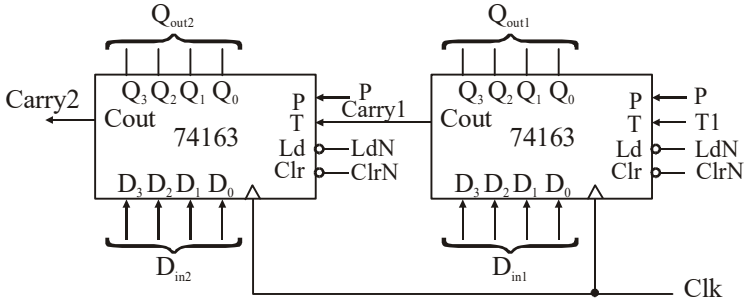
Управляющие сигналы			Следующее состояние				
ClrN	LdN	P·T	$Q3^+$	$Q2^+$	$Q1^+$	$Q0^+$	
0	X	X	0	0	0	0	(сброс)
1	0	X	D3	D2	D1	D0	(параллельная загрузка)
1	1	0	Q3	Q2	Q1	Q0	(сохранение состояния)
1	1	1	текущее состояние +1				(счет на увеличение)

Все изменения состояния счетчика происходят по переднему фронту синхроимпульса. Счетчик генерирует перенос (Cout) в состоянии 15, если T = 1, поэтому

$$Cout=Q_3Q_2Q_1Q_0T$$

VHDL-описание счетчика показано на рисунке 2.32. Перенос вычисляется всякий раз, когда изменяются Q или T. Оператор **wait** в процессе ждет изменения CK, затем продолжает выполнение, если CK изменился на '1'. Учитывая, что операция сброса имеет больший приоритет, чем параллельная загрузка и счет, сигнал ClrN проверяется первым в процессе. Поскольку приоритет операции загрузки выше, чем приоритет операции счета, следующим проверяется LdN. Наконец, значение в счетчике увеличивается на 1, если P и T = 1. Так как Q – бит-вектор, он преобразуется в целое число, увеличивается на 1 и затем переводится обратно в 4-разрядный вектор. Модель с именем c74163 размещена в пакете BITLIB.BitPackage.

Рисунок 2.31. Два каскадных счетчика 74163, формирующие 8-разрядный счетчик



Для тестирования два счетчика 74163 объединяются каскадом в целях формирования 8-разрядного счетчика (см. рисунок 2.31). Когда правый счетчик находится в состоянии 1111, то Carryl = l. Если PT=l, то следующий активный сигнал синхронизации переведет правый счетчик в состояние 0000, в то же время значение в левом увеличится на единицу. Рисунок 2.32 представляет VHDL-код для 8-разрядного счетчика. В нем используется модель *c74163*. Для удобства чтения определен сигнал Count, который является целочисленным эквивалентом значения 8-разрядного счетчика. После тестирования счетчик помещается в BITLIB.BitPackage для дальнейшего использования. После этого объявление компонента может не применяться в коде, а быть записано в форме, как показано на рисунке 2.33.

Рисунок 2.32. Модель счетчика 74163

```
-- 74163 синхронный счетчик
library BITLIB;              -- содержит функции int2vec и vec2int
use BITLIB.BitPackage.all;

entity c74163 is
  port(LdN, CIrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
       Cout: out bit; Q: inout bit_vector(3 downto 0) );
end c74163;

architecture b74163 of c74163 is
begin
  Cout <= Q(3) and Q(2) and Q(1) and Q(0) and T;
  process
  begin
    wait until CK = '1'; -- состояние меняется по переднему фронту
    if CIrN = '0' then Q <= "0000";
      elsif LdN = '0' then Q <= D;
      elsif (P and T) = '1' then
        Q <= int2vec(vec2int(Q)+1,4);
    end if;
  end process;
end b74163;
```

Рисунок 2.33. VHDL - код 8-разрядного счетчика

```
library BITLIB;
use BITLIB.BitPackage.all;

entity c74163test is
  port(CIrN,LdN,P,Tl,Clk: in bit;
       Dini, Din2: in bit_vector(3 downto 0);
       Qouti, Qout2: inout bit_vector(3 downto 0);
       Carry2: out bit);
end c74163test;

architecture tester of c74163test is
  component c74163
    port(LdN, CIrN, P, T, CK: in bit; D: in bit_vector(3 downto 0);
         Cout: out bit; Q: inout bit_vector(3 downto 0) );
  end component;
  signal Carryl: bit;
  signal Count: integer;
  signal temp: bit_vector(7 downto 0);
begin
  ctl: c74163 port map (LdN,CirN,P,Tl,Clk,Dini,Carryl,Qouti);
  ct2: c74163 port map (LdN,CirN,P,Carryl,Clk,Din2,Carry2,Qout2);
  temp <= Qout2 & Qouti;
  Count <= vec2int(temp);
end tester;
```

2.14. Транспортная и инерционная задержки

VHDL предлагает два типа задержек – транспортную и инерционную. Транспортная соответствует модели задержек на проводниках схемы. Выходной сигнал повторяет в этом случае форму входного сигнала с задержкой на указанное время. Инерционная моделирует задержку на вентилях или других устройствах, которые не пропускают короткие импульсы входного сигнала на выход. Эта задержка используется по умолчанию.

Пусть логический элемент имеет инерционную задержку T, тогда любой импульс продолжительностью, меньшей чем T, не достигнет выхода. Например, вентиль имеет инерционную задержку 10 ns, тогда сигнал продолжительностью 10 ns пройдет через вентиль, а сигнал продолжительностью 9.999 ns будет игнорироваться. Реальное устройство не ведет себя таким образом. Обычно максимальная продолжительность импульса, на который реагирует устройство, меньше его задержки. VHDL может моделировать такую ситуацию с помощью выражения **reject** в операторе назначения сигнала. Следующее выражение

```
signal_name <= reject pulse-width expression after delay-time
```

запрещает реакцию схемы на любой импульс продолжительностью меньше, чем pulse-width, и устанавливает новое значение сигнала после задержки, равной delay time.

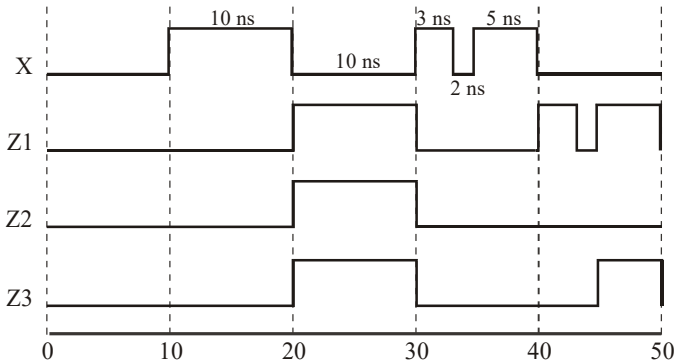
Рисунок 2.34 иллюстрирует различие между транспорными и инерционными задержками. Рассмотрим следующие VHDL-операторы:

```
Z1 <= transport X after 10 ns;  -- транспортная задержка
Z2 <= X after 10 ns;            -- инерционная задержка
Z3 <= reject 4 ns inertial X after 10 ns; -- задержка с заданной
                                    -- длиной отраженных импульсов
```

Z1 имеет такую же форму сигнала, как и X, только смещенную во времени на 10 ns. Z2 подобна Z1, за исключением того, что импульсы, короче 10 ns, отфильтровываются и не появляются в Z2. Z3 подобна Z2, за исключением того, что импульсы, короче 4 ns, будут проигнорированы. В общем случае, использование **reject** представляет собой совмещение инерционной и транспортной задержек. Выражение для Z3 может быть заменено следующими параллельными операторами:

```
Zm <= X after 4 ns;     -- инерционная задержка, отражающая
                        -- короткие импульсы
Z3 <= transport Zm after 6 ns;  -- общая задержка 10 ns
```

Рисунок 2.34. Транспортная и инерционная задержки



Допустим, что следующие параллельные операторы выполняются в момент *T:*

```
A <= transport B after 1 ns;
A <= transport C after 2 ns;
```

Будущее значение A заносится в его драйвер. Сигнал A получает новое значение B в момент времени T + 1 ns. Значение C также заносится в драйвер сигнала A, которое сигнал A получит в момент времени T +2 ns.

Рассморим последовательные операторы, выполняемые в момент времени *T:*

```
A <= B after 1 ns; -- используется инерционная задержка
A <= C after 2 ns; -- используется инерционная задержка
```

В очередь будущих значений драйвера сигнала A заносится B со временем T+1ns. Затем записывается значение C со временем T + 2 ns. Поскольку 2 ns – инерционная задержка, то значение B @ T+1 ns удаляется из очереди, которая содержит только C@T+2 ns.

Если в очередь драйвера заносится изменение сигнала, а затем – следующее значение, которое должно присвоиться ранее или в то же время, что и первое, то первое удаляется из очереди. Допустим, что следующие последовательные операторы выполняются в момент T:

```
A <= transport B after 2 ns;
A <= transport C after 1 ns;
```

В очередь драйвера сигнала *A* будет записано *B@T* + 2 ns. Затем заносится *C @T* + 1 ns, а предыдущее значение будет удалено из очереди.

Общее правило занесения будущих значений сигналов в очередь драйвера сигнала.

Для транспортной задержки. Пусть в момент времени *T* выполняется последовательный оператор назначения сигнала

```
A <= transport B after Tnew ns;
```

В очередь драйвера заносится B со временем T+Tnew (B@T+Tnew). При этом из очереди будут удалены все значения, которые сигнал должен был получить в момент T+Tnew и позже.

Для инерционной задержки. Пусть в момент времени *T* выполняется последовательный оператор назначения сигнала

```
A <= reject Tr inertial B after Tnew ns;
```

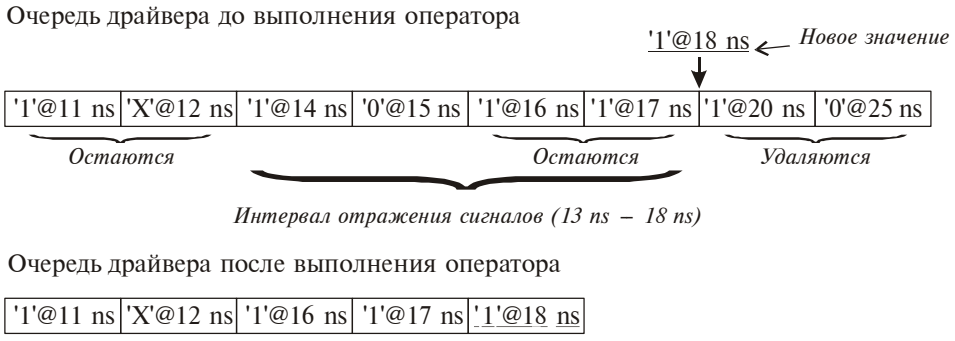
где Tr – полоса пропускания сигнала (импульсы, короче Tr, отбрасываются). Тогда из очереди удаляются все значения, которые сигнал должен был получить в момент времени T+Tnew и позже. Затем анализируются будущие в диапазоне времени T+(Tnew-Tr) и T+Tnew. Из них в очереди остаются только будущие значения сигнала, непосредственно предшествующие и равные заносимому, остальные удаляются.

Пример формирования очереди будущих значений сигнала. Пусть очередь драйвера сигнала A содержит несколько будущих значений. В момент времени 10 ns выполняется последовательный оператор назначения:

```
A <= reject 5 ns inertial '1' after 8 ns;
```

На рисунке 2.35 представлена очередь будущих значений до и после выполнения оператора. После вычисления оператора назначения в драйвер заносится '1' и время 18 ns, где 10ns – время выполнения оператора +8 ns – задержка. Будущие значения со временем назначения позже 18 ns удаляются из очереди. Это '1'@20 ns и '0'@25 ns. Анализируются значения со временем от 13 ns, где 10 ns+ 8 ns-5 ns, до 18 ns, где 10 ns+8ns. Из этих значений остаются '1'@16 ns и '1'@17 ns, непосредственно предшествующие заносимому значению '1'@18 ns и равные ему. Остальные будущие значения из этого диапазона удаляются.

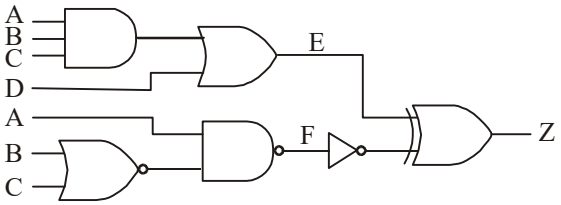
Рисунок 2.35. Очередь драйвера синала A



Выводы. Рассмотрены основные свойства VHDL, реализуемые на языке модели комбинационных и последовательностных логических схем. Поскольку VHDL – язык описания аппаратуры, он имеет ряд отличий от языков программирования. В VHDL операторы выполняются параллельно, потому что они должны моделировать реальную аппаратуру, составные части которой работают одновременно. Операторы в процессах выполняются последовательно, но сами процессы являются параллельными операторами. VHDL-сигналы моделируют реальные линии в схемах, но для внутренних вычислений могут использоваться переменные, которые являются локальными для процессов, процедур и функций. В VHDL существует два вида задержек: транспортная и инерционная, соответствующие линиям и элементам.

2.15. Задачи

2.15.1. Записать VHDL-описание следующей комбинационной схемы, используя параллельные операторы. Каждый элемент имеет задержку 5 ns, за исключением инвертора, имеющего задержку 2 ns.



2.15.2. Записать VHDL-код для устройства вычитания, используя логические уравнения.

2.15.3. Записать VHDL-код для 4-разрядного устройства вычитания, используя модуль, созданный в 2.2, как компонент.

2.15.4. В следующем VHDL-процессе A, B, C и D – целые числа, которые имеют значение 0 в момент времени 10 ns. Сигнал E переключается с '0' в '1' в момент 20 ns. Определить время изменения каждого сигнала и получаемые им значения. Перечислить эти изменения в хронологическом порядке (20, 20 + Δ , 20 + 2 Δ , ...):

```
a)p1: process
begin
   wait on E;
   A <= 1 after 5 ns ;
   B <= A + 1;
   C <= B after 10 ns;
   wait for 0 ns ;
   D <= B after 3 ns;
   A <= A + 5 after 15 ns;
   B <= B + 7;
end process pi;
b)p1: process
begin
```

```
   wait on E;
   A <= 1 after 5 ns ;
   B <= A + 1;
   C <= B after 10 ns;
   wait for 10 ns ;
   D <= B after 3 ns;
   A <= A + 5 after 15 ns;
   B <= B + 7;
end process pi;
c)p1: process
begin
   wait on E;
   A <= 1 after 5 ns ;
   B <= A + 1;
   C <= B after 10 ns;
   wait for 0 ns ;
   D <= B after 3 ns;
   transport A <= A + 5 after 15 ns;
   B <= B + 7;
end process pi;
```

2.15.5. Для следующего VHDL-кода пусть D изменяется на '1' в момент времени 5 ns. Описать значения A, B, C, D, E и F и время их формирования. Другими словами, указать значения сигналов во время 5 ns, 5 ns + Δ, 5 ns + 2Δ, ... Описать первые 20 шагов моделирования или до исчезновения изменений сигналов, или до появления периодически повторяющейся последовательности значений сигналов:

```
entity prob4 is
     port (D: in bit);
end prob4 ;
architecture q1 of prob4 is
   signal A, B, C, E, F: bit;
begin
   C <= A;
   A <= B or D;
   P1: process (A) begin
      B <= A;
   end process P1;
   P2: process
   begin
      wait until A <= '1' ;
      wait for 0 ns;
      E <=  B;
      F <= E;
   end process P2 ;
end architecture q1;
```

2.15.6. Пусть сигнал s в текущий момент имеет значение '0'. Чему будут равны булевы переменные v1 и v2 после выполнения следующих операторов в процессе?

```
S<='1';
v1 :=s= '1';
wait on s;
v2:=s='1';
```

2.15.7. Указать транзакции, занесенные в драйвер, при выполнении следующих операторов и значения, получаемые z во время моделирования:

```
z <= transport '1' after 6 ns;
wait for 3 ns;
z <= transport '0' after 4 ns;
wait for 5 ns;
z <= transport '1' after 6 ns;
```

```
wait for 1 ns;
z <= transport '0' after 4 ns;
```

2.15.8. Создать процесс, эквивалентный условному оператору назначения сигнала:

```
mux_logic:
   z <= a and not b after 5 ns when enable='1' and sel='0' else
        x or y after 6 ns when enable = '1' and sel := '1' else
        '0' after 4 ns;
```

2.15.9. Создать процесс, эквивалентный параллельному оператору выбора назначения сигнала:

```
with bit_vector'(s, r) select
   q <= q when "00",
        '0' when "01",
        '1' when "10",
        'Z' when "11";
```

2.15.10. Записать VHDL-код для SR-триггера с синхронизацией по уровню, используя:

а) условный оператор назначения сигналов;

б) логические уравнения;

в) два логических элемента.

2.15.11. D-триггер с синхронизацией по уровню сохраняет свое значение, если G = 0, и Q=D, если G = 1. Используя оператор процесса, записать VHDL-код D-триггера.

2.15.12. DD-триггер подобен D-триггеру с синхронизацией по фронту, за исключением того, что он может изменять состояние ($Q^+ = D$) при появлении переднего и заднего фронтов синхроимпульса. Триггер имеет вход сброса R. При R = 0 триггер устанавливается в 0 независимо от синхронизации. Записать VHDL-код для DD-триггера.

2.15.13. Триггер-переключатель имеет входы I0, I1, T и Reset, а также выходы Q и QN. Сброс триггера происходит при Reset =1 независимо от значений сигналов на других входах. Триггер работает следующим образом. Если I0=1, он меняет состояние по переднему фронту сигнала T; а если I1 = 1 – по заднему. Если I0=I1= 0, то триггер сохраняет состояние. Пусть задержка передачи сигнала от T до выхода – 8 ns, и от сброса до выхода – 5 ns. Создать VHDL-модель триггера. Записать последовательность команд программы моделирования для его тестирования. Подать на входы I1 = 1, I0 = 0. После двухкратного переключения триггера подать набор I1 = 0, I0 = 1. Завершить моделирование после двух изменений на выходах.

2.15.14. Разработать поведенческую модель мультиплексора 4x1. Использовать тип bit для входных портов. Задержка распространения сигнала от входа до выхода равна 4,5 ns. Следует описать задержку как константу и использовать ее в архитектуре.

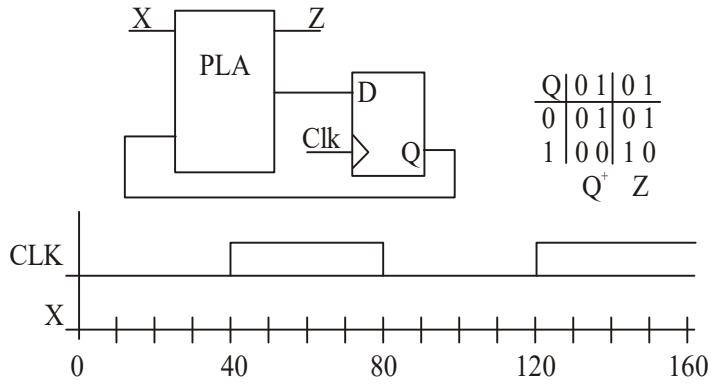
2.15.15. Разработать поведенческую VHDL-модель конечного автомата, имеющего вход X , выходы D и B. На вход X последовательно подается двоичное 4-битовое число N, начиная с младшего бита. D представляет 4-битовое число, равное N–2. Значение поступает на выход, начиная с младшего разряда. Во время приема четвертого бита выход B=1, если N–2 отрицательное число, B=0 – в противном случае. Состояние автомата сбрасывается после поступления четвертого бита на вход X.

Пусть изменения состояния происходят по переднему фронту синхроимпульса. Оператор **case** вместе с операторами **if-then-else** использовать для представления таблицы переходов и выходов. Откомпилировать код. Промоделировать, используя следующую тестовую последовательность: X = 1011 0111 1000. Сигнал X изменяется в момент 1/4 периода после поступления заднего фронта синхронизации. Записать VHDL-код автомата, используя уравнения переходов и выходов. Указать на выходе моменты времени Z для считывания данных. Записать на языке VHDL структурную

модель конечного автомата , состоящую из вентилей и JK-триггеров. Можно использовать для этой части BITLIB библиотеку.

2.15.16. Разработать поведенческую VHDL- модель конечного автомата, представленную на рисунке 2.36.

Рисунок 2.36. Конечный автомат



Задержка автомата находится в пределах $5 \le t_c \le 20$ ns. Задержка между поступлением переднего фронта синхронизации и изменением выходного сигнала находится в пределах $5 \le t_c \le 10$ ns. Необходимое время установки и хранения данных для триггера t_{su} =10 ns и t_n=5 ns. На диаграмме отмечены значения, в которых разрешены изменения сиганала.

Пусть изменения состояний происходят по заднему фронту тактового импульса. Вместо использования операторов **if-then-else** таблицу переходов и выходов представить в виде двух массивов. Промоделировать код, используя следующую тестовую последовательность: X = 1101 1110 1111. Сигнал X изменяется в момент 1/4 периода после поступления заднего фронта синхронизации. Разработать VHDL-модель, используя логические уравнения состояний и выходов. Указать на выходе моменты времени Z для считывания данных. Записать структурную VHDL-модель конечного автомата, состоящую из вентилей и JK-триггеров. Можно использовать для этой части BITLIB библиотеку.

2.15.17. Последовательный автомат Мура с двумя входами (XI и X2) и одним выходом (Z) имеет следующую таблицу переходов и выходов:

	00	01	10	11	Z
1	1	2	2	1	0
2	2	1	2	1	1

Записать VHDL-код, который описывает автомат на поведенческом уровне. Пусть изменения состояния происходят через 10 ns после поступления заднего фронта синхроимпульса, а изменения выхода – через 10 ns после изменения состояния.

2.15.18. Записать функцию VHDL, которая будет вычислять дополнительный код w-разрядного бит-вектора. Вызов функции имеет вид comp2(bit_vec, N), где N – длина вектора.

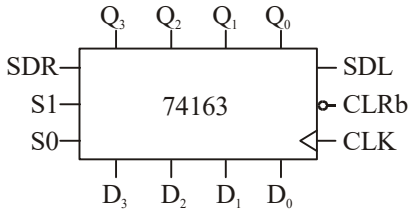
2.15.19. X и Y – бит-векторы длиной N, которые представляют собой знаковые двоичные числа. Отрицательными числа представляются в виде дополнительного кода. Разработать процедуру VHDL, которая вычисляет D = X-Y. Эта процедура должна также возвратить заем от последней позиции двоичного разряда (B) и флаг переполнения (V). Не должно быть никаких других функций или процедур, вызываемых из создаваемой процедуры. Вызов процедуры должен иметь форму

```
SUBVEC (X, Y, D, B, V, N);
```

2.15.20. Записать функцию VHDL для преобразования бит-вектора в целое число. Значение двоичного числа $a_4a_3a_2a_1a_0$ вычисляется по формуле: ((((0 + a_4)*2 + a_3)*2 + +a_2)*2 + a_1)*2 + a_0. Сколько времени займет моделирование функции?

2.15.21. Записать VHDL-модуль, который описывает 16-разрядный сдвиговый регистр с последовательными входом и выходом. Вход SI – последовательный (serial input), EN – вход разрешения синхронизации(Enable) и CK – синхронизация. Сдвиг выполняется по переднему фронту. Последовательный выход S0.

2.15.22. Двунапрвленный сдвиговый 4-разрядный регистр 74194 имеет вход CLRb, асинхронный и активный по низкому уровню, отменяющий все другие сигналы управляющих входов. Все изменения состояния происходят по переднему фронту синхронизации. Если управляющие входы S1 = S0 = 1, то выполняется параллельная загрузка данных в регистр. Если S1 = 1 и S0 = 0 – выполняется сдвиг вправо и данные с SDR записываются в Q3. Если S1 = 0 и S0 = 1, то выполняется сдвиг влево и данные с SDL записываются в Q0. Если S1 = S0 = 0, сохранение состояния.



а) Записать поведенческую VHDL-модель для 74194.

б) Нарисовать блок-схему и записать VHDL-код для 8-разрядного двунаправленного сдвигового регистра, который использует два регистра 74194 в качестве компонент. Параллельные входы и выходы 8-разрядного регистра должны быть – X(7 downto 0) и Y(7 downto 0). Последовательные входы должны быть RSD и LSD.

2.16.23. Синхронный (4-разрядный) реверсивный десятичный счетчик с выходом Q работает следующим образом: все изменения состояния происходят по переднему фронту синхросигнала CLK, кроме асинхронного сброса CLR. Когда CLR = 0, счетчик сбрасывается в 0, независимо от значений на других входах.

Если *LOAD* = 0, то в счетчик загружаются данные со входа D.

Если *LOAD* = *ENT* = *ENP* = *UP* = 1, то значение в счетчике увеличивается на 1.

Если *LOAD* = *ENT* = *ENP* = 1 и *UP* = 0, то значение в счетчике уменьшается на 1.

Если ENT = UP = 1, выход переноса C0 = 1, когда счетчик находится в состоянии 9.

Если ENT = 1 и UP = 0, выход переноса C0 = 1, когда счетчик – в состоянии 0.

Разработать VHDL-описание счетчика.

Нарисовать блок-схему и записать VHDL-код для десятичного счетчика, который использует два счетчика для формирования 2-разрядного десятичного реверсивного счетчика, считающего от 00 до 99 и от 99 до 00.

2.15.24. Записать VHDL-модель для 74HC192 4-разрядного синхронного реверсивного счетчика. Не учитывать никакие временные характеристики. Код должен содержать оператор процесса: process(DOWN, UP, CLR, LOADB).

2.15.25. Разработать VHDL-модель, которая описывает сдвиговый регистр 74198. Использовать следующую систему обозначений: Q – 8-разрядный выход, D – 8-разрядный вход, S0, S1 – управляющие входы, LSI – левый последовательный вход, RSI – правый последовательный вход.

2.15.26. Два 74198 сдвиговых регистра объединены для формирования 16-разрядного циклического сдвигового регистра, управляемого сигналами L и R. Если L=1 и R=0, то данные в 16-разрядном регистре циклически сдвигаются влево; если L=0 и R=1 – сдвигаются вправо. Если L=R=1, то выполняется параллельная загрузка 16-разрядного регистра со входов *D* [15 ... 0]. Записать VHDL-код системы, используя модуль из задачи 2.15.25.

ГЛАВА 3
ИНСТРУМЕНТАЛЬНАЯ СРЕДА ACTIVE-HDL

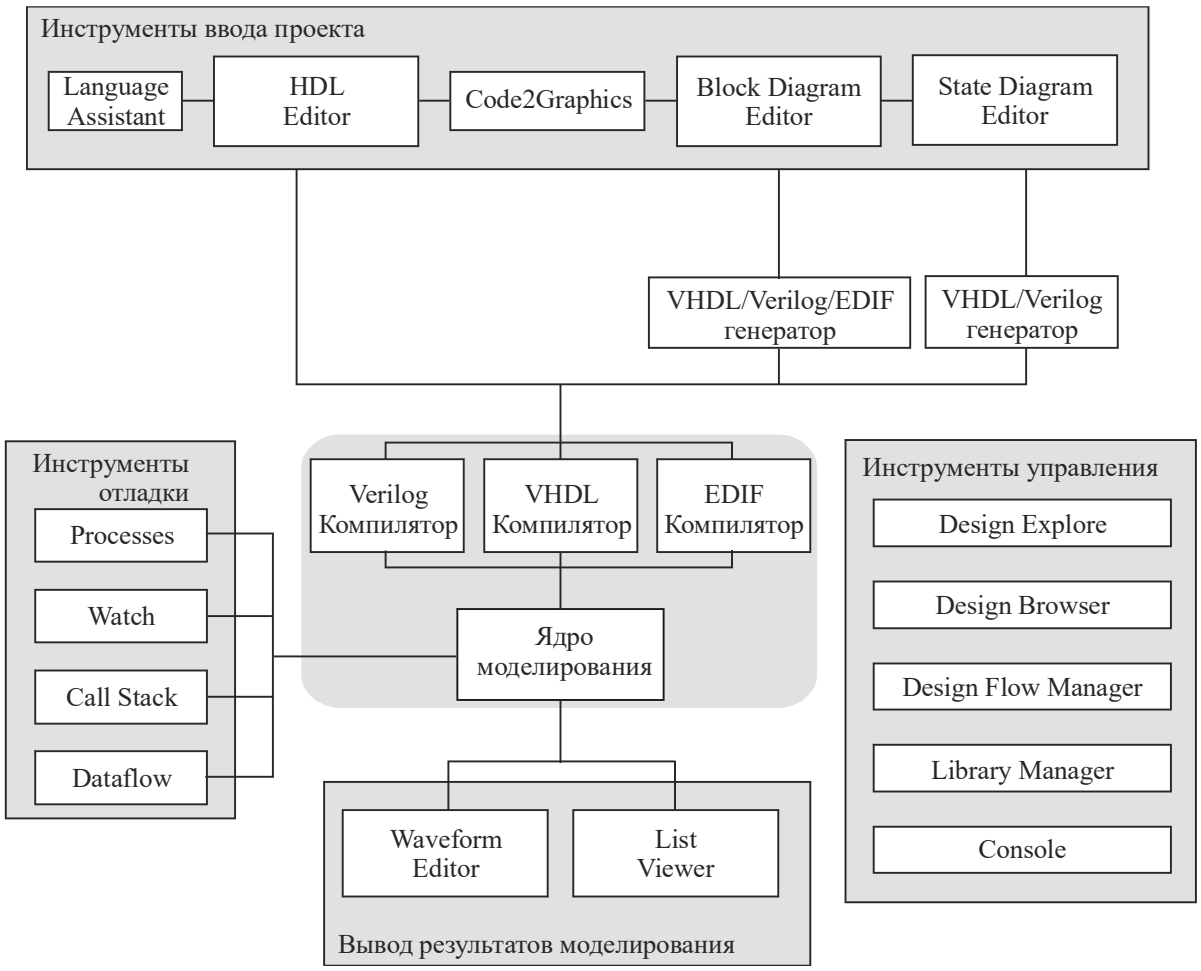
Введение в инструментальную среду проектирования Active-HDL. Представлены ее основные функции. Рассмотрена возможность управления инструментами среды Active-HDL с помощью макрокоманд.

3.1. Введение в Active-HDL

Active-HDL – это интегрированная среда проектирования. Она включает три различные программы для ввода проектов, VHDL'93 и Verilog компиляторы, единое ядро моделирования, программные модули отладки, графический и текстовый вывод результатов моделирования, вспомогательные инструменты для удобного управления файлами ресурсов, проектами и библиотеками (рисунок 3.1).

В Active-HDL включены также несколько мощных мастеров для облегчения создания новых исходных файлов, проектов и TestBench. Большинство операций, которые вызываются через графический пользовательский интерфейс, можно выполнять с помощью команд макроязыка Active-HDL. Командные файлы позволяют автоматизировать процесс проектирования.

Рисунок 3.1. Структурная схема элементов Active-HDL



Компоненты Active-HDL

Все компоненты Active-HDL являются встроенными в интерактивную графическую среду. Каждый инструмент Active-HDL, за исключением ядра моделирования и компилятора, реализован в отдельном окне.

Console (Alt-4) – интерактивное окно для ввода макрокоманд и определенных пользователем скриптов, а также для вывода сообщений, генерируемых инструментами Active-HDL.

Design Explorer – облегчение управления проектами в Active-HDL.

Design Browser – отображение текущего содержания проекта: файлы ресурсов, присоединенных к проекту; состав файлов рабочих библиотек; структура выбранного для моделирования модуля проекта; объявленные в выбранной области текущего проекта VHDL, Verilog или EDIF объекты.

Design Flow Manager – автоматизация процесса проектирования. В графической форме представлен типичный процесс проектирования, а встроенные кнопки позволяют вызывать соответствующие данному этапу приложения.

HDL Editor – текстовый редактор, разработанный для создания исходных файлов VHDL. Он интегрирован с моделирующим устройством, что облегчает отладку исходного текста.

Language Assistant – вспомогательный инструмент, предоставляющий ряд типичных для VHDL или Verilog шаблонов, модели логических примитивов и функциональных блоков. Интегрирован с HDL Editor, что позволяет автоматически размещать выбранный шаблон в редактируемый файл. Допускает определение собственных шаблонов.

State Diagram Editor (редактор автоматов) – графический инструмент, разработанный для редактирования графов конечных автоматов.Осуществляет автоматический перевод графически задаваемых примитивов в коды языка описания аппаратуры VHDL.

Waveform Editor– отображает результаты моделирования в графической форме. Позволяет редактировать форму сигнала для создания требуемых тестовых векторов.

Block Diagram Editor – графический инструмент проектирования, позволяющий создавать структурные модели цифровых устройств. Выполняет автоматическое преобразование графического представления проекта в код VHDL или Verilog языков описания аппаратуры.

List – представление результатов моделирования в виде текстовой таблицы с точностью до дельта-цикла.

Watch (Alt-5) – вывод во время моделирования текущих значений выбранных сигналов и переменных.

Processes (Alt-6) – отображение во время моделирования текущего состояния параллельных процессов проекта.

Call Stack (Alt-7) – вывод списка подпрограмм (процедур и функций), вызванных в текущем процессе.

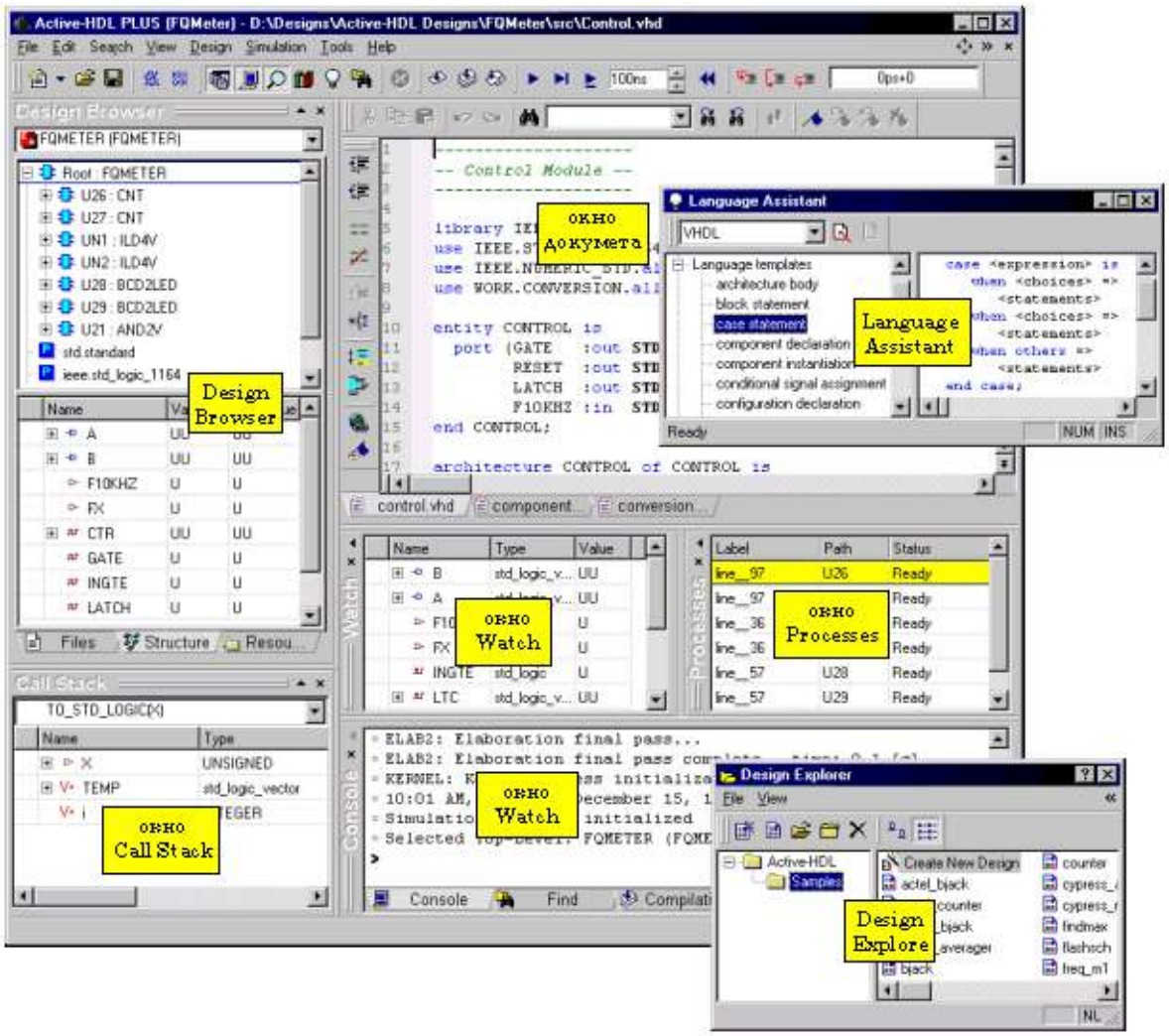
Dataflow (Alt-9) – графическое представление сигналов, входящих и выходящих из процессов во время моделирования.

Library Manager – управление VHDL-библиотеками и их содержанием.

Code2Graphics™ Converter предназначен для автоматического преобразования VHDL, Verilog или EDIF исходных файлов в Active-HDL структурную схему (block diagram) или граф (state diagram).

На рисунке 3.2 представлено задаваемое по умолчанию расположение на рабочем поле Active-HDL компонентов.

Рисунок 3.2. Рабочее окно программы Active-HDL



3.2. Design Browser

Design Browser (браузер проекта) предназначен для управления ресурсами текущего проекта. Состоит из трех подокон (вкладок): Files, Structure и Resources. Позволяет добавлять, удалять, просматривать, модифицировать и выполнять другие операции над файлами проекта. Выводит содержимое рабочей, post-synthesis и timing библиотек текущего проекта.

Подокно Files

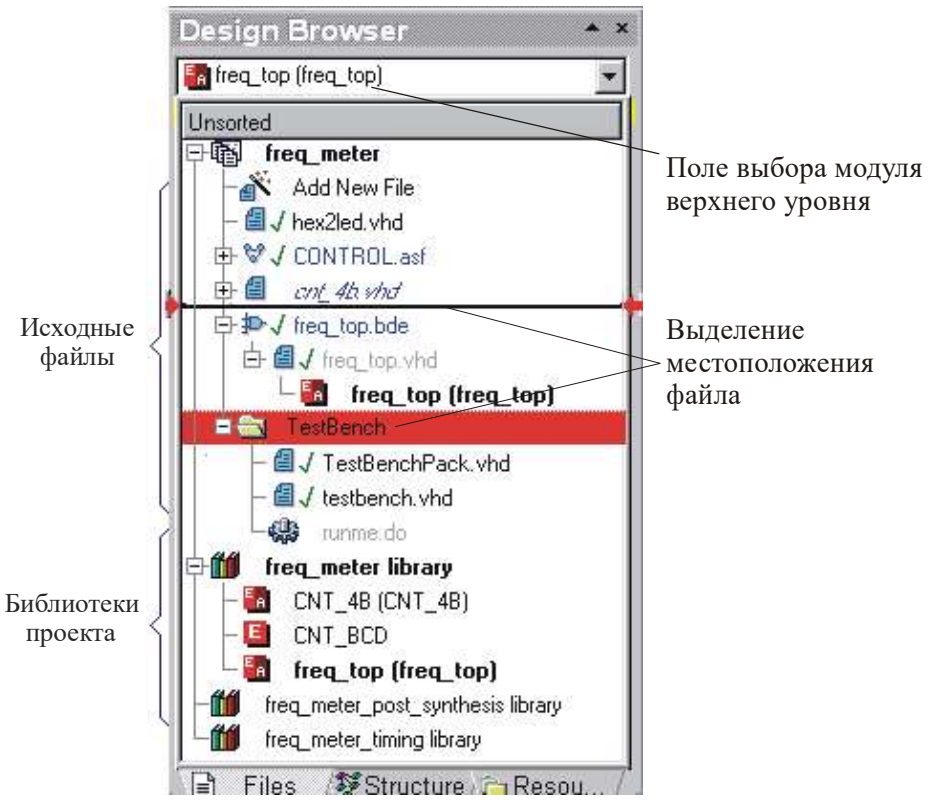
Отображает файлы ресурсов проекта и содержание рабочей, post-synthesis и timing библиотек.

Каждый проект обычно (но не обязательно) сохраняется в папке с тем же именем, что и имя проекта. В среде Active-HDL для обозначения папки текущего проекта может быть использована переменная (an environment variable) $DSN. Разрешается сохранение исходных файлов проекта в любом месте. Тем не менее выделенной для них является папка $DSN\Src. Ее содержание выводится в подокне Files. При необходимости файлы можно размещать в папках нижнего уровня. Если файл расположен вне $DSN\Src, в окне Files его иконка имеет маркер . Например, .

На рисунке 3.3 содержимое проекта представлено в виде структурного дерева, верхняя часть которого содержит исходные файлы, а нижняя – библиотеки проекта. Для каждого файла имеется своя иконка, зависящая от его типа. Ветви дерева,

соответствующие исходным откомпилированным файлам, могут быть раскрыты для просмотра входящих в них модулей (за исключением пакетов). Аналогичным образом можно развернуть содержимое библиотек.

Рисунок 3.3. Подокно Files окна Design Browser



Исходные файлы могут быть отсортированы по имени или типу в восходящем или нисходящем порядке. Для этого используется кнопка Unsorted под полем модуля верхнего уровня.

Значок, следующий за иконкой исходного файла, описывает его состояние после компиляции:

- Успешная компиляция.
- Во время последней компиляции обнаружена ошибка.
- Во время последней компиляции выведено предупреждение.
- Файл не откомпилирован или был модифицирован после компиляции.

Если исходный файл откомпилирован и помещен в библиотеку, отличную от рабочей, ее идентификатор выводится в скобках после имени файла. Можно запретить компиляцию выбранного файла, который обозначается италикой, например, *cnt_4b.vhd* с рисунка 3.3.

Нижняя часть окна Files представляет содержание библиотек проекта, обозначаемых иконкой . Каждый проект имеет рабочую библиотеку. К нему также могут быть добавлены одна или несколько дополнительных библиотек – post-synthesis и/или timing. В подокне Files выводятся только описания, которые могут быть назначены модулями верхнего уровня для моделирования. Такими являются:

- Интерфейс проекта. (Формально в VHDL он не является модулем проекта, а существует только в паре с архитектурой).
- Интерфейс без архитектуры.(Не может быть промоделирован).
- Архитектура.
- Декларация конфигурации.

Verilog модуль.

Модуль, полученный после компиляции EDIF файла.

Типы исходных файлов

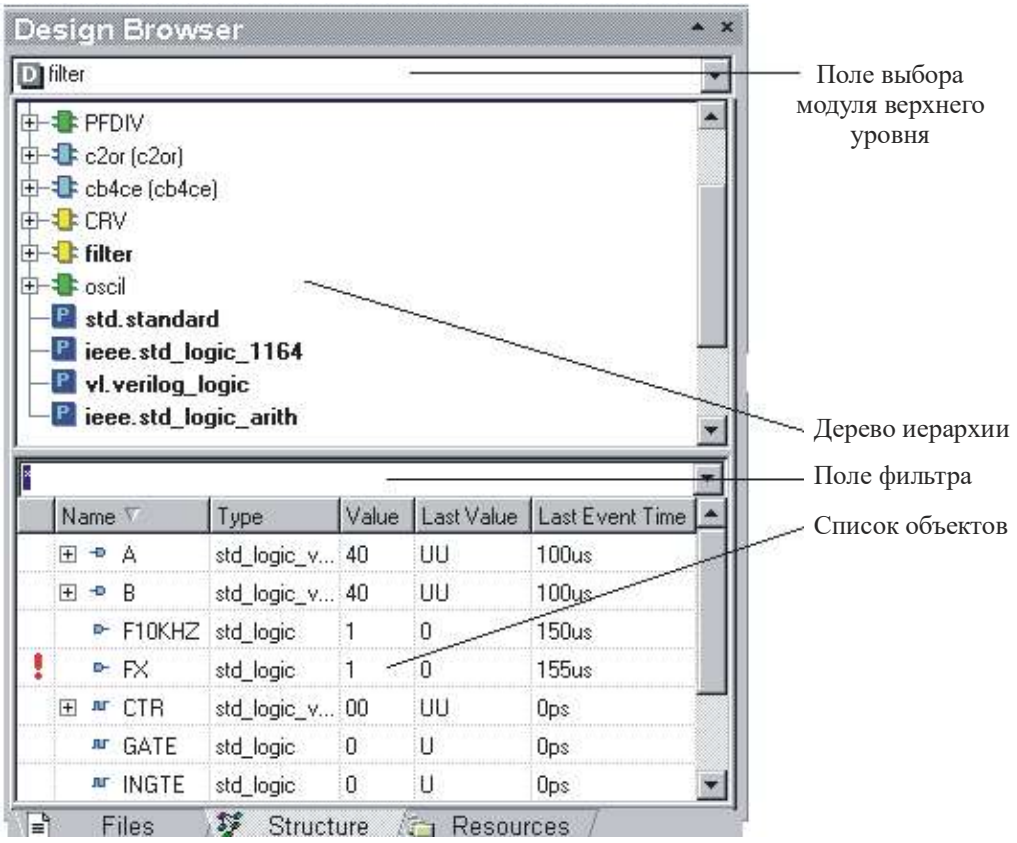
Список, представленный ниже, содержит поддерживаемые типы файлов, их расширения и иконки.

VHDL- код (VHD, VHDL, VHQ, TVHD, VHO, VHM, VHI)	
VHDL TestBench (VHD, VHDL, VHQ, VHO)	
Файл конфигурации	(VHD)
Файл редактора Block Diagram Editor	(BDE)
Файл редактора State Diagram Editor	(ASF)
Файл проекта Active-HDL	(ADF)
List файл табличного представления результатов моделирования	(LST)
Файл временных диаграмм waveform	(AWF)
Проект Active-CAD	(PDF)
Тестовые векторы Active-CAD	(ASC)
Basic скрипт	(BAS)
C++ файл	(CPP, C, H)
Рисунок	(AFC)
EDIF файл	(EDF, EDN, EDO)
Схема EDIF	(EDI, EDS)
Папка	–
Внешний файл	–
HTML document	(HTM, HTML)
Командный файл	(DO)
Perl скрипт	(PL, PM)
SDF файл	(SDF, SDO)
Поле символа	(BDS)
Tcl скрипт	(TCL, TK)
Text file	(TXT)
Verilog модель	(V, VEI, VEO, VCD, VO, VM;VMD;VLB;VLG)
Verilog TestBench	(V, VEI, VEO, VCD, VO)
Схема редактора Viewlogic	(1)
Файл XNF	(XNF)

Подокно Structure

На рисунке 3.4 представлено подокно Structure, разделенное на две части. Верхняя содержит иерархичную структуру проекта. Для сортировки информации следует щелкнуть правой кнопкой мыши и задать необходимые опции из выпадающего меню Sort. Нижняя часть содержит объекты, определенные в модуле, который выделен в верхней части окна Structure. Выводится: имя объекта, тип, текущее и предыдущее значения, время последнего события. Класс объекта (сигнал, переменная, константа, порт, generic-константа, файл) обозначается иконкой рядом с его именем. Поле фильтра позволяет фильтровать выводимые объекты, используя регулярные выражения.

Рисунок 3.4. Подокно Struture окна Design Browser



Иерархическая структура проекта состоит из блоков и процессов, которые могут быть изображены следующими иконками:

- блок,
- параллельный процесс,
- пакет, используемый проектом.

Метка рядом с иконкой соответствует ее имени в исходном коде. Если параллельный оператор не имеет меток, Active-HDL автоматически генерирует их.

Следующие иконки используются для VHDL-объектов, выводимых в нижней части окна:

- порт в режиме **in,**
- порт в режиме **out,**
- порт в режиме **inout,**
- сигнал,
- переменная,
- константа,
- generic-константа,
- файл.

Список объектов содержит колонки:

1. **Event (**событие). Колонка не имеет названия и расположена скраю слева. Если событие имеет место в текущем цикле моделирования, сигнал отмечается красным восклицательным знаком.
2. **Name (**имя). Идентификатор объекта.
3. **Value** (значение). Текущее значение объекта. Для файлов в этой колонке выводится режим, в котором он открыт: **read_mode, write_mode, append_mode**.

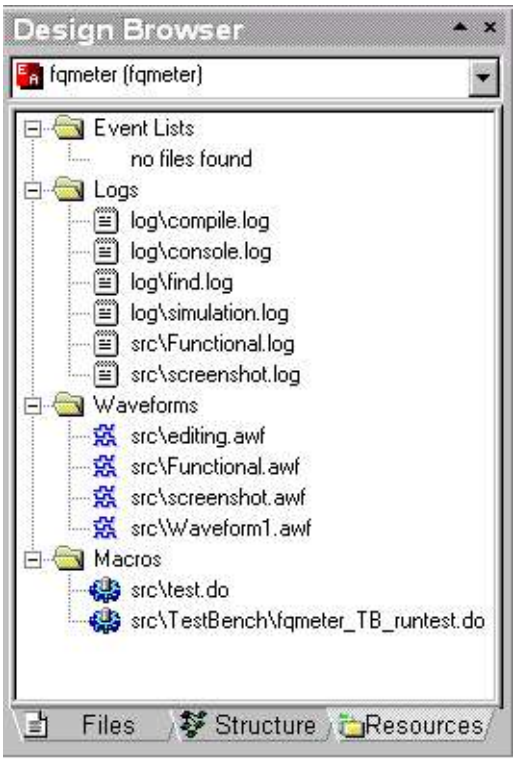
4. **Type (**тип**).** VHDL-тип объекта.
5. **Last Value** (последнее значение). Предыдущее значение объекта, которое он имел до последнего события. Эта колонка используется только для сигналов.
6. **Last Event Time.** Время последнего события. Используется также только для сигналов.

Можно выбирать нужные колонки для вывода. Изучать состояние объектов допускается только в момент приостановки процесса моделирования. Значения, отличные от выведенных в момент предыдущей приостановки, выделяются красным цветом.

Подокно Resources

Представляет файлы ресурсов, отсортированные в соответствии с их типами, которые изображены на рисунке 3.5. Для каждой папки можно определить: её имя, множество расширений файлов, которые будут включены в нее. Разрешается добавлять или удалять папки ресурсов. Они не являются реальными, не влияют на папки, в которых содержатся файлы, и существуют только в подокне **Resource** окна **Design Browser**.

Рисунок 3.5. Подокно Resources окна Design Browser

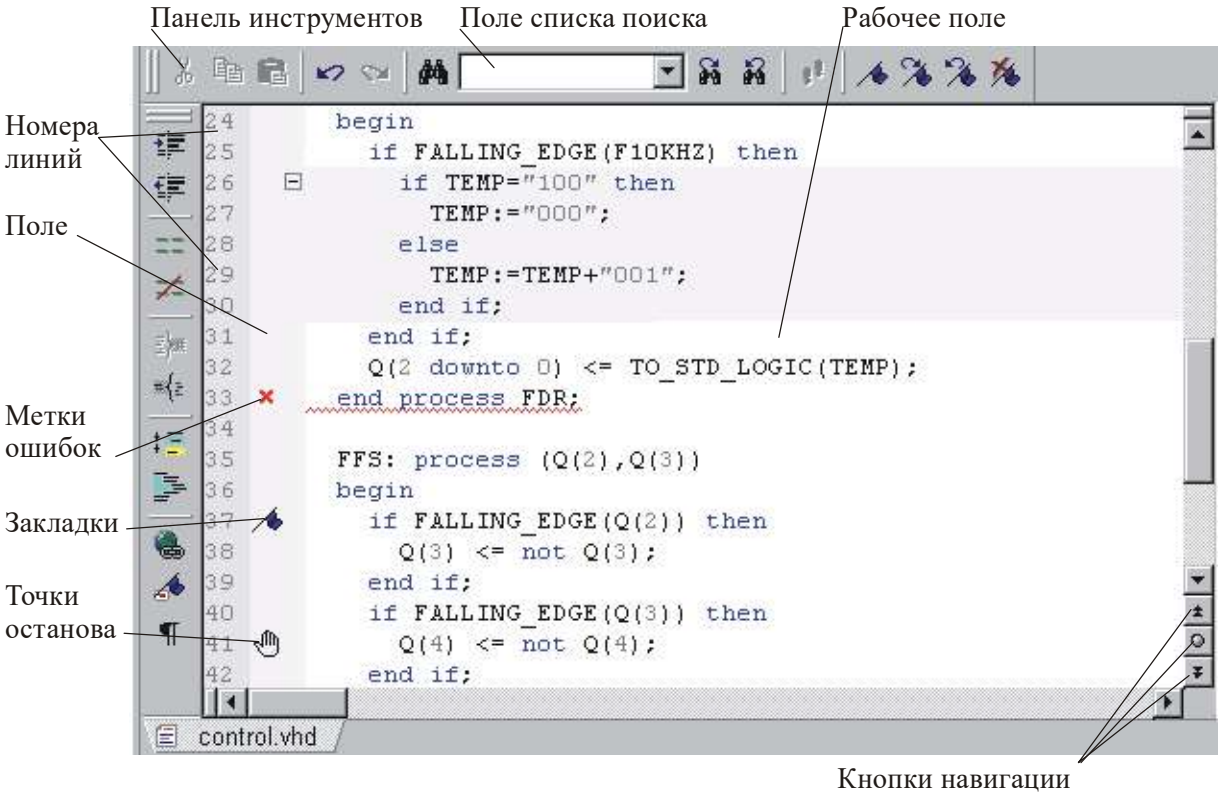


3.3. Редактор HDL Editor

Редактор HDL Editor – это инструмент для ввода моделей, представленных в виде VHDL, Verilog и C/C++ кодов, а также других текстовых файлов. Active-HDL предлагает несколько автоматических методов генерации VHDL-кода, таких как New Source File Wizard.

Рисунок 3.6 содержит окно редактора HDL Editor.

Рисунок 3.6. Окно редактора HDL Editor



Окно редактора HDL Editor содержит:

1) Панель инструментов (**Toolbar**), на которой расположены кнопки для наиболее часто используемых команд.
2) Рабочее поле (**Editing Workspace**), предназначенное для редактирования HDL документов. Для улучшения читаемости исходного кода отдельные синтаксические категории выделяются различным цветом.
3) Поле (**Margin**), на котором размещаются закладки, метки ошибок и номера линий.
4) Номера линий (**Line Numbers**), облегчающие редактирование больших документов. Относительно их выдаются сообщения об ошибках.
5) Закладки (**Bookmarks**), улучшающие навигацию больших документов. Можно определить несколько закладок и затем быстро переходить от одной к другой.
6) Точки останова (**Breakpoints**) в VHDL используются для приостановки процесса моделирования.
7) Метки ошибок (**Error Marks**), помогающие их обнаруживать. Для того чтобы найти метку ошибки из окна **Console**, следует дважды щелкнуть по полю сообщения об ошибке. После этого будет открыт исходный файл и выделена строка с оператором, требующим исправления.
8) Поле списка поиска (**Find List Box**), используемое для быстрого поиска описанной строки. Для этого необходимо ввести строку и нажать кнопки  .
9) Кнопки навигации (**Browse Buttons**) предназначены для быстрой навигации редактируемого текста.

3.4. Создание нового проекта

Для того чтобы создать проект непосредственно из окна Active-HDL, следует выбрать команду New Design из меню File. После этого будет запущен на выполнение мастер New Design Wizard.

В первом окне (рисунок 3.7) следует выбрать содержание проекта из вариантов:

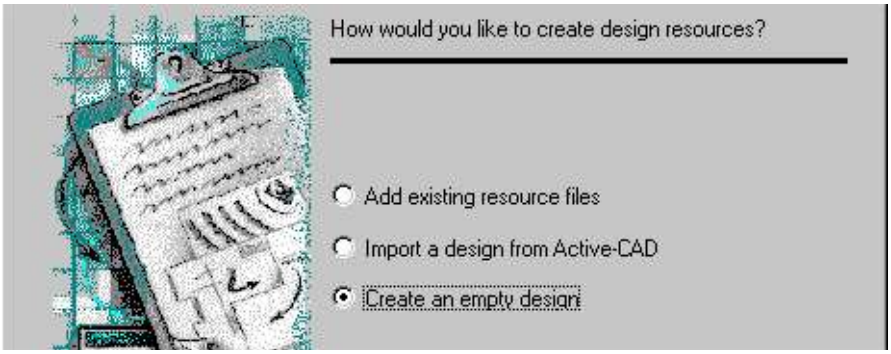
– "Add existing resource files" – добавить существующие файлы.

– "Import a design from Active-CAD" – импортировать проект из Active-CAD.

– "Create an empty design" – создать пустой проект.

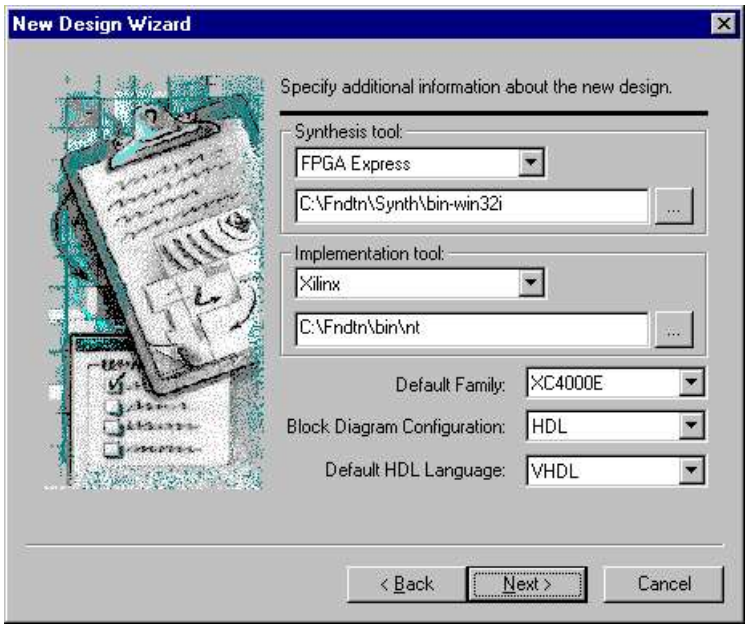
На примере модели полного сумматора проиллюстрируем создание нового проекта. В первом окне отмечается вариант "Create an empty design".

Рисунок 3.7. Фрагмент первого окна мастера New Design Wizard



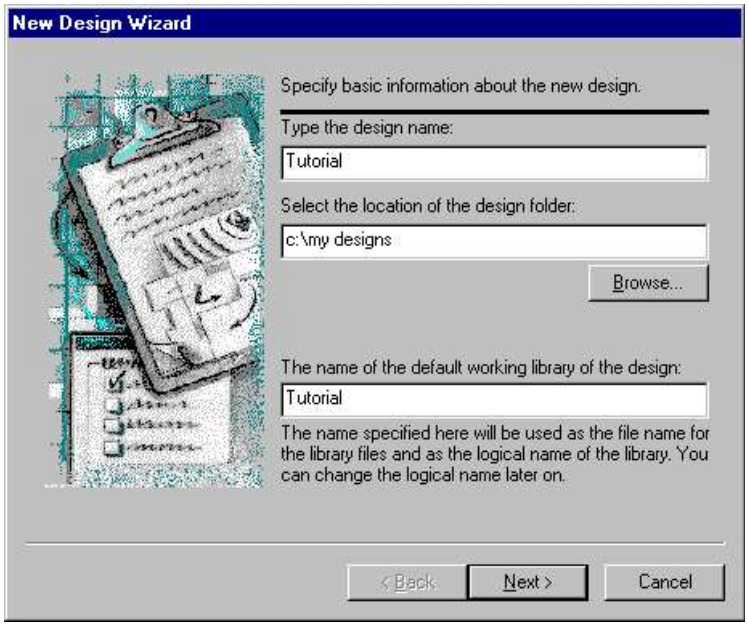
Второе окно мастера (рисунок 3.8) предназначено для выбора инструментов синтеза и имплементации. Необходимо также указать тип микросхемы, конфигурацию редактора Block Diagram и HDL-язык. Поскольку до сих пор не рассматривался синтез и имплементация, данные параметры имеют значения, устанавливаемые по умолчанию. В поле "Default HDL Language" выбирается параметр "VHDL".

Рисунок 3.8. Окно мастера для описания средств синтеза и имплементации



В следующем окне (рисунок 3.9) задаются имена проекта и рабочей библиотеки, описывается расположение проекта на диске.

Рисунок 3.9. Окно мастера для ввода имени проекта



Последнее окно выводит свойства создаваемого проекта для подтверждения их принятия – кнопка Finish, изменения – кнопка Back и отмены создания нового проекта – кнопка Cancel .

Создание нового документа с помощью New Source File Wizard

Мастер New Source File Wizard вызывается командой File/New/VHDL Source или двойным щелчком по иконке Add New File в окне Design Browser. После этого появляется окно Add New File (рисунок 3.10), которое позволяет выбрать создание нового файла мастером (вкладка Wizards) или задание пустого файла (вкладка Empty) с указанием типа создаваемого документа.

Для ввода проекта сумматора следует выбрать тип HDL Source Code из подокна Wizards.

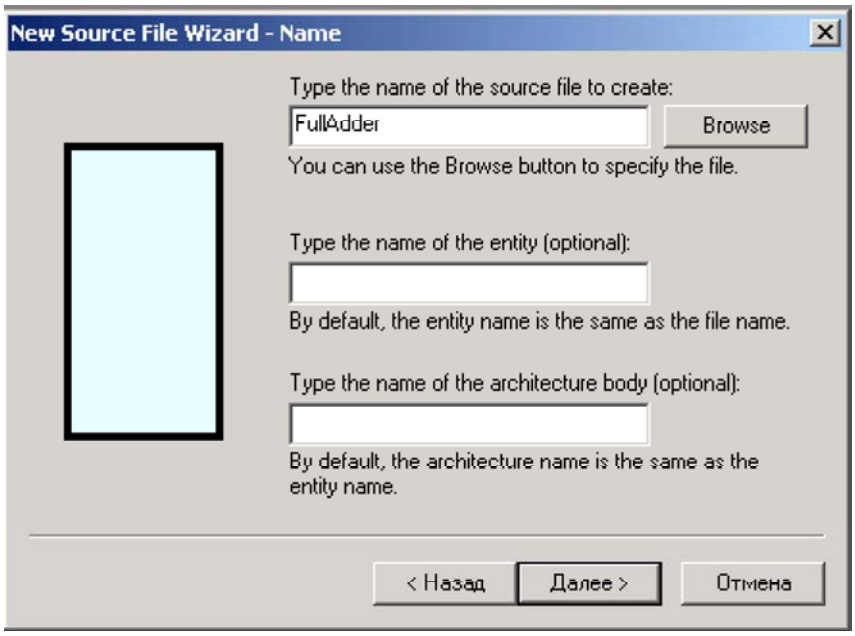
Рисунок 3.10. Окно Add New File

Мастер облегчает ввод VHDL-модели, который выполняется за несколько шагов.

1) Описывает, будет ли создаваемый код добавлен к проекту. Для этого должен быть установлен флажок Add the generated file to design.

2) Вводятся имена файла, интерфейса и архитектуры проекта. Имя файла является обязательным параметром, остальные – нет. Для сумматора это имя FullAdder (рисунок 3.11).

Рисунок 3.11. Окно Name мастера Source File Wizard

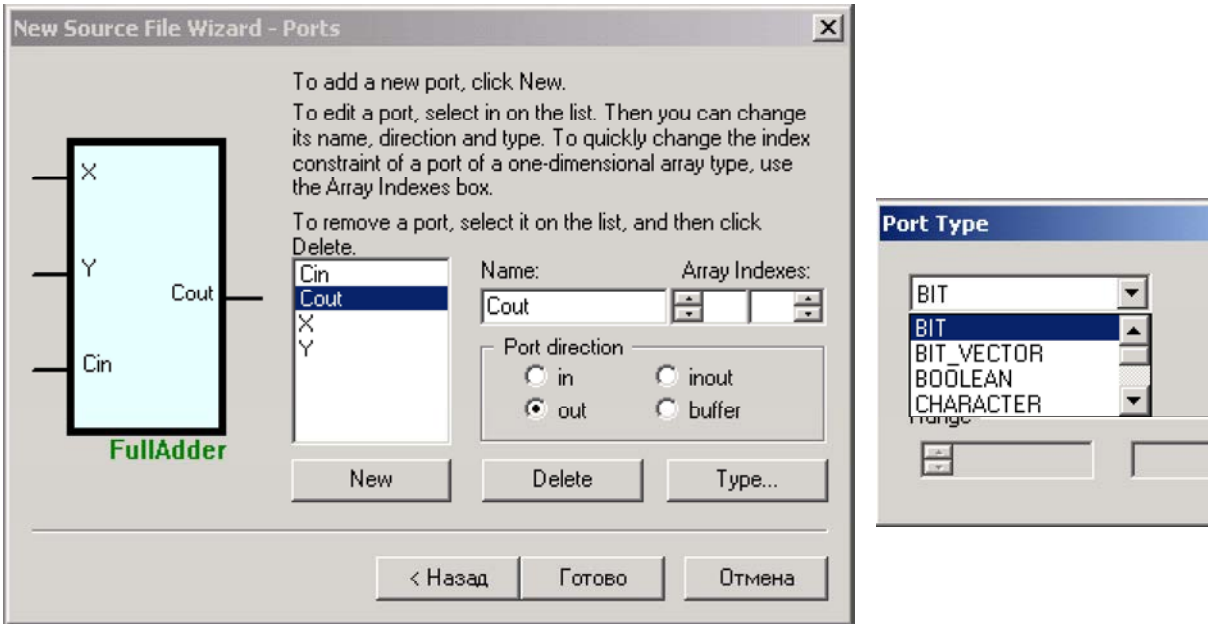


3) Вводится информация о портах (рисунок 3.12). Для создания нового порта необходимо нажать кнопку New. В поле Name ввести его имя. Для выбора направления используется группа параметров – Port direction. Чтобы выбрать тип создаваемого сигнала, следует щелкнуть по кнопке Type и открыть окно Port Type. Параметр Array Indexes используется для ввода диапазонов массивов.

Полный сумматор имеет следующие порты:

```
X, Y, Cin: in bit;
Cout, Sum: out bit.
```

Рисунок 3.12. Окно Ports мастера Design Wizard Window



После того как будет нажата кнопка Готово (Finish), создается шаблон VHDL-кода модели сумматора. В архитектуре вместо надписи " --enter your statement here" необходимо добавить код, описывающий поведение сумматора:

```
Sum <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
```

Полученная VHDL-модель приведена на рисунке 3.13. Структура созданного проекта представлена на рисунке 3.14.

Рисунок 3.13. VHDL-код модели полного сумматора

```
18
19  library IEEE;
20  use IEEE.STD_LOGIC_1164.all;
21
22  entity FullAdder is
23       port(
24            X : in BIT;
25            Y : in BIT;
26            Cin : in BIT;
27            Cout : out BIT;
28            Sum : out BIT
29            );
30  end FullAdder;
31
32  --}} End of automatically maintained section
33
34  architecture FullAdder of FullAdder is
35  begin
36
37  Sum<= X xor Y xor Cin after 10 ns;
38  Cout<= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
39
40  end FullAdder;
```

Рисунок 3.14. Структура проекта, представленная в окне Design Browser

Создание нового пустого документа в HDL Editor

Для создания нового документа следует щелкнуть по стрелке кнопки New  , а затем из выпадающего меню выбрать тип документа:

– VHDL Source (VHDL-код),
– Verilog Source (Verilog-код),
– Text Document (Текстовый документ),
– Perl Script(Perl скрипт),
– Tcl Script (Tcl скрипт),
– Macro (Макрос).

Содержание кнопки New всегда имеет иконку типа последнего созданного документа. По этой причине она может выглядеть по-разному.

Таким образом, можно ввести и добавить к проекту модель 4-разрядного сумматора (см. рисунок 2.5). Рисунок 3.15 представляет содержимое окна Design Browser после выполнения этой операции. Аналогично создается и командный файл Macro, содержащий макрокоманды для моделирования сумматора adder4:

```
asim Adder4
list A B Co C Ci S
```

```
wave A B Co C Ci S
force A 1111
force B 0001
force Ci 1
run 50 ns
```

Рисунок 3.15. Структура проекта

3.5. Language Assistant

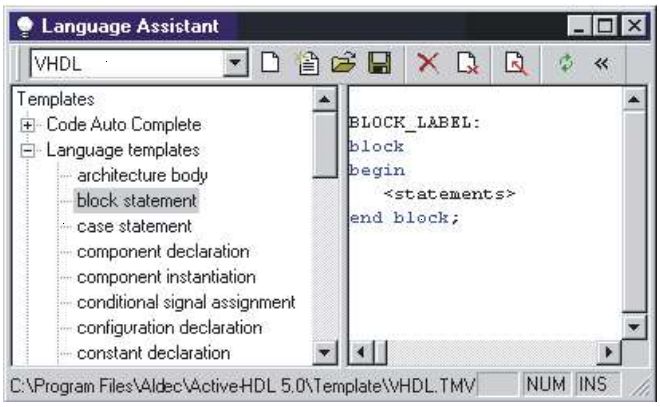
VHDL-файл, генерируемый мастером, представляет собой шаблон, в который добавляются определенные пользователем имена. Более опытные разработчики скорее предпочитают создавать пустые файлы. Тем же, кто только начинает разрабатывать VHDL-проекты, лучше пользоваться мастером, который имеет непосредственную связь с Language Assistant. Этот инструмент позволяет выбирать шаблоны кода и копировать их из окна Language Assistant в исходные файлы. Таким образом, увеличивается скорость разработки моделей и уменьшается количество ошибок. Кроме использования готовых шаблонов есть возможность создавать и свои собственные.

Предлагается несколько групп шаблонов:

1) **Language templates** языковые шаблоны с основными конструкциями языка.
2) **Synthesis templates** синтезируемые шаблоны – модели основных блоков, таких как мультиплексоры, триггеры, счетчики.
3) **Simulation templates** моделируемые шаблоны с примерами полезных конструкций.
4) **Code Auto Complete** автоматическое дополнение кода с шаблонами, требующими Auto-Complete и Interactive templates свойств редактора HDL Editor. Эти свойства позволяют вводить ключевые слова и шаблоны по первым набранным буквам. Создание новых шаблонов для этой группы расширяет возможности автоматического ввода.
5) **Macro Commands** макрокоманды – шаблоны с примерами основных команд, полезных для компиляции, моделирования, выполнения скриптов, управления проектом и наблюдения сигналов.

На рисунке 3.16 представлено окно Language Assistant. Его левая часть задает иерархическое дерево шаблонов. Правая позволяет выполнить предварительный просмотр выбранного шаблона. Добавить шаблон к исходному коду можно, используя технологию drag-and-drop или выполнив команду Use из меню правой кнопки мыши. Также можно использовать кнопку  .

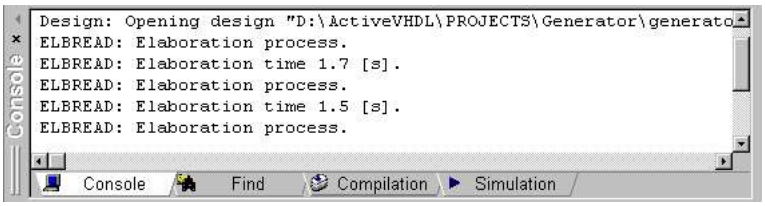
Рисунок 3.16. Окно Language Assistant



3.6. Окно Console

Окно Console (рисунок 3.17) позволяет вводить макрокоманды и скрипты. Все инструменты Active-HDL используют Console для вывода сообщений.

Рисунок 3.17. Окно Console



Окно **Console** содержит несколько вкладок для управления сообщениями от различных Active-HDL инструментов. Вкладка **Console** выводит сообщения, генерируемые всеми инструментами. Она используется также для ввода макрокоманд. Подокно **Compilation** выводит только сообщения, генерируемые компилятором. Подокно **Find** используется для просмотра результатов поиска, выполненного командой **Find in Files** из меню **Search**. Подокно **Simulation** выводит сообщения, генерируемые во время моделирования.

Сообщения, генерируемые средой Active-HDL и выводимые в окне **Console,** сохраняются в log файлах и доступны для просмотра. Сообщения из каждого подокна сохраняются в отдельном файле. Log файлы текущего проекта перечислены в подокне **Resources** окна Design Browser.

3.7. Компиляция

Компиляция – это процесс анализа исходных файлов, которые затем размещаются в рабочей библиотеке в формате, понятном для системы моделирования. По умолчанию все файлы проекта компилируются в рабочую библиотеку. Однако её можно задавать индивидуально для каждого исходного файла проекта. После того как модуль был откомпилирован, можно выполнять его моделирование.

Active-HDL предоставляет несколько способов для выполнения компиляции файлов проекта:

1. Исходные файлы могут быть откомпилированы отдельно. Для этого используется команда **Design /Compile** или кнопка на панели инструментов .
2. Все исходные файлы компилируются за один проход. В этом случае файлы

обрабатываются в том порядке, в котором они представлены в подокне Files окна Design Browser (сверху вниз). Команда **Design /Compile All**. Кнопка **.**

3. Все файлы компилируются за один проход с предварительным изменением порядка обработки. Их сортировка выполняется автоматически, гарантируя таким образом правильный порядок компиляции. Команда **Design /Compile All with File Reorder**. Кнопка .
4. Можно компилировать за один проход все исходные файлы, собранные в одну папку. Она должна быть определена в подокне Files окна Design Browser. Команда Compile All in Folder доступна из меню, вызываемого щелчком правой кнопки мыши по папке.
5. Можно также выполнить макрокоманду **acom**.

Design Browser позволяет исключать выбранные файлы и папки из процесса компиляции. Команды Compile All, Compile All with File Reorder и Compile All in Folder игнорируют такие файлы. Они также игнорируются при выполнении компиляции макрокомандой acom, если в ней не были описаны исходные файлы.

Все сообщения, генерируемые во время компиляции, выводятся в окне **Console.**

3.8. Окно List

Окно **List** (рисунок 3.18) отображает значения выбранных сигналов, полученные во время моделирования, в текстовой табличной форме. Содержание окна может быть сохранено в текстовом формате. Каждому сигналу соответствует колонка значений для последовательных моментов моделирования. Значения сигналов могут быть представлены двумя способами:

– для всех циклов моделирования, включая дельта-циклы, выполняемые для каждого шага моделирования;

– только последние циклы для каждого шага моделирования.

Рисунок 3.18. Окно List

Открыть окно List можно командой New> List из меню File или использовать для этого кнопку панели инструментов. В одно и то же время можно работать с несколькими окнами **List**.

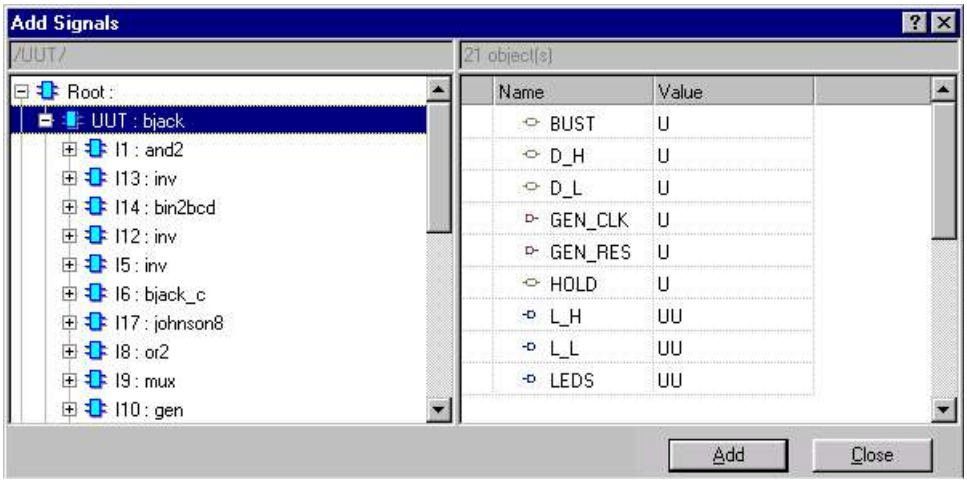
Панель инструментов окна List содержит кнопки: – добавить сигналы, – разрешение вывода дельта-циклов, – прокрутка содержимого окна до заданной временной точки.

Добавить сигналы в окно List можно, используя технологию "Drag-and-drop". Для этого следует выбрать сигнал(ы) из списка, расположенного в нижней части вкладки Structure окна Design Browser, а затем переместить его на рабочее пространство окна List. Если нужно выделить несколько сигналов, следует использовать клавишу Ctrl.

Другим способом добавить сигналы в окно List можно с помощью команды Copy и Paste. Выбранный сигнал или ветвь иерархии копируется с помощью команды Copy. Чтобы добавить их к содержимому окна List, используется команда Paste. Обе команды доступны из меню правой кнопки мыши, в окне List которой следует щелкать по свободному пространству, а не по колонке сигнала.

Также можно добавлять сигналы, используя диалоговое окно Add Signals. Для этого следует щелкнуть правой кнопкой мыши в окне List и выбрать команду Add Signals из контекстного меню, после чего будет открыто окно Add Signals (рисунок 3.19). В левой его части нужно указать область проекта, сигналы из которой будут помещены в окно List. В правой – выбирается сам сигнал. Затем нажимается кнопка Add. Для того чтобы выбрать более одного сигнала, используется кнопка Ctrl.

Рисунок 3.19. Диалоговое окно Add Signals dialog



Чтобы удалить сигнал из окна List, его нужно выделить и нажать клавишу **Delete**.

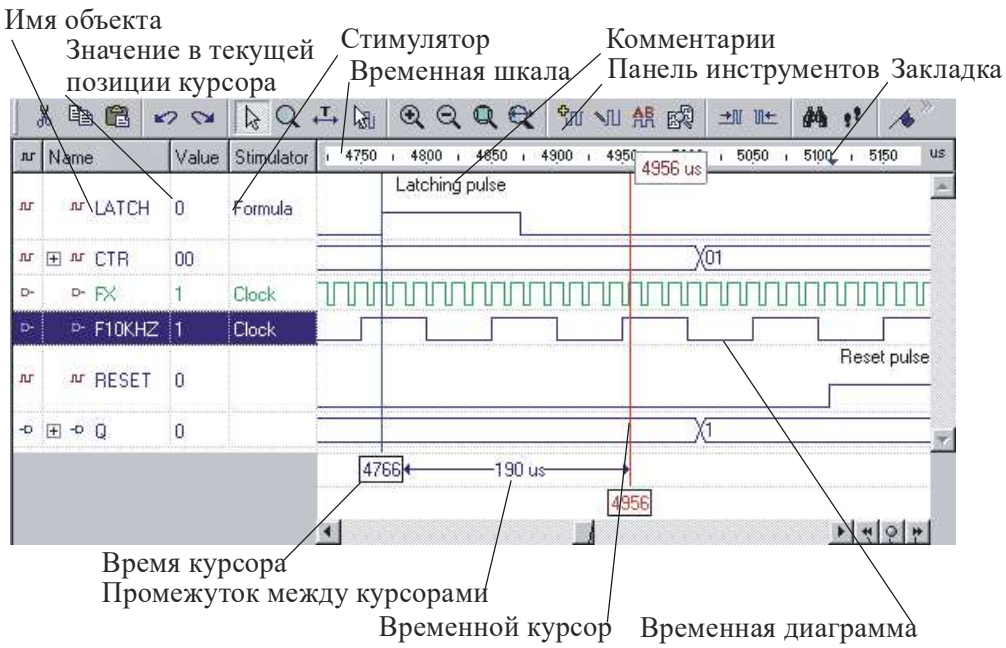
Для выводимых значений сигналов можно указывать различные системы счисления. При этом следует выделить один или несколько сигналов и выбрать команду Properties из меню правой кнопки мыши. Будет открыто окно Signal Properties, в котором задается требуемый параметр системы счисления: Binary (двоичная), Octal (восьмеричная), Decimal (десятичная) или Hexadecimal (шестнадцатеричная).

3.9. Редактор временных диаграмм

Waveform Editor (редактор временных диаграмм) – инструмент, предназначенный для представления результатов моделирования в форме временных диаграмм. Во время моделирования временные диаграммы предварительно выбранных сигналов и переменных выводятся в окно Waveform Editor. Новые объекты можно добавлять к окну в любой момент моделирования. Однако их временные диаграммы будут созданы, начиная с момента, в который они были добавлены в окно.

Waveform Editor является также средством редактирования и позволяет изменять временные диаграммы сигналов. Они могут быть сохранены в файле и использоваться как тестовые последовательности во время последующего моделирования. Существует возможность их экспортирования в различные файловые форматы: VHDL Process (*.VHS), VHDL Waves Vectors (*.VEC), List file (*.LST). В один момент времени можно работать с несколькими окнами **Waveform Editor** .

Рисунок 3.20. Окно Waveform Editor



Левая часть окна, изображенного на рисунке 3.20, содержит колонки, описывающие сигнал. Они представляют его основные характеристики, временные диаграммы которых выводятся в окне. Могут быть выведены следующие типы колонок:

1. Режим. Эта колонка содержит небольшие иконки, обозначающие, является ли сигнал обыкновенным ( ) или портом. Для последних описывается их режим: in , out , inout , buffer , linkage . .
2. Name (имя) содержит идентификатор сигнала.
3. Hierarchy (иерархия) – представляет иерархическую позицию сигнала (путь к нему) в проекте.
4. Type (тип) описывает VHDL-тип сигнала.
5. Value (значение) содержит значение сигнала для текущей позиции временного курсора.
6. Stimulator (стимулятор). В колонке выводится тип стимулятора, назначенного сигналу.

Временная шкала представляет собой масштабную линейку с временными метками и закладками, определенными пользователем. Панель инструментов содержит кнопки для часто используемых команд. Временной курсор (Timing Cursor) позволяет получать значения сигналов в любом месте временной шкалы.

Кнопки панели инструментов окна Waveform Editor

- Вырезать выделенную временную диаграмму и поместить ее в Clipboard.
- Скопировать выделенную временную диаграмму в Clipboard.
- Вклеить временную диаграмму из Clipboard.
- Отменить последнюю операцию.

 Восстановить последнюю операцию.

 Перейти в режим выделения.

 Перейти в режим масштабирования.

 Перейти в режим измерения.

 Перейти в режим редактирования.

 Увеличить масштаб.

 Уменьшить масштаб.

 Настроить масштаб по всей временной диаграмме.

 Добавить сигнал.

 Добавить стимулятор к сигналу.

 Ввести комментарий.

 Сравнить текущие временные диаграммы с сохраненными в файле.

 Переместить курсор к следующему событию выбранного сигнала или всех сигналов, если ни один из них не выделен.

 Переместить курсор к предыдущему событию выбранного сигнала или всех сигналов, если ни один из них не выделен.

 Поиск описанного значения или текстового комментария для выделенного сигнала.

 Переместить курсор в заданную временную точку.

 Установить / удалить закладку.

 Переместить курсор к ближайшей закладке вперед.

 Переместить курсор к ближайшей закладке назад.

 Удалить все закладки.

Для того чтобы открыть новое окно Waveform Editor, можно использовать команду New>Waveform из меню File или кнопку  на основной панели инструментов.

Добавить сигналы к окну Waveform Editor можно таким же образом, как и для окна List. Кроме того, имя сигнала и путь к нему можно ввести вручную в окне Waveform Editor. Для этого нужно перейти в режим редактирования  . Дважды щелкнуть по линии с меткой Click here и ввести имя объекта. Щелкнуть по колонке Hierarchy и ввести путь к сигналу, если это необходимо.

Для описания объектов, которые не объявляются на верхнем уровне, используется путь, показывающий их точное расположение в иерархии проекта. При этом применяется следующее синтаксическое правило:

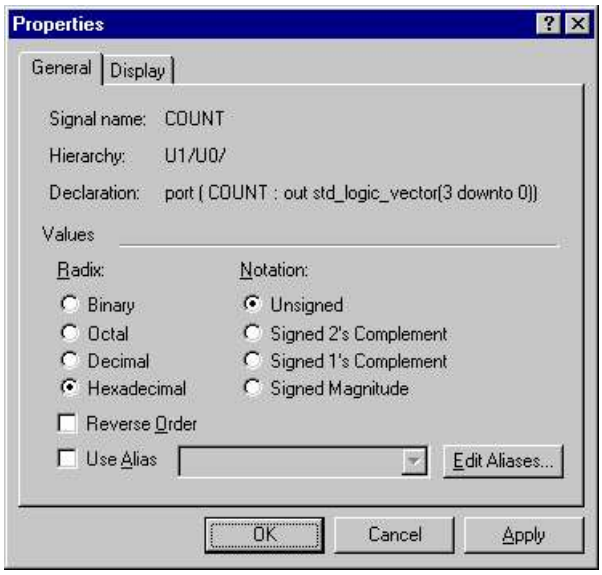
```
object_path::= { instance_name /} object_name
```

Формальные параметры и локальные объекты в VHDL-подпрограммах (функциях и процедурах) выводятся с иерархическим путем следующего вида:

```
local_object_path ::= [ process_name / ] { subprogram_name / }
object_name
```

Для редактирования свойств сигналов и их временных диаграмм существует команда Properties из контекстного меню правой кнопки мыши. Эта команда открывает окно Signal Properties (рисунок 3.21), которое содержит две вкладки: General и Display.

Рисунок 3.21. Окно Signal Properties–General



Вкладка General (см. рисунок 3.21) включает пункты:

– Signal name(имя сигнала).

– Hierarchy (иерархия). Выводится путь к выбранному сигналу.

– Declaration (декларация). Представлено описание сигнала из исходного кода.

– Группа Values описывает систему счисления, в которой представляются сигналы: Binary (двоичная), Octal (восьмеричная), Decimal (десятичная), Hexadecimal (шестнадцатеричная).

– Reverse Order(обратный порядок). В одномерном массиве меняется местоположение старшего и младшего битов на обратное.

– Use Alias (использование псевдонимов). Разрешение использовать выбранные из списка псевдонимы.

– Edit Aliases (редактирование псевдонимов). Открывается окно Alias Editor для создания и редактирования псевдонимов.

– Группа Notations (представление) содержит пункты: Unsigned (беззнаковое), Signed 2's Complement (знаковое двоичное дополнение), Signed 1's Complement (знаковый обратный код), Signed Magnitude (знаковый модуль). Если выбран пункт Unsigned , значение сигналов выводится в беззнаковом формате. В противном случае старший бит вектора рассматривается как знак.

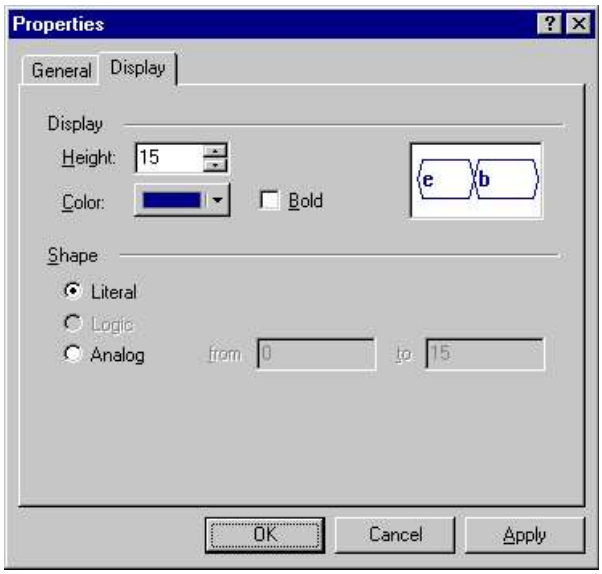
В таблице 3.1 даются значения двоичных чисел, представленных в различных кодах.

Таблица 3.1. Представления чисел в знаковых формах

Decimal	Singed-2's complement	Signed-1's complement	Signed- magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	-	-

Вкладка Display представлена на рисунке 3.22. Она содержит пункты, определяющие графическое изображение временных диаграмм: Height (высота), Color(цвет), Bold (полужирное), Preview (предварительный просмотр) и группу Shape (форма). Height позволяет задавать высоту временной диаграммы в пикселах. Повышение данного параметра увеличивает расстояние между сигналами, а не размер диаграммы. Параметр Preview разрешает вывод примера временной диаграммы с использованием текущих параметров. Группа Shape описывает форму, в которой будут представлены результаты моделирования. Это может быть Literal, когда значения временной диаграммы представлены в виде чисел в заданной системе счисления, или Logic, что соответствует логической форме (1, 0, X). Analog позволяет выводить временную диаграмму в аналоговой форме. После выбора соответствующего параметра аналоговые данные вычисляются в некотором диапазоне from .. to. Эта опция полезна для типов integer, physical и floating point.

Рисунок 3.22. Окно Signal Properties–Display

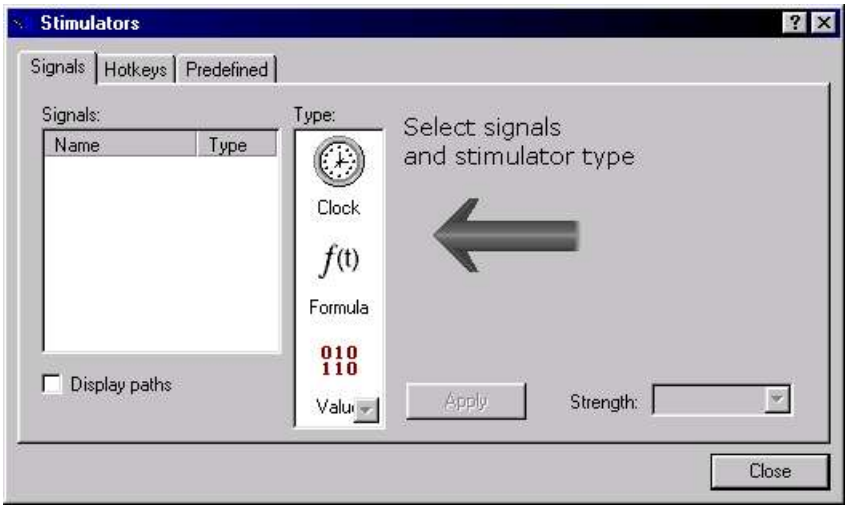


3.10. Стимуляторы

Стимуляторы – это определенные пользователем виртуальные источники входных значений, подключенные к сигналам. Значение, присвоенное стимулятором, может быть константой или изменяться по правилам, заданным типом стимулятора либо параметрами. Это самый быстрый и простой метод задания сигналам требуемых значений. Может применяться к любому сигналу или порту в иерархии проекта.

Active-HDL предлагает графический интерфейс для определения и применения стимуляторов. Для этого используется окно **Stimulators** (рисунок 3.23), вызываемое командой **Stimulators** из меню **Waveform** или краткого меню правой кнопки мыши.

Рисунок 3.23. Окно Stimulators, вкладка Signals



На вкладке Signals в поле Signals выводится список сигналов с соответствующими им стимуляторами. Колонка **Name** содержит имена сигналов, колонка **Type** – типы связанных с ними стимуляторов. Флаговая кнопка слева от имени сигнала позволяет включить или отключить стимулятор. Отключенные стимуляторы игнорируются во время моделирования. Чтобы добавить сигнал, необходимо щелкнуть по его имени в открытом окне **Waveform Editor.** Для того чтобы удалить стимулятор, его следует выделить, затем нажать **Delete.**

Поле Strength определяет, как стимулятор будет влиять на значение выбранной линии. Возможные режимы.

1 **– Deposit** – значения, присвоенные стимулятором, перекрывают текущее состояние сигнала, генерируемое в тестируемом устройстве. Эффект длится пока есть последовательность управляемых транзакций или программа моделирования не будет остановлена.

2 **– Drive** – значение(я), генерируемое стимулятором, воздействует на текущий сигнал так, как если бы еще один драйвер был подсоединен к этой линии. Эффект длится пока есть последовательность управляемых транзакций или программа моделирования не будет остановлена. Может использоваться только для разрешенных сигналов.

3 **– Override** – значение стимулятора заменяет текущее значение сигнала. Эффект длится пока программа моделирования не будет остановлена.

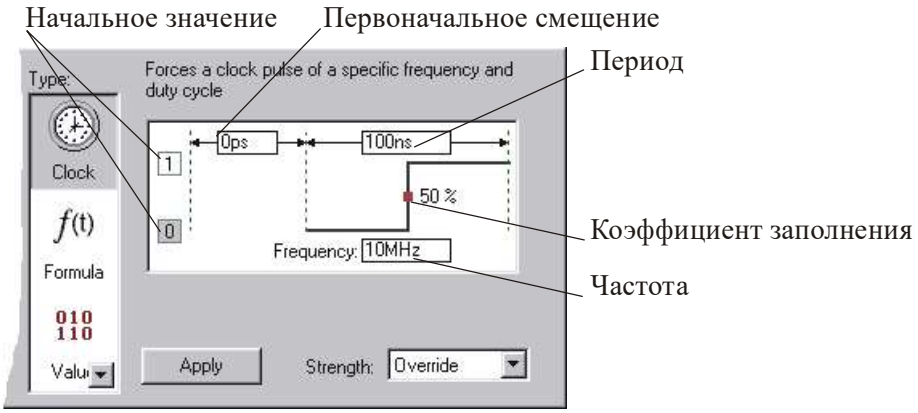
Флаг **Display Path** разрешает вывод пути иерархии сигнала в поле Signals. Кнопка **Apply** используется для назначения стимуляторов.

На вкладке Signals поле Type предназначено для выбора стимулятора выделенному сигналу. Существуют следующие их типы: Clock Stimulators (стимуляторы синхроимпульсов), Counter Stimulators (счетчики), Custom Stimulators (пользовательские стимуляторы), Formula Stimulators (формульные стимуляторы), Value Stimulators

(стимуляторы значений), Hotkey Stimulators (горячие клавиши), Predefined Stimulators (предопределенные стимуляторы).

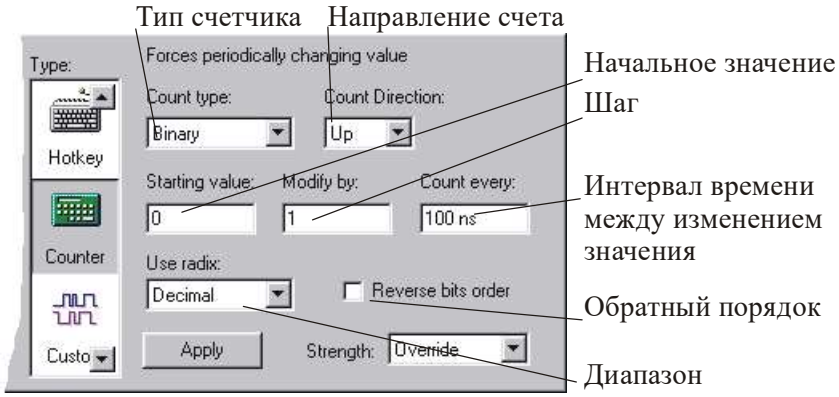
Clock Stimulators (стимуляторы синхроимпульсов) (рисунок 3.24) генерируют синхросигналы, имеющие следующие параметры: частоту/период, первоначальное смещение, коэффициент заполнения и начальное значение. Обычно такие стимуляторы используются для управления синхровходами.

Рисунок 3.24. Стимулятор синхроимпульсов



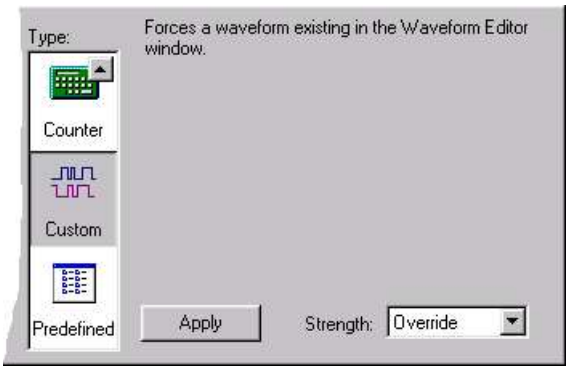
Counter (счетчик) может быть применен к сигналам типа одномерный массив или integer. Стимулятор создает последовательность значений, которые представляют собой очередь состояний счетчика. На рисунке 3.25 показаны параметры, которые можно задавать – это шаг и направление счета, интервал времени между изменением значения, начальное состояние и тип счетчика. Возможные типы счетчиков – binary (двоичный), Gray (Грея), Johnson (Джонсона), Circular One (циклическая единица) и Circular zero (циклический нуль). Параметр Increment by (шаг) применим только при использовании двоичного счетчика и счетчика Грея.

Рисунок 3.25. Стимулятор типа счетчик



Custom Stimulators (пользовательские стимуляторы) назначают сигналу его собственную временную диаграмму, которая уже существует в окне Waveform Editor. Ее можно создать вручную с помощью средств редактора Waveform Editor. Однако чаще используется временная диаграмма, полученная на предыдущем шаге моделирования или загруженная из файла. Например, для генерации формы сигнала сначала использовалась горячая клавиша. Затем для повторного применения полученной временной диаграммы тип стимулятора этого сигнала изменяется на Custom. Как видно из рисунка 3.26, стимулятор Custom не имеет параметров в подокне Signals.

Рисунок 3.26. Пользовательский стимулятор



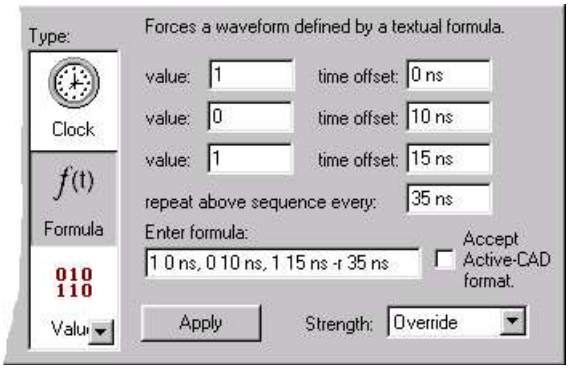
Formula Stimulators (формульные стимуляторы) (рисунок 3.27) генерируют временные диаграммы, заданные с помощью простой текстовой формулы. Она содержит последовательность пар параметров, состоящих из значения сигнала и момента времени, в которое он его получает. Также формула может содержать параметр повтора, означающий, что данная последовательность должна повторяться с заданным периодом.

Синтаксис формулы:

```
<value> <time> [ , <value> <time> ... ] [ -r <period> ]
```

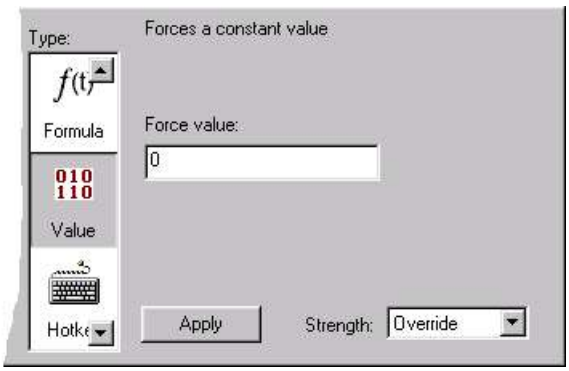
Параметры для ввода: "Value" – значение сигнала, "Time offset" – время присвоения значения, "Repeat above sequence every" – ввод периода для повторяющихся последовательностей, "Enter Formula" – позволяет ввести значение непосредственно в виде формулы. По умолчанию, если не задано явно, значение времени измеряется в ps.

Рисунок 3.27. Формульный стимулятор



Value Stimulators (стимуляторы значений) назначают сигналу константное состояние (рисунок 3.28).

Рисунок 3.28. Стимулятор значения



Hotkey Stimulators (горячие клавиши) подобны стимуляторам значений, но предлагают более удобный механизм для изменения значения. Для этого надо просто нажать описанную (горячую) клавишу, после чего произойдет переключение значения, например, из '0' в '1'. Для определения горячей клавиши или комбинации клавиш используется поле "Press new hotkey" (рисунок 3.29).

Можно определить список значений, которые будут циклически изменяться при нажатии горячей клавиши. Для этого применяется подокно Hotkeys окна Stimulators (рисунок 3.30). Колонка Hotkey содержит список горячих клавиш стимуляторов, колонка Sequence – список значений для каждого стимулятора горячей клавиши. Стимулятор присваивает их сигналу, начиная с самого левого значения и заканчивая самым правым. Для изменения значений или порядка их следования нужно щелкнуть по колонке Sequence, а затем выполнить необходимое редактирование. Значения должны быть разделены запятыми и принадлежать типу сигнала, которому они назначаются.

Рисунок 3.29. Стимуляторы, использующие горячие клавиши

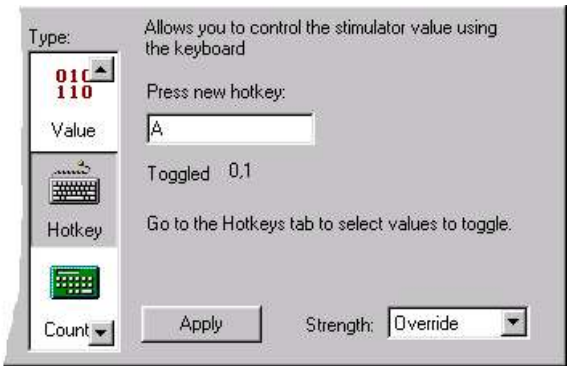
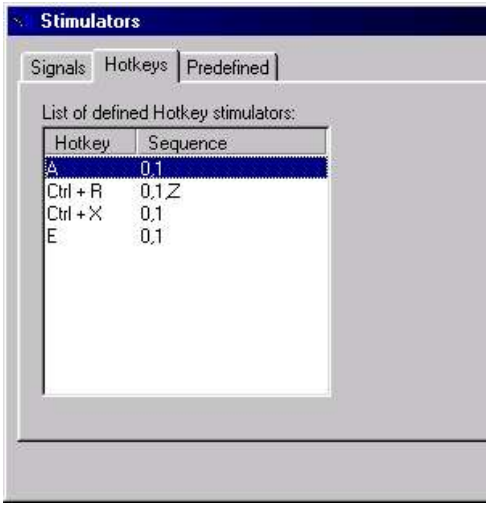


Рисунок 3.30. Подокно Hotkey окна Stimulators



Predefined Stimulators (предопределенные стимуляторы) – это стимуляторы синхросигналов или формульные, имеющие собственное уникальное имя (рисунок 3.31). Поскольку они имеют собственный идентификатор, их легко назначить нескольким сигналам, не повторяя каждый раз ввод параметров. Поле Predefined вкладки Signals окна Stimulators содержит список доступных предопределенных стимуляторов.

Для создания новых предопределенных стимуляторов используется вкладка Predefined (рисунок 3.32). Поле "List of predefined stimulators" содержит список имеющихся предопределенных стимуляторов. В колонке Name описывается имя, в колонке Type – тип. Стимулятор может иметь тип Clock (синхросигнал) или Formula (формула). Кнопка Add используется для создания стимуляторов, а Remove – для их удаления.

Примечание. Описание новых предопределенных стимуляторов должно быть сохранено в системном реестре. Они не могут быть сохранены в файле временных

диаграмм (*.awf). Поэтому их нельзя эффективно использовать при моделировании на других компьютерах.

Рисунок 3.31. Предопределенные стимуляторы

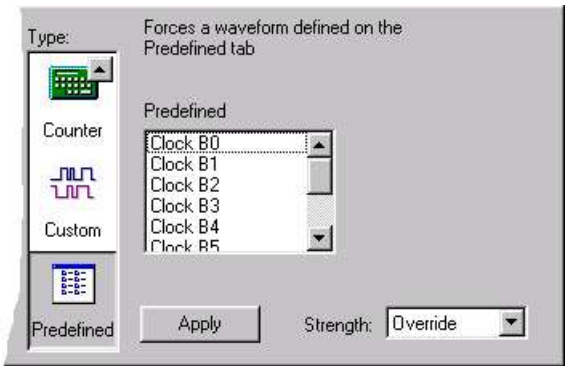
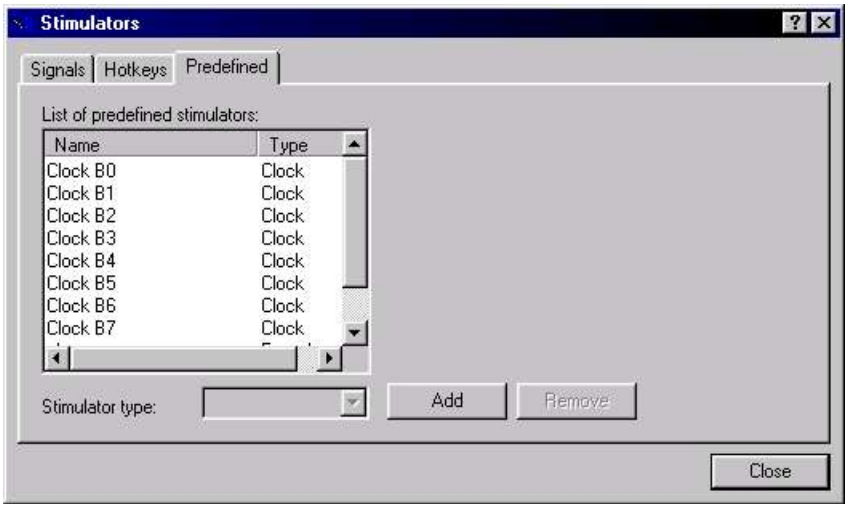


Рисунок 3.32. Вкладка Predefined окна Stimulators



3.11. Моделирование

После того как VHDL-модель устройства успешно откомпилирована, для ее верификации следует использовать моделирование. Перед этой процедурой, если в проекте существует несколько HDL-файлов, необходимо установить модуль верхнего уровня описания.

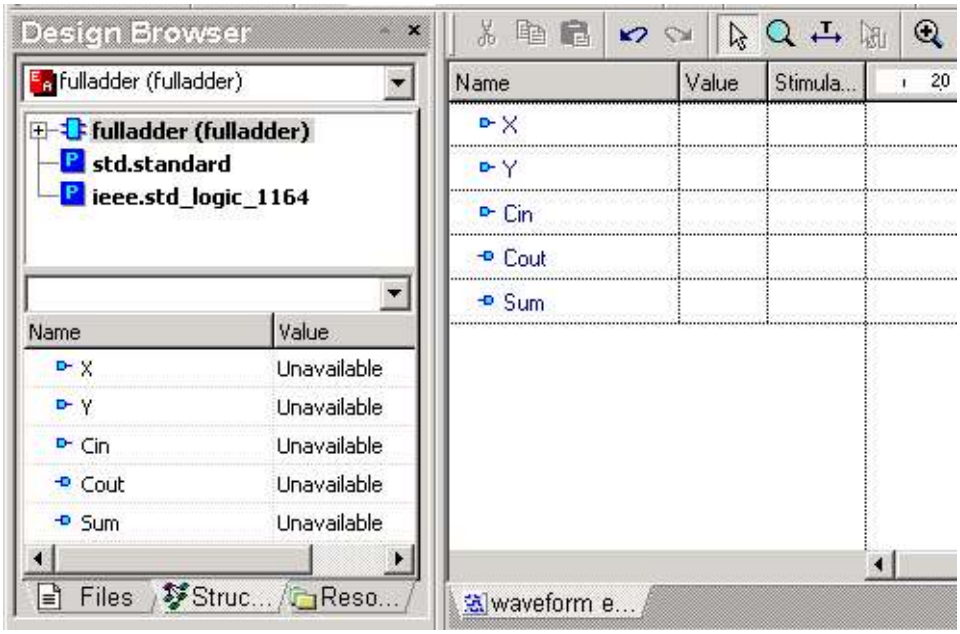
Для того чтобы промоделировать модель полного сумматора с рисунка 3.13 и 3.14, следует в окне Design Browser на вкладке Files или Structure в поле выбора модуля верхнего уровня выбрать FullAdder, как это показано на рисунке 3.33. Затем выполняется инициализация моделирования – команда Initialize Simulation из меню Simulation.

Рисунок 3.33. Выбор модуля верхнего уровня

Назначение стимуляторов

Командой File/New/Waveform или щелчком по кнопке открывается новое окно Waveform Editor. В него с вкладки Structure окна Design Browser переносятся сигналы X, Y, Cin, Cout, Sum (рисунок 3.34).

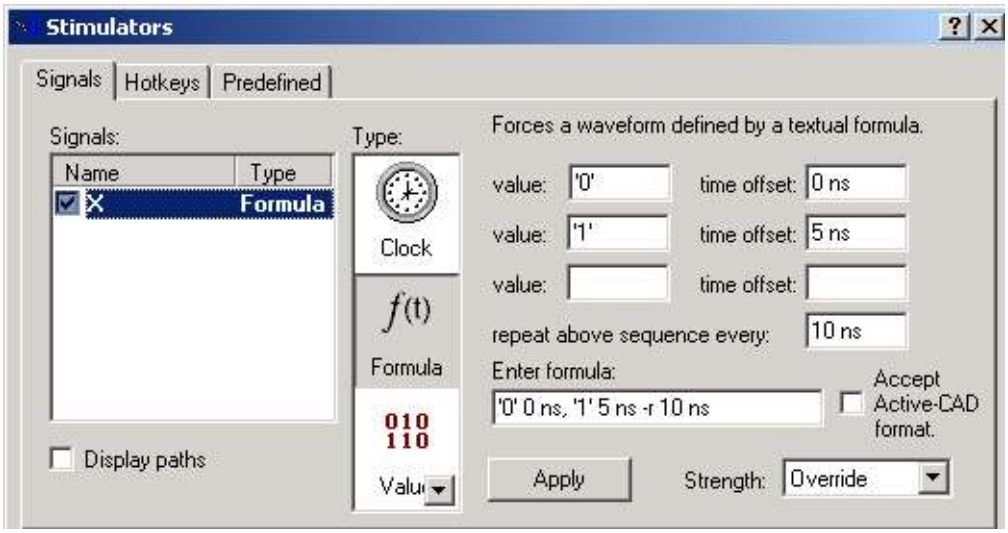
Рисунок 3.34. Окно Waveform Editor с добавленными к нему сигналами



Добавить сигналы можно также с помощью команд Copy и Paste или из окна Add Signals, которое вызывается командой Add Signals из контекстного меню правой кнопки мыши.

Чтобы открыть окно Stimulators, необходимо выделить порт X, щелкнуть правой кнопкой мыши и из контекстного меню выбрать команду Stimulators. В поле Type вкладки Signals выбрать формульный стимулятор, ввести параметры, как это показано на рисунке 3.35, нажать кнопку Apply.

Рисунок 3.35. Ввод параметров формульного стимулятора



Не закрывая окна Stimulators, выбрать сигнал Y в окне Waveform Editor. Для него также назначить формульный стимулятор и ввести формулу: 0 0 ns, 1 10 ns, 0 20 ns, 1 30 ns. Нажать кнопку Apply. Выбрать сигнал Cin, не закрывая окна Stimulators, и задать для него горячую клавишу (рисунок 3.36). Нажать клавишу Apply и закрыть окно

Stimulators. После этого окно Waveform Editor должно содержать информацию, такую же как и на рисунке 3.37. Обратите внимание: в начале моделирования сигнал Cin должен иметь значение '0'.

Рисунок 3.36. Назначение горячей клавиши сигналу Cin

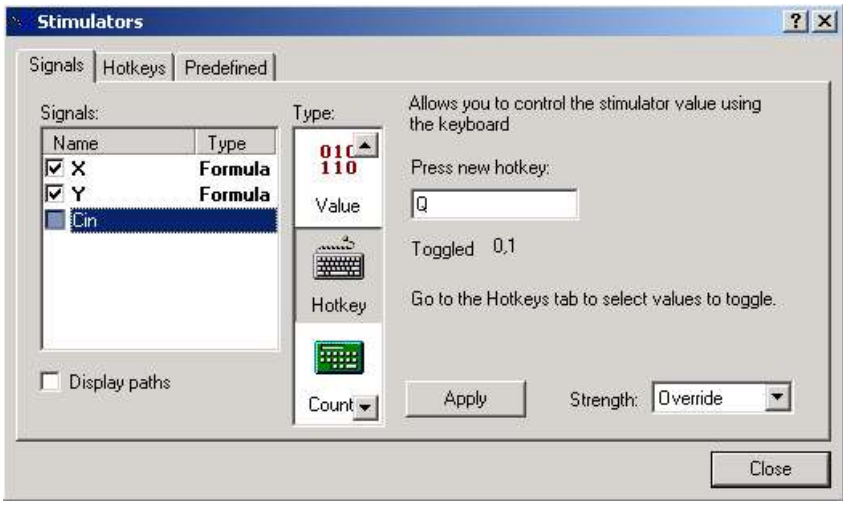
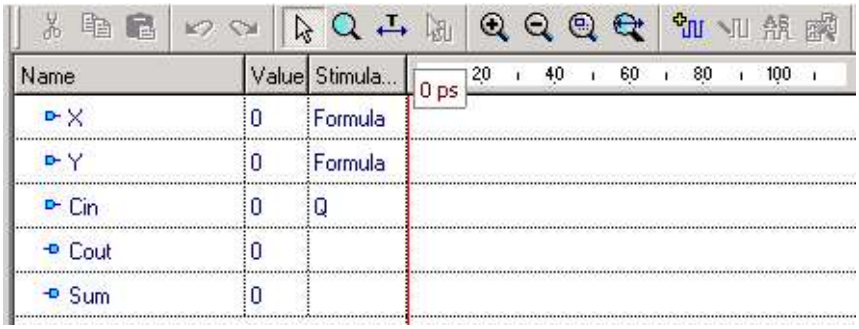


Рисунок 3.37. Окно Waveform Editor с назначенными стимуляторами



Выполнение моделирования

Для проведения моделирования в Active-HDL предлагаются три команды:

1. Команда Simulation /Run запускает моделирование на неопределенный промежуток времени. Моделирование будет закончено при выполнении одного из условий: когда текущее время моделирования станет равным TIME'HIGH; нет ни одного события или другой причины для возобновления выполнения процесса.
2. Команда Simulation /Run For выполняет моделирование в течение описанного временного промежутка (command advances simulation by a specified time step). Для задания этого промежутка используется поле Simulation Step на основной инструментальной панели:

3. Команда Simulation /Run Until запускает выполнение моделирования до описанной временной точки.

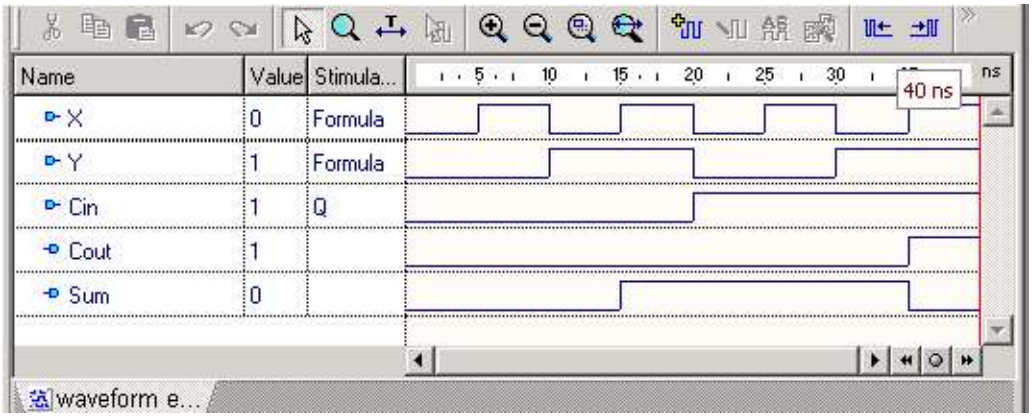
В каждом из представленных выше способов моделирование можно приостановить командой Stop или кнопкой .

Для моделирования полного сумматора в поле Simulation Step следует ввести время 20 ns и нажать кнопку . Затем с помощью горячей клавиши изменить

значение сигнала Cin с '0' в '1'. И снова нажать кнопку , чтобы запустить моделирование еще на 20 ns. Таким образом, будет достигнута точка 40 ns. Иначе можно было бы использовать команду Simulation /Run Until, кнопка . При этом задать точки моделирования сначала 20 ns, затем 40 ns.

Результаты моделирования приведены на рисунке 3.38.

Рисунок 3.38. Результаты моделирования полного сумматора



Для завершения процесса моделирования используется команда Simulation /End Simulation. Программа освобождает память, занимаемую рабочей моделью, и деинициализирует программу моделирования.

Чтобы повторить моделирование, нет необходимости перезагружать модель в программу. Вместо этого следует реинициализоровать процесс моделирования командой Simulation /Restart Simulation или с помощью кнопки .

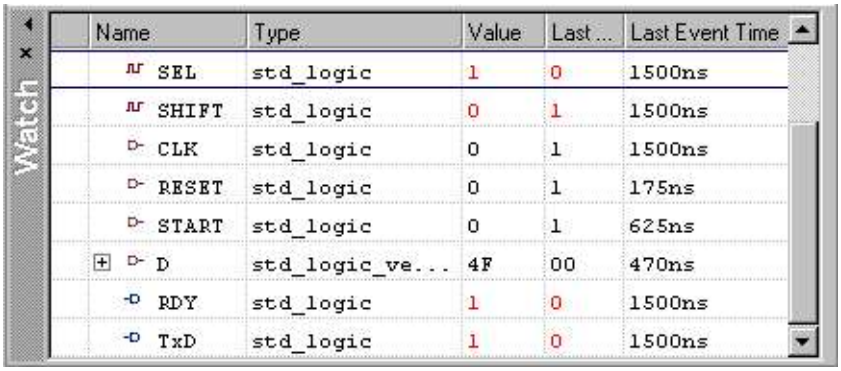
3.12. Другие инструменты для наблюдения за результатами и процессом моделирования

Окно Watch

Окно Watch – это инструмент отладки, предназначенный для вывода значений объектов (сигналов, переменных), выбранных в тестируемой модели (рисунок 3.39). Отображаются только текущие значения, без какой-либо информации об истории их изменения. Объекты представляются в таком же формате, как и на вкладке Structure окна Design Browser (см. рисунок 3.4).

Окно Watch также позволяет выводить значения формальных параметров и локальных объектов VHDL-подпрограмм. Кроме того, для таких объектов существует и специальный инструмент наблюдения – окно Call Stack.

Рисунок 3.39. Окно Watch



Для того чтобы получать значения объектов, их надо добавить в окно Watch. Это можно сделать до инициализации моделирования или в любой момент после. Объект, добавленный в окно Watch, остается в нем, пока не будет удален или до изменения проекта. Окно позволяет наблюдать значения объектов только в момент приостановки моделирования. Формальные параметры и локальные объекты подпрограмм могут быть добавлены в окно только после инициализации моделирования.

Объекты в окне Watch, которые не соответствуют ни одному объекту из тестируемой модели, отмечаются словом Unavailable. Объекты, для которых не указан путь и которые могут одновременно принадлежать модулю верхнего уровня или быть локальными объектами выполняемой подпрограммы, считаются локальными для неё, чтобы избежать неоднозначности.

Для того чтобы открыть или спрятать окно Watch, используется команда Watch из меню View или кнопка  .

Для того чтобы разрешить или запретить вывод отдельных колонок окна Watch, следует щелкнуть правой кнопкой мыши по заголовкам колонок и из краткого меню выбрать необходимые пункты. Возможны варианты: Name, Type, Value, Last Value, Event, Last Event Time.

Добавить сигналы в окно Watch можно несколькими способами:

– Переместить их с вкладки Structure окна Design Browser.

– Переместить их из открытого в окне HDL Editor VHDL файла.

– Использовать команды Copy и Paste.

– Набрать имя объекта и путь непосредственно в окне Watch.

В первых двух случаях путь к объекту копируется вместе с именем, если моделирование было инициализировано.

Окно Watch позволяет редактировать значения переменных и констант. Для этого нужно щелкнуть по соответствующей строке в колонке Value и ввести новое значение. Изменение можно вносить только в момент приостановки процесса моделирования.

Для изменения системы счисления, в которой представляются значения, выводимые в окне Watch, необходимо щелкнуть правой кнопкой мыши и выбрать команду Display Options из контекстного меню. Будет открыто окно Preferences. Вкладка Debug позволяет выбрать требуемую систему счисления в полях Numbers и Vectors. Numbers даёт возможность задавать систему счисления для численных значений, а Vectors – для одномерных векторов. Возможны варианты: Binary (двоичная), Octal (восьмеричная), Decimal (десятичная) и Hexadecimal (шестнадцатеричная) системы счисления. Система счисления, задаваемая в диалоговом окне Preferences, является глобальным параметром и оказывает влияние на представление значений в Watch, Call Stack и Design Browser окнах.

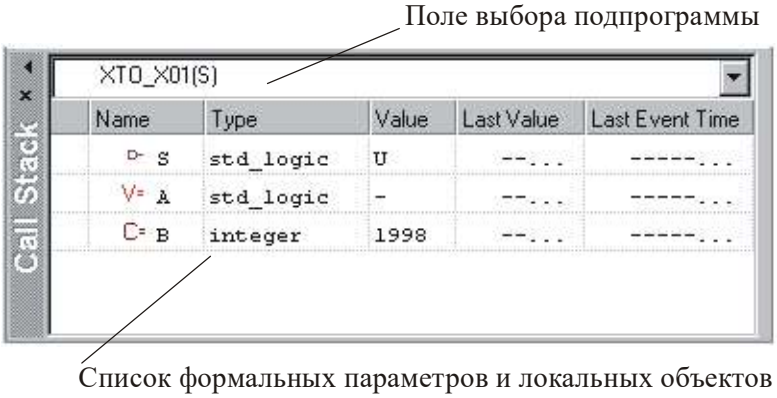
Окно Call Stack

Окно Call Stack (рисунок 3.40) является инструментом отладки, позволяющим просматривать список подпрограмм (процедур и функций), вызванных в выполняемом в данный момент процессе. Под словом процесс в данном случае подразумевается любой параллельный оператор, например: оператор process, параллельное назначение сигнала, параллельный оператор assert, параллельный вызов процедуры.

Для каждой подпрограммы в окне выводится информация: формальные параметры с реальными значениями, переменные, константы и файлы, являющиеся локальными для подпрограммы (они продекларированы в ней), вместе со своими значениями. Если в моделируемом проекте имеется более одного процесса, можно использовать окно Processes для выбора процесса, подпрограммы которого будут выведены в активном окне Call Stack.

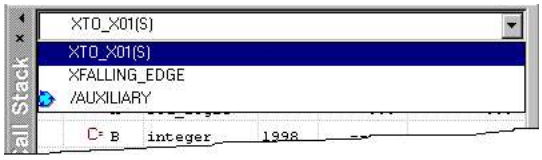
Последнее доступно только во время выполнения моделирования.

Рисунок 3.40. Окно Call Stack



Форма представления списка формальных параметров и локальных объектов такая же, как и в окне Watch.

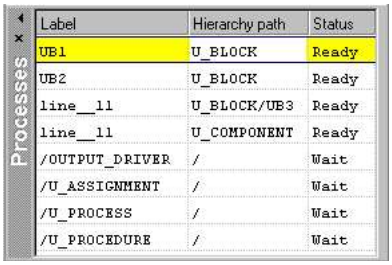
Щелчком по полю выбора подпрограмм открывается список вложенных подпрограмм текущего процесса. Имя процесса расположено внизу списка, а имя самой вложенной подпрограммы – вверху. После ее выбора будет открыто окно **HDL Editor**, содержащее исходный файл этой подпрограммы:



Окно Processes

Окно Processes является отладочным инструментом, отображающим процессы, которые входят в тестируемую модель. Окно доступно только после инициализации моделирования. Оно содержит три колонки, описывающие метку (label), путь (hierarchical path) и состояние или статус (status) каждого процесса (рисунок 3.41). Для процессов, не имеющих метки, компилятор генерирует псевдометку, представляющую собой номер строки, с которой начинается процесс в исходном коде, например, line__12.

Рисунок 3.41. Окно Processes



Процесс может находиться в одном из следующих состояний, которое выводится в колонке Status:

– *Ready(готов)* – означает, что процесс занесен в список (is scheduled) на выполнение во время текущего цикла моделирования. После этого процесс получает статус <*Wait*>.

– *Wait* (ожидает) – процесс ожидает изменения (for an HDL item to change) или окончания описанного периода времени.

Процессы выполняются в порядке, в котором они приведены в окне: сверху вниз. Желтая подсветка означает их выполнение в текущий момент. Порядок следования процессов можно изменить вручную.

Для выбранной области тестируемого проекта окно Processes может выводить или все процессы, независимо от их статуса в текущем цикле моделирования, или только активные в данном цикле. Выделить область для вывода процессов можно в подокне Structure tab окна Design Browser.

3.13. Active-HDL Macro Language

Active-HDL-макроязык предоставляет возможность работать в Active-HDL среде, не используя ее графический интерфейс. Макрокоманды могут быть введены непосредственно в окне **Console**. После того как будет нажата кнопка **Enter**, осуществляется выполнение команды. Для инициализации единичных команд такой способ работы не очень удобный. Но макрокоманды незаменимы, когда следует выполнить большую последовательность операций. В таком случае создается текстовый файл, содержащий команды – макрофайл или командный файл, который запускается на выполнение.

Для этого следует набрать команду в окне **Console**:

```
do <filename> [ <parameter_value> ...]
```

Параметр <filename> соответствует имени выполняемого макрофайла. Можно указать полный путь расположения файла. Если никакой путь не указан, по умолчанию подразумевается текущая рабочая директория Active-HDL.

Можно также ввести необязательные аргументы <parameter_value>, которые будут переданы в макрофайл через параметры $1...$99. Если в командной строке описано меньше параметров, чем в макрофайле, неописанные параметры будут заменены пустой строкой.

Пример вызова макрофайла может иметь следующий вид:

```
do $dsn\macrofilename.do
```

Следует обратить внимание, что оператор **do** является макрокомандой и может вызывать командный файл из аналогичного файла. В среде Active-HDL макрофайлы являются файлами ресурса, поэтому могут быть включены в проект как и любые другие ресурсы.

Для того чтобы запустить на выполнение макрофайл из окна Design Browser, необходимо выделить его на вкладке Files, щелкнуть правой кнопкой мыши и выбрать команду Execude из контекстного меню.

Разработка макрофайла

Макрофайл можно создать, используя любой текстовый редактор. Тем не менее, лучше использовать Active-HDL Editor, поскольку он поддерживает цветовое выделение ключевых слов макрокоманд.

Для создания нового командного файла следует выполнить команду File /New/ Macro или, щелкнув по стрелке кнопки  , выбрать из выпадающего меню пункт Macro. Будет открыто пустое окно HDL Editor, в которое вводится набор макрокоманд. Чтобы добавить созданный файл к проекту, можно использовать команду Add Current Document из меню Design. Если документ до этого момента не был сохранен на диске, появляется диалоговое окно Save As. Далее следует указать имя файла, тип и выбрать папку. Нажать OK.

Использование макрокоманд для моделирования

Рассмотрим макрокоманды, которые были использованы для моделирования 4-битного сумматора Adder (см. рисунок 2.4). Этот макрофайл (рисунок 3.42) инициализирует

процесс моделирования, открывает окно List, добавляет к нему наблюдаемые сигналы и выполняет процесс моделирования. Ниже описаны команды, использованные в командном файле.

Рисунок 3.42. Макрофайл для моделирования 4-битного сумматора

```
asim Adder4
list A B Co C Ci S
-- размещение сигналов в окне List
force A 1111
-- присвоение сигналу A значения "1111"
force B 0001
-- присвоение сигналу B значения "0001"
force Ci 1
-- установка Ci в '1'
run 50 ns
-- выполнение моделирования в течение 50 ns
force Ci 0
force A 0101
force B 1110
run 50 ns
```

Команда asim

Инициализирует моделирование.

Синтаксис:

```
asim [ -help ] [ -file <filename> ] [ -i <iteration_limit> ]
<configuration> | <entity> [ <architecture> ]
```

Аргументы:

-help – вывод короткого описания синтаксиса команды;

-file <filename> – описывает необязательный командный файл, содержащий аргументы для команды asim. Эти аргументы можно использовать вместо того, чтобы непосредственно вводить их в команде;

-i <iteration_limit> – устанавливает максимальное число дельта-итераций, которые могут быть выполнены в одно и то же время моделирования. Это позволяет избежать бесконечных циклов;

<configuration> – задает имя конфигурации верхнего уровня для моделирования;

<entity> – имя интерфейса верхнего уровня для моделирования;

<architecture> – имя архитектуры для моделирования. Если имя не указано, будет использована последняя откомпилированная архитектура для данного интерфейса. Этот аргумент может использоваться только с интерфейсом.

Команда list

Открывает окно List или добавляет сигналы к существующему окну.

Синтаксис:

```
list [-in] [-out] [-inout] [-internal] [-ports] [-signals]
     [ -collapse ] [ -<radix> ][ -width <n> <item_name> ... ]
```

Аргументы:

-in – команда влияет только на порты в режиме IN;

-out – команда влияет только на порты в режиме OUT;

-inout – команда влияет только на порты в режиме INOUT;

-internal – команда влияет только на внутренние объекты;

-ports – команда влияет только на порты;

-signals – команда влияет только на сигналы;

-collapse – переключатель между промежуточными и результирующими значениями, которые выводятся на экран. Результирующим считается значение последнего цикла моделирования в описанном времени моделирования. Все значения из промежуточных циклов считаются промежуточными;

-<radix> – задает систему счисления для сигналов, описанных в команде. Доступные опции:
– binary (abbr. **bin**) – двоичная;
– octal (abbr. **oct**) – восьмеричная;
– decimal (abbr. **dec**) – десятичная;
– **hex –** шестнадцатеричная;

-width <n> – параметр “n” задает ширину колонки;

< item_name>– имя сигнала для размещения в окне List.

Команда force

Присваивает значение или последовательность значений сигналу.

Синтаксис:

```
force [ -rec | -recursive ][ -repeat <period> ]
      <signal_name> <value> [ <time> ] [ , <value> <time> ... ]
```

Аргументы:

-rec | -recursive – поиск описанного сигнала по всему проекту. Если аргумент не указан, команда применяется только к текущей области проекта;

-repeat <period> – повторяет последовательность описанных значений с указанным периодом;

<signal_name> – имя сигнала. Можно указывать элементы массива, диапазон и весь массив целиком;

<value> – значение, которое получит сигнал. Тип значения должен соответствовать типу сигнала.

Одноэлементные массивы типа BIT, STD_ULOGIC и их подтипов могут быть описаны как последовательность символов (01XZ011Z) или как число с указанием системы счисления 2, 8, 10 или 16.

Например, сигналу типа BIT_VECTOR (3 downto 0) можно назначить следующие значения:

```
1011    символьная последовательность
2#1011  двоичная система счисления
10#11   десятичная система счисления
16#B    шестнадцатеричная система счисления
```

<time> – время, в которое сигнал должен получить значение. Это время является относительным для текущего цикла моделирования. Символ @ перед временным параметром обозначает абсолютное время.

Команда run

Выполняет моделирование.

Синтаксис:

```
run [ <time_step> | @<time> | -all | -next ]
```

Аргументы:

<time_step> – промежуток времени для моделирования;

@<time> – абсолютное время, до которого будет осуществляться моделирование;

-all – выполняются все возможные шаги моделирования, пока не будет обработано последнее событие из очередей драйверов;

-next – выполнение до следующей записи в очередь драйвера.

Команда wave

Вместо команды **list** или вместе с ней можно использовать команду **wave**, которая позволяет отображать результаты моделирования в виде диаграмм в окне Wave.

Команда добавляет указанные сигналы в окно Wave.

Синтаксис:

```
wave [-in] [-out] [-inout] [-internal] [-ports] [-signals][-<radix>]
     [ -<format> ] [ -height <pixels> ][ -color <red_value,
     green_value, blue_value> ] [ <item_name> ] ... ] ...
```

Аргументы:

-<format> – необязательный параметр, задающий один из следующих типов:

– literal **(abbr.** li**) –** символьный;

– logic **(abbr.** lo**) –** логический;

– analog – аналоговый.

Символьная waveform представляет собой прямоугольники, в которые вписаны значения. Логические сигналы могут быть либо 1, 0, X или Z.

-height <pixels> – высота waveform в пикселах;

-color <red_value, green_value, blue_value> – цвет waveform, описываемый тремя параметрами с диапазоном значений 0-255, соответствующими красной, зеленой и синей составляющей цвета (RGB-модель цвета);

<item_name> – имя сигнала.

Аргументы -in, -out, -inout, -internal, -ports, -signals, -<radix> описаны в команде list.

Команда close

Закрывает указанное окно документа.

Синтаксис:

```
close -wave | -list | -hde | -fsm | -bde
```

Аргументы:

-wave – окно **Waveform**;

-list – окно **List**;

-hde – окно **HDL Editor**;

-fsm – окно **State Editor**;

-bde – окно **Block Diagram Editor**.

Среда проектирования Active-HDL фирмы Aldec – это мощный программный пакет, позволяющий создавать и управлять проектами цифровых устройств. Окно Design Browser предоставляет инструменты для управления файлами проекта, облегчающие процесс их создания, просмотра, копирования, удаления. Существуют мастеры для создания новых проектов и генерации шаблонов исходных файлов. Окно Language Assistant содержит стандартные шаблоны VHDL-конструкций и VHDL-модели типовых элементов. Среда Active-HDL реализует возможность моделирования поведения разрабатываемых проектов цифровых устройств. Для просмотра результатов моделирования создано несколько инструментов, облегчающих наблюдение за работой тестируемой модели. Это окна List, Waveform, Watch, Call Stack, Processes.

Макрокоманды позволяют работать со средой Active-HDL без использования ее графического интерфейса.

3.14. Задачи

3.14.1. Разработать проект, состоящий из одного VHDL-файла, созданного в задаче 2.15.1. Откомпилировать. Промоделировать, использовав для вывода результатов окно Waveform. Для входных сигналов назначить формульные стимуляторы, так чтобы выполнялся перебор всех 16 входных последовательностей. Будьте внимательны с определением задержек переключения входных сигналов.

3.14.2. Для устройства, разработанного в задаче 2.15.2, создать проект. Выполнить моделирование полного и 4-разрядного устройства вычитания. Для инициализации моделирования, присвоения входных значений и вывода результатов моделирования разработать и использовать макрофайл.

3.14.3. В среде Active-HDL создать проект и выполнить тестирование модели, разработанной в задаче 2.15.9.

3.14.4. В среде Active-HDL создать проект и выполнить тестирование модели, разработанной в задаче 2.15.10.

3.14.5. В среде Active-HDL создать проект и выполнить тестирование модели, разработанной в задаче 2.15.11.

3.14.6. В среде Active-HDL создать проект и выполнить тестирование модели, разработанной в задаче 2.15.12.

3.14.7. В среде Active-HDL создать проект и выполнить тестирование модели, разработанной в задаче 2.15.13.

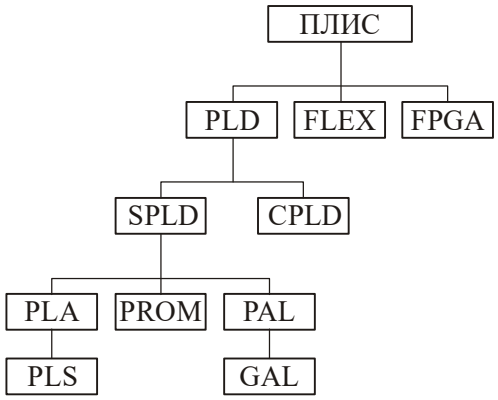
3.14.8. Протестировать функцию, разработанную в задаче 2.15.17. Для этого создать VHDL-модель устройства, в котором бы использовалась эта функция. Для наблюдения за значениями формальных параметров и локальных объектов функции использовать окно Call Stack.

3.14.9. Протестировать функцию, разработанную в задаче 2.15.18. Для этого создать VHDL-модель устройства, в котором бы использовалась эта функция. Для наблюдения за значениями формальных параметров и локальных объектов функции использовать окно Call Stack.

ГЛАВА 4
СТАНДАРТНЫЕ ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ УСТРОЙСТВА

На рынке микроэлектронных технологий все более популярными становятся программируемые логические интегральные схемы (ПЛИС). Реализация цифровых устройств на ПЛИС достойно конкурирует с проектами на основе использования базовых матричных кристаллов (БМК), сигнальных процессоров. Такой успех определяется: применением новых технологий разработки и реализации цифровых систем на основе использования Hardware-Software Cooperation-Design; уменьшением времени проектирования, верификации и изготовления сложного вычислительного устройства до 4-5 месяцев; высоким быстродействием выполнения операций в ПЛИС (до 500 МГц); большой степенью интеграции элементов на кристалле (до 10 млн). Общая классификация этих устройств приведена на рисунке 4.1. Согласно ей устройства программируемой логики принято делить на: программируемые логические устройства (ПЛУ– programmable logic device – PLD), программируемые пользователем вентильные матрицы (Field Programmable Gate Array – FPGA) и FLEX. Программируемые логические устройства, в свою очередь, делятся на стандартные (Standart PLD) или классические (Classic PLD) и сложные ПЛУ (Complex PLD – CPLD) (см. рисунок 4.1). К классу стандартных ПЛУ относятся ППЗУ (PROM), ПЛМ (PLA), ПМЛ (PAL).

Рисунок 4.1. Классификация ПЛИС



PLD (Programmable Logic Device) – программируемое логическое устройство; FLEX (Flexible Logic Element Matrix) – матрица элементов гибкой логики; FPGA (Field Programmable Gate Array) – программируемая пользователем вентильная матрица; SPLD (Standart PLD или Classic PLD) – стандартное или классическое программируемое логическое устройство; CPLD (Complex PLD) – сложное программируемое логическое устройство; PLA (Programmable Logic Array) – программируемая логическая матрица; PROM (Programmable Read Only Memory) – программируемое постоянное запоминающее устройство; PAL (Programmable Array Logic) – программируемая матричная логика; PLS (Programmable Logic Sequencer) – программируемый логический секвенсер; GAL (Generic Array Logic) – типовая или обобщенная матричная логика

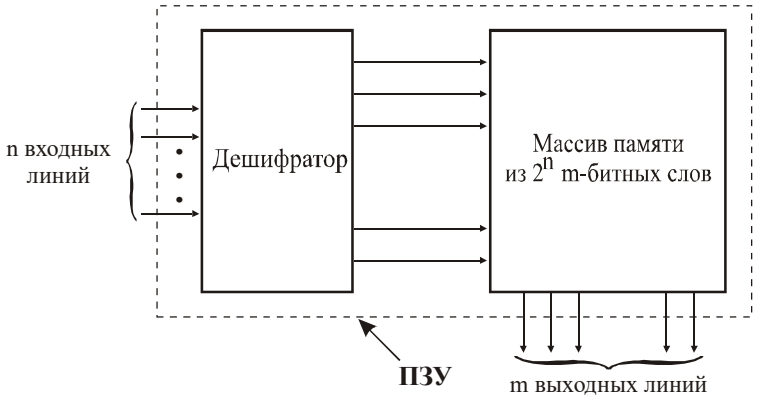
4.1. Постоянные запоминающие устройства (ПЗУ)

ПЗУ – это полупроводниковые матрицы, предназначенные для хранения массивов двоичной информации. Такие данные можно читать, но нельзя изменять в процессе эксплуатации. ПЗУ с n входами и m выходами может содержать 2^n m-битных слов (рисунок 4.2). Входные линии являются адресными для выбора одного из 2^n слов. Теоретически ПЗУ можно представить устройством, состоящим из дешифратора и массива памяти. Когда двоичный код подается на входы дешифратора, активным, равным 1, будет только один из его выходов, который выбирает одно из слов массива

памяти, поступающих на выходы ПЗУ. С помощью ПЗУ размерностью (2^n x m) можно реализовать m функций от n переменных, поскольку в нем можно сохранить таблицу истинности, состоящую из n строк и m столбцов.

По способу записи информации ПЗУ можно разделить на устройства с масочной и электрической записью. В первом случае информация записывается один раз при производстве микросхемы. Это достигается путем селективного создания переключающих элементов на пересечениях строк и столбцов массива памяти. Для этого разрабатываются специальные маски, которые используются на стадии производства микросхемы. Поскольку изготовление масок – дорогостоящий процесс, их применение оправдано при создании большого числа (несколько тысяч) ПЗУ с одинаковой информацией. Если требуется небольшое количество одинаковых микросхем, лучше использовать ПЗУ с электрической записью, позволяющей перезапись информации.

Рисунок 4.2. Структура ПЗУ



Изменение данных, хранящихся в ПЗУ, часто бывает необходимо на этапе создания цифровых систем. В таких случаях используются перезаписываемые ПЗУ вместо устройств с масочной записью. Для разрешения или запрещения переключательных элементов в матрице памяти таких ПЗУ применяется специальный сохраняющий заряд механизм. Программирование перезаписываемых ПЗУ выполняется с помощью импульсов напряжения, создающих электрические заряды в определенных местах матрицы памяти. Данные, сохраненные таким способом, могут быть стерты с помощью ультрафиолетового излучения. После этого в ПЗУ может быть записана другая информация. В последнее время стали очень популярны электрически стираемые ПЗУ. Их отличие от предыдущих в том, что сохраненные данные можно стирать, используя электрические импульсы вместо ультрафиолетовых лучей. Такие ПЗУ можно перепрограммировать ограниченное число раз, обычно от 100 до 1000. Следующий тип перезаписываемых ПЗУ – flash, использует иной механизм сохранения зарядов. Такие микросхемы обычно имеют встроенный механизм для программирования и стирания информации, поэтому данные во flash-памяти могут быть сохранены непосредственно в схеме, без внешнего программатора.

Используя ПЗУ и триггеры, можно легко построить любую последовательностную схему. В структурной модели устройства, заданного автоматом Мили (см. рисунок 2.15), комбинационная часть может быть реализована на ПЗУ. Таким образом, с помощью ПЗУ можно реализовать функции переходов и выходов. Для хранения текущего состояния могут быть использованы D-триггеры, сигналы с которых подаются на входы ПЗУ. Применение D-триггеров предпочтительнее, чем JK. Применение 2-входовых триггеров увеличивает в два раза число необходимых выходных линий ПЗУ. То, что уравнения для D-триггеров более сложные и требуют большее число вентилей для реализации, в данном случае не имеет значения, поскольку размер ПЗУ определяется числом входов и выходов, а не сложностью реализуемых уравнений. По

этой же причине не столь важно и кодирование состояний, которое выполняется двоичными кодами в любом порядке.

На рисунке 4.3 представлена реализация автомата, изображенного на рисунке 2.15. Таблица 4.1 содержит запись переходов и выходов автомата, а таблица 4.2 – полученную на ее основе таблицу истинности для ПЗУ, в которой символ "X" заменен на "0". Поскольку ПЗУ имеет 4 входа, в нем содержится 2^4=16 слов. Обычно автомат с i входами, j выходами и k состояниями может быть реализован с использованием k D-триггеров и ПЗУ с i+k входами (2^{i+k} слов) и j+k выходами.

Рисунок 4.3. Реализация автомата Мили с использованием ПЗУ

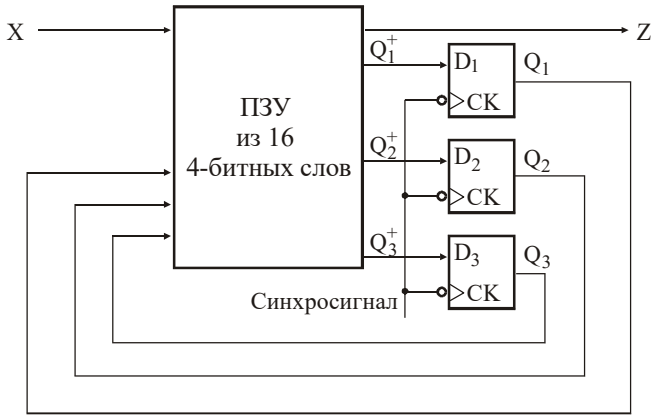


Таблица 4.1. Таблица переходов и выходов автомата

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$		Z	
	X=0	X=1	X=0	X=1
000	100	101	1	0
100	111	110	1	0
101	110	110	0	1
111	011	011	0	1
110	011	010	1	0
011	000	000	0	1
010	000	xxx	1	x
001	xxx	xxx	x	x

Таблица 4.2. Таблица истинности ПЗУ

Q_1	Q_2	Q_3	X	Q_1+	Q_2+	Q_3+	Z
0	0	0	0	1	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	0	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	0	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	0	1	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	1	0
1	1	1	1	0	1	1	1

VHDL-код для реализации автомата на ПЗУ (рисунок 4.4) аналогичен рисунку 2.18, за исключением процедуры вычисления следующего состояния и выходных

значений по коду из ПЗУ. Регистр текущего состояния задан сигналом Q, описанным бит-вектором (Q_1, Q_2, Q_3). Регистр следующего состояния – Qplus. На языке VHDL ПЗУ можно представить в виде константы, типа одномерный массив из бит-векторов. В данном случае – это шестнадцать 4-битных слов, а константа имеет имя FSM_ROM. Входами FSM_ROM является Q&X. Поскольку индекс массива имеет тип integer, функция vec2int используется для преобразования Q&X в этот тип данных. Переменная ROMValue соответствует выходам ПЗУ. ROMValue делится на Qplus и Z. Регистр состояний Q обновляется по заднему фронту синхроимпульса.

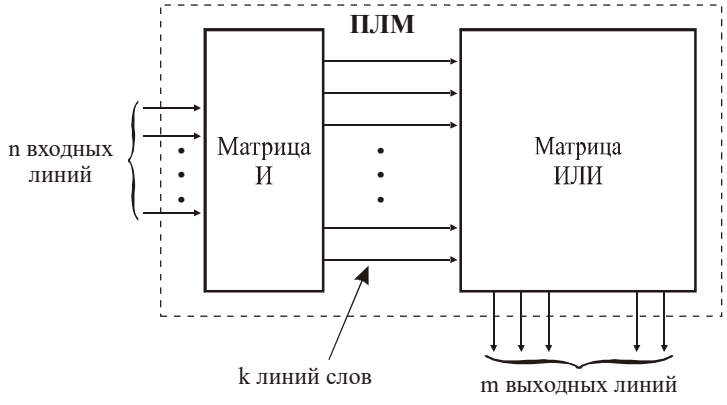
Рисунок 4.4. Реализация устройства на ПЗУ

```
library BITLI;
use BITLIB.bit_pack.all;
entity ROM1_2 is
   port(X,CLK: in bit;
        Z: out bit) ;
end ROM1_2;
architecture ROM1 of ROM1_2 is
signal Q, Qplus: bit_vector(1 to 3) := "000";
type ROM is array (0 to 15) of bit_vector(3 downto 0);
constant FSM_ROM:  ROM :=
              ("1001","1010","0000","0000",
               "0001","0000","0000","0001",
               "1111","1100","1100","1101",
               "0111","0100","0110","0111");
begin
   process(Q,X)          -- determines the next state and output
   variable ROMValue: bit_vector(3 downto 0);
   begin
      ROMValue := FSM_ROM(vec2int(Q & X));  -- read ROM output
      Qplus <= ROMValue(3 downto 1);
      Z <= ROMValue(0);
   end process;
process(CLK)
   begin
      if CLK='1' then Q <= Qplus; end if;   -- update state register
   end process;
end ROM1 ;
```

4.2. Программируемые логические матрицы (ПЛМ)

ПЛМ выполняют те же функции, что и ПЗУ. На них (рисунок 4.5) можно реализовать m функций от n переменных. По внутренней структуре ПЛМ отличаются от ПЗУ. Дешифратор заменяет матрица И, реализующая выбор конъюнктивных термов, из которых в массиве ИЛИ формируются выходные функции.

Рисунок 4.5. Структурная схема ПЛМ



На рисунке 4.6 представлена схема n-МОП ПЛМ, реализующая функции:

$$F_0 = \sum m(0,1,4,6) = A'B' + AC',$$
$$F_1 = \sum m(2,3,4,6,7) = B + AC', \qquad (4.1)$$
$$F_2 = \sum m(0,1,2,6) = A'B' + BC',$$
$$F_3 = \sum m(2,3,5,6,7) = AC + B.$$

Внутри ПЛМ используется ИЛИ-НЕ–ИЛИ-НЕ логика, но добавленные на входы и выходы инвертирующие буферы делают ее эквивалентной И-ИЛИ логике. Логические элементы формируются n-МОП коммутирующими транзисторами на пересечении горизонтальных и вертикальных линий. На рисунке 4.7 представлена схема двухвходового ИЛИ-НЕ вентиля. Транзистор функционирует как переключатель. Если на входе вентиля находится логический 0, то транзистор закрыт. Если на входе 1 – транзистор открыт и пропускает ток на землю. Если $X_1 = X_2 = 0$, то оба транзистора закрыты и напряжение сопротивления создает на выходе Z уровень логической 1 (+V). Если X_1 или X_2 равняется 1, соответствующий транзистор открыт и Z = 0. Таким îáðàçîì , $Z=(X_1+X_2)' = X_1'X_2'$, что соответствует ИЛИ-НЕ вентилю. Фрагмент ПЛМ, реализующий функцию F_0 и эквивалентный схеме из ИЛИ-НЕ вентилей, представлен на рисунке 4.8. Сократив дополнительные инверторы, можно получить И-ИЛИ структуру. Матрица И-ИЛИ, изображенная на рисунке 4.9, соответствует ПЛМ с рисунка 4.6.

Рисунок 4.6. ПЛМ с 3 входами, 5 термами и 4 выходами

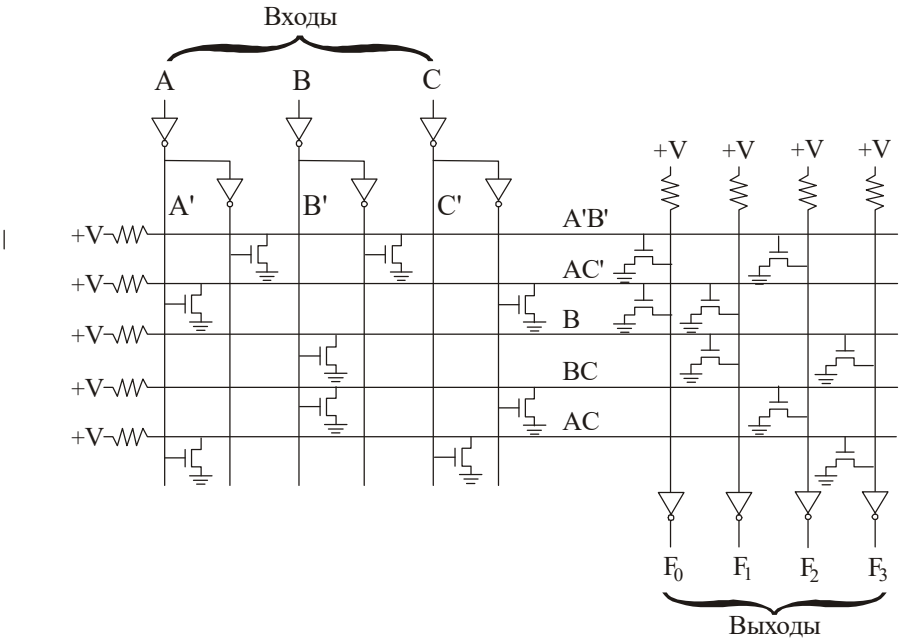


Рисунок 4.7. nМОП ИЛИ-НЕ элемент

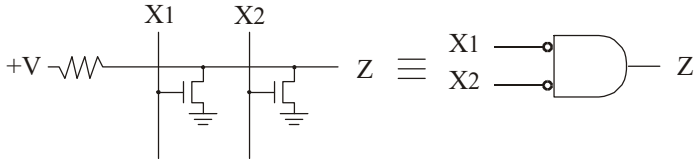


Рисунок 4.8. Преобразование ИЛИНЕ-ИЛИНЕ логики в И-ИЛИ

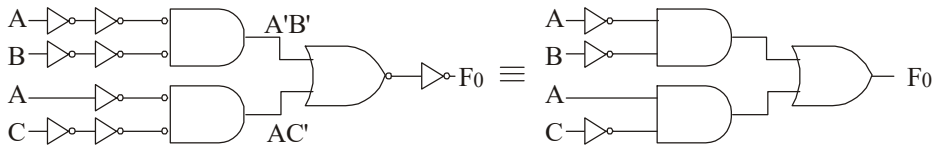
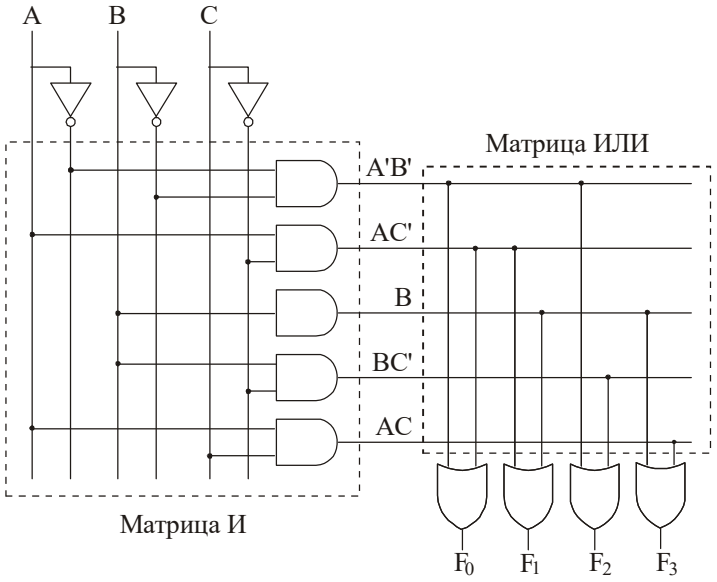


Рисунок 4.9. И-ИЛИ матрица – структурная схема ПЛМ



Содержание ПЛМ может быть описано с помощью модифицированной таблицы истинности. Таблица 4.3 отражает содержимое ПЛМ (см. рисунок 4.6). Первая колонка описывает термы произведения. Символы "0", "1" или "-" обозначают инверсию переменной, ее отсутствие или несущественность в образовании данного терма. В последней колонке указаны термы, участвующие в формировании функций. Символы "0" или "1" обозначают представление терма или его отсутствие в соответствующей функции выходов. Таким образом, в первой строке таблицы записано, что терм A'B' принадлежит функциям F_0 и F_2 , во второй – терм AC' принадлежит F_0 и F_1.

Таблица 4.3. Таблица истинности для структуры ПЛМ

Термы произвед.	Входы			Выходы			
	A	B	C	F_0	F_1	F_2	F_3
A'B'	0	0	-	1	0	1	0
AC'	1	-	0	1	1	0	0
B	-	1	-	0	1	0	1
BC'	-	1	0	0	0	1	0
AC	1	-	1	0	0	0	1

Используя ПЛМ, реализовать функции:

$$F_1 = \sum m(2,3,5,7,8,9,10,11,13,15),$$
$$F_2 = \sum m(2,3,5,6,7,10,11,14,15), \qquad (4.2)$$
$$F_2 = \sum m(6,7,8,9,13,14,15).$$

Минимизировав каждую функцию отдельно, можно получить:

$$F_1=bd+b'c+ab', \qquad (4.3)$$
$$F_2=c+a'bd,$$
$$F_3=bc+ab'c'+abd.$$

Если реализовать эти уравнения на ПЛМ, потребуется 8 термов произведения.

Вместо того, чтобы минимизировать каждую функцию в отдельности, можно минимизировать общее число строк в таблице ПЛМ. В данном случае неважно, сколько термов содержится в каждом уравнении: размер ПЛМ от этого не зависит. Карты Карно, соответствующие уравнениям функций, представлены на рисунке 4.10.

Поскольку терм ab'c' необходим для функции F_3, его можно использовать и в F_1 вместо ab', а две другие единицы из терма ab'c' преобразуются в терм b'c. Это исключает строку ab' из таблицы для ПЛМ. Аналогичным образом a'bd и abd из функций F_2 и F_3 могут заменить терм bd из F_1: a'bd + abd. Это позволяет убрать строку bd из таблицы. И, наконец, терм *c* из функции F_2 реализуется термами b'c и bc из F_1 и F_3: b'c + bc. Ниже представлены полученные уравнения (4.4) и сокращенная таблица 4.4 для ПЛМ. Существуют и другие алгоритмы для минимизации строк таблицы.

Рисунок 4.10. Многовыходные карты Карно

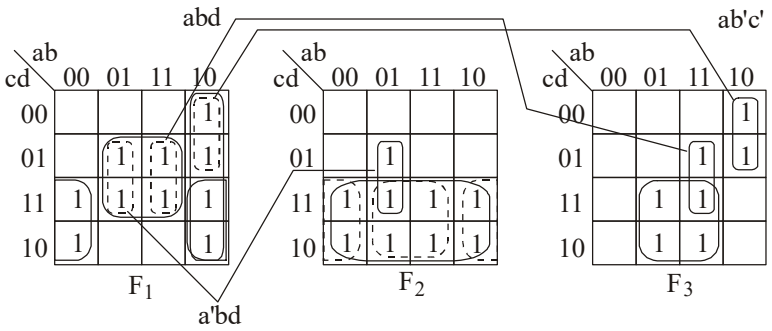


Таблица 4.4. Минимизированная таблица ПЛМ

a	b	c	d	F_1	F_2	F_3
0	1	-	1	1	1	0
1	1	-	1	1	0	1
1	0	0	-	1	0	1
-	0	1	-	1	1	0
-	1	1		0	0	1

```
F1 = a'bd + abd + ab'c' + b'c,                    (4.4)
F2 = a'bd + b'c + bc,
F3 = abd + ab'c' + be.
```

Уравнения (4.4) включают только 5 различных термов, а таблица ПЛМ – 5 строк, что значительно лучше по сравнению с 8 термами уравнений (4.3). На рисунке 4.11 представлена соответствующая ПЛМ структура, имеющая 4 входа, 5 термов и 3 выхода. Точка на пересечении линий означает присутствие переключательного элемента в матрице.

Рисунок 4.11. Схема ПЛМ, реализующей уравнения (4.4)

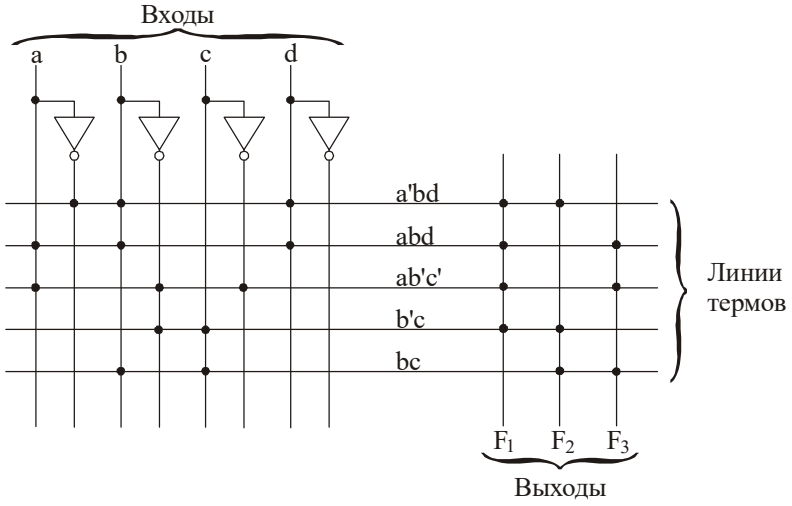


Таблица для ПЛМ значительно отличается от таблицы истинности ПЗУ. В последней каждая строка соответствует минтерму, поэтому только одна строка будет

выбрана любой комбинацией входных сигналов. Эти 0 и/или 1 из колонки выходов будут определять соответствующие значения выходов. В ПЛМ таблице, наоборот, каждая строка соответствует простой импликанте функции (минимизированный терм). Для того чтобы определить значение F для данной входной комбинации, выбранные строки из ПЛМ таблицы должны быть объединены по ИЛИ. Следующий пример использует ПЛМ, представленную в таблице 4.4. Если abed =0001, ни одна из строк не будет выбрана и все функции равны 0. Если abed = 1001, будет выбрана только третья строка и $F_1F_2F_3 = 101$. Если abed = 0111, – первая и пятая строки, поэтому $F_1 = 1+0=1$, $F_2 = 1+1=1$, $F_3 = 0+1=1$.

С помощью ПЛМ и трех D-триггеров можно реализовать автомат Мили (см. рисунок 2.15). Схема устройства аналогична схеме с ПЗУ (см. рисунок 4.3), за исключением того, что ПЗУ заменяется ПЛМ. Таблица 4.5 представляет ПЛМ, соответствующую уравнениям, поданным на рисунке 4.12.

Рисунок 4.12. Уравнения состояний и выхода автомата

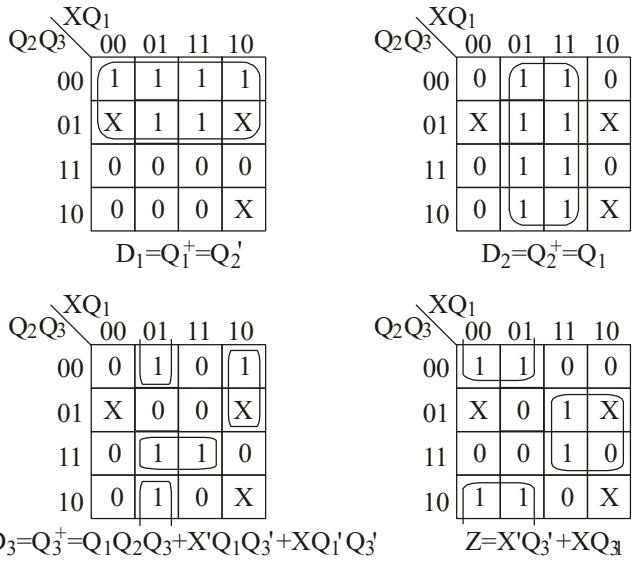


Таблица 4.5. Описание структуры ПЛМ

Термы	Q1	Q2	Q3	X	Q1+	Q2+	Q3+	Z
Q2'	-	0	-	-	1	0	0	0
Q1	1	-	-	-	0	1	0	0
Q1Q2Q3	1	1	1	-	0	0	1	0
Q1Q3'X'	1	-	0	0	0	0	1	0
Q1'Q2'X'	0	0	-	1	0	0	1	0
Q3'X'	-	-	0	0	0	0	0	1
Q3X	-	-	1	1	0	0	0	1

На рисунке 4.13 представлено VHDL описание реализации автомата на ПЛМ. Функцию считывания информации с выходов для ПЛМ описывать сложнее, чем с выходов ПЗУ. Поскольку входы ПЛМ могут включать несколько строк, которые затем будут соединены по ИЛИ, используется функция, выполняющая последовательное сканирование ПЛМ матрицы и определяющяя значение выхода. В коде VHDL ПЛМ описывается двумерным массивом с именем типа PLAmtrx и константами FSM_PLA. Функция PLAout вызывается для определения значений выходов FSM_PLA, зависимых от входов Q&X, которые присваиваются переменной PLAValue. После разделения PLAValue на Qplus и Z значение сигнала Q обновляется по переднему фронту синхроимпульса.

Поскольку в ПЛМ таблице используется символ "-", для ее описания необходима многозначная логика. Можно определить VHDL тип с элементами '1', '0', '-', однако предпочтительнее использовать предопределенные типы. Поэтому применяется тип std_logic, определенный в библиотеке IEEE. Тип std_logic имеет 9 значений, включая '0', '1', 'X'. При преобразовании ПЛМ таблицы в VHDL-код вместо символа "-" используется символ 'X'. Описания типа *PLAmtrx* и функции *PLAout* из пакета *MVLLIB* (multivalued logic library) созданы с помощью пакета IEEE standard logic.

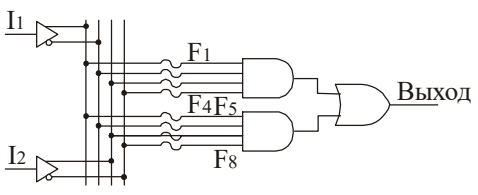
Рисунок 4.13. Реализация автомата на ПЛМ

```
library ieee;
use ieee.std_logic_1164.all;     -- Пакет IEEE standard logic
library MVLLIB;                  --  подключение типа PLAmtrx type и
use MVLLIB.mvl_pack.all;         --  функции PLAout
entity PLA1_2 is
   port(X,CLK: in std_logic;
        Z: out std_logic);
end PLA1_2 ;
architecture PLA of PLA1_2 is
signal Q, Qplus: std_logic_vector(1 to 3) := "000";
constant FSM_PLA: PLAmtrx (0 to 6, 7 downto 0) :=
        ("X0XX1000", "1XXX0100", "111X0010",
        "1X000010", "00X10010",
        "XX000001", "XX110001");
begin
   process(Q,X)
   variable PLAValue: std_logic_vector(3 downto 0);
   begin
      PLAValue := PLAout(FSM_PLA,Q & X);  -- чтение значений выходов
ПЛМ
      Qplus <= PLAValue(3 downto 1);
      Z <= PLAValue(0) ;
   end process;

   process(CLK)
   begin
      if CLK='1' then Q <= Qplus; end if; -- обновление состояния реги-
стра
   end process;
end PLA;
```

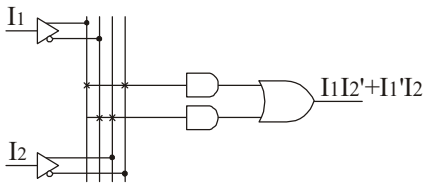
4.3. Программируемая матричная логика (ПМЛ)

ПМЛ или PAL (Programmable Array Logic) — это особый вид программируемых логических матриц, в которых массив И является программируемым, а ИЛИ – нет. Структурная схема ПМЛ аналогична ПЛМ, представленной на рисунке 4.5. Поскольку в ПМЛ программируется только один массив И, они дешевле ПЛМ и легче в программировании. По этой причине проектировщики чаще используют ПМЛ вместо отдельных логических элементов или ПЛМ, когда необходимо реализовать несколько функций.

Рисунок 4.14. Фрагмент ПМЛ: а - незапрограммированный; б - запрограммированный

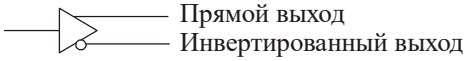


а

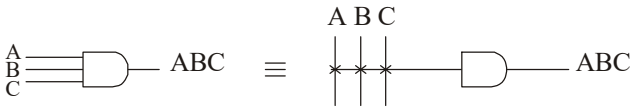


б

На рисунке 4.14, а представлен фрагмент незапрограммированной ПМЛ. Символ



соответствует входному буферу с инверсным и неинверсным выходами. Буфер необходим для подключения каждого входа к большому количеству вентилей И. В момент программирования ПМЛ связи $F_1, F_2 ..., F_8$ выборочно пережигаются и остаются только необходимые для вентилей И входы. Подключенные входы вентиля И обозначаются крестиками ($\times$), как показано ниже:



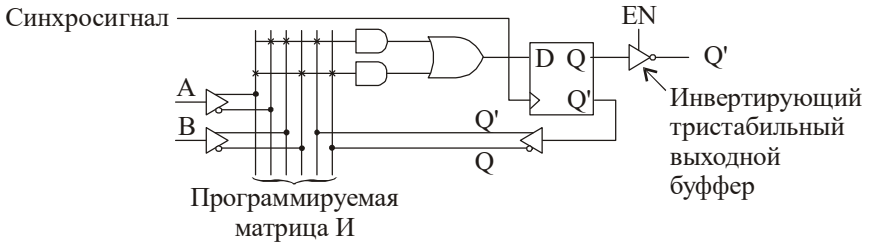
В качестве примера реализована функция $I_1I_2'+I_1'I_2$ на фрагменте ПМЛ, указанного на рисунке 4.14, а. Символ $\times$ обозначает подключение линий I_1 и I_2' к первому И-элементу, I_1' и I_2 – ко второму (рисунок 4.14, б).

Обычно комбинационные ПМЛ имеют 10 - 20 входов, 2 - 10 выходов и от 2 до 8 И-элементов, подключенных к каждому ИЛИ. Например, в устройстве PAL16L8 буква L обозначает комбинационную ПМЛ, которая имеет 16 входов для матрицы И (10 внешних и 6 – по обратной связи) и 8 выходов. ПМЛ может содержать также D-триггеры со входами, управляемыми программируемой матричной логикой. Такие ПМЛ удобно использовать для создания последовательностных схем. Обозначаются они буквой R. На рисунке 4.15 представлен фрагмент последовательностной ПМЛ. Вход D-триггера соединен с выходом вентиля ИЛИ, который в свою очередь подключен к двум элементам И. Сигналы с выхода триггера поступают через буфер обратно на программируемую матрицу И. Таким образом, входы вентиля И могут быть соединены с *A, A', B, B', Q* или *Q'* линиями. Символами $\times$ обозначена реализация следующего уравнения переходов:

$$Q^+=D=A'BQ'+AB'Q.$$

Выход триггера соединен с тристабильным инвертирующим буфером, который открывается, когда EN=1.

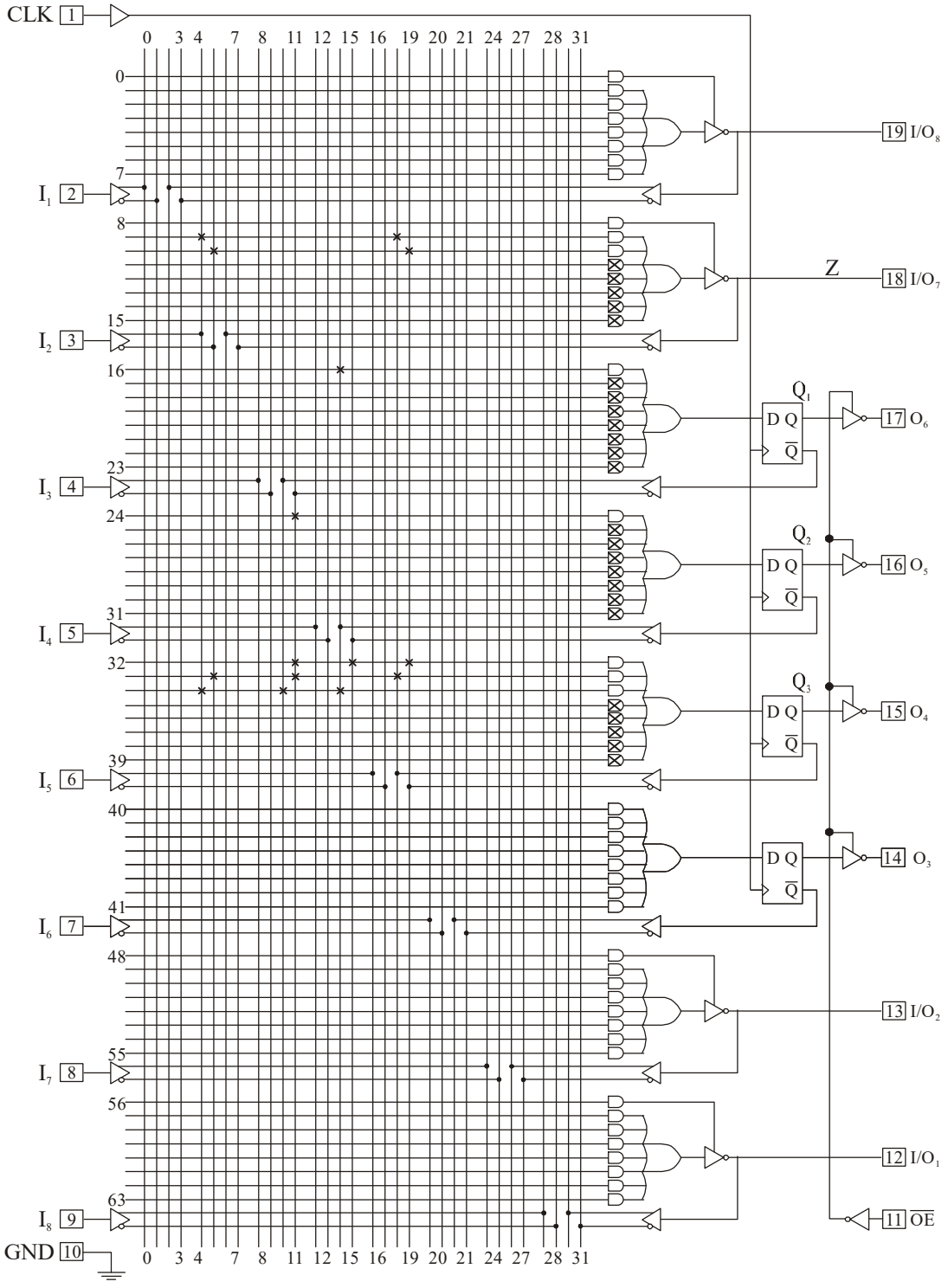
Рисунок 4.15. Фрагмент последовательностной ПМЛ



На рисунке 4.16 представлена схема типичной последовательностной ПМЛ 16R4, которая имеет матрицу вентилей И с 16 входными переменными и 4 D-триггера.Выходы триггеров соединены с тристабильными буферами – выходные контакты 14-17. Вход 11 используется для управления этими буферами. Значения триггеров изменяются по переднему фронту синхроимпульса. Их входы соединены с выходом элементов

ИЛИ. К каждому ИЛИ-элементу подключено 8 вентилей И. Сигналы на них могут подаваться с внешних входов (контакты 2-9) или по обратной связи с выходов триггеров. Дополнительные 4 вход-выходных контакта – 12, 13, 18 и 19 – могут быть использованы как выходы схемы либо как входы вентилей И. Поэтому каждый из них может иметь до 16 входов: 8 внешних, 4 входа с выходов триггеров и 4 – с вход-выходных контактов. Если вход-выходной контакт используется как выход, он управляется тристабильным инвертирующим буфером. Сигнал на буфер поступает с ИЛИ-вентиля, соединенного с семью И-элементами. Восьмой И-элемент используется для управления буфером.

Рисунок 4.16. Логическая схема для 16R4 ПМЛ



Если на 16R4 ПМЛ реализуется последовательностная схема, вход-выходные контакты используются для Z выходов. Таким образом, на одной микросхеме 16R4 без дополнительных элементов может быть реализован автомат, имеющий 8 входов, 4 выхода и 16 состояний. Каждое уравнение состояний может содержать до 8 термов, а уравнение функций – до 7. В качестве примера реализован преобразователь кода (рисунок 4.17). Для хранения состояния используются три триггера Q_1, Q_2, Q_3 , а матричная логика запрограммирована для реализации функций D_1, D_2 и D_3 , что показано на рисунке 4.16. Символ $\times$ обозначает связи с входами И-вентилей. Символ $\times$ внутри вентиля указывает на то, что он не используется. Для реализации функции D_3 необходимо три И-вентиля:

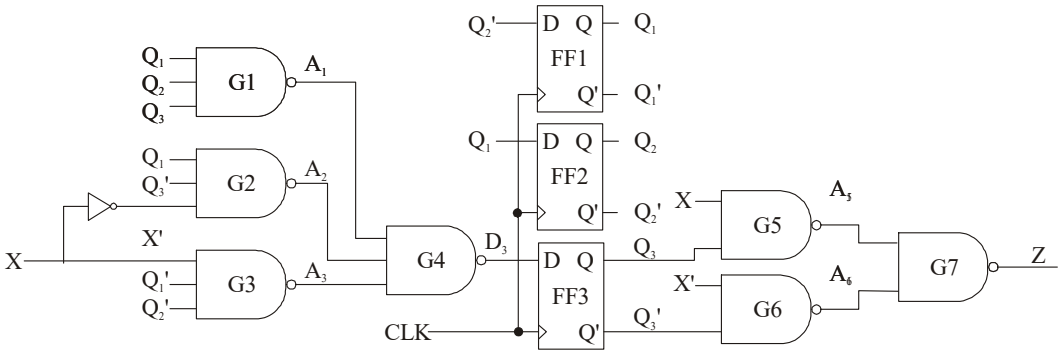
$$D_3=Q_1Q_2Q_3+X'Q_1Q_3'+XQ_1'Q_2'.$$

Выходы триггеров не применяются, поэтому выходные буферы закрыты. Поскольку Z выход проходит через инвертирующий буфер, матричная логика должна реализовывать функцию:

$$Z'=(X+Q_3)(X'+Q_3') = XQ_3'+X'Q_3.$$

В данном примере выходной буфер Z постоянно открыт, поэтому нет соединений для входов И-вентиля, управляющего буфером и на его выходе – логическая 1.

Рисунок 4.17. Схема преобразователя кода



При проектировании на основе ПМЛ необходимо упрощать логические уравнения, чтобы их можно было реализовать с использованием одной или нескольких существующих ПМЛ. В отличие от универсальных здесь И-вентиль не может быть общим для нескольких ИЛИ, поэтому каждую функцию можно минимизировать отдельно, не учитывая общее число термов. Для данного типа ПМЛ число И-термов, подключенных к И-вентилю, фиксировано и ограничено. Если число И-термов в упрощенной функции значительно больше, приходится выбирать ПМЛ с большим числом входов для вентилей ИЛИ и меньшим числом выходов.

В настоящее время широко доступны компьютерные программы для проектирования ПМЛ. Такие программы на основе исходной информации, представленной логическими уравнениями, таблицами истинности, графами состояний и таблицами состояний, автоматически генерируют карту прошивки. Эти данные можно загрузить в ПМЛ-программатор, который пережигает перемычки и проверяет работу ПМЛ.

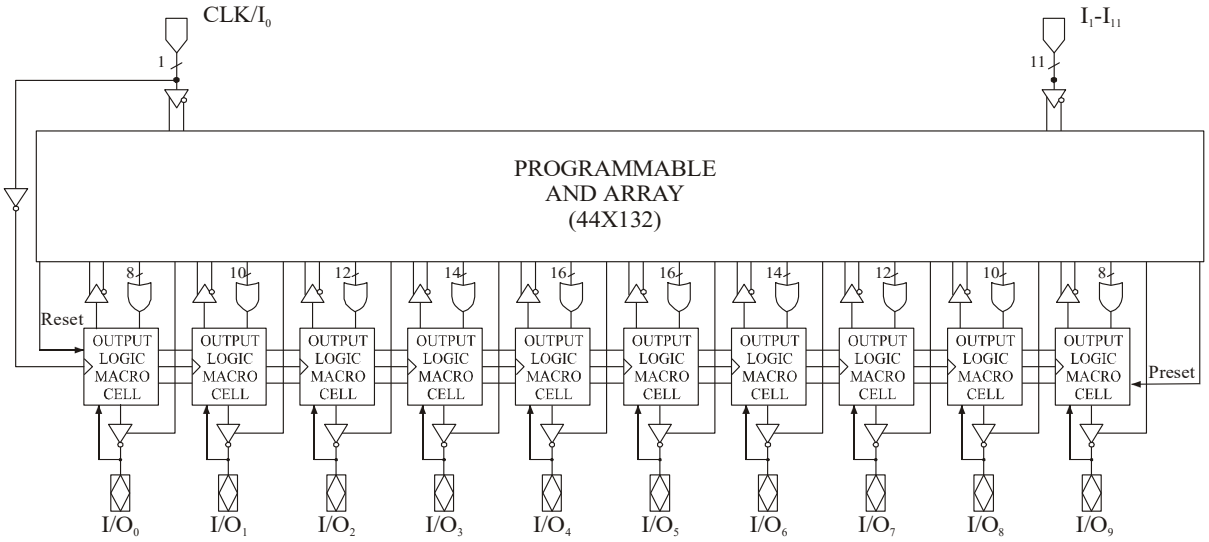
4.4. Другие последовательностные программируемые логические устройства (ПЛУ)

16R4 является примером простого последовательностоного ПЛУ. В связи с прогрессом в области технологии производства интегральных схем стали доступны другие типы ПЛУ. Некоторые из них базируются на развитии ПМЛ-технологии, другие – на вентильных матрицах.

Усовершенствование выходных макроячеек ПМЛ привело к появлению универсальных ПМЛ, которые обозначаются буквой V. Иногда их называют обобщенными матрицами логики (General array logic (GAL)).

Например, 22CEV10 (рисунок 4.18) – это КМОП электрически стираемое ПЛУ, которое может быть использовано для реализации комбинационных и последовательностных схем. Оно имеет 12 специальных входов и 10 контактов, которые могут быть запрограммированы как входы или выходы, 10 D-триггеров и 10 ИЛИ-вентилей. Число И-вентилей, подаваемых на элементы ИЛИ, изменяется от 8 до 16. Каждый ИЛИ-вентиль управляет выходной логической макроячейкой (*output logic macrocell*). Каждая макроячека содержит один из десяти триггеров. Они имеют общие входы синхронизации, асинхронного сброса (AR) и синхронной установки (SP). Название 22V10 означает универсальную ПМЛ, имеющую 22 ножки, 10 из которых – двунаправленные вход-выходные контакты.

Рисунок 4.18. Структурная схема 22V10



На рисунке 4.19 приведена схема выходной макроячейки 22CEV10. Направление передачи данных через контакт определяется программированием этой ячейки. Выходной мультиплексор MUX, управляемый входами S_1 и S_0, выбирает один из входов данных. Например, S_1S_2 = 10 означает вход 2. Каждая ячейка имеет два программируемых бита межсоединений: S_1 или S_0, соединяемых с землей (логический 0), когда соответствующий бит запрограммирован. Стирание бита отключает управляющую линию S_1 или S_0 от земли и позволяет ей перейти в состояние логической 1. Когда S_1 = 1, значение на выход поступает непосредственно с вентиля ИЛИ в обход триггера. Вентиль соединяется с вход-выходным контактом через мультиплексор и выходной буфер. Сигнал с ИЛИ также подается назад и может быть использован как вход И-вентиля. Когда S_0 = 1, выход не инвертируется и активным является высокий уровень сигнала. Когда S_0 = 0, выход инвертируется и активным является низкий уровень сигнала. Выходной контакт управляется тристабильным инвертирующим буфером. Когда выходной буфер находится в состоянии высокого импеданса, ИЛИ-вентиль и триггер отключены от внешнего контакта, который может быть использован как вход. Пунктирной линией на рисунке 4.19, а и б, показан путь сигнала через макроячейку, когда $S_1 = S_0 = 0$ и $S_1 = S_0 = 1$ соответственно. В первом случае Q-выход триггера инвертируется однажды выходным буфером, а во втором случае выход ИЛИ-вентиля инвертируется дважды. Таким образом, он не имеет инверсии.

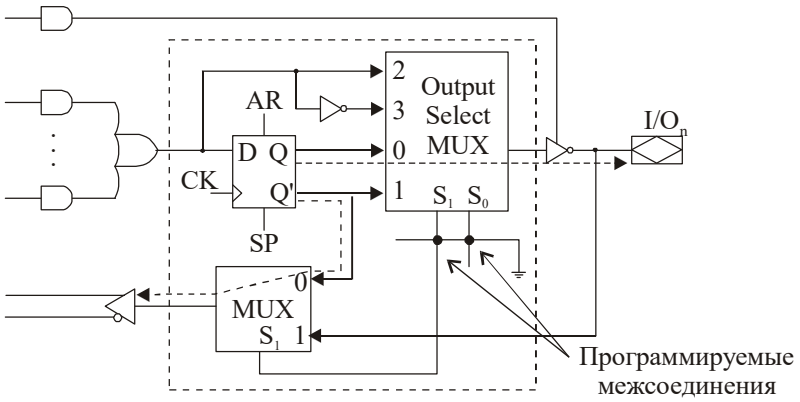
В таблице 4.6 представлены характеристики нескольких ПЛУ, похожих на 22V 10. Все эти ПЛУ имеют выходные макроячейки и каждая из них имеет один D-триггер. Вход-выходной контакт может быть запрограммирован как вход или как комбинаци-

онный, или как регистровый выход. Некоторые из ПЛУ имеют выделенные входы синхронизации, обозначенные в таблице как clk, другие имеют двухцелевые контакты, которые могут быть использованы как входы синхронизации или как обычный вход. Все ПЛУ имеют тристабильные буферы на выходах, а некоторые – специальный вход разрешения (OE).

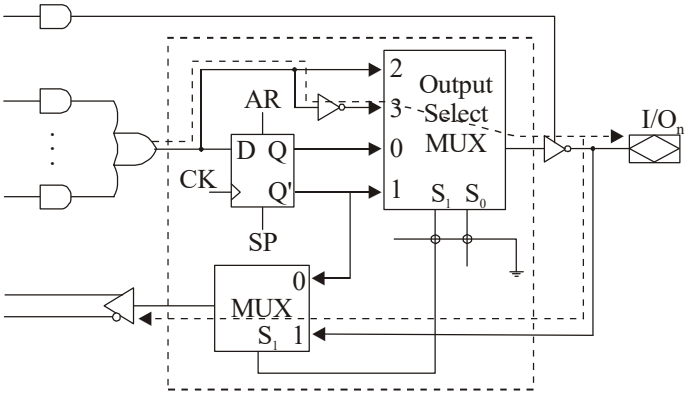
Таблица 4.6. Характеристики простых КМОП ПЛУ

Тип микросхемы	Количество входов	Входы/ выходы	Макроячейки= триггеры	Количество элементов И на один элемент ИЛИ
PALCE16V10	8 + OE + clk	8	8	8
PALCE20V8	14	8	8	8
PALCE22V10	12	10	10	8-16
PALCE24V10	14	10	10	8
PALCE29MA16	5+clk	16	16	4-12
CY7C335	12+OE+clk	12	12 in/12 out	9-19

Рисунок 4.19. Выходная макроячейка: а – путь сигнала, когда $S_1 = S_0 = 0$; б – путь сигнала, когда $S_1 = S_0 = 1$



а



б

GAL22V10

Так же как и PAL22V10, эта микросхема имеет переменное число И-термов, подключенных к ИЛИ-элементам: восемь – контакты 14 и 23, десять – контакты 15 и 22, двенадцать – контакты 16 и 21, четырнадцать и шестнадцать – контакты 17,20 и 18, 19 соответственно. Также доступны дополнительные термы для логики. Выход каждой

макроячейки может использоваться как прямой или инверсный, комбинационный или регистровый сигнал. Каждый выход возможно индивидуально запрограмировать на активный высокий или активный низкий сигнал. Имеются термы для асинхронного сброса и синхронной установки. Они являются дополнительно выделенными термами и общими для всех регистров. На рисунке 4.20 представлена структурная схема GAL22V10, а на рисунке 4.21 – его вход-выходная макроячейка.

При аналогичных функциях с PAL22V10 микросхема GAL22V10 более быстрая (PAL задержка 5ns, частота 142.8МГц, GAL задержка 4 ns, частота 250 МГц). Также GAL22V10 имеет дополнительные функции по защите хранящейся в ней информации.

Рисунок 4.20. Структурная схема GAL22V10

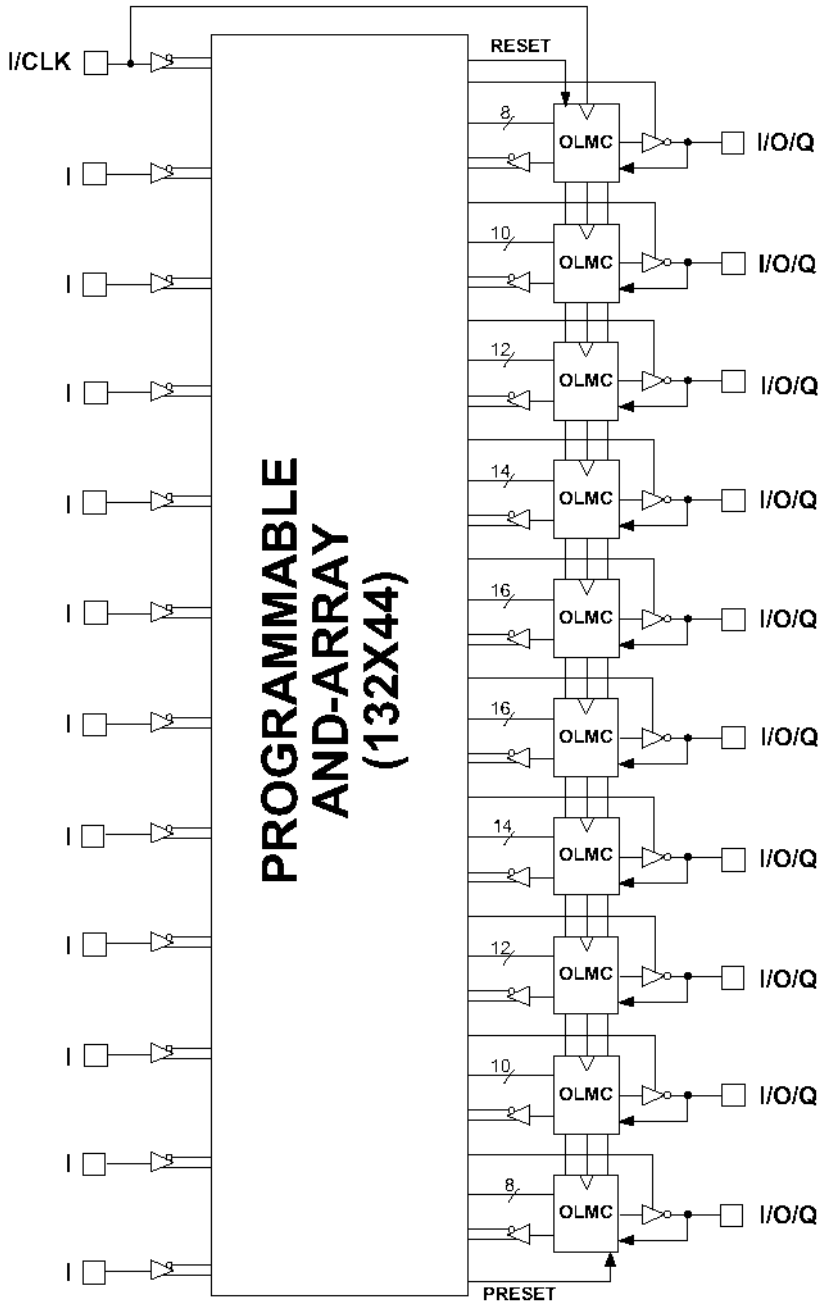
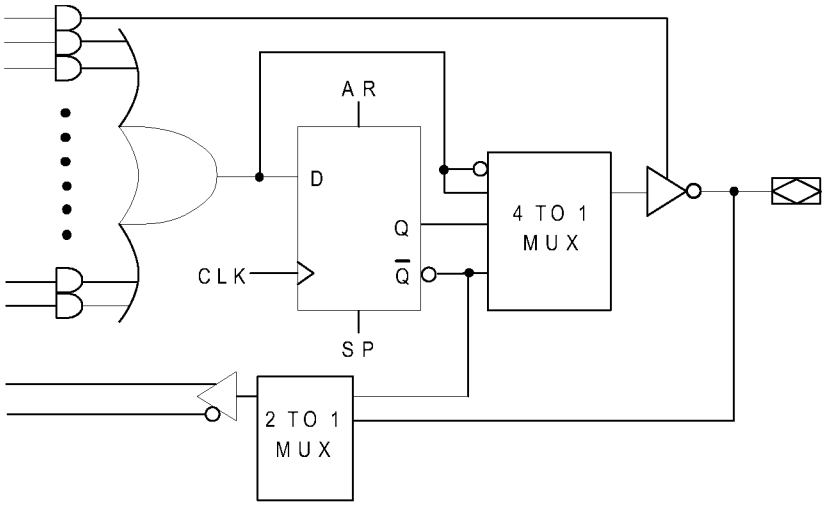


Рисунок 4.21. Входная/выходная макроячейка GAL22V10



Примеры проектирования с использованием микросхем 22V10

Проектируется контроллер светофора, расположенный на пересечении улиц "A" и "B". Каждая улица имеет датчик движения, который определяет наличие транспортных средств, приближающихся или остановившихся на перекрестке. *Sa* = 1 означает, что транспортное средство подъезжает к перекрестку по улице "A", а *Sb* = 1 – по улице "B". Улица "A" является главной и зеленый свет на светофоре горит до тех пор, пока машина не появляется на улице "B". Тогда свет светофора изменяется на зеленый для улицы "B". Через 50 с светофор переключается обратно, однако если на улице "B" есть транспорт, а на улице "A" нет, то зеленый свет для улицы "B" продлевается еще на 10 с. Продолжительность цикла зеленого света для улицы "A" равняется 60 с и свет изменяется, если на улице "B" есть транспорт. На рисунке 4.22 представлена структурная схема контроллера. Три выхода (Ga, *Ya* и *Ra)* управляют зеленым, желтым и красным сигналами светофора для улицы "A", а три других (Gb, Yb и Rb) – аналогичными огнями для улицы "B".

На рисунке 4.23 изображен граф переходов автомата Мура для контроллера. Последовательностная схема управляется синхроимпульсом с периодом в 10 с. По этой причине изменение состояния может происходить не чаще, чем один раз в 10 с. Используемые в графе состояний обозначения: *GaRb – Ga = Rb* = 1 , а все остальные выходные переменные равны нулю; *Sa'Sb* на дугах переходов – данный переход происходит при условии *Sa = 0* и *Sb* = 1. Переход по дуге без метки не зависит от входных переменных и выполняется по фронту синхроимпульса. Таким образом, зеленый свет на улице "A" будет сохраняться 6 синхроимпульсов (60 с) и затем изменится на желтый при наличии машин на улице "B".

Рисунок 4.22. Структурная схема контроллера светофора

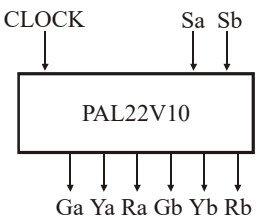
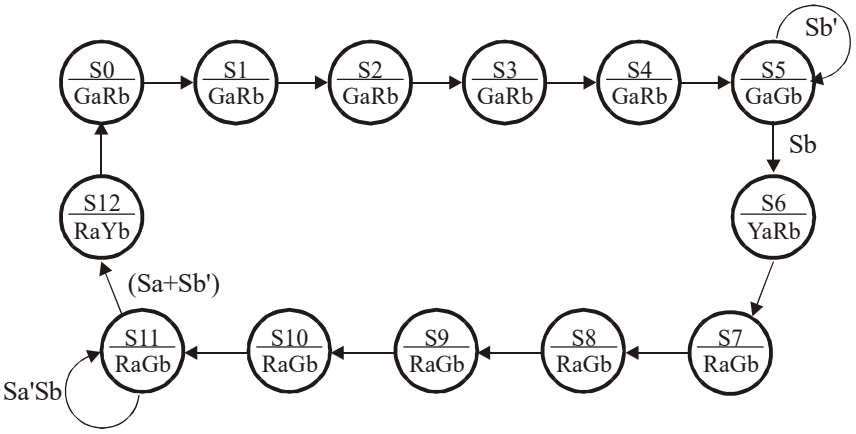


Рисунок 4.23. Граф состояний для контроллера светофора



VHDL-код для контроллера светофора (рисунок 4.24) описывает автомат с двумя процессами. Всякий раз, когда состояния *Sa* или *Sb* изменяются, первый процесс обновляет значения выходов и пересчитывает следующее состояние. Оператор **case** иллюстрирует использование выражения с диапазоном. Поскольку состояния от S0 до S4 имеют одни и те же значения на выходах и переход между ними происходит последовательно, вместо пяти операторов используется один – **when** с диапазоном:

```
when 0 to 4 => Ga <= '1'; Rb <= '1'; nextstate <= state + 1;
```

Рисунок 4.24. VHDL-модель контроллера светофора

```
entity traffic_light is
   port (clk, Sa, Sb: in bit;
         Ra, Rb, Ga, Gb, Ya, Yb: inout bit);
end traffic_light;
architecture behave of traffic_light is
   signal state, nextstate: integer range 0 to 12;
   type light is (R, Y, G) ;
   signal lightA, lightB: light;  -- определение выходных сигналов
begin
   process(state, Sa, Sb)
   begin
      Ra <= '0'; Rb <= '0';
      Ga <= '0'; Gb <= '0';
      Ya <= '0'; Yb <= '0';
      case state is
         when 0 to 4 => Ga <= '1'; Rb <= '1'; nextstate <= state+1;
         when 5 => Ga <= '1'; Rb <= '1' ;
                   if Sb = '1' then nextstate <= 6; end if;
         when 6 => Ya <= '1'; Rb <= '1'; nextstate <= 7;
         when 7 to 10 => Ra <= '1'; Gb <= '1'; nextstate <= state+1;
         when 11 => Ra <= '1'; Gb <= '1';
              if (Sa='1' or Sb='0') then nextstate <= 12; end if;
         when 12 => Ra <= '1'; Yb <= '1'; nextstate <= 0;
      end case;
   end process;
   process(clk)
   begin
      if clk = '1' then state <= nextstate;
      end if;
   end process;
```

```
    lightA <=  R when Ra='1' else
               Y when Ya='1' else
               G when Ga='1';
    lightB <=  R when Rb='1' else
               Y when Yb='1' else
               G when Gb='1';
end behave;
```

Для каждого состояния в операторе **case** упоминаются сигналы, получающие значение '1'. Остальные необходимо обнулять при смене состояний. Например, при идентификации состояния 6 сигнал Ga следует установить в '0', далее при состоянии 7 значение Ya = '0', ... Это можно сделать, поместив соответствующие выражения в операторы **when**. Например, в оператор "when 6 = >" можно добавить Ga <= '0' . Но гораздо проще устанавливать все выходы в '0' до выполнения оператора **case**, как это показано на рисунке 4.25. Может показаться, что это вызывает переключение сигнала, когда он должен постоянно оставаться в '1'. Однако это не так, поскольку последовательные операторы в процессе выполняются мгновенно. Например, пусть в момент времени time=30 произошел переход из состояния S2 в S3. Ga и Rb равны '1', но как только процесс начинает выполняться, будет вычислена первая линия кода. В очередь драйвера Ga и Rb заносится '0' со временем $20 + \Delta$. Затем выполняется оператор **case** и в драйверы сигналов Ga и Rb записывается значение '1' @ $20 + \Delta$, а предыдущий '0' удаляется.

Прежде чем закончить проектирование контроллера светофора, необходимо протестировать VHDL-код на его соответствие техническому описанию. Иначе, тестовая последовательность должна проверить каждый переход автомата как минимум один раз. Можно выполнить дополнительное тестирование, чтобы проверить временные характеристики при разных значениях транспортного потока, таких как оживленное движение на обеих улицах "A" и "B", слабое движение на них, оживленное движение только на "A", оживленное движение только на улице "B", особые ситуации: машина не смогла поехать на зеленый свет, она едет через перекресток на красный.

Чтобы облегчить чтение результатов моделирования, определен тип light со значениями R**,** Y, G и два сигнала *UghtA* и *HghtB* этого типа. Затем добавлен код, устанавливающий *UghtA* в R, Y или G, когда свет светофора красный, желтый или зеленый соответственно. Следующий командный файл устройства моделирования вначале тестирует ситуацию, когда обе петли графа проверяются, а затем – когда ни одна петля графа не проверяется:

```
wave clk SA SB state lightA lightB
force -repeat 10 sec clk 0 0 sec, 1 5 sec
force SA 1 0 sec, 0 40 sec, 1 170 sec, 0 230 sec, 1 250 sec
force SB 0 0 sec, 1 70 sec, 0 100 sec, 1 120 sec, 0 150 sec,
         1 210 sec, 0 250 sec, 1 270 sec
run 300 sec
```

Результаты тестирования, приведенные на рисунке 4.25, отображают тот факт, что огни светофора переключаются в заданные моменты времени.

Рисунок 4.25. Результаты тестирования контроллера светофора

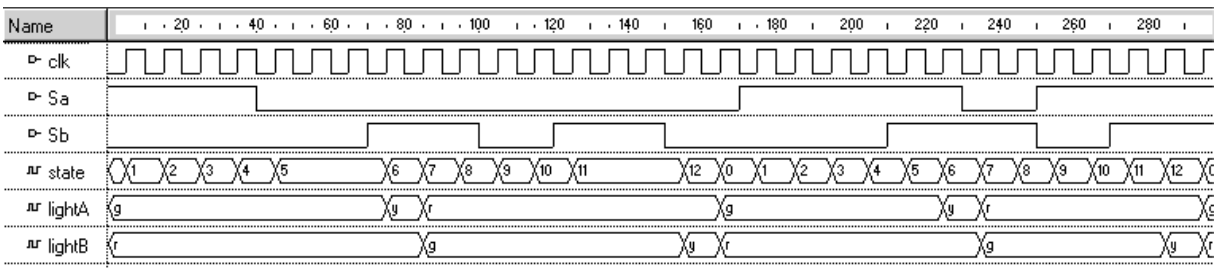


Таблица 4.7. Состояния для контроллера светофора

	SaSb 00	01	10	11	Ga	Ya	Ra	Gb	Yb	Rb
S0	S1	S1	S1	S1	1	0	0	0	0	1 (Green A, Red B}
S1	S2	S2	S2	S2	1	0	0	0	0	1
S2	S3	S3	S3	S3	1	0	0	0	0	1
S3	S4	S4	S4	S4	1	0	0	0	0	1
S4	S5	S5	S5	S5	1	0	0	0	0	1
S5	S5	S6	S5	S6	1	0	0	0	0	1
S6	S7	S7	S7	S7	0	1	0	0	0	1 {Ya.Rb]
S7	S8	S8	S8	S8	0	0	1	1	0	0 {Ra, Gb]
S8	S9	S9	S9	S9	0	0	1	1	0	0
S9	S10	S10	S10	S10	0	0	1	1	0	0
S10	S11	S11	S11	S11	0	0	1	1	0	0
S11	S12	S11	S12	S12	0	0	1	1	0	0
S12	S0	S0	S0	S0	0	0	1	0	1	0{Ra, Yb}

Таблица 4.7 представляет собой функции переходов контроллера. Такой автомат может быть реализован с использованием четырех D-триггеров со входами D_1, D_2, D_3, D_4 и выходами Q_1, Q_2, Q_3, Q_4. При использовании унарного кодирования, когда код любого состояния содержит только одну единицу, из таблицы были получены следующие уравнения:

```
D1=Q1Q2'+Q2Q3Q4
D2=Q1'Q2'Q3Q4+SaQ1Q3Q4+Sb'Q1Q3Q4+Q1'Q2Q4'+Q1'Q2Q3'
D3=Q3Q4'+SbQ3'Q4+Q2'Q3'Q4+Sa'SbQ1Q4
D4=Sa'SbQ1Q3+Q2'Q4'+Q1'Q4'+SaSb'Q2Q3'Q4
Ga=Q1'Q3'+Q1'Q2'  Ya=Q2Q3Q4'  Ra=Q1+Q2Q3Q4
Gb=Q1Q2'+Q2Q3Q4       Yb=Q1Q2     Rb=Q1'Q2'+Q1'Q4'+Q1'Q3'
```

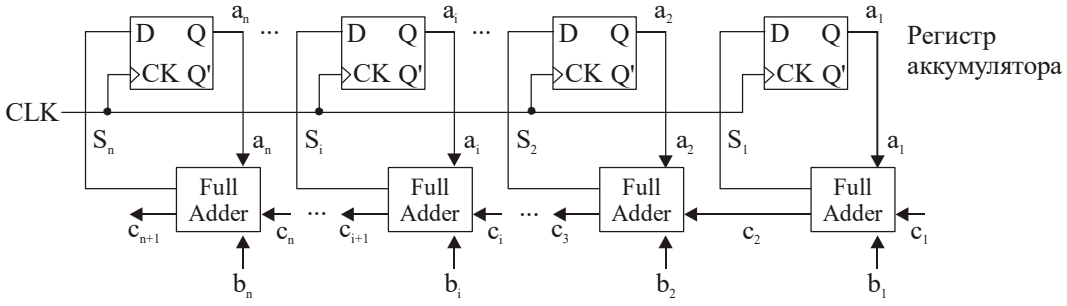
Поскольку в каждом из этих уравнений меньше восьми термов И, их можно легко реализовать на микросхеме 22V10. Если бы некоторые уравнения имели большее количество термов и не годились для реализации на 22V10, следовало бы использовать другие способы кодирования состояний.

Параллельный сумматор с накапливанием

Следующий пример иллюстрирует реализацию параллельного сумматора с накапливанием на 22V10. На рисунке 4.26 представлена структурная схема сумматора, который складывает n-битное число $B = b_n \ldots b_2 b_1$, с аккумулятором $A = a_n \ldots a_2 a_1$ и выдает n-битную сумму и перенос.

Первое число А загружается в аккумулятор. Затем на вход полного сумматора подается число В. После того как перенос прошел через схему, результирующая сумма появляется на ее выходах. По синхроимпульсу сумма заносится в аккумулятор. Один из способов поместить А в аккумулятор – это очистить его, используя входы сброса триггеров, после чего подать число А на В входы и выполнить сложение чисел.

Рисунок 4.26. Параллельный сумматор с накапливанием



В проект вводится управляющий вход Ad, разрешающий сложение, если на него подана 1. Это можно сделать, добавив вентиль на синхровход таким образом, чтобы синхроимпульсы на триггеры поступали только при Ad =1. Лучший способ, не требующий использования вентиля на синхровходе – это добавить переменную Ad в уравнения входов триггеров аккумулятора:

$$a_i^+ = Si = Ad(a_i'b_i'c_i + a_i'b_ic_i' + a_ib_i'c_i' + a_ib_ic_i) + Ad'a_i \qquad (4.5)$$

таким образом, чтобы a_i не изменяла значения при $Ad = 0$. То же самое следует сделать и для уравнения переноса c_{i+1}.

При реализации сумматора на микросхеме 22V10 необходимо вычислить максимальное число разрядов. Пусть число триггеров равно F, а количество комбинационных функций сложения – C. Поскольку число ячеек равно 10, должно выполняться:

$$F+C \le 10. \qquad (4.6)$$

В микросхеме имеется 12 выделенных входов и каждая неиспользуемая макроячейка может быть использована как вход, поэтому число внешних входов (I) равно

$$I \le 12+[10-(F+C)]=22-F-C. \qquad (4.7)$$

Также нужно быть уверенным, что число И-термов в комбинационных функциях и функциях D-триггеров не превышает допустимое число И-вентилей. Для каждого бита сумматора требуется один триггер, вход которого обозначается как S_i. К тому же необходимо генерировать перенос из каждого бита, который используется в старшем разряде. Значение функции c_{i+1}, должно подаваться обратно на И-массив через макроячейку, даже если внешний перенос не требуется. Для N-битного сумматора F=C = N, поэтому из уравнения (4.6) следует $2N \le 10$. Число входов для него определяется суммой: количества битов N и переменных синхровхода, сброса, переноса (c_i) и сигнала Ad, что дает в соответствии с уравнением (4.7) выражение $I=N+4 \le 22-2N$. Решением данного неравенства является значение $N \le 5$.

Операция сложения, реализованная таким способом, будет довольно медленной, потому что определенное время будет затрачено на распространение переноса через пять секций И-ИЛИ в кристалле 22V10. Один из способов повысить скорость операций сложения и одновременно увеличить число бит, которые можно реализовать на одной микросхеме 22V10 – это выполнить сложение в блоках по два бита без генерации дополнительных переносов. Ниже представлен фрагмент таблицы истинности и часть уравнений для такого двухбитного сумматора:

A_2	A_1	B_2	B_1	C_1	C_3	S_2	S_1
0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1
0	0	0	1	0	0	0	1
0	0	0	1	1	0	1	0
0	0	1	0	0	0	1	0
-	-	-	-	-	-	-	-
1	1	0	1	0	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	0	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	0
1	1	1	1	1	1	1	1

$C_3=B_1B_2'C_1+A_1B_2C_1+A_1B_2B_1+A_2B_1C_1+A_2B_2+A_1A_2C_1+A_1A_2B_1$

$S_2=(A_2'B_1B_2'C_1+A_1'A_2'B_1'B_2+A_1'A_2'B_2C_1'+A_1A_2'B_2'C_1+A_1A_2'B_1B_2'+A_2'B_1'B_2C_1'+$

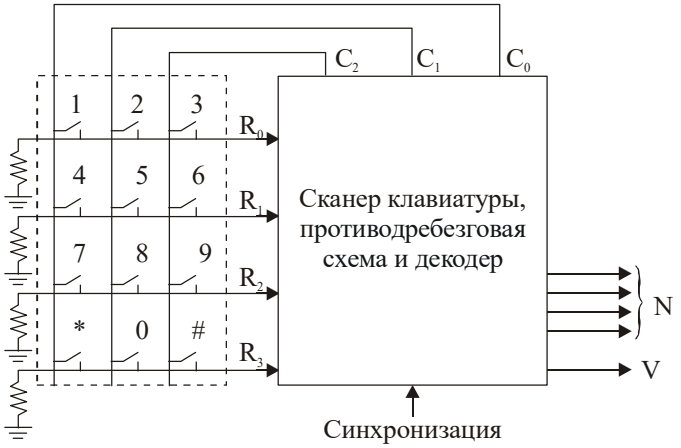
```
+ A1'A2B1'B2'+A1'A2B2'C1'+A2B1B2C1+A2B1'B2'C1'+A1A2B2C1+A1A2B1B2)Ad+Ad'A2;
S1=(A1'B1'C1+A1'B1C1'+A1B1'C1'+A1B1C1)Ad+Ad'A1
```

Поскольку самое длинное уравнение содержит 13 И-термов, а их максимальное число для 22V10 равно 16, эти уравнения могут быть запрограммированы на упомянутом кристалле. Можно также реализовать три 2-битных сумматора на одной микросхеме 22V10, потому что F+C=6+3=9 ≤ 10 и I=6+4 ≤ 22-9. В данном случае перенос должен распространяться только через три И-ИЛИ секции. Таким образом, данный 6-битный сумматор значительно быстрее 5-битного, спроектированного ранее.

4.5. Проектирование сканера клавиатуры

Рассмотрим пример проектирования на ПЛУ сканера клавиатуры телефона. Рисунок 4.27 содержит структурную схему устройства. Клавиатура представляет собой матрицу из проводов с переключательными элементами на пересечении строк и столбцов. Нажатием клавиши создается связь между ними. Задача сканера – определить, какая кнопка была нажата, и сформировать двоичное число $N = N_3N_2N_1N_0$, соответствующее номеру кнопки. Например, если нажата кнопка 5, на выходе будет 0101, кнопка * – 1010, кнопка # – 1011. После того, как номер кнопки определен, сканер должен выдать сигнал V, длительностью в один такт синхронизации. Резистор, соединяющий каждый ряд с землей, подключен таким образом, что $R_1=R_2=R_3=R_4=0$, если ни одна кнопка не нажата.

Рисунок 4.27. Структурная схема сканера клавиатуры

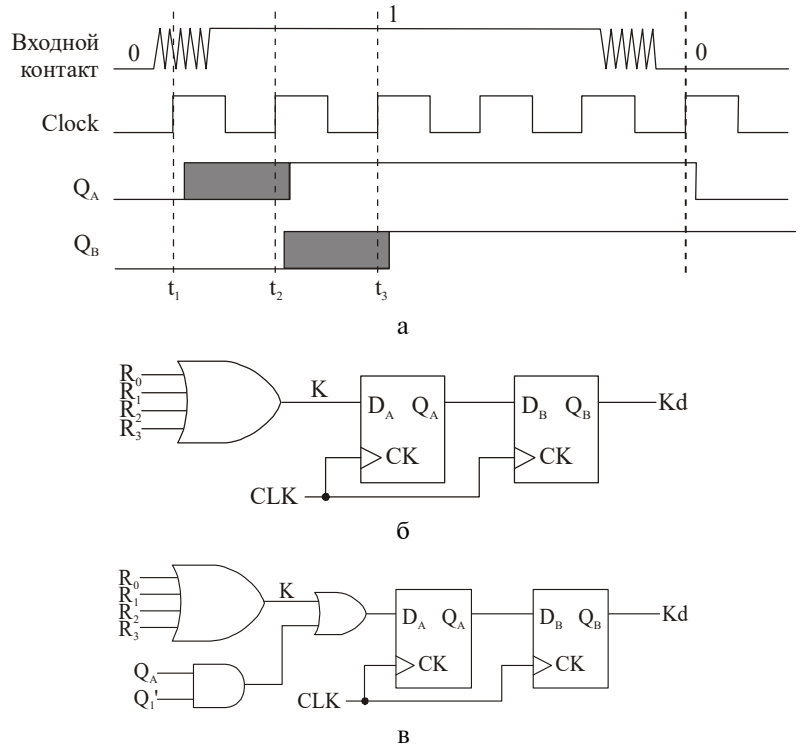


Для сканирования клавиатуры будет использована следующая процедура. Сначала на линии C_0, C_1 и C_2 подаются логические 1. Если какая-нибудь клавиша нажата, 1 появится на соответствующей линии R_0, R_1, R_2 или R_3. Затем подается 1 только на линию C_0. Если R_i=1, клавиша определена, сигнал V= 1 и на выход выдается номер N. Если нажата клавиша не из первой колонки, единица подается на C_1. Затем, если кнопка не определена, 1 подается на C_2. После определения кнопки единицы подаются и сохраняются на линиях C_0, C_1 и C_2, пока какая-нибудь из кнопок не будет нажата. Последний шаг необходим, чтобы при каждом нажатии кнопки генерировался только один сигнал.

В процессе вычисления нажатой клавиши сканер должен учитывать дребезг контакта. Когда механический контакт замыкается или размыкается, переключающий контакт в течение нескольких миллисекунд, пока он не установится в окончательную позицию, будет иметь дребезг, вызывая помехи на выходе, как это показано на рисунке 4.28, а. Поэтому после того, как произошло замыкание контакта перед сканированием клавиатуры, необходимо подождать, пока помехи исчезнут. Также сигнал, появляющийся в результате нажатия кнопки, необходимо синхронизировать, поскольку он является входом синхронной последовательностной схемы.

Рисунок 4.28, б содержит противодребезговую и синхронизирующую схему. Период синхроимпульса должен быть больше времени дребезга. Если $C_0 = C_1 = C_2 = 1$ в момент нажатия кнопки, то K станет равным 1 после того, как дребезг успокоится. Если передний фронт синхроимпульса поступит в момент дребезга, 0 или 1 сохранится в триггере в момент t_1. При наличии 0 единица будет получена при следующем активном фронте синхроимпульса (t_2). Поэтому сигнал Q_A будет беспомеховой и синхронизированной версией сигнала K. Тем не менее возможность ошибки существует, если изменение сигнала K будет близко к активному фронту синхроимпульса, что приведет к нарушению времени установки и хранения. В таком случае выход триггера может начать генерировать или выдать неправильный результат. Хотя такая ситуация встречается редко, лучше исключить ее, добавив еще один триггер. Нужно выбрать такой период синхроимпульса, чтобы любые колебания на выходе Q_A закончились до прихода следующего активного фронта. Тогда вход D_B будет всегда стабильным в этот момент. Противодребезговый сигнал Kd будет отфильтрован и синхронизирован, хотя он может иметь задержку в два временных такта после нажатия.

Рисунок 4.28. Противодребезговая и синхронизирующая схема



Сканер состоит из трех модулей, как показано на рисунке 4.29. Противодребезговый модуль генерирует сигнал K и очищенный Kd. Сканирующий модуль генерирует сигналы столбцов для опроса клавиатуры. Когда кнопка определена, декодер вычисляет ее номер по идентификаторам ряда и столбца. Рисунок 4.30 представляет граф переходов сканирующего модуля. До нажатия кнопки он находится в состоянии S_1 с выходами $C_1 = C_2 = C_3 = 1$. В состоянии S_2 сигнал $C_0 = 1$, если нажата кнопка из нулевого столбца. При K = 1 схема выдает сигнал V и переходит в состояние S_5. Если нажатая кнопка не принадлежит столбцу 0, проверяется первый – состояние S_3, и если необходимо, столбец 2 – состояние S_4. В состоянии S_5 схема ожидает, пока все кнопки не будут отпущены, после чего Kd переходит в 0 до следующей установки.

Рисунок 4.29. Модули сканера

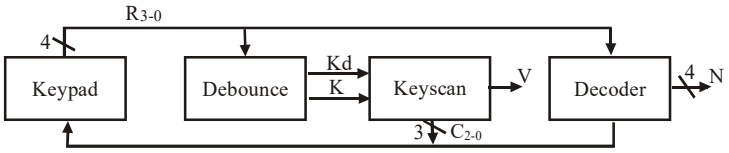
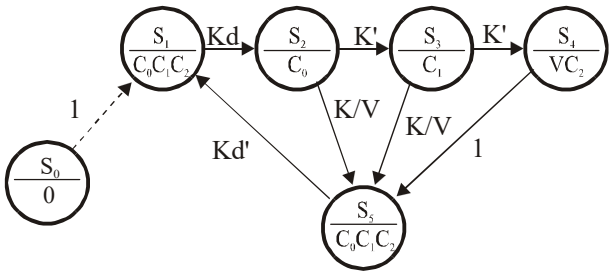


Рисунок 4.30.Граф переходов для сканера



Декодер, используя таблицу истинности (таблица 4.8), определяет номер кнопки по идентификаторам ряда и столбца. Каждая строка соответствует одной из 12 кнопок. Остальные ряды не влияют на выходы, поскольку можно допустить, что нажата только одна кнопка. Поскольку декодер является комбинационной схемой, его выходы изменятся, как только кнопка будет определена. В этот момент, когда $K = 1$ и $V = 1$, на выходах декодера находятся правильные значения сигналов, которые могут быть сохранены в регистре как состояние S_5.

Таблица 4.8. Таблица истинности декодера

R_3	R_2	R_1	R_0	C_0	C_1	C_2	N_3	N_2	N_1	N_0
0	0	0	1	1	0	0	0	0	0	1
0	0	0	1	0	1	0	0	0	1	0
0	0	0	1	0	0	1	0	0	1	1
0	0	1	0	1	0	0	0	1	0	0
0	0	1	0	0	1	0	0	1	0	1
0	0	1	0	0	0	1	0	1	1	0
0	1	0	0	1	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0
0	1	0	0	0	0	1	1	0	0	1
1	0	0	0	1	0	0	1	0	1	0(*)
1	0	0	0	0	1	0	0	0	0	0
1	0	0	0	0	0	1	1	0	1	1(#)

Логические уравнения для декодера:

$N_3=R_2C_0'+R_3C_1'$
$N_2=R_1+R_2R_0$
$N_1=R_0C_0'+R_2'C_2+R_1'R_0'C_0$
$N_0=R_1C_1+R_1'C_2+R_3'R_1'C_1'$

Можно попробовать реализовать противодребезговый сканирующий и декодирующий модуль на одной микросхеме 22V10, если потребуется, с небольшим добавлением аппаратуры. Из 22V10 используются входы: R_0, R_1, R_2 , R_3, синхровход и сброс. Выходы: C_0, C_1, C_2, N_3, N_2, N_1, N_0 и V– занимают 8 из 10 макроячеек. Если реализовать граф состояний, используя 3 триггера, потребуется 11 макроячеек, что не годится для выбранной микросхемы. Одним из решений данной проблемы может быть применение двух ПМЛ и реализация декодера на отдельной ПМЛ. Но лучше использовать четыре триггера для реализации графа состояний, которые закодировать так, чтобы выходы C_0, C_1 и C_2 можно было читать прямо из триггеров Q_2, Q_3, Q_4. Используется следующее

обозначение состояний для $Q_1Q_2Q_3Q_4$: $S_1 - 0111; S_2 - 0100; S_3 - 0010; S_4 - 0001; S_5 - 1111$. Первые три триггера формируют С выходы. Состояния S_1 и S_5 отличаются значанием Q_1. При таком кодировании состояний требуется 9 макроячеек для реализации сканирующего и декодирующего модулей. Для дребезгового модуля остается один триггер, поэтому необходим всего один внешний триггер для Kd. Если выполнить сброс 22V10, триггеры установятся в 0000. Таким образом, появляется дополнительное состояние S_0 с переходом в S_1, добавляемое к графу. Ниже приведены уравнения для графа состояний:

$$Q_1^+ = Q_1Kd + Q_2Q_3'K + Q_2'Q_3K + Q_2'Q_4,$$
$$Q_2^+ = Q_2'Q_3' + K + Q_4,$$
$$Q_3^+ = Q_3' + Q_1 + Q_4Kd' + Q_2'K,$$
$$Q_4^+ = Q_2' + Q_1 + Q_3Kd' + Q_3'K,$$
$$V = KQ_2Q_3' + KQ_2'Q_3 + Q_2'Q_4.$$

Для устранения генерации на линии К необходима дополнительная ячейка. При этом в уравнениях К заменяется на $R_0+R_1+R_2+R_3$:

$$Q_1^+ = Q_1Kd + Q_2Q_3'(R_0 + R_1 + R_2 + R_3) + Q_2'Q_3(R_0 + R_1 + R_2 + R_3) + Q_2'Q_4.$$

Максимальное число термов в любом уравнении равно 10, поэтому их можно реализовать на 22V10. Конечные уравнения могут быть введены в программу автоматизированного проектирования для генерации битовой карты прошивки ПМЛ.

При тестировании схемы с помощью VHDL был обнаружен один недостаток, связанный с противодребезговой схемой (см. рисунок 4.28, б). Оригинальный проект работает корректно, если нажата кнопка в столбце 0 или 1, но если нажата кнопка из столбца 2, могут возникнуть проблемы. В этом случае К переходит в 0 при сканировании столбцов 0 и 1. Таким образом, Kd переходит в 0 при достижении состояния S_5, даже если кнопка до сих пор нажата. Чтобы исправить эту ошибку, необходимо изменить уравнение состояния для Q_A. на $Q_A^+ = K + Q_AQ_1'$. Введение дополнительного терма гарантирует, что если Q_A установится в 1, переменная будет оставаться в этом состоянии, пока устройство не перейдет в состояние S_5 и Q_1 не станет равным 1. Исправленная противодребезговая схема показана на рисунке 4.28, в.

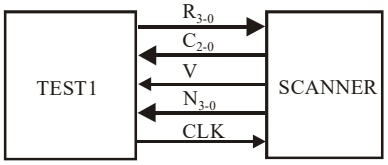
VHDL-код проекта представлен на рисунке 4.31. Уравнения декодера, также как и уравнения для K и V, реализованы с помощью параллельных операторов. Процесс использован для реализации уравнений состояний и противодребезговых триггеров. Этот код было бы трудно протестировать, подавая сигналы на входы R_0, R_1 и R_2, поскольку они зависят от выходов столбцов (C_0, C_1, C_2). Лучший способ тестирования сканера – это написать на VHDL программу, названную scantest . Такие тестирующие программы часто называют test bench (испытательный стенд – функциональный тест – тест верификации изделия), по аналогии с аппаратурным испытательным стендом. При тестировании сканер описывается как компонент и встраивается в тестовую программу. Сигналы, сгенерированные в scantest, передаются на сканер по интерфейсу, представленному на рисунке 3.32. Scantest моделирует нажатие кнопки, подавая соответствующие R-сигналы в ответ на С-сигналы со сканера. Когда scantest получит сигнал V = 1 со сканера, будет выполнена проверка – соответствует ли сгенерированное сканером число N нажатой кнопке.

Рисунок 4.31. VHDL-код для сканера

```
entity scanner is
  port (R0,R1,R2,R3,CLK: in bit;
        C0,C1,C2: inout bit;
        N0,N1,N2,N3,V: out bit);
```

```
end scanner;

architecture scani of scanner is
signal Q1,QA, K, Kd: bit;
alias Q2: bit is C0;          -- выходы C0, C1 и C2 должны совпадать
alias Q3: bit is C1;          -- со значениями Q2, Q3 и Q4
alias Q4: bit is C2;          -- из-за выбранного кодирования
begin
   K <=  R0 or R1 or R2 or R3;         -- это секция декодера
   N3 <= (R2 and not C0) or (R3 and not C1);
   N2 <= R1 or (R2 and C0) ;
   N1 <= (R0 and not C0) or (not R2 and C2) or
         (not R1 and not RO and CO);
   NO <= (R1 and C1) or (not R1 and C2) or
         (not R3 and not R1 and not C1);
   V <=  (Q2 and not Q3 and K) or (not Q2 and Q3 and K) or
         (not Q2 and Q4);
   process(CLK)                      -- процесс, обновляющий триггеры
   begin
      if CLK = '1' then
         Q1 <= (Q1 and Kd) or (Q2 and not Q3 and K) or (not Q2 and q3
               and K) or (not Q2 and Q4) ;
         Q2 <= (not Q2 and not Q3) or K or Q4;
         Q3 <= not Q3 or Q1 or (Q4 and not Kd) or (not Q2 and K) ;
         Q4 <= not Q2 or Q1 or (Q3 and not Kd) or (not Q3 and K) ;
            -- первый противодребезговый триггер
         QA <= K or (QA and not Q1);
         Kd <= QA;                  -- второй противодребезговый триггер
      end if;
   end process;
end scani;
```

Рисунок 4.32. Интерфейс сканера



VHDL-код scantest представлен на рисунке 4.33. Копия сканера встроена в архитектуру, а соединение с ним выполняется через карту портов (port map). Используемая для тестирования последовательность номеров кнопок хранится в массиве KARRAY. Тестер моделирует аппаратуру с помощью параллельных операторов для R_0, R_1, R_2 и R_3. Всякий раз, когда C_0, C_1, C_2 или номер кнопки (KN) изменяются, вычисляются новые значения для Rs. Например, если KN = 5 – моделируется нажатие кнопки 5, то значение $R_0R_1R_2R_3 = 0100$ посылается на сканер, когда $C_0C_1C_2 = 010$. Процесс тестирования будет выполнен следующим образом:

1. Чтение номера кнопки из массива для моделирования нажатия кнопки.
2. Ожидание V = 1 и активного фронта синхроимпульса.
3. Сравнение результирующего номера N с выходов сканера и номера кнопки.
4. Установка KN = 15 для моделирования отсутствия нажатия кнопки. Поскольку не существует кнопки с номером 15, все Rs устанавливаются в 0.
5. Ожидание Kd = 0 до выбора новой кнопки.

Scantest выдает сообщение об ошибке, если сканер сгенерирует неправильный номер кнопки, и "Testing complete", когда все номера будут протестированы.

Рисунок 4.33. VHDL для Scantest

```
library BITLIB;
use BITLIB.bit_pack.all;
```

```
entity scantest is
end scantesfc;

architecture testi of scantest is
component scanner
  port (RO,R1,R2,R3,CLK: in bit;
        CO,C1,C2: inout bit;
        NO,N1,N2,N3,V: out bit);
end component;
type arr is array(0 to 11) of integer;
constant KARRAY:arr := (2,5,8,0,3,6,9,11,1,4,7,10);
signal C0,Cl,C2,V,CLK,RO,Rl,R2,R3: bit;       -- сигналы интерфейса
signal N: bit_vector(3 downto 0);
signal KN: integer;                        -- номер тестируемой кнопки

begin
  CLK <= not CLK after 20 ns;      -- генерирование синхросигнала
  -- Эта секция эмулирует клавиатуру
  RO <= '1' when   (C0='l' and KN=1) or (Cl='r and KN=2) or
                   (C2='1' and KN=3)  else
        '0';
  Rl <= '1' when   (C0='l' and KN=4) or (Cl='l' and KN=5) or
                   (C2='1' and KN=6) else
        '0' ;
  R2 <= '1' when(C0='1' and KN=7) or (C1='1' and KN=8) or
                   (C2='1' and KN=9) else
       '0';
  R3 <= '1' when(C0='l' and KN=10) or (Cl='l' and KN=0) or
                   (C2='l' and KN=11) else
       '0';
  process                            -- эта секция тестирует сканер
  begin
    for i in 0 to 11 loop    -- тестирование каждого номера кнопки
      KN <= KARRAY(i);    -- моделирование нажатия кнопки
      wait until (V='1' and rising_edge(CLK)) ;
      assert (vec2int(N) = KN) -- проверка результатов на выходах
        report "Numbers don't match"
        severity error;
      KN <= 15;          -- эквивалентно отсутствию нажатия кнопки
      wait until rising_edge(CLK);     -- ожидание сброса сканера
      wait until rising_edge(CLK) ;
      wait until rising_edge(CLK);
    end loop;
    report "Test complete.";
  end process;
  scanneri: scanner          -- подключение сканера для тестирования
    port map(RO,R1,R2,R3,CLK,CO,C1,C2,N(0),N(1),N(2),N(3),V) ;
end testi;
```

Поскольку в предыдущей программе встречаются не рассмотренные ранее операторы, существует необходимость их подробного описания, приведенного ниже.

Оператор Alias

Оператор **alias** позволяет задавать альтернативный идентификатор (псевдоним) для обращения к объекту. Его синтаксис

```
alias indentifier is name;
```

Используя новый идентификатор, можно точно так же обращаться к объекту, как если бы это был первоначальный.

Одним из случаев применения псевдонима может быть ситуация, когда необходимо задавать простые имена объектам, импортированным из пакетов.Например, два разных пакета **alu_types** и **io_types** декларируют константу с именем data_with, имеющую различные значения в каждом пакете. В обычной ситуации только одна из версий этой константы не может быть использована в одном коде из-за их одинакового имени.Тем не менее к этим константам можно обращаться, как **work.alu_types.data_with** и **work.io_types.data**. Применение оператора **alias** позволяет избежать таких длинных имен:

```
library ieee;
use work.alu_types.all, work.io_types.all;
architecture structural of controller_system is
   alias alu_data_width is work.alu_types.data_width;
   alias io_data_width is work.io_types.data_width;
   signal alu_in1, alu_in2,
         alu_result: std_logic_vector(0 to alu_data_width-1);
   signal io_data: std_logic_vector(0 to io_data_width-1);
. . .
```

Как и для обозначения целых объектов данных псевдоним может быть назначен отдельным элементам сложных объектов данных: массивов и записей, а также сектору массива. Например,

```
alias SP is GPR(15);
alias Interupt is GRP(30 downto 26);
```

В некоторых случаях может быть полезным обозначение подтипа псевдонима:

```
alias alias_name [: alias_type] is object_name;
```

Подтип псевдонима указывает, как будут видимы структуры этих данных. Можно включить подтип в конструкцию **alias** при создании альтернативных идентификаторов для объектов скалярных типов данных. Но границы и направление этого подтипа должны быть такими же, как и у оригинала. Поэтому такое описание в большей степени является документированием, предназначенным для повышения читаемости кода. В качестве подтипа при записи псевдонима для элемента данных или его сектора можно использовать массив с неограниченным размером. Тогда его направление и размер будет такой же, как и у исходного объекта. Однако при создании альтернативного идентификатора для массива или части массива можно применять подтип с другими индексными границами или направлением. При этом базовый тип подтипа должен быть таким же, как и у оригинального идентификатора. Другими словами, он должен иметь тот же тип элементов и индексации, что и исходный объект, а также одинаковое число элементов массива. Например, для четырех младших разрядов 32-битного сигнала DataBus типа bit_vector создается псевдоним с именем FirstNibble. Этот массив имеет противоположное направление индексации по отношению к исходному сигналу:

```
signal DataBus : Bit_Vector(31 downto 0);
alias FirstNibble : Bit_Vector(0 to 3) is DataBus(31 downto 28);
```

Оператор Assert

В архитектуре scantest для сравнения результатов на выходах сканера с заданным номером используется оператор assert. Он проверяет заданное условие на истинность, в противном случае выдается сообщение об ошибке. Одна из форм оператора assert имеет вид

```
assert boolean-expression
     report string-expression
     severity severity-level;
```

Если булево выражение (boolean-expression) имеет значение false, на экране появится строковое выражение (string-expression), зависящее от уровня серьезности (severity-level). Иначе – никакое сообщение не выводится. Существует четыре возможных уровня серьезности: note, warning, error и failure. Действия, которые будут выполняться для каждого уровня, зависят от устройства моделирования.

Если оператор assert опущен, сообщение в операторе report будет выводиться всегда. Указывать уровень серьезности (severity) не обязательно. Следующий оператор всякий раз при выполнении

```
report "ALL IS WELL";
```

будет выдавать сообщение "ALL IS WELL".

Выводы. Описаны несколько разных ПЛУ и их использование для проектирования последовательностных схем. Существует много других типов ПЛУ, доступных вместе с программным обеспечением, облегчающим их применение. Некоторые из этих программ принимают входные данные только в виде логических уравнений, в то время как другие позволяют вводить таблицы состояний, графы переходов или логические схемы. Они генерируют файлы данных, используемые программаторами для прошивки ПЛУ при выполнении определенных функций.

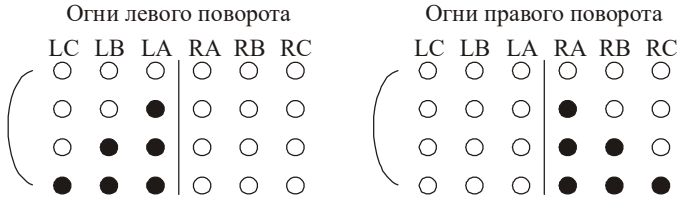
4.6. Задачи

4.6.1. Автомат, описанный следующей таблицей переходов и выходов, реализован на ПЗУ и двух D-триггерах с задним фронтом синхронизации:

Q_1Q_2		$Q_1^+Q_2^+$		Z	
		X=0	X=1	X=0	X=1
0	0	01	10	0	1
0	1	10	00	1	1
1	0	00	01	1	0

Нарисовать структурную схему. Написать таблицу прошивки ПЗУ. Написать VHDL-код, описывающий систему, в которой ПЗУ имеет задержку 10 ns и каждый триггер имеет задержку распространения 15 ns.

4.6.2. В старых моделях Thunderbird машин было три левых и три правых задних огня, которые посылали световые сигналы в определенном порядке для обозначения левого или правого поворота:



Спроектировать автомат Мура для контроля за этими огнями. Схема имеет три входа: LEFT, RIGHT и HAZ. LEFT и RIGHT приходят от переключателя сигнала поворота и не могут быть оба равны 1 в одно и то же время. Как показано выше, если сигнал LEFT = 1, огни переключаются по схеме: LA включен, LA и LB включен, LA, LB и LC включен, все выключены; затем последовательность повторяется. Когда RIGHT = 1, огни зажигаются аналогичным образом. Если переключатель "слева

направо" (и наоборот) проходит среднее положение, схема должна немедленно перейти в состояние IDLE (огни погашены) и затем начать новую последовательность. Сигнал HAZ приходит от переключателя *риска сбоя*. Когда HAZ = 1, все шесть огней включаются и выключаются одновременно. HAZ имеет больший приоритет, чем LEFT или RIGHT. Пусть синхросигнал имеет частоту, равную требуемой для формирования сигналов.

1) Нарисовать граф переходов (8 состояний).

2) Создать схему с шестью D-триггерами. Закодировать состояния так, чтобы каждый непосредственно управлял одним из шести огней.

3) Реализовать схему на трех D-триггерах.

4) Найти компромисс между большим числом триггеров и большим количеством вентилей. Выбрать подходящие ПМЛ или ПЛУ для каждого случая.

5) Написать VHDL-код для варианта 2 и для его моделирования.

4.6.3. Найти минимальное число строк в ПЛМ-таблице для реализации функций:

а) $f_1\ (A, B, C, D) = \sum$ m(4, 5, 10, 11, 12),
$f_2\ (A, B, C, D) = \sum$ m(0, 1, 3, 4, 8, 11),
$f_3\ (A, B, C, D) = \sum$ m(0, 4, 10, 12, 14);

б) $f_1\ (A, B, C, D) = \sum$ m(3, 4, 6, 9, 11),
$f_2\ (A, B, C, D) = \sum$ m(2, 4, 8, 10, 11, 12),
$f_3\ (A, B, C, D) = \sum$ m(3, 6, 7, 10, 11).

4.6.4. В N-битном двунаправленном сдвиговом регистре имеется N параллельных входов данных, N выходов, левый последовательный вход LSI, правый последовательный вход RSI, вход синхронизации и управляющие сигналы. Операция Load – параллельная загрузка данных в регистр, имеет больший приоритет, чем команды сдвига. Rsh – сдвиг регистра вправо. Значение с LSI заносится в левый разряд. Lsh – сдвиг регистра влево. Сигнал RSI поступает в правый разряд.

Чему равно максимальное значение N, если регистр реализован на кристалле 22V10? Привести уравнения для двух самых правых ячеек.

4.6.5. N-битный двоичный реверсивный счетчик реализован на 22V10. Счетчик имеет управляющие входы U и D и вход синхронизации. При U = 1 – выполняется счет на увеличение; D = 1 – счет на уменьшение; U = D = 0 – сохранение состояния; U = D = 1 – запрещенная комбинация. Определить максимальное значение N.

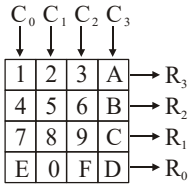
4.6.6. Спроектировать 6-разрядный реверсивный счетчик на 22V10, минимизировав число внешних вентилей. Привести все входные уравнения для триггеров. Написать VHDL-код для счетчика, реализованного с использованием ПЛМ. Протестировать работу счетчика.

4.6.7. Автомат Мили с четырьмя выходами реализован на 22V10. Какое максимальное число входов и состояний он может иметь? Может ли любой Мили-автомат с таким числом входов и выходов быть реализованым на 22V10? Обосновать свой ответ.

4.6.8. Клавиатура имеет четыре ряда и три колонки (см. рисунок 4.27). Одновременно могут быть нажаты не более двух кнопок. Записать 10 первых рядов таблицы истинности для дешифратора клавиатуры, подобной таблице 4.8. Если две кнопки нажаты в одной колонке, N соответствует кнопке в первом из двух рядов.

4.6.9. Разработать поведенческую VHDL-модель сканера клавиатуры, функционально эквивалентную модели с рисунка 4.31, включающую автомат с рисунка 4.30 и таблицу дешифратора 4.8.

4.6.10. Клавиатура 4 x 4 имеет структуру



1) В один момент времени может быть нажата только одна кнопка. Написать уравнения для дешифратора номера, который, получая $R_{3\text{-}0}$ и $C_{3\text{-}0}$, генерирует двоичный эквивалент кнопки. Для кнопки F значение на выходах равно $N_{3\text{-}0} = 1111$ или 15.

2) Спроектировать противодребезговый триггер, определяющий нажатие кнопки. Помехи будут затухать за один или два синхроцикла. Когда кнопка нажата, K = 1 и Kd – отфильтрованный сигнал.

3) Нарисовать граф-схему автомата, выполняющего сканирование клавиатуры и выдающего сигнал наличия нажатия, используя входы из задачи 4.6.10, 2.

4) Записать VHDL-код сканера клавиатуры, включающего декодер, противодребезговую схему и сканер.

4.6.11. Реализовать контроллер светофора с рисунка 4.23, используя 74163-счетчик с дополнительной логикой. ПЗУ применяется для формирования значений на выходах. Написать VHDL-код структурной модели и разработать TestBench.

ГЛАВА 5
СТРУКТУРНЫЕ VHDL-МОДЕЛИ

Представлены технологии создания и использования структурных VHDL-моделей. Рассмотрен редактор Block Editor, предназначенный для их графического ввода, который для введенной схемы автоматически генерирует VHDL-описание устройства.

Структурная модель устройства представляет собой соединение подсистем с помощью сигналов. Каждая из них может в свою очередь состоять из компонентов, объединенных в структурную модель, и так вниз по иерархиии, до тех пор пока подсистемы нижнего уровня не будут представлять собой структурно-неделимые функциональные модели. Существует несколько способов для описания структурных моделей в VHDL-языке.

5.1. Прямая реализация интерфейса

Следует заметить, что слову "реализация" здесь и далее соответствует "instantiation". Для того чтобы создать структурное описание архитектуры, необходимо использовать параллельный оператор, называемый *реализацией компонентов* (component instantiation). Простейший его синтаксис имеет вид:

```
instantiation_label:
   entity entity_name [(architecture_identifier)]
      [port map (port_association_list)];
```

Такая форма обработки компонента называется прямой реализацией интерфейса (direct instantiation of entity). Ее можно рассматривать как создание копии интерфейса, которая подставляется в тело соответствующей архитектуры. При этом карта портов описывает связь портов интерфейса компонента с сигналами архитектуры:

```
Port_association_list <=
   ([port_name] (signal_name | expression | open)){....}
```

Каждый элемент в списке ассоциации "Port_association_list" отображает связь портов интерфейса вложенных модулей с сигналами архитектуры или значениями в выражениях. Можно также оставить порт несвязанным, обозначив его ключевым словом **open**.

Следующий пример иллюстрирует связь портов с сигналами в операторе реализации интерфейса. Пусть интерфейс имеет следующий вид:

```
entity DRAM_controller is
   port (rd, wr, mem: in bit;
      ras, cas, we, ready: out bit);
end entity DRAM_controller;
```

Имя соответствующей интерфейсу архитектуры – **fpld**. Его реализация в архитектуре может иметь вид:

```
main_mem_controller: entity work.DRAM_controller(fpld)
   port map (cpu_rd, cpu_wr, cpu_mem,
      mem_ras, mem_cas, mem_we, cpu_rdy: out bit);
```

В данном примере слово **work** относится к имени текущей рабочей библиотеки, в которой хранятся тела интерфейсов и архитектур. Карта портов содержит список сигналов архитектуры, которые подключаются к портам копии интерфейса. Такой вид

связи называется *позиционной ассоциацией.* При этом каждый сигнал связывается с портом, расположенным в описании интерфейса в той же самой позиции, что и этот сигнал в операторе реализации интерфейса. Таким образом, сигнал **cpu_rd** будет связан с портом **rd**, а сигнал **cpu_wr** – с портом **wr**.

Одним из недостатков позиционной ассоциации является то, что не совсем ясно, какому порту соответствует сигнал из оператора реализации интерфейса. Чтобы узнать это, необходимо искать описание интерфейса и изучать порядок следования портов. Лучшим решением может быть использование именной ассоциации, пример которой показан ниже:

```
main_mem_controller: entity work.DRAM_controller(fpld)
   port map (rd => cpu_rd, wr => cpu_wr, mem => cpu_mem,
      ready => cpu_rdy, ras => mem_ras, cas => mem_cas,
      we => mem_we: out bit);
```

Здесь каждый порт явно связан с сигналом. Порядок следования соединений при использовании позиционной ассоциации портов несущественен.

В предыдущем примере указывалось имя архитектуры, связанной с интерфейсом. Тем не менее в синтаксическом правиле этот параметр указан как необязательный и в данном случае его можно опустить.

На рисунке 5.1 приведена модель четырехразрядного регистра, построенного с использованием функциональных моделей триггеров, синхронизированных по переднему фронту.

Рисунок 5.1. Структурная модель четырехразрядного регистра

```
-- Функциональная модель D-триггера с синхронизацией по переднему
-- фронту
entity edge_triggered_Dff is
   port (D: in bit; clk: in bit; clr: in bit;
        Q: out bit);
end edge_triggered_Dff;

architecture behavioral of edge_triggered_Dff is
begin
   state_change: process (CLK, clr)
      begin
         if clr ='1' then  -- асинхронный сброс clr
            DOUT <= '0';
         elsif (CLK'event and CLK='1') then  -- передний фронт CLK
            DOUT <= DIN;
         end if;
      end process state_change;
end architecture behavioral;


-- Структурная модель четырехразрядного регистра
entity reg4 is
   port(clk, clr, d0, d1, d2, d3: in bit;
      q0, q1, q2, q3: out bit;
end entity reg4;

architecture struct of reg4 is
begin
   bit0: entity work.edge_triggered_Dff(behavioral)
      port map (d0, clk, clr, q0);
   bit1: entity work.edge_triggered_Dff(behavioral)
      port map (d1, clk, clr, q0);
   bit2: entity work.edge_triggered_Dff(behavioral)
      port map (d2, clk, clr, q0);
   bit3: entity work.edge_triggered_Dff(behavioral)
```

```
      port map (d3, clk, clr, q0);
end architecture struct;
```

Разработанную модель регистра можно использовать в архитектуре счетчика, показанного на рисунке 5.2. Пусть каждая десятичная цифра представляется четырехразрядным бит-вектором, описанным как подтип:

```
subtype digit is bit_vector(3 downto 0);
```

VHDL-код счетчика приведен на рисунке 5.3. Он иллюстрирует несколько основных моментов, связанных с оператором реализации и картой портов. Во-первых, два компонента val0_reg и val1_reg являются копиями одной и той же пары интерфейс/архитектура. Во-вторых, в каждой карте порты связываются с отдельными элементами массива. Таким образом, сложный сигнал, например, имеющий тип **массив**, может рассматриваться как несколько одномерных сигналов.

Рисунок 5.2. Двухразрядный десятичный счетчик

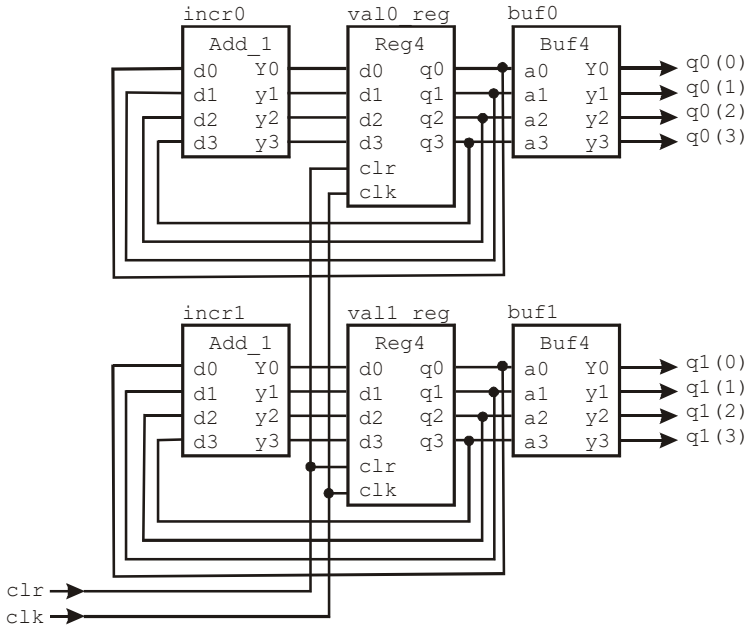


Рисунок 5.3. Структурная модель двухразрядного десятичного счетчика

```
entity counter is
  port (clk, clr: in bit; q0, q1: out digit);
end entity counter;

architecture registered of counter is
  signal current_val0, current_val1, next_val0, next_val1: digit;
begin
  val0_reg: entity work.reg4(struct)
    port map (d0=>next_val0(0), d1=>next_val0(1),
      d2=>next_val0(2), d3=>next_val0(3),
      q0=>current_val0(0), q1=>current_val0(1),
      q2=>current_val0(2), q3=>current_val0(3),
      clk => clk, clr => clr);
  val1_reg: entity work.reg4(struct)
    port map (d0=>next_val1(0), d1=>next_val1(1),
      d2=>next_val1(2), d3=>next_val1(3),
      q0=>current_val1(0), q1=>current_val1(1),
      q2=>current_val1(2), q3=>current_val1(3),
      clk => clk, clr => clr);
  incr0: entity work.add_1 (boolean_eqn)...;
  incr1: entity work.add_1 (boolean_eqn)...;
  buf0: entity work.buf4 (basic)...;
```

```
  buf0: entity work.buf4 (basic)...;
end architecture registered;
```

Допускается также выполнять связь сегмента порта типа **одномерный массив** с сигналом аналогичного типа. Для этого следует указать границы диапазона, например, как это сделано в следующем примере:

```
port map( d(7) => interrupt_reg,
          d(6 downto 4) => interrupt_level,
          d(3) => carry_flag,
          d(2) => negative_flag,
          d(1) => overflow_flag,
          d(0) => program_status, ...
```

Поэлементную связь можно создаватьтакже для портов, имеющих тип **неограниченный массив**. Диапазон типа определяется самым старшим и самым младшим подэлементом массива в реализации компонента. Например, существует вентиль, для которого имеет место интерфейс вида:

```
entity and_gate is
  port (i: in bit_vector; y : out bit);
end entity and_gate;
```

В архитектуре, где предполагается реализовать компонент and_gate, описаны сигналы:

```
signal serial_select, write_en, bus_clk, serial_wr: bit;
```

Тогда оператор прямой реализации интерфейса and_gate может выглядеть следующим образом:

```
serial_write_gate: entity work.and_gate
  port map(  i(1) => serial_select,
             i(2) => write_en,
             i(3) => bus_clk,
             y => serial_wr);
```

Из синтаксического правила видно, что может быть выполнена связь порта с выражением в списке ассоциации портов. В таком случае значение выражения используется как константное значение порта во время моделирования. Если из модели будет синтезировано реальное устройство, порт компонента будет сохранять фиксированное значение, определенное выражением. Ассоциация с выражением полезна, когда используется библиотека готовых компонентов и нет необходимости применять все функциональные возможности каждого из них.

Следующий пример иллюстрирует использование выражений в операторе реализации интерфейса, который для четырехвходового мультиплексора имеет вид:

```
entity mux4 is
  port (i0, i1, i2, i3, sel0, sel1: in bit;
     z: out bit);
end entity mux4;
```

Этот мультипрексор может быть использован как двухвходовой. В этом случае оператор реализации компонентов будет иметь вид:

```
a_mux: entity work.mux4
  port map ( sel0 => select_line, i0 => line0, i1 => line1,
             z => result_line, sel1 => '0', i2 => '1', i3 => '1');
```

Поскольку на старший разряд сигналов sel0 подается значение '0', то доступными для выбора будут только линии i0 и i1. Следуя практическому правилу, для многих

семейств элементов на неиспользуемые входы элемента подается константное значение '1'.

В описании некоторых интерфейсов можно предусмотреть возможность оставлять некоторые порты несвязанными (открытыми); такой порт инициализируется значением, заданным по умолчанию. Если порт остается открытым в списке ассоциации, он обозначается ключевым словом **open.** Например, существует описание интерфейса, в котором указаны значения, принимаемые по умолчанию каждым входным портом:

```
entity and_or_inv is
  port (a1, a2, b1, b2: in bit:='1';
     y: out bit);
 end entity and_or_inv;
```

Реализацию такого компонента можно выполнить следующим образом:

```
f_cell: entity work.and_or_inv
  port map (a1 => A, a2 =>B, b1 => C, b2 => open, y => F);
```

Порт b2 остался открытым, следовательно, он получает значение '1', задаваемое ему по умолчанию.

Выходной порт также может быть оставлен открытым с помощью ключевого слова **open.** В таком случае все значения этого порта игнорируются. Можно оставлять открытым и порт в режиме inout.

Порт, пропущенный в списке ассоциации, также считается открытым, как если бы его связали с ключевым словом **open**. Различие лишь в том, что использование **open** делает код более ясным и понятным для чтения.

5.2. Generic-константы

Generic-константы используются для описания параметров компонентов, которые будут заданы в момент их реализации. Например, времена задержек возрастания и спада сигнала для вентиля могут быть описаны как generic, которым можно назначить различные численные значения в операторе реализации компонентов. На рисунке 5.4 описан двухвходовой вентиль И-НЕ(NAND), для которого времена задержек возрастания и спада зависят от числа, загруженного в вентиль. В декларации интерфейса параметры Trise, Tfall и load определены как константы generic, описывающие незагружаемые параметры задержки возрастания и спада, а также загружаемое число соответственно. В архитектуре задержка возрастания для выхода вычисляется по формуле:

```
Trise + 3 ns * load,
```

где 3 ns – дополнительная задержка к каждой загружаемой.Задержка спада вычисляется по формуле

```
Tfall + 2 ns * load,
```

здесь 2 ns – дополнительная задержка к каждой загружаемой.

Рисунок 5.4. Моделирование задержек возрастания и спада с использованием Generic

```
entity NAND2 is
  generic (Trise, Tfall: time; load: natural);
  port (a,b : in bit;
        c: out bit);
end NAND2;

architecture behavior of NAND2 is
```

```
   signal nand_value : bit;
begin
   nand_value <= a nand b;
   c <= nand_value after (Trise + 3 ns * load) when nand_value = '1'
        else nand_value after (Tfall + 2 ns * load);
end behavior;

entity NAND2_test is
   port (in1, in2, in3, in4 : in bit;
         outi, out2 : out bit);
end NAND2_test;

architecture behavior of NAND2_test is
   component NAND2 is
      generic (Trise: time := 3 ns; Tfall: time := 2 ns;
               load: natural := 1);
      port (a,b : in bit;
            c: out bit) ;
   end component ;
begin
   U1: NAND2 generic map (2 ns, 1 ns, 2) port map (in1, in2, outi);
   U2: NAND2 port map (in3, in4, out2);
end behavior;
```

Интерфейс NAND2_test тестирует компонент NAND2. В его декларации, относящейся к архитектуре, указаны значения, получаемые константами generic Trise, Tfall и load по умолчанию. В реализации U1 описаны другие значения для Trise, Tfall и load. В реализации U2 generic map не описывается, следовательно, в данном случае будут использоваться значения generic-констант, заданные по умолчанию.

Синтаксическое правило задания generic-констант:

```
entity_declaration <=
   entity identifier is
      [ generic (generics_interface_list);]
      [ port (port_interface_list);]
      {entity_declarative_item}
   end [ entity] [identifier];
```

Определение generic происходит до задания портов. Описание объекта в списке generic подобно представлению любого другого объекта интерфейса с ограничениями: объекты класса констант имеют режим in:

```
generic_interface_list<=
   ( identifier {,...}: subtype_indication[:=expression])
{;...}
```

Другой пример использования констант generic – это определение параметров структуры устройства. Generic можно применять для описания ширины портов типа **массив**. Для примера рассмотрим описания регистра, имеющего интерфейс:

```
entity reg is
   port (d:in bit_vector; q:out bit_vector; ...);
end entity reg;
```

Несмотря на то, что это полностью правильное описание, оно допускает, чтобы массивы d и q были различных размеров. Следовательно, реализация такого компонента может выглядеть следующим образом:

```
signal small_data: bit_vector (0 to 7);
signal large_data: bit_vector (0 to 15);
. . .
```

```
problem_reg: entity work.reg
     port map (d => small_data, q => large_date, ...);
```

Ошибка обнаружится только в момент, когда вектор меньшего размера будет присвоен большему. Этого можно избежать, если использовать константу generic для определения размера портов. Тогда описание интерфейса может выглядеть следующим образом:

```
entity reg is
   generic (width: possible);
   port (d: in  bit_vector (0  to width-1);
         q:  out bit_vector (0  to width-1);
         ...);
end entity reg;
```

При таком определении интерфейса требуется, чтобы пользователь указал размер шины при каждой реализации компонента:

```
sigmal in_data, out_data: bit_vector (0 to bus_size-1);
. . .
ok_reg: entity work.reg
   generic map  (width => bus_size)
   port map (d=> in_data, q=> out_data, ...);
```

Если реальные сигналы имеют различную длину массивов, анализатор кода выдаст сообщение об ошибке.

5.3. Использование описания компонента

В примере VHDL-кода (см. рисунок 5.4) использовалось описание компонента, реализуемого в операторе. Описание компонента объявляет интерфейс виртуального проекта, который может быть использован в операторе реализации компонентов. Это альтернативный способ создания структурных моделей. Компонент объявляется в декларативной части архитектуры или пакета.

Синтаксис:

```
component component_name [ is ]
   [generic (generic_list); ]
   [port (port_list); ]
end component [component_name; ]
```

Оператор **компонент** описывает внешний интерфейс элемента в терминах портов и констант generic. Поскольку используется только эта информация, архитектура оказывается полностью независимой от внутренней структуры компонента.

Ниже приведен пример декларации компонента, представляющего собой триггер со входами синхронизации **clk**, сброса **clk**, данных **d** и выходом **q.** Также использованы generic-константы для параметров: задержки распространения, времени установки данных и времени хранения данных:

```
component flipflop is
   generic (Tprop, Tsetap, Thold: delay_length);
   port (clk: in bit; clr: in bit; d: in bit;
      q: out bit);
end component flipflop;
```

Следует обратить внимание на сходство между объявлениями компонента и интерфейса. Это сходство не случайно, поскольку оба описания служат для декларации внешних интерфейсов устройств. Несмотря на явное сходство, в основе их использования лежат разные подходы. Это может быть причиной путаницы у начина-

ющих пользователей VHDL. Тем не менее гибкость, предоставляемая этими двумя конструкциями, является большим достоинством языка VHDL.

Разница между описаниями **entity** и **component** заключается в том, что в их основе лежат разные уровни готовности модели. **Entity**-описание определяет реальное устройство, которое может быть откомпилировано и помещено в библиотеку проекта. **Component**-описание определяет виртуальный модуль, который находится в теле архитектуры. После определения имен, режимов и типов портов виртуального модуля продолжается разработка архитектуры проекта, использующего этот идеальный элемент.

Естественно, такое предположение не делается произвольным образом. Возможно, существует реальный модуль, доступный для использования, и **component**-описание строится, исходя из его свойств. Преимущество данного подхода в том, что идеализированное опускание несущественных деталей реального модуля облегчает работу с проектом. Другой подход состоит в том, что при проектировании сверху-вниз создается виртуальный модуль, который затем станет основой для разработки реального устройства. Естественно, необходимо указывать связь между виртуальным компонентом и реальным интерфейсом.

Реализация компонента

После того, как с помощью объявления компонента выполнено описание внешнего интерфейса устройства, необходимо указать, как модуль используется в проекте. Для этого применяется оператор реализации компонента. Синтаксис такой формы оператора имеет вид:

```
instantiation_label :
   [ component ] component_name
   [generic map ( generic_association_list ) ]
   [port map ( port_association_list ); ]
```

Синтаксис показывает, что в данном случае можно использовать только имя компонента. Если необходимо, можно также указать точные значения generic-констант, а также реальные сигналы, связанные с портами. Метка, используемая для идентификации компонентов, является обязательной.

На рисунке 5.5 приведен VHDL-код 4-битного регистра, иллюстрирующий использование описания компонента и оператора его реализации для построения структурной модели устройства. Созданная на данном уровне модель не включает какого-либо описания о реализации использованного в ней компонента flipflop.

Рисунок 5.5. Структурная модель 4-битного регистра

```
entity reg4 is
   port(clk, clr: in bit; d: in bit_vector (0 to 3);
      q: out bit_vector (0 to 3);
end entity reg4;

architecture struct of reg4 is

component flipflop is
   generic (Tprop, Tsetap, Thold: delay_length);
   port (clk: in bit; clr: in bit; d: in bit;
      q: out bit);
end component flipflop;

begin
   bit0: component flipflop
         generic map  (Trop => 2 ns, Tsetap => 2 ns, Thold => 1 ns);
         port map (clk => clk, clr => clr, d => d(0), q =>q(0));
   bit1: component flipflop
```

```
        generic map  (Trop => 2 ns, Tsetap => 2 ns, Thold => 1 ns);
        port map (clk => clk, clr => clr, d => d(1), q =>q(1));
   bit2: component flipflop
        generic map  (Trop => 2 ns, Tsetap => 2 ns, Thold => 1 ns);
        port map (clk => clk, clr => clr, d => d(2), q =>q(2));
   bit3: component flipflop
        generic map  (Trop => 2 ns, Tsetap => 2 ns, Thold => 1 ns);
        port map (clk => clk, clr => clr, d => d(3), q =>q(3));
end architecture struct;
```

Описание компонента можно размещать в декларативной части пакета. Это может быть полезно, например, если большой проект разрабатывается несколькими проектировщиками. Тогда они могут использовать этот компонент, не включая его описание в создаваемые модели. Таким образом, можно избежать возникновения потенциальных ошибок и неоднозначностей.

5.4. Конфигурация копии компонента

После того как на одном уровне была создана модель, использующая все компоненты, необходимо опуститься на следующий уровень, уточнив реализацию каждой копии компонента. Это можно сделать с помощью конфигурации (configuration declaration). В ней явно указываются имена реальных интерфейсов и архитектур, которые будут использованы для каждой копии компонента. Это называется связыванием (binding) копии компонента с интерфейсом проекта. Нет необходимости выполнять связывание для копии компонента, который реализован с помощью оператора прямой реализизации интерфейса, поскольку в этой форме явно указывается имя интерфейса и архитектуры.

Простейший случай создания конфигурации – это модуль, с которым связывается **component,** имеющий поведенческое описание архитектуры. В этом случае получается один уровень иерархии. Для этой ситуации используется синтаксис:

```
configuration configuration_name of entity_name is
-- configuration declarations
   for architecture_name
      {for component_specification binding_indication;
      end for;}
   end for;

component_specification <=
   (instantiation_label {,...}| others | all ): component_name

binding_indication <= use entity entity_name
[(architecture_identifier)]
```

Имя архитектуры в идентификаторе связи (binding_indication) указывается, если для одного интерфейса существует несколько архитектур. Например, выполнить связь с интерфейсом для копий компонентов **bit0** и **bit1** (см. рисунок 5.5) можно следующим образом:

```
for bit0, bit1: flipflop
   use entity work.edge_triggered_Dff(basic);
end for;
```

Это означает, что копиям компонентов **bit0** и **bit1** соответствует реальный интерфейс проекта edge_triggered_Dff и архитектура basic, находящаяся в текущей рабочей библиотеке.

Следует обратить внимание на то, что связь можно выполнять отдельно для каждой копии компонента и использовать различные имена интерфейсов и архитектур. Главное, чтобы эти интерфейсы имели такие же порты, которые были указаны в

конструкции описания компонента. После того, как были описаны связи для некоторых копий компонентов, для остальных можно применять ключевое слово **others**. Также, если нужно привязать все назначения компонентов к одному модулю, можно использовать ключевое слово **all.**

Следующий пример, приведенный на рисунке 5.6, описывает конфигурацию для 4-разрядного регистра с рисунка 5.5. В ней сделано предположение, что существует библиотека проекта **star_lib**, которая содержит реализацию необходимых для проекта модулей. Оператор use делает интерфейс edge_triggered_Dff видимым для конфигурации.

Конфигурация имеет имя reg4_gate_level и относится к архитектуре struct интерфейса reg4. В архитектуре копия bit0 компонента flipflop связывается с интерфейсом edge_triggered_Dff и архитектурой hi_fanout. Остальные копии компонента связываются с интерфейсом edge_triggered_Dff и архитектурой basic.

Рисунок 5.6. Конфигурация для 4-разрядного регистра с рисунка 5.5

```
library star_lib;
use star_lib.edge_triggered_Dff;

configuration reg4_gate_level of reg4 is
  for struct --архитектура интерфейса reg4
    for bit0: flipflop
      use entity edge_triggered_Dff(hi_fanout);
    end for;
    for other: flipflop
      use entity edge_triggered_Dff(basic);
    end for;
  end for;-- конец архитектуры struct
end configuration reg4_gate_level;
```

Примечание. Если для копии компонента нет конфигурации, то выполняется его связывание по умолчанию. Это значит, что для компонента будет выбран такой модуль, который имеет аналогичные имена интерфейса, портов, generic-констант, какие указаны в его описании. Если интерфейс имеет несколько архитектур, то будет использована последняя, откомпилированная. Инструменты синтеза обычно не поддерживают конфигурацию и требуют выполнять связь по умолчанию.

Большинство реальных проектов имеют многоуровневую иерархическую структуру. Компоненты верхнего уровня включают, в свою очередь, архитектуру, содержащую элементы более низкого уровня. Тело архитектуры, с которым связываются компоненты второго уровня, может также содержать операторы реализации последних. В этом случае в описание конфигурации может быть внесена ссылка на конфигурацию нижнего уровня:

```
binding_indication <= use configuration configuration_name
```

Таким образом, для устройства, использующего в качестве подэлемента с меткой **flag_reg** четырехразрядный регистр, описание конфигурации будет включать следующие строки:

```
for  flag_reg: reg4
  use configuration work.reg4_gate_level;
end for;
```

На рисунке 5.3 был приведен VHDL-код двухразрядного десятичного счетчика, реализованного с применением двуразрядных регистров. Пусть в пакете **couter_types** существует тип **digit,** определенный следующим образом:

```
subtype digit is bit_vector(3  downto 0);
```

На рисунке 5.7 показано, как можно переписать архитектуру схемы, используя описание компонента для регистров. Затем для копий компонентов создается конфигурация (рисунок 5.8). В ней записано, что каждый используемый компонент **digit_register** ограничен конфигурирующей информанией **reg4_gate_level** из текущей библиотеки проекта, представленной на рисунке 5.6. Вторая конфигурация, в свою очередь, точно задает интерфейс (**reg4**), соответствующую ему архитектуру (**struct**) и связывает каждый компонент внутри тела архитектуры. Эти две конфигурации создают двухуровневую структурную схему.

Рисунок 5.7. Двухразрядный десятичный счетчик

```
use workcounter_types.digit;
entity counter is
  port (clk, clr: in bit;
    q0, q1: out digit);
end entity counter;

architecture registered of counter is
  component digit_registered  is
    port (clk, clr: in bit;
      d:  in digit;
      q: out digit);
  end component digit_registered;
  signal current_val0, current_val1, next_val0, next_val1:digit;
begin
  val0_reg: component  digit_register
    port map (clk => clk, clr =>clr, d => next_val0,
      q => current_val0);
  val1_reg: component  digit_register
    port map (clk => clk, clr =>clr, d => next_val1,
      q => current_val1);
--другие операторы реализации компонентов
  . . .
end architecture registered;
```

Рисунок 5.8. Конфигурация для копий компонента digit_register

```
configuretion counter_down_to_gate_level of counter is
  for registered
    for all: digit_register
      use configuretion work.reg4_gate_level;
    end for;
    . . . -- связь для других реализованных компонент
  end for; -- конец конф. информации для архитектуры registered
end configuration counter_down_to_gate_level;
```

Этот пример иллюстрирует использование отдельных конфигураций на каждом иерархическом уровне проекта. И хотя этот способ создания конфигурации является хорошим стилем, существует альтернативный метод, позволяющий задавать все связи в одной конфигурации. В этом случае применяется более сложное синтаксическое правило:

```
configuration_declaration <=
  configuration identifier of entity_name is
    block_conflguration
  end [configuration] [identifier];

block_configuration <=
  for architecture_name
    {for component_specification
      bindmg_mdication;
      [block_configuration]
      end for;}
  end for;
```

На рисунке 5.9 представлена конфигурация, аналогичная приведенной на рисунке 5.8, но в ней явно указаны все связи для всех копий компонентов данного уровня и более низких архитектур. Этот пример показывает, как сложно понимать вложенные архитектуры. Отдельные конфигурации более просты для понимания и более гибки для управления.

Рисунок 5.9. Конфигурация двухразрядного десятичного счетчика с рисунка 5.7

```
library star_lib;
use star_lib.edge_triggered_Dff;

configuration full of counter is
   for  registered    -- ссылка на архитектуру счетчика
      for all: digit_register
         use entity work. reg4(struct);
         for struct    --  ссылка на архитектуру регистра reg4
            for bit0: flipflop
               use entity edge_triggered_Dff(hi_fanout);
            end for;
            for others: flipflop
               use entity edge_triggered_Dff(basic);
            end for;
         end for;      -- конец архитектуры struct
      end for;
      . . .    -- связывание других копий компонентов
   end for;   -- конец архитектуры registered
end configuration full;
```

5.5. Использование редактора Block Diagram Editor

Редактор Block Diagram Editor (BDE) – это инструмент для графического ввода проектов VHDL, Verilog и EDIF. Поскольку здесь и далее рассматривается использование языка VHDL, ниже будут показаны возможности применения этого инструмента для создания VHDL-проектов. С помощью EDIF-формата описываются схемы, полученные в результате синтеза.

Если большая часть проекта представляет собой структурные модели, то может оказаться проще ввести их в виде рисунка, чем вручную набирать сотни операторов. Редактор автоматически преобразует графическое описание в структурный VHDL- или EDIF-код. Система Active-HDL позволяет смешивать различные типы описания. Проект может содержать фрагменты, написанные сразу в HDL-коде, точно так же, как и структурные схемы и графы автоматов. Например, проект верхнего уровня является структурной схемой, в то время как реализуемые компоненты описаны с помощью HDL- или EDIF-кода, или представляют собой граф автомата.

На рисунке 5.10 описаны этапы создания моделей цифровых устройств с помощью редактора Block Diagram Editor.

Рисунок 5.11 изображает окно редактора Block Diagram Editor. Обе панели инструментов окна являются плавающими и имеют кнопки для часто используемых команд. На главной панели расположены кнопки для операций общего редактирования. Панель инструментов с элементами схемы (Diagram Items toolbar) содержит кнопки только для определенных режимов. Графическая и текстовая панели инструментов (Graphics and Text toolbar) имеют кнопки для создания и редактирования графических и текстовых объектов.

Границы страницы определяют размеры печатаемой области. Части диаграммы, которые не помещаются в странице, не будут выведены на печать, за исключением случая применения коэффициента маштабирования менее 100%.

На рабочем поле схемы может находиться информационная таблица, содержащая имя проекта, имя автора, производителя, версию. По умолчанию, таблица помещается в правый нижний угол и содержит несколько конфигурируемых ячеек. Имеется возможность редактирования заданной по умолчанию формы таблицы.

Рисунок 5.10. Этапы создания моделей цифровых устройств в Block Diagram Editor

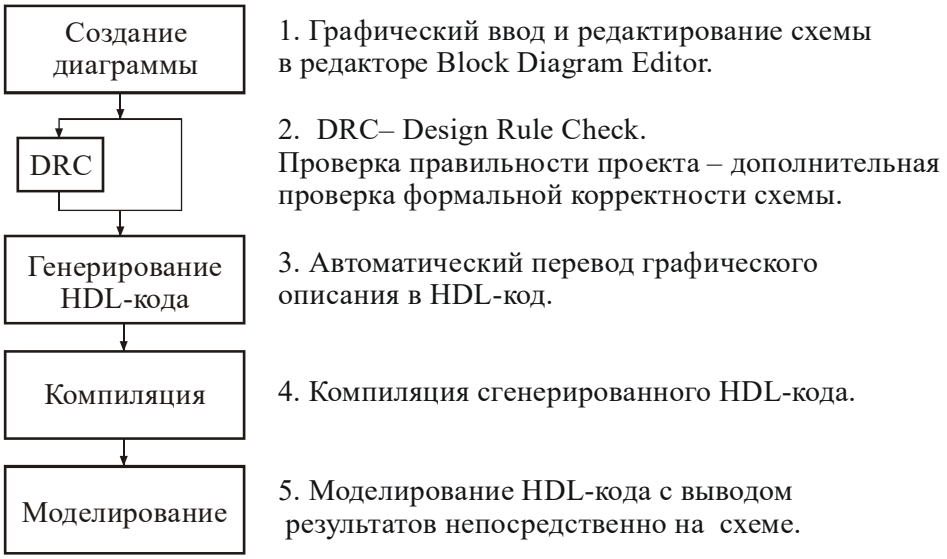


Рисунок 5.11. Окно Block Diagram Editor

Кнопки на главной палитре инструментов

Вырезать выделенные элементы схемы и занести их в Clipboard.

Копировать выделенные элементы схемы в Clipboard.

 Разместить содержащиеся в Clipboard элементы схемы в верхнем левом углу рабочего поля.

 Отменить последнюю операцию редактирования.

 Восстановить последнюю операцию редактирования.

 Войти в режим масштабирования **Zoom**.

 Войти в режим прокрутки **Pan**.

 Увеличение масштаба отображения.

 Уменьшение масштаба отображения.

 Вывести целиком содержимое диаграммы.

 Вывести полностью рабочее поле.

 Генерировать VHDL-, Verilog- или EDIF-код из текущей схемы.

 Вывести HDL-код, сгенерированный для текущей схемы.

 Открыть диалоговое окно Goto Page для перехода на другую страницу многостраничного документа.

 Открыть документ самого верхнего уровня иерархии.

Кнопки инструментальной панели элементов схемы

 Войти в режим выбора (Select).

 Войти в режим создания Fub (Draw Fub).

 Войти в режим Graphical Process.

 Войти в режим Graphical Always.

 Открыть или закрыть окно Symbol Toolbox.

 Режим рисования линий (Draw Wire).

 Режим рисования шин (Draw Bus).

 Режим ввода прикрепленного текста (Attach Text).

    Режим размещения терминалов (Place Terminal) и ввод входных, выходных, двунаправленных терминалов и терминалов типа **буфер** соответственно.

    Режим размещения терминалов (Place Terminal) и ввод входных, выходных, двунаправленных терминалов шин и терминалов шин типа **буфер** соответственно.

Режимы ввода HDL-оператора (Add HDL Statement mode)

 Используется для ввода и редактирования контекстных операторов, предшествующих декларации интерфейса.

 Используется для ввода и редактирования деклараций в теле архитектуры.

 Используется для ввода и редактирования деклараций в интерфейсе.

 Используется для создания параллельных операторов в теле архитектуры.

 Используется для создания дополнительных деклараций, которые будут размещены перед описанием компонента в теле архитектуры.

 Используется для ввода деклараций generic-констант.

 Используется для ввода параллельных операторов в описании интерфейса.

 Используется для ввода и редактирования параллельных операторов, предшествующих телу архитектуры.

 Режим создания символов питания (Place Power Symbol). Размещение VCC- и GND-символа соответственно.

 Режим создания глобальных соединений (Place Global Connector). Создание соединений глобальных линий и шин.

 Войти в режим ввода текста.

 Войти в режим рисования линий.

 Войти в режим рисования кривых Безье.

 Войти в режим рисования прямоугольников.

 Войти в режим рисования эллипсов.

 Войти в режим рисования дуг.

 Возможность добавить рисунок к схеме.

 Открыть или закрыть окно Add New Pin.

 Открыть или закрыть окно Query.

 Открыть окно Objects View.

5.6. Элементы схем редактора Block Diagram Editor

Каждый элемент структурной схемы имеет свой эквивалент в генерируемом VHDL-коде. Например, терминалы (terminals) соответствуют портам интерфейса, символы – реализациям копий компонентов в теле архитектуры.

На рисунке 5.12 изображен пример схемы цифрового устройства с указанием составляющих ее элементов.

Символы

Символ (*symbol*) – это графический эквивалент оператора реализации компонента в теле архитектуры. В VHDL-проекте компоненты должны быть связаны с интерфейсами проектов. Такая связь может быть описана в конфигурации или выполнена по умолчанию. Файл-источник, содержащий описание интерфейса, который будет связан с символом (компонентом в VHDL-коде), называется содержанием (*contents)* символа. Тело архитектуры, соответствующее интерфейсу, называется реализацией (*implementation)* символа. Block Diagram Editor поддерживает следующие типы файлов, реализующие символ: VHDL-, Verilog- или EDIF-код, граф автомата (State diagram), другая структурная схема (Block diagram).

Имя символа идентично имени интерфейса, соответствующего символу. Точно так же имя компонента всегда совпадает с именем интерфейса (рисунок 5.13). Порты компонентов эквивалентны входным/выходным контактам символа, а их имена –

именам портов. После того как символ размещается на схеме, он получает уникальное имя копии (instance name), которое используется для его идентификации в схеме.

Рисунок 5.12. Пример структурно-функциональной схемы

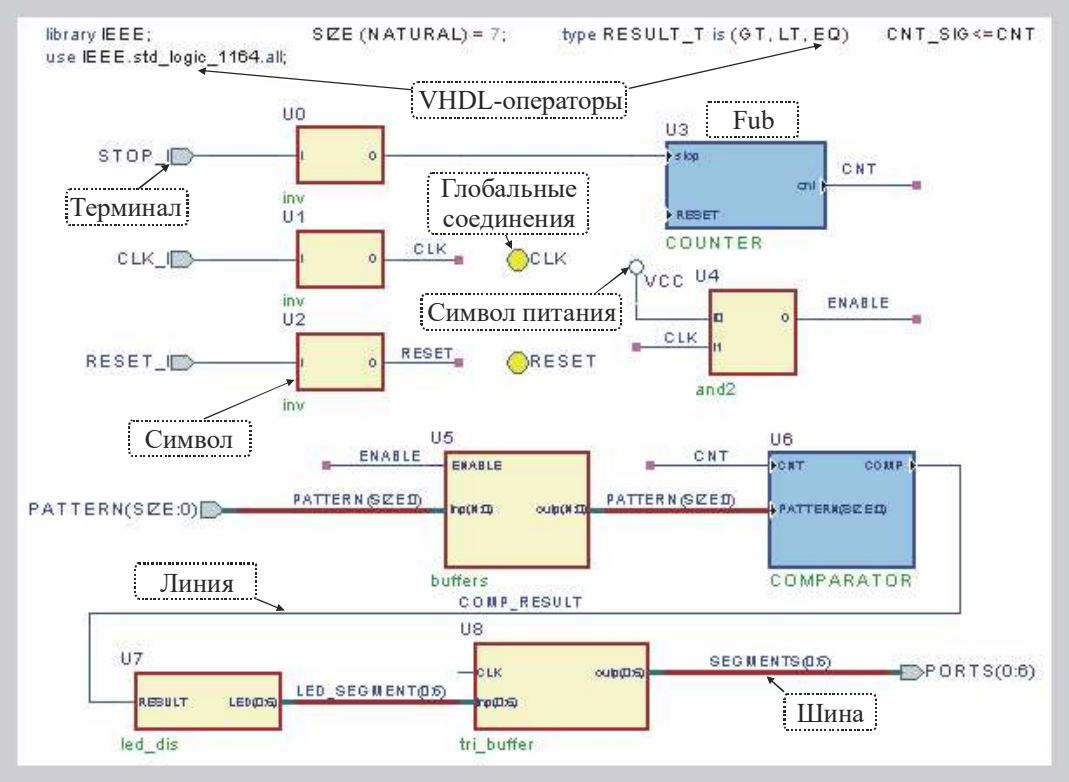
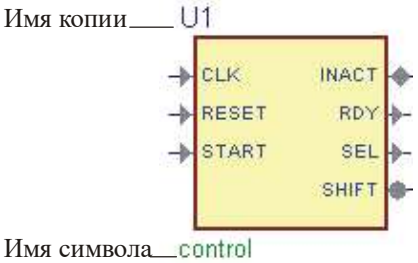


Рисунок 5.13. Пример символа



Большинство символов имеет один файл реализации. Он компилируется в библиотеку и хранится в ней вместе с графическим изображением символа.

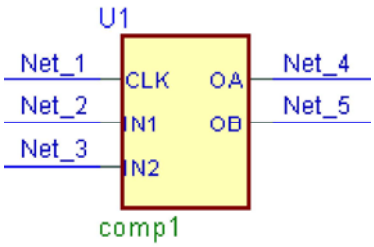
Тем не менее, символ может иметь и несколько реализаций, при условии, что все они основаны на VHDL-описании (исходный код VHDL, структурная схема и граф автомата, преобразованные в VHDL-код). Это связано с тем, что в VHDL интерфейс может иметь несколько альтернативных архитектур. После размещения символа, имеющего несколько реализаций, можно указать активную архитектуру в окне Symbol Properties или не делать этого. Обратите внимание, такое описание относится к копии символа, а не к нему самому. Для неописанной копии символа используется правило связывания по умолчанию, другими словами, он связывается с последней откомпилированной архитектурой.

Block Diagram Editor позволяет создавать символы до выполнения их реализации, которые также хранятся в библиотеке. Они называются пустыми (empty symbols). Если сгенерировать VHDL-код из схемы, содержащей пустые символы, копии компонентов, соответствующие таким символам, останутся не связанными.

На рисунке 5.14 изображен символ и соответствующий ему VHDL-код. Декларативная часть тела архитектуры содержит описание компонента, в то время как ее

операторная часть может включать один или несколько операторов реализации компонентов. Каждая копия компонента соответствует одному символу в схеме.

Рисунок 5.14. Пример символа и соответствующего ему VHDL-кода




```
architecture SAMPLE_ARCH of SAMPLE is
   ...
component COMP1 is
   port (IN1: in STD_LOGIC;
         IN2: in STD_LOGIC;
         CLK: in STD_LOGIC;
         OA: out STD_LOGIC;
         OB: out STD_LOGIC);
end component COMP1;
   ...
begin
   ...
U1: component COMP1
      port map (IN1 => Net_1, IN2 => Net_2,CLK => Net_3,
                OA => Net_4,OB => Net_5);
   ...
end architecture SAMPLE_ARCH;
```

В дополнение к портам символы могут иметь generic-константы. В отличие от портов, они не выводятся в графическом изображении символа, а доступны через окно Symbol Properties.

Инструмент Symbol Toolbox

Окно Symbol Toolbox позволяет размещать символы на диаграмме. Оно содержит список модулей из выбранной библиотеки. По умолчанию, модули выводятся из текущей рабочей библиотеки проекта. Однако имеется возможность использования любой библиотеки, доступной в Active-HDL.

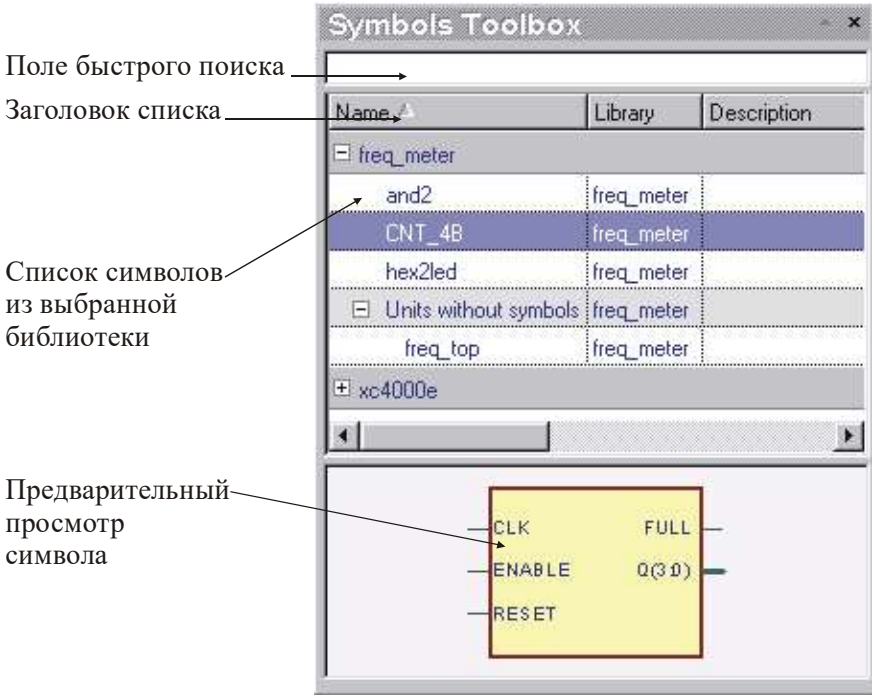
Открыть или закрыть окно Symbol Toolbox можно с помощью кнопки  . На рисунке 5.15 изображено окно Symbol Toolbox. В его верхней части выводится список модулей из выбранной библиотеки. Можно выбирать отображаемые колонки и сортировать список в алфавитном порядке, и наоборот. Нижняя часть окна предназначена для предварительного просмотра выделенного символа.

Для того чтобы разместить выбранный символ на схеме, его надо перетащить из окна Symbol Toolbox на рабочее поле. Если модуль из библиотеки не имеет символа, Block Diagram Editor генерирует его автоматически, в момент первого использования. Модули библиотеки, не имеющие символов, группируются в отдельную секцию в выводимом списке.

Генерация символа выполняется один раз. Если интерфейс, уже имеющий символ, был модифицирован, то последний автоматически не будет обновлен. Редактирование символа необходимо выполнить вручную или с помощью окна Compare Interfaces. Оно открывается командой Compare Symbol with Contents из контекстного меню правой кнопки мыши или Compare Document with Symbol из меню Design.

Можно включать или отключать режим предварительного просмотра символов. Для этого необходимо щелкнуть правой кнопкой по окну Symbol Toolbox и из контекстного меню выбрать команду Preview Visible. Вдавленная иконка рядом с пунктом меню означает, что режим предварительного просмотра обычно активен.

Рисунок 5.15. Окно Symbols Toolbox



Один символ на схеме можно заменить другим, при условии, что символы имеют контакты с одинаковыми именами. Если в новом символе отсутствуют какие-либо входные или выходные линии, соединения будут разорваны. Новые контакты всегда несоединены.

Чтобы заменить символ, необходимо в режиме Select щелкнуть по нему правой кнопкой мыши и выбрать Replace Symbol из контекстного меню. Будет открыто окно Replace Symbol. Выделить более одного символа можно с помощью клавиши Ctrl. Выбрать новый символ в поле New Symbol. Нажать OK.

Встроенные символы редактора Block Diagram Editor

Редактор Block Diagram Editor содержит основные логические элементы, доступные через окно Symbols Toolbox – встроенные символы. При генерации VHDL-модели для структурной схемы, включающей такие символы, они описываются RTL-кодом вместо операторов реализации компонентов. По умолчанию, библиотека встроенных символов (Built-in symbols library) добавлена в список окна Symbols Toolbox (рисунок 5.16). Для их использования не требуются дополнительные библиотеки. Тем не менее, эта упомянутая библиотека не является глобальной Active-HDL-библиотекой или локальной, и ее компоненты доступны только для BDE исходных файлов. Таблица 5.1 содержит список доступных элементов библиотеки встроенных символов (Built-in symbols).

Компоненты Built-in symbols библиотеки имеют также следующие свойства:

1) Генерируемый код может быть записан на VHDL- или Verilog-языке.
2) Компонент может быть соединен со сложной шиной.
3) Компонент может быть использован в виде массива из n копий.
4) Встроенный символ может быть размещен как и BDE стандартные символы с помощью опции Replace Symbol.
5) Нельзя редактировать библиотеку встроенных символов и входящие в нее

компоненты.

6) Эти компоненты могут быть заменены большинством символов из XC4000E библиотеки.

Рисунок 5.16. Окно Symbols Toolbox с библиотекой встроенных символов

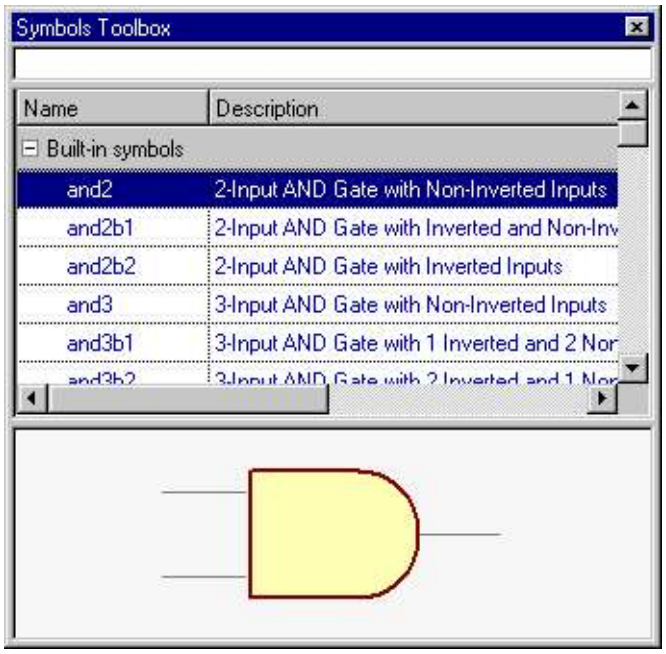


Таблица 5.1. Элементы библиотеки встроенных символов

Элемент	Описание
ANDn	n-входовой элемент И
ANDnbm	n-входовой элемент И с m инверсными входами (n=2-4, m=1-4)
Buf	Буфер
Buft	3-стабильный буфер
INV	Инвертер
M2_1	Мультипрексов 2 в 1
M2_1bm	Мультипрексов 2 в 1 с m инверсными входами (m=1-2)
NANDn	n-входовой элемент И-НЕ
NANDnbm	n-входовой элемент И-НЕ с m инверсными входами (n=2-4, m=1-4)
NORn	n-входовой элемент ИЛИ-НЕ
NORnbm	n-входовой элемент ИЛИ-НЕ с m инверсными входами (n=2-4, m=1-4)
ORn	n-входовой элемент ИЛИ
ORnbm	n-входовой элемент ИЛИ с m инверсными входами (n=2-4, m=1-4)
XOR	2- или 3-входовой элемент сложения по модулю 2
XNOR	2- или 3-входовой элемент сложения по модулю 2, с инверсным выходом

На рисунке 5.17 изображен пример использования встроенных символов при построении структурных схем в редакторе Block Diagram Editor. Данная диаграмма содержит три И-элемента, реализованных с помощью встроенных символов, и три копии обычного символа CNT_4B.

На рисунке 5.18 представлен VHDL-код, сгенерированный для диаграммы с рисунка 5.17. Для символа CNT_4B создано описание компонента, а его копии U1, U2 и U3 реализуются с помощью операторов создания компонента. Копиям встроенных символов U5, U6 и U7 соответствует следующее RTL-описание, представленное параллельными операторами:

```
NET157 <= GATE and NET91;
NET240 <= NET157 and NET212;
NET266 <= NET240 and NET248;
```

Рисунок 5.17. Пример схемы , содержащей встроенные символы

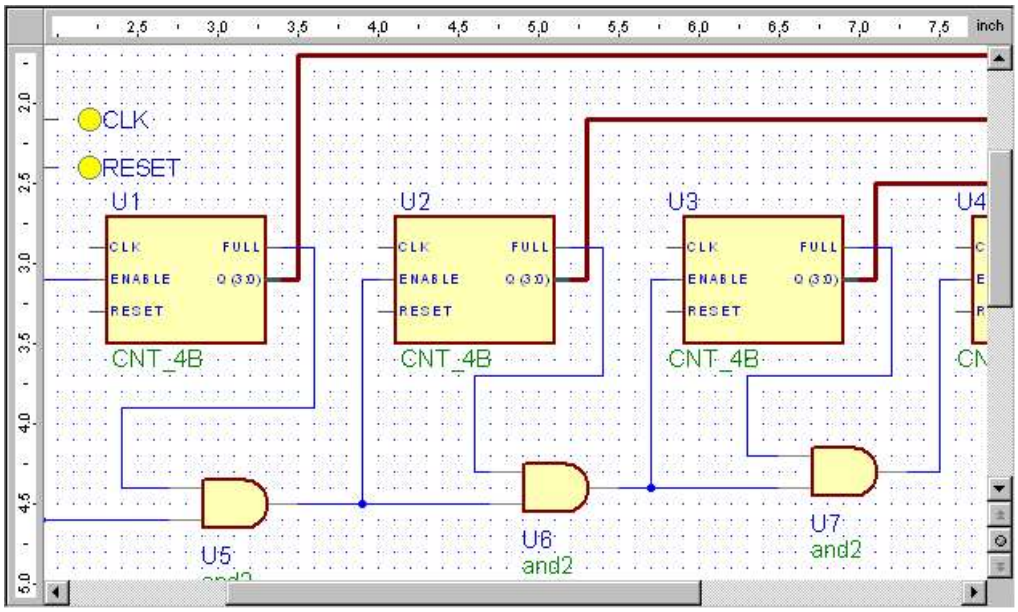


Рисунок 5.18. VHDL-код, сгенерированный для схемы с рисунка 5.16

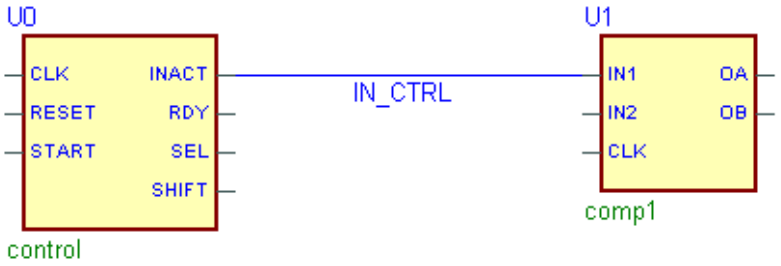
```
library IEEE;
use IEEE.std_logic_1164.all;

entity CNT_BCD is
   port( CLK: in std_logic;
         GATE: in std_logic;
         RESET: in std_logic;
         BCD_A: out std_logic_vector(3 downto 0);
         BCD_B: out std_logic_vector(3 downto 0);
         BCD_C: out std_logic_vector(3 downto 0);
         BCD_D: out std_logic_vector(3 downto 0));
end CNT_BCD;

architecture CNT_BCD of CNT_BCD is
   ---- Описание компонента  ----
   component CNT_4B
      port (CLK: in std_logic;
            ENABLE: in std_logic;
            RESET: in std_logic;
            FULL: out std_logic;
            Q: out std_logic_vector(3 downto 0));
   end component;
   -- Декларация сигналов, использованных в схеме --
   signal NET157: std_logic;
   signal NET212: std_logic;
   signal NET240: std_logic;
   signal NET248: std_logic;
   signal NET266: std_logic;
   signal NET91: std_logic;
begin
-- Операторы реализации компонентов --
U1: CNT_4B
   port map(CLK => CLK, ENABLE => GATE, FULL => NET91,
            Q => BCD_D, RESET => RESET);
U2: CNT_4B
   port map(CLK => CLK, ENABLE => NET157, FULL => NET212,
            Q => BCD_C, RESET => RESET);
U3: CNT_4B
   port map(CLK => CLK, ENABLE => NET240, FULL => NET248,
```

```
          Q => BCD_B, RESET => RESET);
U4: CNT_4B
  port map(CLK => CLK, ENABLE => NET266, Q => BCD_A,
          RESET => RESET);
-- RTL код --
NET157 <= GATE and NET91;
NET240 <= NET157 and NET212;
NET266 <= NET240 and NET248;
end CNT_BCD;
```

Линии (Wires)

Линии используются для соединения символов и fub в схеме. Каждая линия может иметь или не иметь имя, в зависимости от этого они подчиняются различным правилам соединения. На рисунке 5.19 приведен пример линии с именем IN_CTRL.

Рисунок 5.19. Пример линии



В VHDL-описании линии и скалярному сигналу соответствует одинаковое имя. Если оно не определено (безымянная линия), редактор автоматически генерирует имя, основываясь на внутреннем номере линии.

Можно выбирать любой тип данных для генерируемого сигнала. Если требуемый тип не определен явно в пакетах STD.STANDARD или IEEE.STD_LOGIC_1164, его можно создать непосредственно на схеме. Если тип определен в другом пакете, необходимо внести изменения в заголовок VHDL-кода таким образом, чтобы пакет стал видимым. Например, линии IN_CTRL с рисунка 5.18 может соответствовать следующий VHDL-код:

```
architecture SAMPLE_ARCH of SAMPLE is
 ...
signal IN_CTRL: STD_LOGIC;
 ...
begin
   ...
   UO: component CONTROL
        port map (..., INACT => IN_CTRL, ...);
   ...
   U1: component COMP1
        port map (..., IN1 => IN_CTRL, ...);
...
```

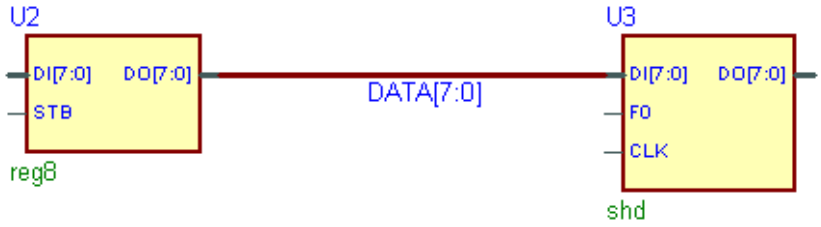
Шины

Шина – это объединение нескольких линий. Она изображается одной жирной линией. Подобно линиям шины используются для соединений символов и fubs в схеме. Имя шины задается так же, как и для линий.

Шинам в VHDL-коде соответствуют сигналы типа **одномерный массив**. Обычно это BIT_VECTOR или STD_LOGIC_VECTOR. Можно произвольно выбирать или определять тип данных, однако он должен быть одномерным массивом. В дополнение к имени и типу для каждой шины задается диапазон, который описывается направлением (**to** или **downto**), левой и правой границами.

На рисунке 5.20 приведен пример шины Data[7:0], соединяющей два символа, и соответствующий ей VHDL-код.

Рисунок 5.20. Пример шины



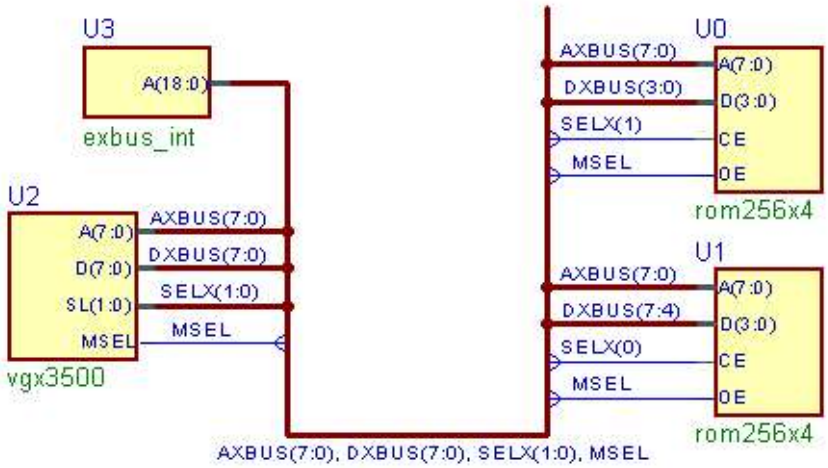

```
architecture SAMPLE_ARCH of SAMPLE is
   ...
signal DATA: STD_LOGIC_VECTOR (7 downto 0);
...
begin
   ...
   U2: component REG8
        port map (..., DO => DATA, ...);
   ...
   U3: component SHD
        port map (..., DI => DATA, ...);
```

Сложные шины (Compound Buses)

Block Diagram Editor позволяет также создавать сложные шины, которые представляют собой объединение отдельных линий и/или шин. Сложные шины, как и простые, изображаются на схеме жирной линией.

Они не представляют собой дополнительные связи и не оказывают влияния на генерируемый HDL-код; также не имеют своего имени. Оно образуется из имен составляющих их линий и шин, записанных через запятую. Пример сложной шины представлен на рисунке 5.21.

Рисунок 5.21. Пример использования сложной шины

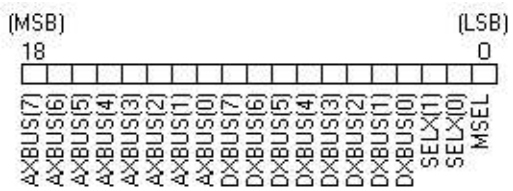


Причиной использования сложных шин может быть:
1) Повышение читаемости цифровой схемы из-за уменьшения числа отдельных линий и шин.

2) Возможность создания соединения между шинным контактом и множественными связями. Например, на рисунке 5.20 изображена шина A(18:) символа U3, соединенная со сложной шиной AXBUS(7:0),DXBUS(7:0),SELX(1:0),MSEL. Это объединение образует следующие дискретные соединения:

```
A(18) – AXBUS(7)   A(10) – DXBUS(7)   A(2) – SELX(1)
A(17) – AXBUS(6)   A(9) – DXBUS(6)    A(1) – SELX(0)
A(16) – AXBUS(5)   A(8) – DXBUS(5)    A(0) – MSEL
A(15) – AXBUS(4)   A(7) – DXBUS(4)
A(14) – AXBUS(3)   A(6) – DXBUS(3)
A(13) – AXBUS(2)   A(5) – DXBUS(2)
A(12) – AXBUS(1)   A(4) – DXBUS(1)
A(11) – AXBUS(0)   A(3) – DXBUS(0).
```

Предполается, что левая составляющая сложной шины содержит ее старший разряд. Исходя из этого допущения, структура схемы может быть представлена следующим образом:



Безымянные шины и линии не могут быть составляющимим сложных шин.

Терминалы (Terminals)

Терминал – графический символ окончания шины или линии, соответствующий порту в VHDL-коде. Ему может быть присвоено имя или он может оставаться безымянным.

Линия или шина, соединенная с терминалом, становится портом, если имя терминала совпадает с одной из них. Иначе, терминал в VHDL-коде связывается с ними с помощью оператора назначения сигнала.

Существует четыре режима терминалов, соответствующих аналогичным режимам портов: **in**, **out**, **inout**, **buffer**. Режим порта **linkage** не поддерживается Block Diagram Editor. За исключением режима, все остальные свойства терминалов аналогичны их линиям или шинам.

Глобальная связь

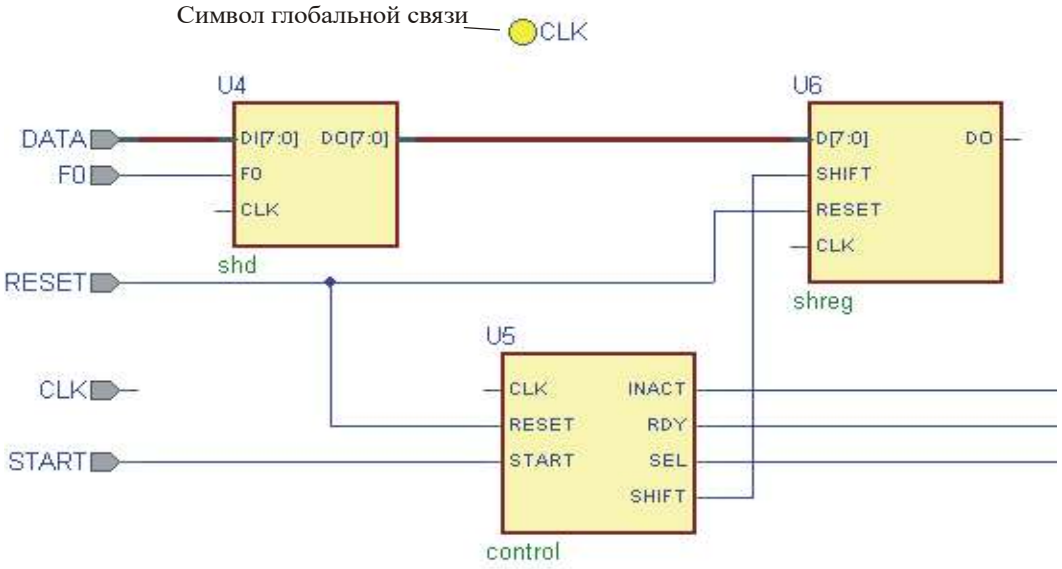
Символ глобальной связи (global connector) создает соединение между всеми несвязанными входными/выходными контактами, имеющими то же имя, что и он. Может использоваться для шин и для линий.

Пример применения символа глобальной связи представлен на рисунке 5.22. Связь CLK объявлена глобальной. В результате ее символ будет соединен с CLK входами U4, U5 и U6 копий символов. Иначе, CLK является фактически внешним входом схемы.

С помощью символов глобальной связи можно создавать соединения для глобальных сигналов, например, таких как сброс, установка и синхронизация. Это упрощает ввод схемы и делает ее более понятной. Однако рамки использования символов глобальной связи ограничиваются диаграммой, в которой они созданы.

Символы глобальной связи представляются в VHDL-коде точно так же, как шины и линии.

Рисунок 5.22. Пример использования символа глобальной связи



Использование Fub

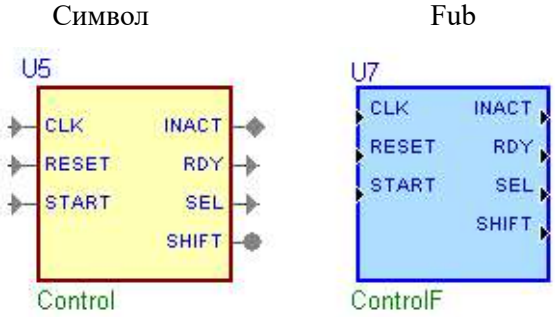
Fub – это графическое представление логических блоков, которые создаются и редактируются непосредственно на диаграмме.

Несколько специфических свойств fub отличает его от обычных символов. Размеры символа fub можно изменять непосредственно на диаграмме, а также добавлять и удалять входные/выходные контакты. Последние создаются автоматически, при соединении линии или шины с fub. Начальное имя контакта аналогично имени подключенной линии или шины. Контакт удаляется автоматически, как только происходит рассоединение с линией/шиной. Это правило не выполняется, если имя контакта отличается от имени линии/шины. Fub может существовать только в единственном экземпляре. Другими словами, на схеме не может быть более одной копии данного fub. По этой причине он не появляется в окне Symbol Toolbox.

Fub может быть применим только при использовании метода проектирования сверху-вниз, поскольку он создается до определения его содержания. Тем не менее он может быть преобразован в обычный символ. Для этого в режиме Select (кнопка ) необходимо щелкнуть правой кнопкой по fub и из местного меню выбрать команду Convert Fub to Symbol.

Графическое представление fub отличается от обычных символов (рисунок 5.23).

Рисунок 5.23. Сравнение изображений символа и fub



При создании проекта fub рисуется прямо на схеме. Затем выполняется подсоединение линий или шин с автоматическим построением входных контактов. Для их определения можно также использовать окно **Add New Pin**.

Каждый fub изначально не имеет какой-либо реализации, т. е. он является пустым компонентом. После того как fub и его интерфейс были описаны, применяется команда Push для создания файла-источника, реализующего fub. Возможные типы исходных файлов реализации аналогичны обычным символам.

Реализация fub компилируется в библиотеку и сохраняется там как откомпилированный модуль с графическим изображением символа fub. В библиотеке модуль, соответствующий fub, имеет специальную отметку, препятствующую выводу fub в списке символов окна Symbol Toolbox. Fub может иметь несколько реализаций, если они основаны на VHDL-описании.

В VHDL-, Verilog- и EDIF-описаниях fub имеет то же представление, что и символ. Как и символы, fub может иметь generic-константы, доступные через окно Fub Properties.

Символы питания (Power Symbols)

Block Diagram Editor предлагает два вида символов питания:

VCC символ – логический уровень единицы.

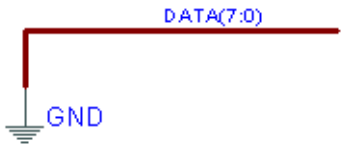
GND символ – логический уровень нуля.

Символы питания определяют связь с именем, таким же как и у символа, задающего константные значения нуля и единицы. По умолчанию, символы питания имеют имена VCC и GND, следовательно, аналогичные имена имеют созданные с символами соединения. Заданные по умолчанию имя и значение символов питания могут быть изменены с помощью диалогового окна Code Generation Settings.

Обычно символы питания соединяют с линиями. Если линия имеет такое же имя, как и символ питания, они составляют единую *power связь*. Если линия и символ питания имеют разные имена, они являются различными элементами и в VHDL-коде реализуются оператором назначения сигнала.

Символы питания могут быть непосредственно соединены с шиной (рисунок 5.24). Поскольку они не имеют диапазона, а всегда представляют дискретную связь, символ питания и шина являются двумя соединенными элементами и в VHDL-коде реализуются с помощью нескольких операторов назначений сигналов.

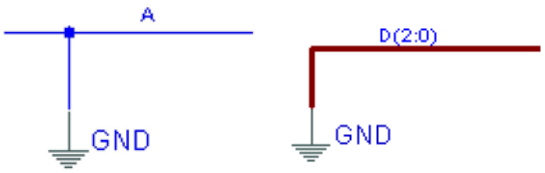
Рисунок 5.24. Соединение шины и символа питания



Если символ питания графически не подключен, он может быть соединен с некоторыми контактами или линиями с помощью имени.

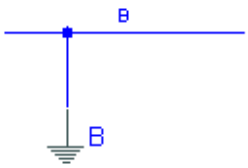
На рисунке 5.25 приведены примеры символов питания и сгенерированных для них VHDL-кодов. В отличие от обычных связей символам питания присваиваются константные значения через операторы назначения сигналов. По умолчанию, VCC – сильная '1' и GND – сильный '0'.

Рисунок 5.25. Представление символов питания в генерируемом VHDL-описании




```
architecture SAMPLE_ARCH of SAMPLE is
...
constant GND_CONSTANT: STD_LOGIC := '0';
signal GND : STD_LOGIC:= GND_CONSTANT;
...
begin
...
GND <= GND_CONSTANT;
A <= GND;
D(0) <= GND;
D(1) <= GND;
D(2) <= GND;
```

а




```
architecture SAMPLE_ARCH of SAMPLE is
...
constant GND_CONSTANT: STD_LOGIC:= '0';
...
begin
...
B <= GND_CONSTANT;
...
```

б

Graphical Process Blocks

Graphical Process text blocks – это специальные текстовые блоки, представляющие собой VHDL-процесс или другие VHDL-операторы, которые непосредственно могут быть добавлены в структурную схему.

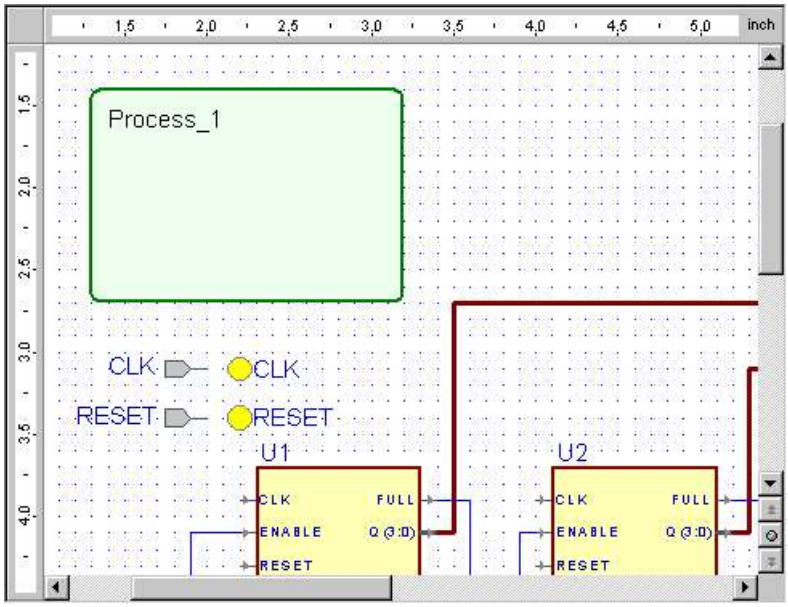
Будет ли использован символ Graphical Process или Graphical Always, зависит от HDL-языка, для которого создается структурная схема. В зависимости от этого панель инструментов с элементами схемы может содержать: Add Process для VHDL или Add Always для Verilog.

Процедура размещения Graphical Process на структурной схеме подобна рисованию fub. Для этого следует щелкнуть по кнопке Add Process и нарисовать квадрат, представляющий Graphical Process. Создать новый Graphical Process можно также с помощью команды Diagram | VHDL | Process.

Graphical Process содержит VHDL-код, который может быть отредактирован непосредственно в окне Block Diagram Editor или в отдельном окне HDL Editor. По умолчанию, блок Graphical Process выводит начальное имя (Process_1, Process_2, ...),

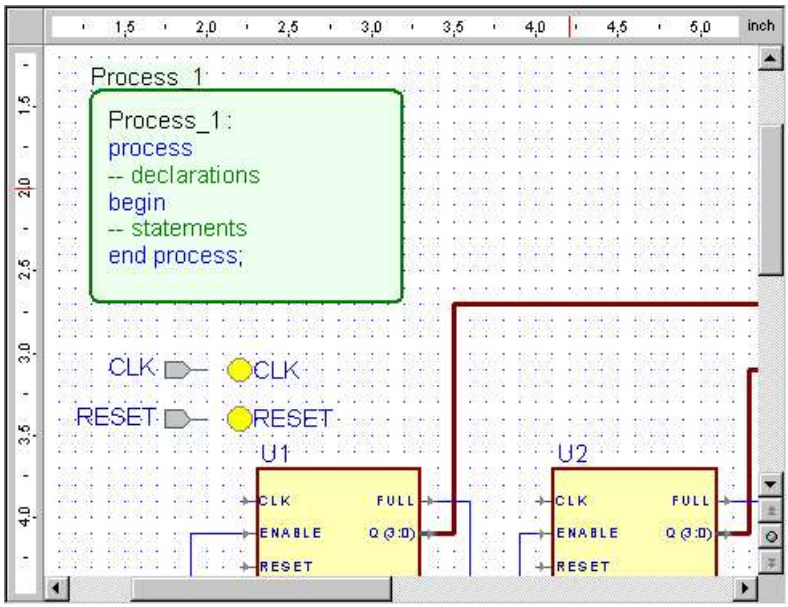
как это показано на рисунке 5.26. Выбрав "Show text" на вкладке General окна Process/ Always Properties, можно сделать HDL-код постоянно видимым на структурной схеме (рисунок 5.27).

Рисунок 5.26. Создание нового Graphical Process



В зависимости от установок, код из Graphical Process может быть модифицирован непосредственно на рабочем поле или в отдельном окне редактора HDL Editor. Если HDL-код видим на диаграмме, то двойным щелчком его можно перевести в режим редактирования. Чтобы открыть окно HDL Editor, необходимо щелкнуть правой кнопкой по блоку Graphical Process и из раскрывшегося меню выбрать команду Edit in Separate Window, если установлена опция Show Text, или Edit – в противном случае. В отдельном окне можно использовать команды редактирования, предоставляемые окном HDL Editor. После внесения изменений в VHDL-код окно HDL Editor может быть закрыто, а изменения сохраняются в исходном файле. Если на вкладке General окна ProcessProperties не выбрана опция "Save text in external file", изменения сохраняются в *.bde-файле, иначе – создается или обновляется внешний *.vhd-файл.

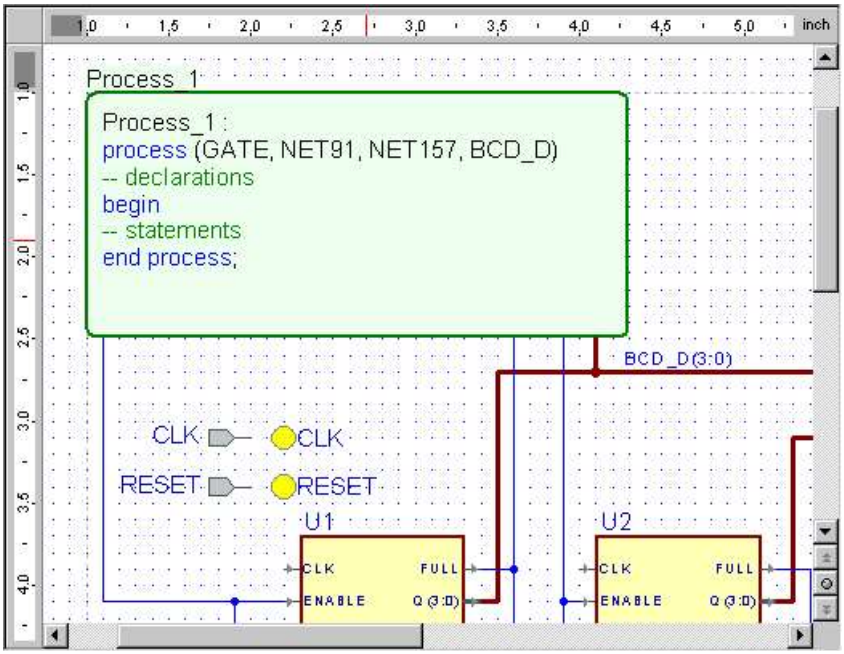
Рисунок 5.27. Graphical Process



После того как специальный текстовый блок, Graphical Process, размещается на диаграмме, он может быть соединен с другими объектами схемы, например с fub, символами, терминалами или другими Graphical Process. Для создания связи могут быть использованы линии, шины, глобальные связи, как это показано на рисунке 5.28.

Сразу после прорисовки связи от любого указаного объекта к Graphical Process имя связывающего сигнала заносится в список чувствительности процесса, если его текст видим, или в список на вкладке Sensitivity List окна Process Properties. Если связь рисуется от Graphical Process к другому объекту, этот сигнал не заносится в список чувствительности, а считается, что он обновляется процессом.

Рисунок 5.28. Пример Graphical Process с выполненными связями



HDL-операторы

Пользователю может понадобиться поместить в генерируемый HDL-код дополнительные операторы, которые нельзя графически представить в виде схемы. Это можно сделать, создав специальные текстовые блоки, содержащие HDL-код. Существует несколько типов таких блоков. Для их создания можно использовать соответствующую кнопку на панели инструментов или команду, доступную в Diagram/VHDL-меню. Поскольку HDL-операторы представляют собой текстовые блоки, их можно редактировать и конфигурировать таким же образом, как и Graphical Process.

Следующий список содержит типы HDL-блоков, возможных для VHDL:

- Заголовок интерфейса (Entity Header) – содержит операторы, предшествующие описанию интерфейса. Может быть создан только один заголовок интерфейса.
- Заголовок архитектуры (Architecture Header) – содержит операторы, предшествующие телу архитектуры. Может быть создан только один заголовок архитектуры.
- Декларация для интерфейса (Declaration for Entity) – содержит дополнительное описание, которое будет добавлено в интерфейс генерируемого VHDL-кода. Может быть создана только одна декларация для интерфейса.
- Декларация для архитектуры (Declaration for Architecture) – содержит дополнительное описание, которое будет размещено в декларативной части архитектуры. Может быть создана только одна декларация для архитектуры.
- Операторы для архитектуры (Statements for Architecture) – содержат дополни-

тельные параллельные операторы, которые будут размещены в теле архитектуры. Число операторов не ограничено.


Операторы для интерфейса (Statements for Entity) – содержат параллельные операторы, которые будут размещены в описании интерфейса. Число операторов не ограничено.


Декларация перед описанием компонента (Architecture declarations before components) – содержит дополнительные декларации, размещаемые перед описанием компонента в архитектуре. Может быть создана только одна декларация.


Generic – включает описание generic-констант, которые будут размещены в описании интерфейса. Число описаний generic-констант не ограничено. Один блок может содержать только одну generic-константу.

Комментарии

Комментарии представляют собой дополнительный текст, добавленный к каждому классу объектов структурной схемы. Объекты: линии, шины, fub, символы, встроенные символы, графически представленные VHDL-процессы, дополнительные VHDL-операторы, глобальные сигналы – могут иметь дополнительное текстовое описание своих свойств на вкладке Comment.

Кроме комментариев, добавленных к объектам структурных схем, можно создавать комментарии, размещаемые в генерируемом VHDL-коде. Комментарии, введенные в окне Preferences (категория: Generation - File Headers - VHDL), используются как заголовок, размещаемый в первых строках генерируемого кода.

5.7. Создание структурных схем

Основные операции для работы с файлами

Создать новую структурную схему можно щелкнув по стрелке кнопки New  и выбрав Block Diagram из открывшегося меню. После этого будет открыто пустое окно Block Diagram Editor. Иначе, можно использовать мастер New Source File Wizard, который вызывается командой New/Block Diagram меню File.

Для того чтобы задать язык, на котором будет генерироваться описание структурной схемы, следует применить команду Set Target HDL из меню Diagram. Доступные варианты: VHDL, Verilog или EDIF.

С помощью команды Diagram/Generate VHDL Code выполняется генерирование HDL-кода для текущей структурной схемы. Команда Diagram/View VHDL Code позволяет открыть HDL-описание для текущей структурной схемы.

Режимы редактирования

При создании структурных схем указатель мыши всегда находится в одном из режимов редактирования, предназначенном для выполнения определенных операций: рисования линий или шин, размещения символов и терминалов. Кнопки, отвечающие за эти режимы, находятся на инструментальной панели элементов схемы.

Рисование и редактирование линий и шин

Для рисования линий (шин) следует перейти в режим Draw Wire (Draw Bus) – кнопка  (). Существует два метода для рисования линий (шин). Первый основан на последовательных щелчках, второй требует удерживать кнопку мыши нажатой во время рисования.

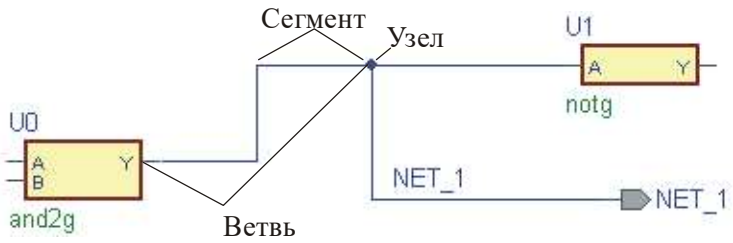
При использовании первого способа щелчком необходимо обозначить начало создаваемой линии, затем переместить курсор к ее концу. Между началом линии и текущим положением курсора будет прорисовываться временная связь. Щелчком ее можно закрепить. Также щелчком обозначается конец рисуемой линии. Для ее завершения на пустом месте рабочего поля делается двойной щелчок.

При использовании второго способа кнопка мыши после первого щелчка не отпускается, курсор перемещается к концу линии, и только после этого кнопка освобождается. При движении курсора между концом и началом линии рисуется временная связь. Если ее необходимо закрепить, следует нажать клавишу Space.

Структура линий (шин)

Графическая структура линий (шин) представляет собой объединение ветвей, соединенных с помощью узлов (junction). Каждая линия (шина) имеет, как минимум, одну ветвь, состоящую из сегментов. Каждая ветвь имеет, как минимум, один сегмент. Линия, пример которой представлен на рисунке 5.29, состоит из трех ветвей и шести сегментов.

Рисунок 5.29. Структура линии

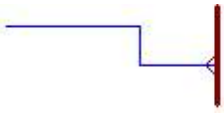


Новая линия не имеет имени. Линия (шина) может начинаться:

1) На пустом пространстве рабочего поля. Такой отрезок считается свободным (hanging).

2) На контакте, терминале, символе питания или символе fub – линия соединяется с этими объектами.

3) На шине (для линии). В таком случае создается ответвление от шины (Bus tap**).**



4) На свободном (несоединенном) конце другой линии (шины). Тогда новая линия (шина) будет представлять единое целое с существующим фрагментом линии (шины).

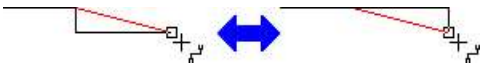
5) На другой линии (шине). В этом случае линия формирует новую ветвь. Если таким образом соединяются две линии (шины), имеющие имена, то создается короткая связь.

Если установлена опция **Autorouting**, редактор автоматически будет рисовать линии, не пересекающие расположенные на диаграмме символы. Внутренний алгоритм вычисляет оптимальную маршрутизацию и рисует временную линию. Эта опция доступна, если линии рисуются удерживанием кнопки мыши.

Может случиться, что для очень сложных структурных схем процесс вычисления оптимальной маршрутизации будет занимать много времени. Поэтому в свойствах есть возможность задать время выхода из алгоритма. По достижении этого времени

процесс вычисления оптимальной линии прекращается и рисуется самая короткая линия. Для того чтобы включить режим Autorouting, а также задать время выхода из алгоритма, используется команда Preferences из меню Tools, чтобы открыть окно Preferences. На вкладке Block Diagram в группе Autorouting выбирается Enabled. Для установки времени выхода используется ползунок Timeout.

Некоторые клавиши клавиатуры облегчают рисование линий. Удерживаемая клавиша Alt временно отменяет опцию Snap to grid (привязать к сетке), если она была установлена в окне Preferences. Tab – создает терминал в месте, где расположен курсор мыши. Можно несколько раз нажимать Tab для выбора желаемого типа терминала. Если указатель мыши находится над прямоугольником fub, Tab позволяет выбрать необходимый тип fub-контакта. Enter – создает зеркальное отображение по диагонали последних двух сегментов линии или шины, как показано на рисунке:



Это свойство доступно, если линия или шина рисуется способом последовательных щелчков и угол между двумя сегментами не был закреплен. Esc – отменяет рисование линии или шины. Backspace – освобождает последний закрепленный угол линии или шины.

Входные и выходные контакты в редакторе Block Diagram обозначаются терминалами, которые создаются в режиме Place Terminal. С помощью щелчка они размещаются на рабочем поле. Терминал по умолчанию получает имя, определяемое редактором. Кроме того, для шины он получает еще и диапазон. Терминал, размещенный на свободном конце линии (шины) или близко к нему, будет автоматически с ней связан и получит ее имя и диапазон.

Изменить имя терминала можно в окне Terminal Properties на вкладке General в поле Name. Для шины можно также задать ее диапазон в поле Range. Иначе – имя терминала можно изменить непосредственно на рабочам поле, отредактировав для этого метку, расположенную рядом с символом терминала. Двойным щелчком по метке выполняется переход в режим ее редактирования. Для выхода используется клавиша **Enter**.

Связи (Nets)

Связь – это логическое соединение между символами на схеме, включая fub и символы питания. Бывает двух видов: дискретная связь и шинная. Представляется линиями и шинами, соответственно. В VHDL связь задается сигналом.

В результате редактирования линий и шин может случиться, что две или более линии (шины), имеющие различные имена, будут соединены через узел – возникнет короткая связь. Такая ситуация приводит к возникновению ошибки, которую необходимо устранять.

Если линия (шина) соединена с терминалом и не имеет имени, они образуют единую связь, имеющую имя терминала. То же самое будет, если и линия (шина), и терминал имеют одинаковое имя:

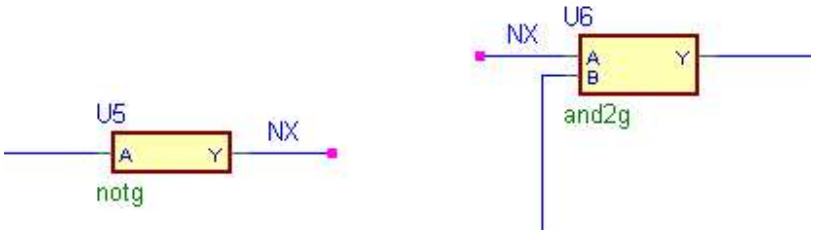
Если же линия (шина) и терминал имеют различные имена, они образуют две различные связи, соединение которых реализуется в VHDL с помощью оператора назначения сигнала:

Если два или более терминала соединены с линией (шиной), не имеющей имени, то линия (шина) и терминалы представляют собой различные связи.

Соединение по имени

Различные несоединенные линии (шины) с одинаковым именем образуют единую связь.Такая связь называется соединением по имени. Она позволяет избежать прорисовки лишних линий и делает схему более понятной. Пример такой связи представлен на рисунке 5.30. Выход инвертора U5 соединяется по имени со входом вентиля U6. Соединением по имени можно выполнять связь между линией (шиной) и терминалом. Это правило также действует для глобальных сигналов и символов питания.

Рисунок 5.30. Соединение по имени



Задание и редактирование имен линий и шин

Считается, что линия (шина) имеет имя, если хотя бы один из ее сегментов в любой из ветвей имеет имя. В свою очередь, сегмент имеет имя, если оно было явно присвоено ему.

Чтобы задать имя сегменту, необходимо перейти в режим ввода прикрепленного текста (Attach Text)  ; щелкнуть по сегменту; в выпадающем меню доступных свойств выбрать NAME. Мерцающий курсор появится на текстовой метке, связанной с сегментом. Метка будет содержать имя линии (шины), если она его уже получила. Если линия (шина) не имеет имени и соединения с каким-либо терминалом, то будет выводиться задаваемое редактором по умолчанию имя. Если линия (шина) не имеет имени, но имеет соединение с каким-либо терминалом, метка будет содержать имя последнего. По желанию можно отредактировать имя, представленное в метке, и нажать Enter, чтобы выйти из данного режима.

Отредактировать имя линии (шины) можно также в окне Wire Properties (Bus Properties), которое вызывается двойным щелчком по необходимому сегменту. Имя вводится или редактируется в поле Segment на вкладке General. Для ввода диапазона шины предназначено поле Range.

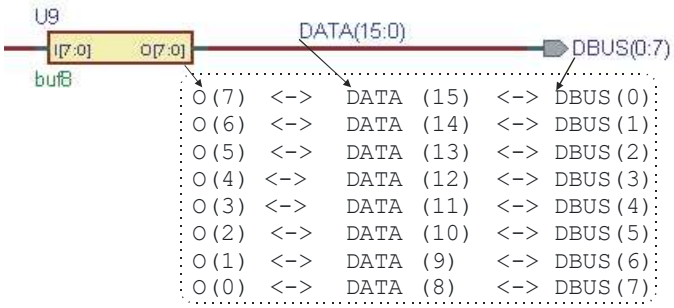
Если линия (шина) имеет более одной метки связи, операция изменения имени коснется их всех. Для того чтобы отменить имя линии (шины), необходимо удалить все ее метки связи.

Индексный диапазон шин

Для шины необходимо указывать индексный диапазон, определяющий ее ширину. Если шина имеет имя, то в этом случае ее метка связи состоит из имени и индексного диапазона.

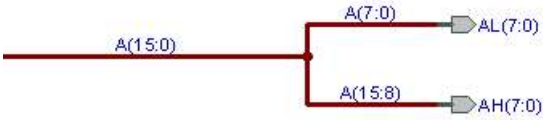
Если шина соединяется с шинным контактом, имеющим другую ширину, то несколько элементов шины или контакта останутся несоединенными. В этом случае выравнивание происходит по левым битам. Это правило действует и при соединении шины с шинным терминалом. На рисунке 5.31 представлен один их типовых примеров такого соединения.

Рисунок 5.31. Пример соединения шины и терминала разной ширины



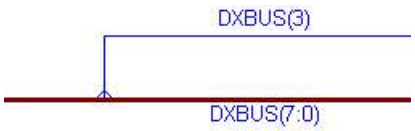
Сектор шины – это подмножество последовательности элементов шины, представленных в том же порядке, что и в оригинале. Пример таких секторов приведен на рисунке 5.32. Два сектора A(7:0) и A(15:8) соединены с 8-битными терминалами, хотя ширина целой шины A(15:0) равна 16 битам. Соединение секторов с шиной выполнено через узел. В данном случае допускается также использовать соединение по имени.

Рисунок 5.32. Использование секторов шины для выполнения соединения с терминалами



Ответвление от шины (Bus Tap) (рисунок 5.33) – это графическое представление соединения между линией и шиной. Линия, соединенная с шиной, должна быть элементом этой шины. Из этого факта вытекает, что линия и шина должны иметь одинаковые имена, а линия – индекс, входящий в диапазон шины.

Рисунок 5.33. Пример ответвления от шины



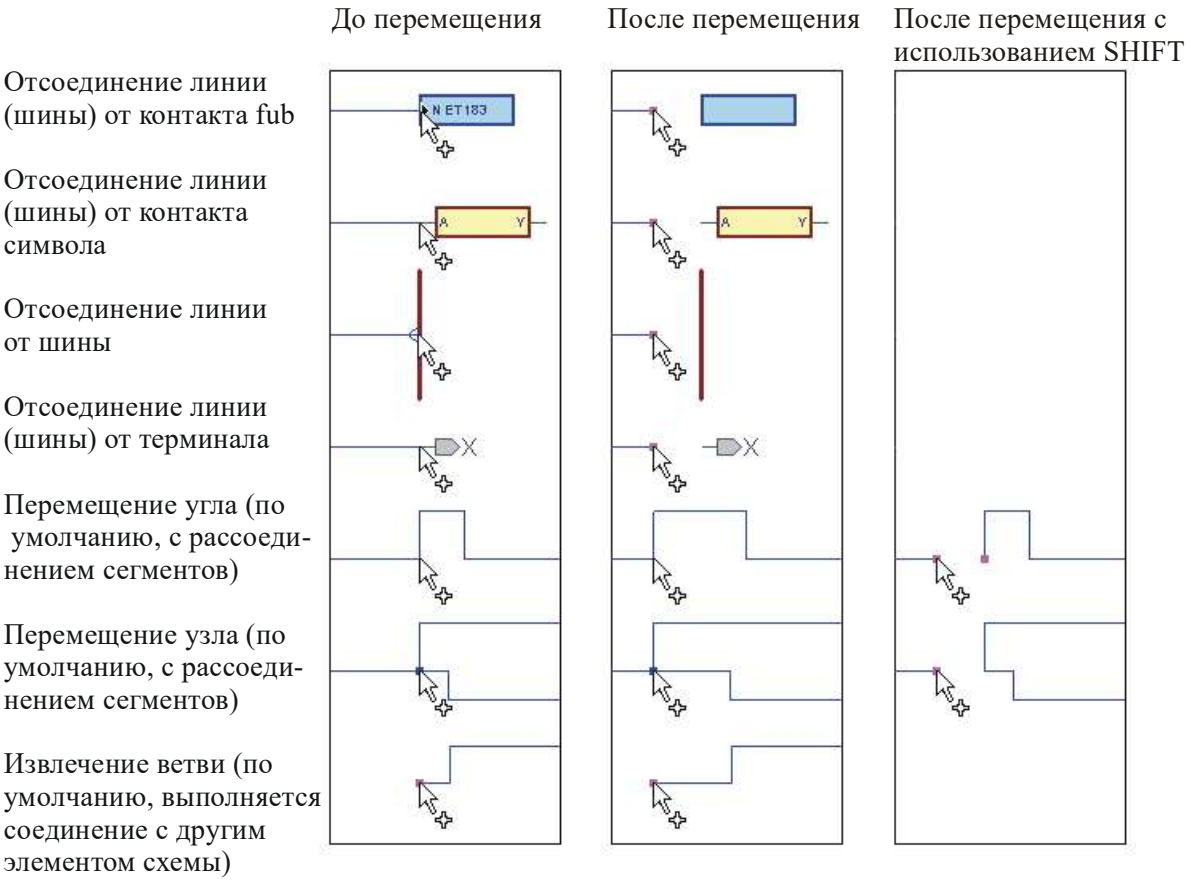
Редактирование линий (шин)

Выделить любой элемент схемы можно в режиме Select  . Щелчком определяется один сегмент линии (шины). Для того чтобы выделить линию (шину) целиком, следует выделить все ее сегменты. Это можно сделать, щелкнув по сегменту линии (шины), удерживая при этом клавишу Ctrl. Или щелкнуть правой кнопкой по линии и из выпадающего меню выбрать команду Select Wire/Bus. Или же выделить любой сегмент линии (шины), а затем щелкнуть по нему повторно.

Если необходимо выделить несоединенные на рисунке линии (шины), представляющие единую связь, следует щелкнуть правой кнопкой мыши по любому ее фрагменту и выполнить команду Select Net из контекстного меню.

Если задержать курсор над концом сегмента линии (шины), будет произведен временный переход в режим редактирования Edit Wire/Bus. В нем можно выполнять перемещение угла, соединение, извлекать или разрывать ветви. Все типы доступных операций изображены на рисунке 5.34.

Рисунок 5.34. Операции редактирования сегментов линий (шин)

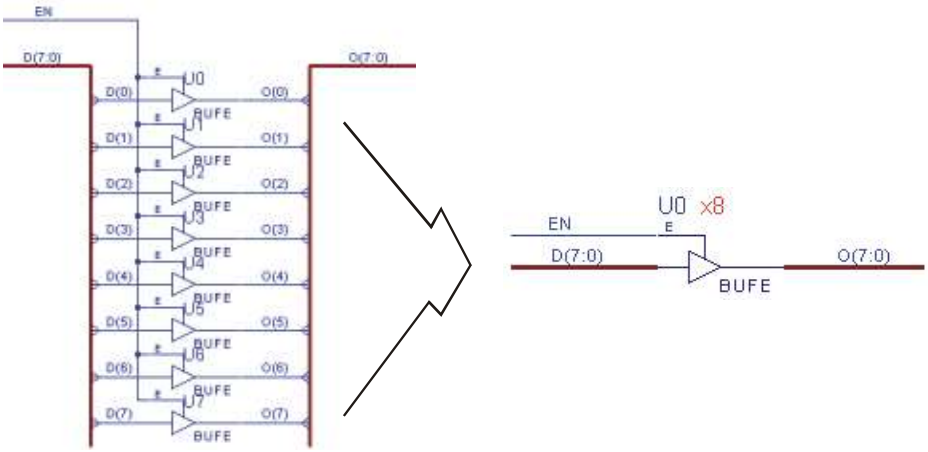


Размещение символов

Для размещения символов используется окно Symbol Toolbox, представленное на рисунке 5.15, которое открывается щелчком по кнопке . Чтобы создать символ, его нужно выбрать и перетащить на рабочее поле.

Иногда необходимо разместить несколько копий одного символа, формирующих массив. Типичный пример такой ситуации приведен на рисунке 5.35. На нем восемь буферов BUFE (U0 –U7), управляемых сигналом EN, отделяют шину D(7:0) от O(7:0).

Рисунок 5.35. Пример массива буферов



Вместо того, чтобы рисовать массив из 8 символов BUFE, можно разместить один символ и указать, чтобы редактор Block Diagram Editor рассматривал его как массив.

Для этого следует открыть окно Symbol Properties, на вкладке General выбрать опцию "Generate array of [ ] symbols" и указать необходимое число копий, которое на схеме выводится рядом с именем копии символа, например “x8”.

Как показано на рисунке 5.35, дискретные контакты символа BUFE объединяются в 8-битную шину. В соотвествующем HDL-коде такие интерфейсы обозначаются как U0_ARRAY<n>, где <n> обозначает номер копии. Нумерация начинается с 0.

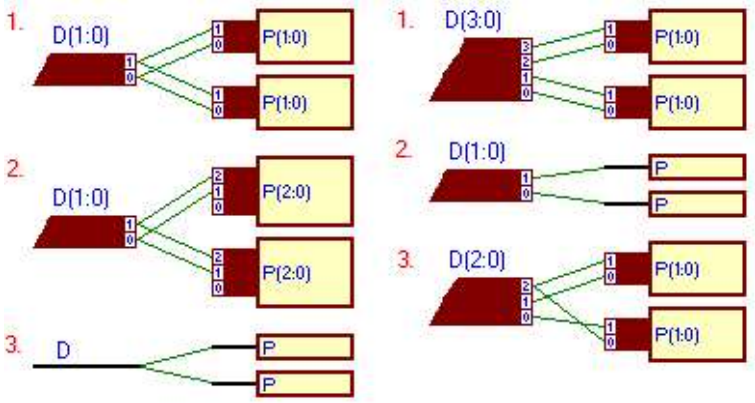
Соединения между контактами массива символов и связями (дискретными или шиной) осуществляются по следующим правилам:

1) Если ширина контактов больше или равна ширине связи (рисунок 5.36, а), контакты каждой копии генерируемых символов соединяются со связью точно так же, как и обычные символы. Другими словами, контакты каждой копии, начиная с крайнего левого, последовательно соединяются с элементами связи, начиная с крайнего левого.

2) Если ширина связи больше ширины контактов (рисунок 5.36, б), контакты копий символа, начиная с крайнего левого символа копии с номером 0, последовательно соединяются с элементами связи, начиная с крайнего левого. Если общее число контактов всех копий массива превышает число элементов связи, то последовательный алгоритм выполняется итеративно, начиная с крайнего левого элемента, как это показано на рисунке 5.36, б, 3.

Дискретная связь (линия) состоит из одного элемента. Ширина шины должна быть задана явно. В этом случае не допускается использование generic-констант.

Рисунок 5.36. Выполнение соединений с контактами массива копий символов



а б

Создание Fub

В отличие от символов создание fub происходит в момент его размещения на диаграмме. Входные и выходные контакты fub формируются путем подключения к нему или отсоединения линий и шин. Иначе, входные контакты можно редактировать в режиме Edit Symbol. Таким образом, имеется возможность создавать контакты, не подключенные к линиям или шинам. Если отредактировать контакты fub, содержание которого уже было определено, может возникнуть необходимость сравнить существующие контакты fub и определенный для него интерфейс. Для этого можно использовать окно Compare Interfaces.

Fub – это компонент, имеющий одну копию. По этой причине его нельзя найти в окне Symbol Toolbox, и хотя fub сохраняется в рабочей библиотеке, он скрыт от пользователя. Можно создать копию fub, используя Clipboard. В таком случае оригинал и копия представляют собой два разных fub и должны иметь различные имена. Если оригинальный fub имеет содержание, копия иметь его не будет.

Если интерфейс fub имеет описания generic-констант, значения им задаются в окне Fub Properties. Открыть окно можно, щелкнув правой кнопкой мыши по символу fub и выбрав команду Properties из контекстного меню. В генерируемом VHDL-коде эти значения будут помещены в карту generic-констант в соответствующем операторе реализации компонентов. Аналогично входным и выходным контактам, при изменении generic-констант для fub, имеющих содержание, необходимо сравнить значения, определенные в fub-символе и в fub-описании. Сделать это также можно в окне Compare Interfaces.

Для того чтобы создать новый символ fub, необходимо перейти в режим **Draw Fub,** кнопка  . Поместить указатель в место, где будет размещен левый верхний угол fub, нажать кнопку мыши и переместить курсор в точку расположения противоположного угла, после этого отпустить кнопку мыши.

Другой быстрый метод рисования fub использует правую кнопку мыши. Если в режиме Select щелкнуть ее правой кнопкой, указатель временно переключится в режим Draw Fub. Теперь можно нарисовать fub, как было описано выше, используя правую кнопку мыши вместо левой.

Изменить размер fub можно в режиме Select. Для этого его нужно выделить, после чего можно редактировать контуры символа.

Чтобы создать контакты для fub, нужно в режиме Draw Wire() или Draw Bus () нарисовать линию или шину, один конец которой размещен на символе fub. Для выбора типа контакта (in, out, inout, buffer) следует использовать клавишу Tab в момент, когда указатель мыши находится над контуром fub.

Если начать рисовать линию или шину с контура fub, нет необходимости переходить в режим Draw Wire или Draw Bus. Вместо этого в режиме Select следует задержать указатель мыши над контуром символа fub, указатель временно переключится в режим Draw Wire, а если нажать Ctrl, то – в режим Draw Bus. После этого можно рисовать линию или шину обычным способом, создавая при этом контакт fub.

Контакты, созданные путем рисования линии или шины, называются автоматическими (*automatic pin)*. Они могут быть удалены, если ликвидировать линии или шины, создавшие их.

Иначе, создавать, редактировать и удалять контакты можно также в режиме Edit Symbol.

Fub можно преобразовать в символ. Для этого в режиме Select нужно щелкнуть правой кнопкой мыши по символу fub и выбрать Convert Fub to Symbol из открывшегося меню.

Редактирование символов и fub

Active-HDL позволяет редактирвоать символы прямо на рабочем поле (in-place editing) или в режиме Symbol Editor.

Редактирование символов и fub на рабочем поле

При редактировании символов и fub непосредственно на схеме необходимо перейти в режим Edit Symbol. Для этого в режиме Select следует щелкнуть по символу правой кнопкой мыши и выполнить команду Edit из контекстного меню. Режим редактирования позволяет изменять размеры, добавлять графические и текстовые элементы, осуществлять редактирование контактов и generic-констант. В этом режиме Block Diagram Editor открывает окно Add New Pin, которое дает возможность выбирать и добавлять контакты к редактируемым символам и fub.

Если символ имеет несколько копий, все они будут обновлены в момент принятия изменений и выхода из режима Edit Symbol.

При редактировании контактов и generic-констант символов/fub не выполняется коррекция их HDL-описания. Проверять согласование изображения символа/fub и их HDL-кода следует вручную, с помощью окна Compare Interfaces. Если не сделать этого, в VHDL-модели, генерируемой на основе схемы, могут быть ошибки.

Такое же несоответствие может возникнуть, если модифицировать описание портов и generic- констант в HDL-коде, описывающем символ/fub. Для решения этой проблемы также следует использовать окно Compare Interfaces.

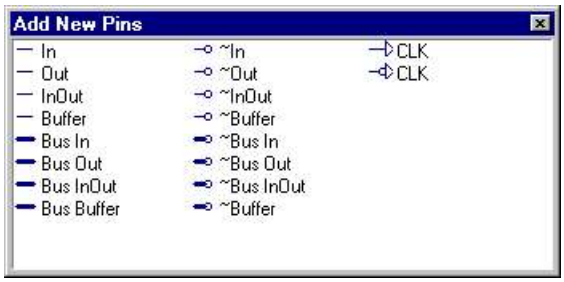
Редактирование пустых символов и fub без реализации является особым случаем. Если такой символ или fub не имеет содержания, его можно редактировать без каких-либо последствий для последнего.

Некоторые свойства fub могут быть отредактированы без перехода в режим Edit Symbol. Например, изменить значение generic-констант можно в окне Fub Properties, а удалить или добавить контакты – путем рисования или удаления линий/шин. Однако добавить или удалить свободные контакты можно только в режиме Edit Symbol.

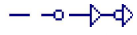
Панель Add New Pin

Панель Add New Pin, окно которой изображено на рисунке 5.37, используется для создания входных/выходных контактов и/или редактирования generic-констант символов и fubs в режиме Edit Symbol. Контакт может быть добавлен к символу с помощью drag-and-drop метода. В режиме Edit Symbol открывать и закрывать окно можно с помощью кнопки  .

Рисунок 5.37. Окно Add New Pin



Панель содержит восемь основных типов контактов: 4 дискретных и 4 шины. Каждый тип, в свою очередь, имеет несколько вариантов. Эти различия имеют место только в графическом представлении. Их кодирование в HDL-коде одинаковое. Некоторые контакты созданы специально для символов, некоторые для fub:

Тип контакта	Для символов	Для fub
in		
out		
inout		
buffer		
in (bus)		
out (bus)		
inout (bus)		
buffer (bus)		

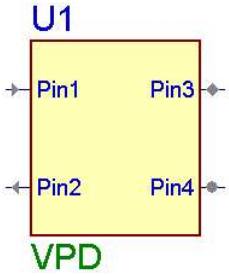
Список контактов, выведенных в окне панели Add New Pin, будет отличаться в зависимости от того, редактируется символ или fub.

Направление порта

Можно определить, будет ли графически изображаться направление портов или нет. Если опция "Pins direction visible" окна Block Diagram Editor Preferences включена,

направление портов будет отмечаться так, как это показано на рисунке 5.38. Они не видимы, если символ редактируется.

Рисунок 5.38. Изображение направления портов



Использование Symbol Editor для редактирования

В дополнение к редактированию символов и fub на схеме Active-HDL предлагает инструмент Symbol Editor. Он предоставляет в основном те же функции, которые доступны в режиме Edit Symbol, и несколько дополнительных свойств.

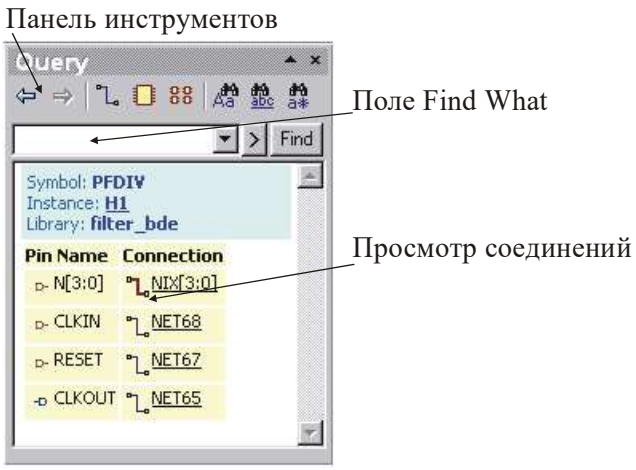
Чтобы открыть окно Symbol Editor, необходимо щелкнуть правой кнопкой по символу и выбрать команду Edit Symbol in Separate Window из раскрывшегося меню или выполнить команду Open Symbol из меню File.

5.8. Окно Query

Окно Query спроектировано для просмотра и поиска соединений между символами, fub и связями на текущей схеме. На рисунке 5.39 изображен внешний вид окна Query.

Открыть или закрыть окно можно командой View/Query или с помощью кнопки  на панели инструментов окна Block Diagram Editor.

Рисунок 5.39. Окно Query



Панель инструментов окна Query имеет следующие кнопки:

 Загрузить предыдущую информацию.

 Загрузить следующую информацию, если перед этим использовалась кнопка, загружающая предыдущую информацию.

 Разрешить поиск элементов диаграммы, представляющих собой связи: линии, шины, терминалы, символы питания и глобальные соединения.

 Разрешить поиск по имени символов и fub.

 Разрешить поиск символов и fub по именам их копий.

При нажатой кнопке будет выполняться поиск текста, имеющего идентичный шаблон больших и маленьких букв, который был записан в поле Find What.

При нажатой кнопке будет выполняться поиск целых слов, не являющихся частью длинных слов.

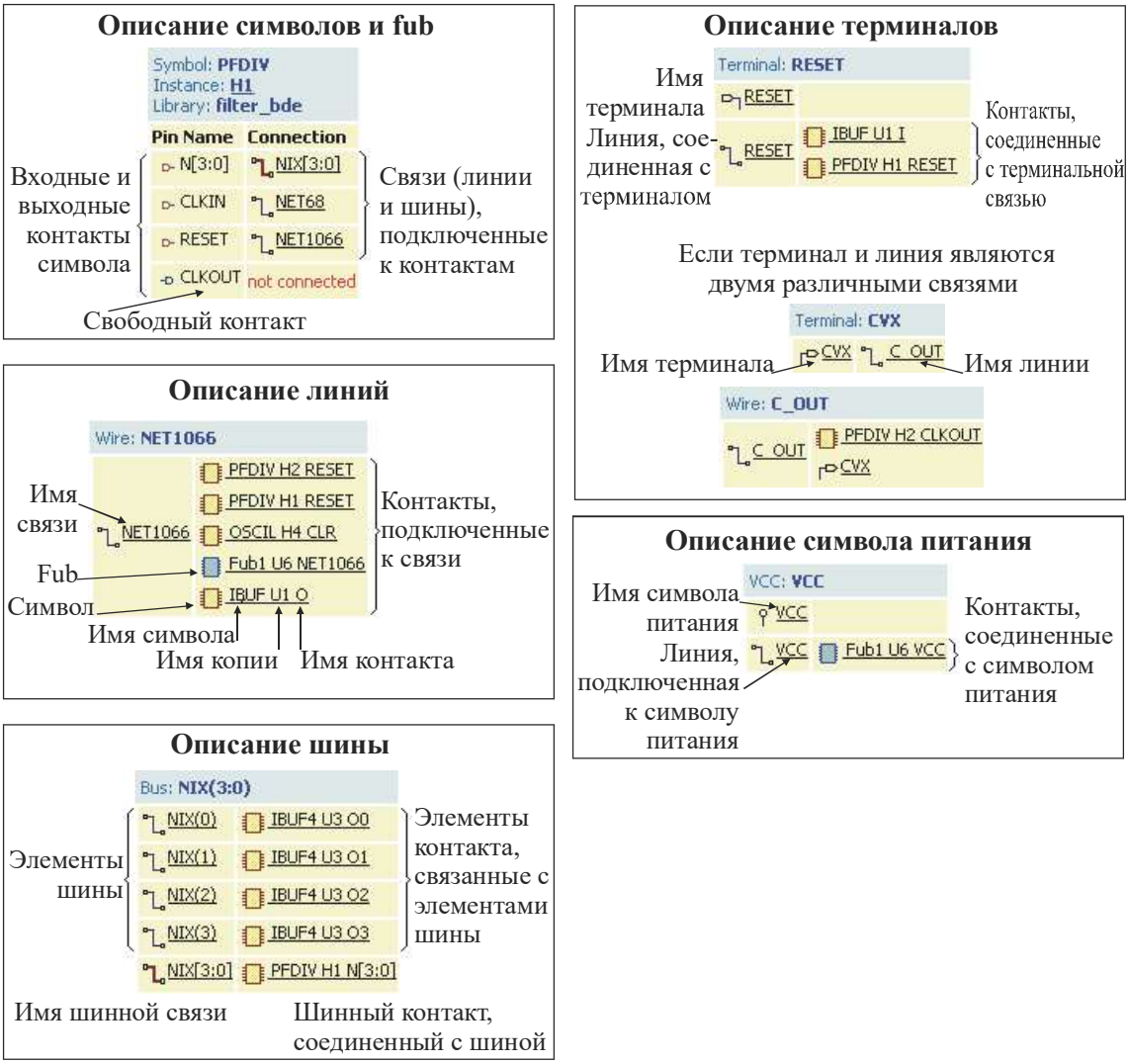
При нажатой кнопке можно вводить регулярные выражения в поле Find What.

Окно Query выводит соединения для элемента схемы, являющегося выделенным в данный момент на ней. Можно выбирать одну или несколько копий следующих объектов: копии символов, fub, линии, шины, глобальные соединения, терминалы, символы питания.

Список соединений в окне Query делится на секции, в которых описываются отдельные элементы схемы.

На рисунке 5.40 представлены примеры типичных соединений.

Рисунок 5.40. Примеры вывода соединений и элементов схемы в окне Query



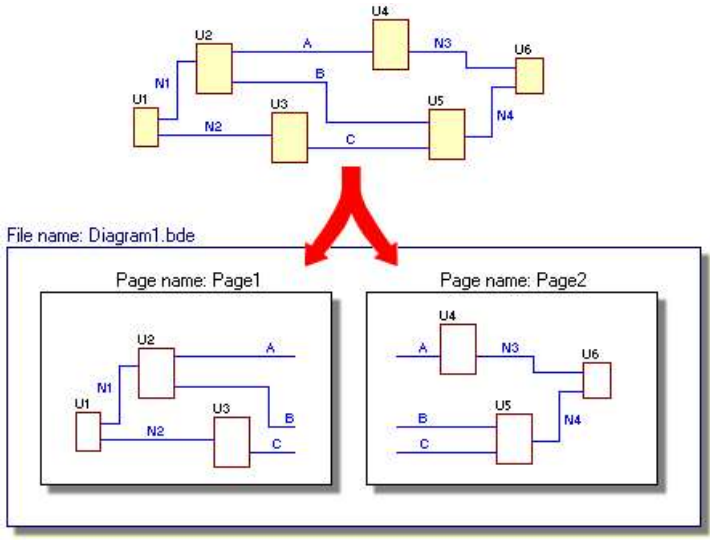
5.9. Многостраничные структурные схемы

Многостраничные структурные схемы представляют собой схемы, состоящие из нескольких страниц, при этом их содержание формирует единую схему. Таким образом, упрощается отображение на экране и обработка больших схем.

На рисунке 5.41 представлен пример схемы, разделенной на две страницы: Page1 и Page 2. Линии A,B,C со страницы Page1 соединяются по имени с идентичными линиями на странице Page 2. Имена копий компонентов должны быть уникальными в пределах всей схемы. Появление одного и того же имени на различных страницах приведет к ошибке.

Многостраничные схемы сохраняются в одном файле.

Рисунок 5.41. Многостраничная схема



Обычно Block Diagram Editor выводит только одну выбранную страницу. Однако при необходимости можно открыть их несколько в отдельных окнах редактора.

Block Diagram Editor предлагает специальные кнопки для управления страницами многостраничных схем. Они размещаются в правом нижнем углу окна (рисунок 5.42) и позволяют открывать следующую и предыдущую страницы, вызывать диалоговое окно, содержащее кнопки:

Добавить новую страницу к текущему документу.

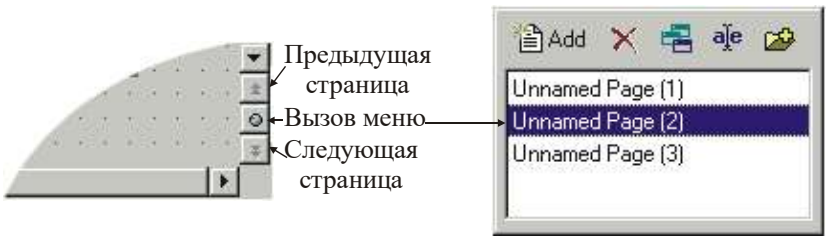

Удалить выделенную в списке страницу.


Открыть выбранную в списке страницу.


Переименовать выбранную в списке страницу.


Добавить к текущему документу новую страницу из файла.

Рисунок 5.42. Управление отображением многостраничных схем



5.10. Проверка правильности проекта

Проверка правильности проекта – Design Rule Check (DRC) – формальная проверка правильности логических соединений между элементами схемы. Сообщения об ошибках и несоответствиях выводятся в окне Console. Каждое DRC-сообщение соответствует одному из двух уровней серьезности:

1) Error. Если было выведено сообщение об ошибке, VHDL-код не будет

генерироваться. Если отключить автоматическое выполнение DRC до генерации кода, его можно получить в любой ситуации.

2) Warning. Предупреждения не мешают генерации кода. Они сигнализируют о потенциальной возможности ошибки в проекте, которая может проявиться на этапе компиляции или моделирования проекта.

Запустить DRC проверку схемы можно командой Diagram/ Check Diagram**.**

Серьезность некоторых DRC сообщений может настраиваться в окне Check Diagram Settings, которое открывается командой Diagram/Check Diagram Settings**.** Можно задать так, что некоторое сообщение будет рассматриваться как ошибка, предупреждение или не будет выводиться вообще. Установки применяются только к открытой в данный момент схеме.

Для того чтобы разрешить или запретить выполнение автоматической проверки до генерации HDL-кода, следует выполнить команду Diagram/Check Diagram Settings. В открывшемся окне на вкладке Generation установить или сбросить флажок "Check diagram before code generation" ("Выполнить проверку до генерации.").

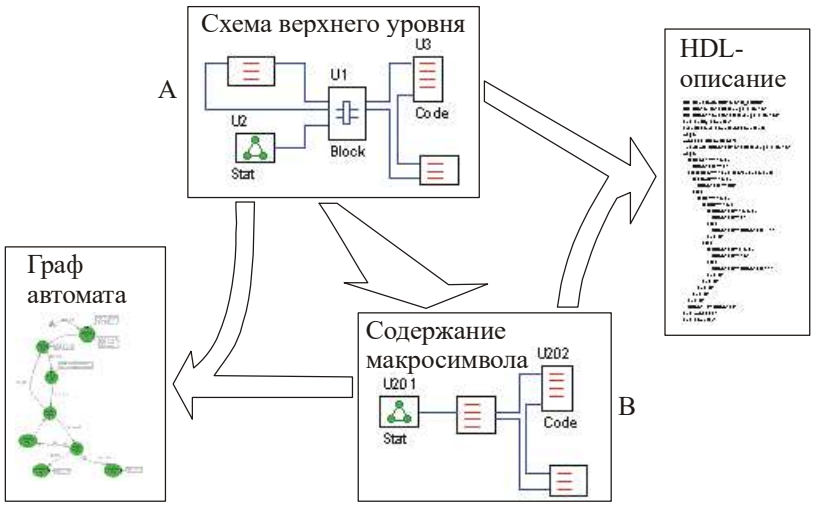
5.11. Создание иерархических структурных схем

При проектировании цифровых устройств дизайнер может использовать два различных способа: снизу-вверх (восходящее проектирование) и сверху-вниз (нисходящее проектирование). На практике обычно используются два этих метода одновременно.

С этой точки зрения символы могут быть разделены на макрос (macros) и примитивы. Примитивные – это символы, содержание которых представлено в виде VHDL-, Verilog- или EDIF-кода, или в виде графа автомата. Макросимвол – это символ, содержание которого описано с помощью структурной схемы. К нему также относится и fub-символ.

Концепция макросов и примитивов иллюстрируется рисунком 5.43. Структурная схема A имеет копию символа U1, содержание которого представлено структурной схемой B. Следовательно, компонент U1 является макросимволом. Схема A соответствует верхнему уровню иерархии , а схема B – второму нижнему. Остальные компоненты схем A и B не могут быть разделены, следовательно, они являются примитивами.

Рисунок 5.43. Пример иерархической структурной схемы



Block Diagram Editor предлагает простой способ для навигации исходных документов, структурных схем, VHDL-файлов, формирующих иерархическую структурную схему. Для того чтобы просмотреть содержимое символа или fub, необходимо щелкнуть по нему правой кнопкой мыши и выбрать из контекстного меню команду

Push. Active-HDL откроет редактор, соответствующий типу исходного документа, который описывает символ. Обратной командой является Hierarchy pop из меню Design или кнопка  . После этого происходит возврат к структурной схеме, содержащей символ.

5.12. Компиляция и моделирование структурных схем

Компиляции структурной схемы должна предшествовать генерация VHDL-кода. Каждый файл, содержащий структурную схему, преобразуется в один VHDL-файл.

Сгенерировать VHDL-код можно командой Generate HDL Code из меню Design. Это также можно сделать в окне Besign Browser, щелкнуть правой кнопкой по имени файла со структурной схемой и из контекстного меню выбрать команду Generate HDL Code.

По умолчанию, процесс компиляции включает этапы:

– проверка правильности проекта;

– генерация VHDL-кода;

– компиляция сгенерированного VHDL-кода.

Запустить компиляцию можно командой Design/Compile или кнопкой  .

Во время моделирования проекта можно наблюдать значения сигналов непосредственно на схеме в редакторе Block Diagram Editor. Вывод моделируемых значений управляется двумя командами из меню Diagram:

– Probes/On Signals – разрешает вывод значений для всех линий и шин.

– Probes/On Pins – разрешает вывод значений для контактов всех символов и fub.

Отметка ✓ рядом с командой меню означает, что она выбрана. Результаты моделирования выводятся, если моделирование инициализировано.

Зондирование (Crossprobing)

Зондирование – это метод локального сравнения объектов из различных документов. В Active-HDL он позволяет пользователю просматривать объекты структурной схемы и соответствующий им сгенерированный VHDL-код. Для этого предлагается команда Show in Generated Code (F3) из контекстного меню. Можно просматривать сгенерированный код для символов, fub, встроенных символов, терминалов, линий, шин, глобальных сигналов, символов питания, HDL-операторов.

5.13. Library Manager

Library Manager спроектирован для управления VHDL-библиотеками. Он позволяет выполнить следующие операции над библиотеками и их содержанием:

1) Создавать новую библиотеку и устанавливать ее в рабочий режим.

2) Подключать, отключать и удалять библиотеки.

3) Редактировать логические имена библиотек.

4) Упаковывать (Compacting) и распаковывать библиотеки.

5) Просматривать содержимое библиотек.

6) Просматривать исходный код модуля, содержащегося в библиотеке.

7) Удалять модули из библиотеки.

Открыть окно Library Manager можно командой Library Manager из меню View или щелчком по кнопке  . Внешний вид окна представлен на рисунке 5.44. Оно содержит две панели. Левая – список подключенных библиотек и их параметры, состоит из следующих колонок:

1) Library (библиотека) – представляет логические имена библиотек.

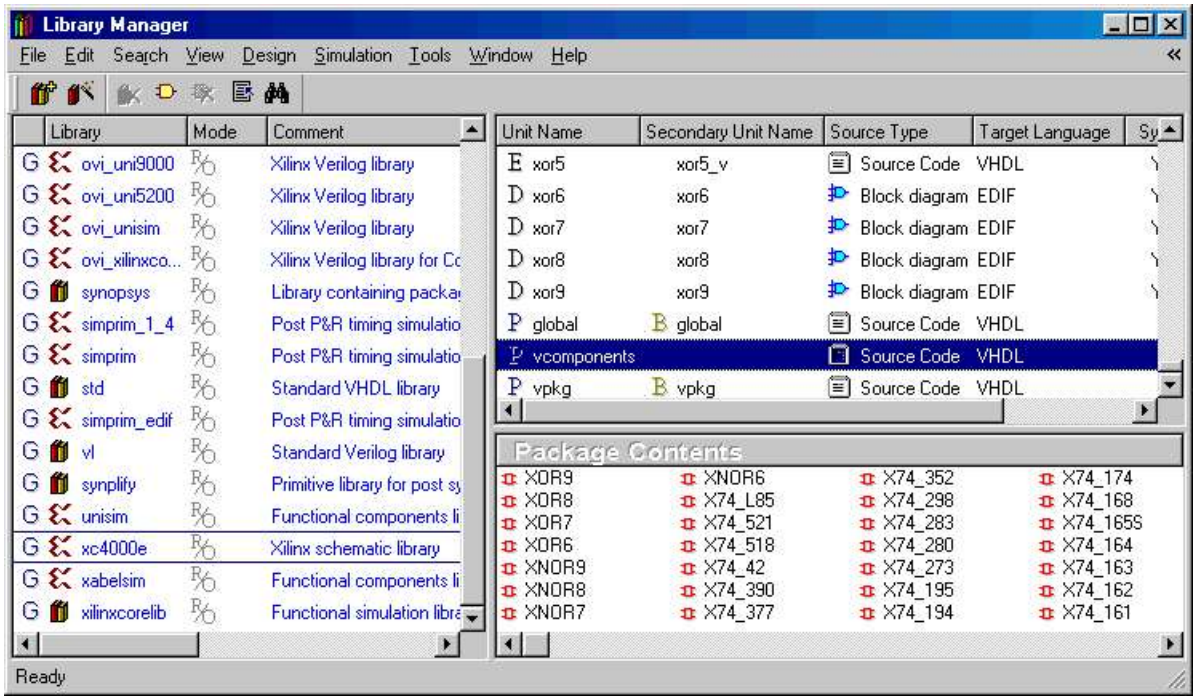
2) Mode (режим) – описывает режим библиотеки, который бывает двух типов: чтение/запись (R/W) и только чтение (R/O).

3) Comment – содержит необязательные комментарии, представляющие короткое описание содержания библиотек.

4) Directory (директория) – выводит полный путь к файлу библиотеки.

Стандартные библиотеки подключаются во время инсталяции. При создании новых происходит их автоматическое подключение. По умолчанию, рабочая библиотека проекта появляется в списке в момент его создания.

Рисунок 5.44. Окно Library Manager



Правая панель представляет модули, содержащиеся в выделенной на левой панели библиотеке. Модули бывают первичные и вторичные. К первичным относятся интерфейс, пакет и конфигурация. Ко вторичным – архитектура и тело пакета. Если выбрать имя пакета, то список модулей, определенных в нем, будет выведен в нижней части правой половины окна. В правой панели имеются колонки:

– Unit Name (имя модуля) – содержит список первичных модулей библиотеки, выделенной в левой панели окна.

– Secondary Unit Name (имя вторичного модуля) – содержит список вторичных модулей библиотеки. Только интерфейс и пакет могут иметь вторичные модули. Для остальных типов модулей в колонке повторяется имя, указанное в предыдущей колонке.

– Source Type (тип источника) – отображает тип исходного документа с описанием модуля, который может быть: Source Code, если описан на языке VHDL или Verilog;Netlist, если исходный файл модуля имеет формат EDIF; State Diagram (граф автомата) или Block Diagram(структурная схема).

– Target Language (язык). Указывается язык, из которого после компиляции была получена библиотека. Если исходный файл был создан в редакторах Block Diagram или State Diagram, колонка отображает язык, который использовался для генерации файла в редакторах. Это может быть VHDL, Verilog или EDIF.

– Symbol (символ) – указывается, если модуль имеет в библиотеке символ для структурной схемы.

–Simulation Data (данные моделирования). Отметка о наличии или отсутствии в модуле данных о моделировании.

Стандартные и определенные пользователем библиотеки отмечаются иконкой . Разработанные производителями (Vendor-specific) библиотеки имеют собственные иконки.

Панель окна Library Manager содержит следующие кнопки:

Создание новой библиотеки с помощью New Library Wizard.

Подключение библиотеки

Отключение библиотеки. Библиотека становится недоступной в среде Active-HDL, но ее файл на диске остается неизменным.

Открыть окно Symbol Editor и загрузить в него символ выбранного модуля.

Показать исходный файл для выбранного модуля библиотеки.

В окне Library Manager используется несколько иконок для отображения содержания библиотек. Для библиотек, описанных на языке VHDL, их четыре:

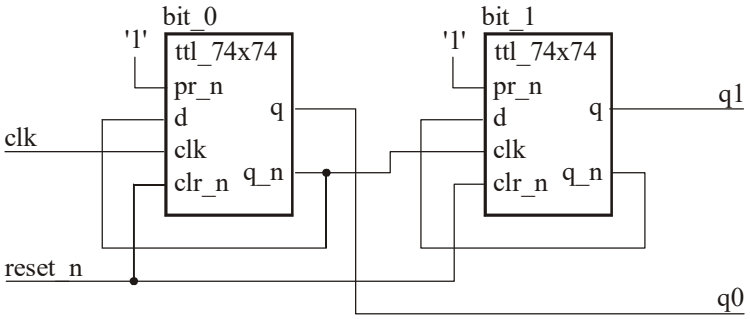
E интерфейс;
P пакет;
C конфигурация;
A архитектура.

Следующие иконки применяются для представления содержания VHDL-пакетов:

f декларация функции,
p декларация процедуры,
декларация компонента,
c декларация константы,
s декларация сигнала,
v декларация общей переменной.

5.14. Задачи

5.14.1. Написать операторы прямой реализации интерфейса, описывающие модель цифрового устройства, изображенного на приведенном ниже рисунке. Пусть интерфейс ttl_74x74 и его архитектура находятся в библиотеке.



5.14.2. Нарисовать схему, эквивалентную следующим операторам реализации компонентов:

```
decode_1 : entity work.ttl_74x138(basic)
     port map ( c => a (2), b => a(1), a => a(0),
                g1 => a(3), g2a_n => sel_n, g2b_n => '0',
                y7_n => en_n(15), y6_n => en_n(14),
                y5_n => en_n(13), y4_n => en_n(12),
                y3_n => en_n(11), y2_n => en_n(10),
                y1_n => en_n(9), y0_n => en_n(8));
decode_0: entity work.ttl_74x138(basic)
```

```
     port map ( c => a(2), b => a(1), a => a(0),
                g1 => '1', g2a_n => sel_n, g2b_n => a(3),
                y7_n => en_n(7), y6_n => en_n(6),
y5_n => en_n(5), y4_n => en_n(4), y3_n => en_n(3),
y2_n => en_n(2), y1_n => en_n(1), y0_n => en_n(0));
```

5.14.3. Разработать структурную модель восьмибитного устройства проверки на четность, используя операторы прямой реализации интерфейса для элементов "исключающее ИЛИ". Устройство имеет 8 входов, i0 – i7, и выход p. Все порты относятся к типу std_ulogic. Логическая функция, описывающая поведение устройства, имеет вид:

$$P = ((I_0 \oplus I_1) \oplus (I_2 \oplus I_3)) \oplus ((I_4 \oplus I_5) \oplus (I_6 \oplus I_7)).$$

5.14.4. Разработать поведенческую модель 4-битного счетчика с синхронизацией по заднему фронту и с асинхронной параллельной загрузкой. Описание интерфейса имеет вид:

```
entity counter is
     port (  clk_n, load_en: in std_ulogic;
             d: in std_ulogic_vector(3 downto 0);
             q: out std_ulogic_vector(3 downto 0));
end entity counter;
```

Используя данное описание, разработать структурную модель 14-битного счетчика с параллельной загрузкой. Удостовериться, что все неиспользованные входы правильно получили константные значения.

5.14.5.Написать декларацию компонента для компаратора, выполняющего сравнение величин двоичных чисел. Два двоичных вектора a и b являются входами компоратора. Их длина описывается с помощью genetic-константы. Значения двух выходов указывают, что a = b и a < b. Компонент также включает genetic-константу, описывающую задержку распространения.

5.14.6. Написать оператор реализации компонента, описанного в задаче 5.14.5. Входные сигналы подключаются к current_position и upper_limit. Выход, означающий a < b, соединяется с position_ok, второй выход остается открытым. Задержка распространения 12 ns.

5.14.7. Написать декларацию пакета, в котором определен подтип для восьмибитного натурального числа, и декларацию компонента для сумматора слагаемых данного подтипа.

5.14.8. Существует архитектура цифрового фильтра, имеющая следующий шаблон:

```
architecture registeMransfer of digital_filter is
. . .
   component multiplier is
      port(...);
   end component multiplier;
begin
   coefMjnultiplier: component multiplier
      port map (...);
end architecture register_transfer;
```

Написать конфигурацию декларации, которая связывает копию компонента мультиплексора multiplier с интерфейсом fixed_point_mult из библиотеки dspjib и архитектурой algorithmic.

5.14.9. Пусть библиотека dspjib из задачи 5.14.8 включает конфигурацию интерфейса fixed_point_mult с именем fixed_point_mult_std_cell. Написать альтернативную декла-

рацию конфигурации для фильтра, описанную в задаче 5.14.8, используя конфигурацию fixed_point_mult_std_cell.

5.14.10. Модифицировать шаблон архитектуры фильтра из задачи 5.14.8 таким образом, чтобы в нем был оператор прямой реализации компонента fixed_point_mult_std_cell без использования описания компонента multiplier.

5.14.11. Декларация компонента и оператор, реализующий его в теле архитектуры, имеет вид:

```
component multiplexer is
  port (s, d0, d1: in bit; z: out bit);
end component multiplexer;

serial_dataj7iux: component multiplexer
     port map ( s => serial_source_select,
                d0 => rx_data_0, d1 => rx_data_1,
                z => internal_rx_data);
```

Написать конфигурацию, которая связывает реализацию компонента со следующим интерфейсом из рабочей библиотеки. Будет использоваться последняя анализируемая архитектура. Задержка распространения устанавливается равной 3,5 ns:

```
entity multiplexer is
     generic (Tpd: delay_length:= 3 ns);
     port (s, d0, d1: in bit; z: out bit);
end entity multiplexer;
```

5.14.12. Пусть в библиотеке gate_lib описан интерфейс nand4:

```
entity nand4 is
  generic (Tpd_01, Tpd_10: delay_length := 2 ns);
  port (a, b, c, d: in bit := '1'; y: out bit);
end entity nand4;
```

Связь интерфейса с копией компонента выполняется с помощью конфигурации:

```
for gatel: nand3
  use entity get_lib.nand4(basic);
end for;
```

Написать карты generic-констант и портов, которые включают связывание по умолчанию, использованное в конфигурации.

5.14.13. Разработать структурную модель 8-битного последовательного на входе и параллельного на выходе регистра на основе таких же 4-битных регистров. Включить описание компонента для 4-битного регистра; 4-битный регистр имеет вход синхронизации по переднему фронту, инверсный асинхронный сброс, последовательный вход данных и четыре параллельных выхода данных.

5.14.14. Разработать пакет, содержащий декларацию компонентов для двухвходовых логических элементов и инвертора, соответствующих логическим VHDL-операторам. Каждый компонент имеет порты типа bit и generic-константы, описывающие задержки возрастания и спада для выхода (generic constants for rising output and falling output propagation delays).

5.14.15. Разработать структурную модель 32-битного двунаправленного трансивера, использующую как компонент функциональную модель 8-битного трансивера. 8-битный трансивер имеет два двунаправленных входа данных, a и b. Асинхронный, активный по низкому уровню порт разрешения выходов, oe_n. Порт направления dir. Когда oe_n='0' и dir='0', данные передаются с b в a; если oe_n='0' и dir='1', то данные

передаются с a на b. Если oe_n='1', то a и b имеет значение высокого импеданса. Задержка распространения для выходов равна 5 ns.

5.14.16. Написать конфигурацию для 32-битного трансивера из задачи 5.14.15, которая связывает каждую копию компонента 8-битного трансивера с интерфейсом 8-битного.

5.14.17. Используя пакет, созданный в задаче 5.14.14, разработать структурную модель полного сумматора, описываемого уравнениями:

$$S = (A \oplus B) \oplus Cin$$

$$Gout = AB + (A \oplus B)Cin.$$

Написать поведенческие модели интерфейсов, соответствующих каждому компоненту логических элементов, и разработать конфигурацию, связывающую каждую копию компонента из полного сумматора с соответствующим интерфейсом логического элемента.

5.14.18. Разработать структурную модель 4-битного сумматора, используя компоненты полного сумматора. Написать конфигурацию, описывающую связь каждой копии полного сумматора и использующую конфигурацию из задачи 5.14.17. Как альтернативу, написать полную конфигурацию 4-битного сумматора, без использования других конфигураций.

5.14.19. Разработать поведенческую модель памяти RAM с адресными входами, входами и выходами данных, имеющих тип bit-vector. Размер портов в интерфейсе должен описываться с помощью generic- констант. Затем разработать TestBench, включающий описание компонента RAM без generic-констант с фиксированным размером портов адресов и данных. Написать конфигурацию для TestBench, связывающую RAM-интерфейс с копией RAM-компонента, используя локальный размер компонента для определения величин формальных generic-констант интерфейса.

5.14.20. Мажоритарная функция трех переменных может быть описана уравнением:

$$M(a, b, c) = abc + ab\overline{c} + a\overline{b}c + \overline{a}bc\,.$$

Разработать структурную трехвходовую мажоритарную схему, используя компоненты инвертора, элементов И и ИЛИ. Разработать поведенческие модели для инвертора и вентилей, применяющих generic-константы для описания задержек нарастания и спада выхода. В структурную модель включить описание конфигурации, выполняющей связь копий компонентов с интерфейсами, а также устанавливающей значение задержки равной 2 ns для всех вентилей.

Затем разработать конфигурации для мажоритарной схемы, которые включают дополнительные связи для переопределения задержек следующим образом:

	задержка возрастания	задержка спада
inverter	1.8 ns	1.7 ns
and gate	2.3 ns	1.9 ns
or gate	2.2 ns	2.0 ns

5.14.21. Разработать модель цифрового секундомера. Схема имеет три входа: 100 kГц синхровход, переключающие входы start/stop и lap/reset. Два последних входа в нормальном состоянии имеют значения '1' и переключаются в '0' при нажатии внешней кнопки. В схеме есть выходы для управления внешним семисегментным дисплеем минут, секунд и сотых долей секунд (''), форматированных следующим образом:

В устройстве имеется по одному выходу для управления каждым индикатором минут (') и секунд ("). Когда выход имеет значение 1, соответствующий сегмент и индикатор светится; когда выход равен нулю, он выключен. Секундомер содержит счетчик времени, считающий минуты, секунды и сотые доли секунд. Счетчик секундомера изначально сбрасывается в 00'00"00, эта же информация отображается и на дисплее. В таком состоянии нажатием кнопки start/stop выполняется инициализация счета, на дисплее отображается текущее время. Повторное нажатие кнопки останавливает счет. Затем последовательное нажатие кнопки будет возобновлять или останавливать счет. Если кнопка lap/reset будет нажата в момент остановки счета, дисплей будет отражать состояние счетчика, который сбрасывается в 00'00"00. Если кнопка lap/reset будет нажата во время счета, дисплей зафиксирует время, в которое была нажата кнопка, счетчик продолжит счет, а мерцание индикаторов минут и секунд с частотой 1 Гц будет означать, что счетчик продолжает считать. Если кнопка start/stop будет нажата, счет будет остановлен, индикаторы минут и секунд перестанут мерцать и время на дисплее будет изменено. Последовательное нажатие кнопки start/stop приведет к продолжению или остановке счета, с измененным временем на дисплее. Во время счета индикаторы минут и секунд будут мерцать. Нажатие кнопки lap/reset при зафиксированном значении на дисплее отобразит на нем время, в которое счетчик был запущен или остановлен.

Первая модель должна быть поведенческой, для тестирования которой следует написать TestBench. Для TestBench реализовать конфигурацию, связывающую копию компонента и поведенческую модель секундомера. Затем нужно детализировать поведенческую модель, преобразовав ее в структурную, включающую устройства управления последовательностью функционирования, регистры, счетчики, декодеры и другие необходимые компоненты. Написать поведенческие модели, соответствующие каждому из компонентов, и конфигурацию, связывающую копии компонентов с интерфейсами. Изменить конфигурацию TestBench, чтобы выполнялось тестирование поведенческой модели. Сравнить результаты работы поведенческой и структурной моделей. Продолжить разработку устройства управления последовательностью функционирования таким образом, чтобы оно представляло собой автомат, состоящий из логики, вычисляющей следующее состояние и состояние регистра. Так же изменить модели других компонентов, чтобы они включали логические элементы и триггеры. На каждом этапе изменять соответствующим образом конфигурации. Протестировать результирующую модель.

ГЛАВА 6
STATE DIAGRAM EDITOR

State Diagram Editor предназначен для графического ввода диаграмм состояний автоматов. Их использование представляет собой альтернативный способ моделирования последовательностных схем. HDL-модель генерируется на основе введенной диаграммы состояний, описывающей поведение устройства. Редактор полностью интегрирован с компилятором и средой моделирования, что позволяет тестировать модели в графическом режиме в окне State Diagram Editor.

В одном проекте имеется возможность совмещать HDL-код и диаграммы состояний, используемые, например, для описания компонентов, применяемых на верхнем уровне иерархии. Проект такого уровня можно также описать с помощью диаграмм состояний. В этом случае он представляется на верхнем уровне иерархии моделью поведенческого типа, который имеет единственный интерфейс, поскольку в диаграмме состояний нет возможности задавать другие интерфейсы.

State Diagram Editor позволяет делить сложные последовательностные устройства на несколько совместно действующих автоматов. Все параллельные автоматы определяются внутри единственного поля диаграммы и имеют общий интерфейс (порты). Для связи между автоматами можно также декларировать внутренние сигналы. В редакторе State Diagram имеется возможность создавать многоуровневые схемы, которые строятся с использованием стандартных и иерархических состояний. Это дает возможность разделить обычную диаграмму на несколько логических уровней и сделать описание устройства более наглядным и понятным.

6.1. Окно редактора State Diagram

На рисунке 6.1 изображено стандартное окно редактора. Оно включает несколько панелей инструментов: главную, содержащую кнопки для основных операций и встроенных инструментов верификации/отладки; вспомогательную панель инструментов графа автомата, дублирующую большинство команд из FSM-меню для создания и размещения компонентов графа; дополнительную панель создания иерархического графа, на которой расположены кнопки управления.

Главная панель инструментов


Вырезать выделенный фрагмент диаграммы и поместить его в Clipboard.


Скопировать вырезанный фрагмент диаграммы в Clipboard.


Вставить фрагмент диаграммы из Clipboard.


Отменить и восстановить последнюю операцию.


Режим масштабирования.


Увеличить масштаб.


Уменьшить масштаб.


Отобразить все элементы диаграммы в окне.


Отобразить всю диаграмму целиком.


Откорректировать масштаб по ширине диаграммы.

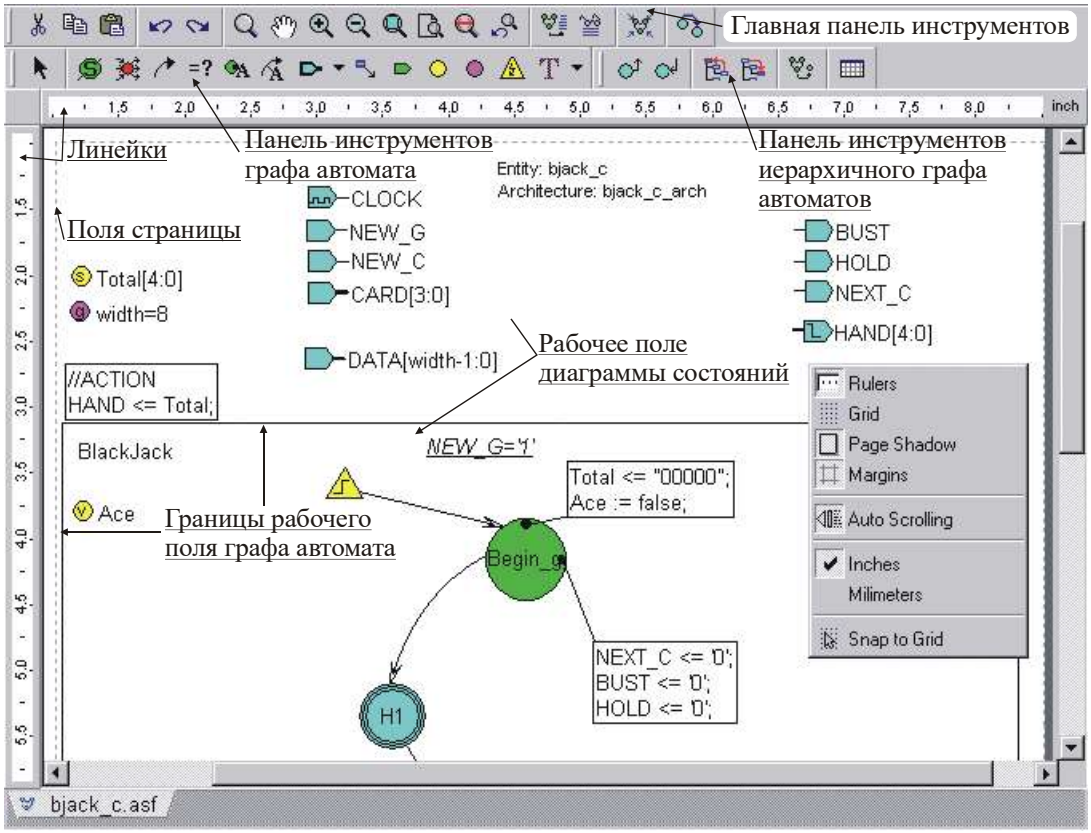

Сгенерировать HDL-код для текущей диаграммы состояний.


Отобразить HDL-код, сгенерированный для текущей диаграммы состояний.


Генерация Testbench файла для текущей диаграммы состояний.

Продолжить моделирование до следующего состояния графа.

Рисунок 6.1. Окно редактора State Diagram



Кнопки панели инструментов графа

Перейти в режим выделения.

Добавить состояние к схеме.

Добавить связывание к схеме.

Нарисовать переход между состояниями.

Назначить условие перехода.

Назначить состоянию действие на входе.

Назначить действие в состоянии.

Назначить состоянию действие на выходе.

Назначить действие на переходе.

Добавить входной порт к диаграмме.

Добавить выходной порт к диаграмме.

Добавить двунаправленный порт к диаграмме.

Добавить порт типа буфер.

Добавить иерархический вход.

Добавить иерархический выход.


Добавить прямую связь для состояния.


Добавить сигнал или переменную к диаграмме.


Добавить состоянию константу или generic-константу.


Добавить символ сброса (reset).


Добавить текст к диаграмме.


Нарисовать кривую Безье.


Нарисовать круг.


Нарисовать прямоугольник.

Кнопки панели инструментов иерархического графа


Перейти на верхний уровень иерархии графа автомата.


Перейти на нижний уровень иерархии графа автомата.


Вывести корень иерархической диаграммы состояний и его порты.


Вернуться к предыдущему уровню диаграммы состояний и масштабу.


Преобразовать множество выбранных объектов в иерархическое состояние.


Открыть окно View Objects.

6.2. Элементы диаграмм

На рисунках 6.2 и 6.3 перечислены основные компоненты, из которых может складываться диаграмма состояний.

Рисунок 6.2. Основные элементы диаграммы состояний

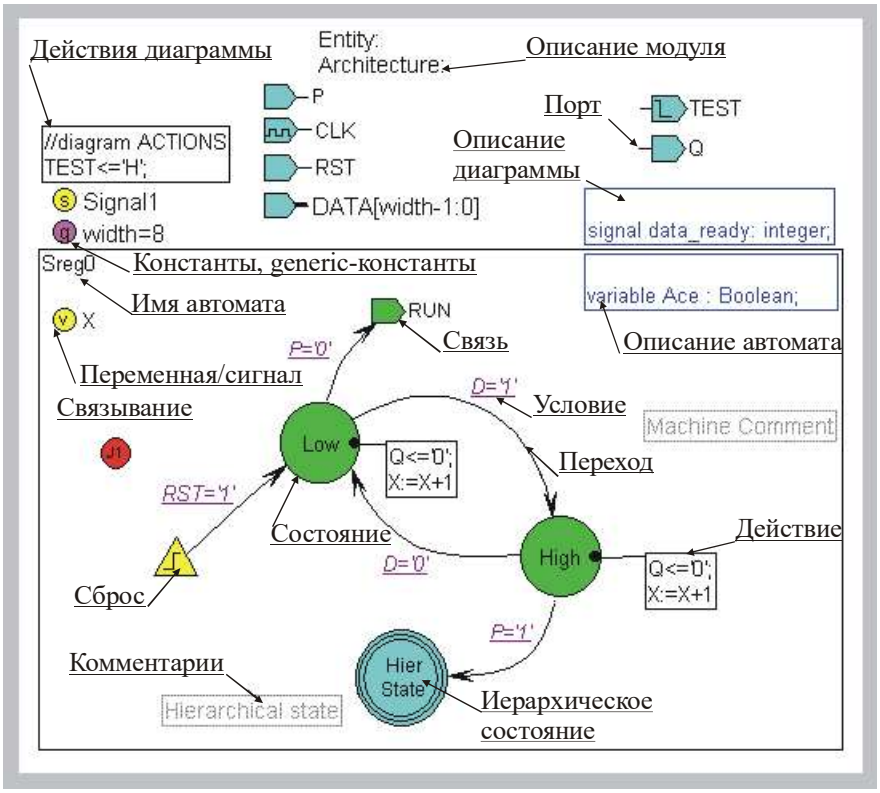
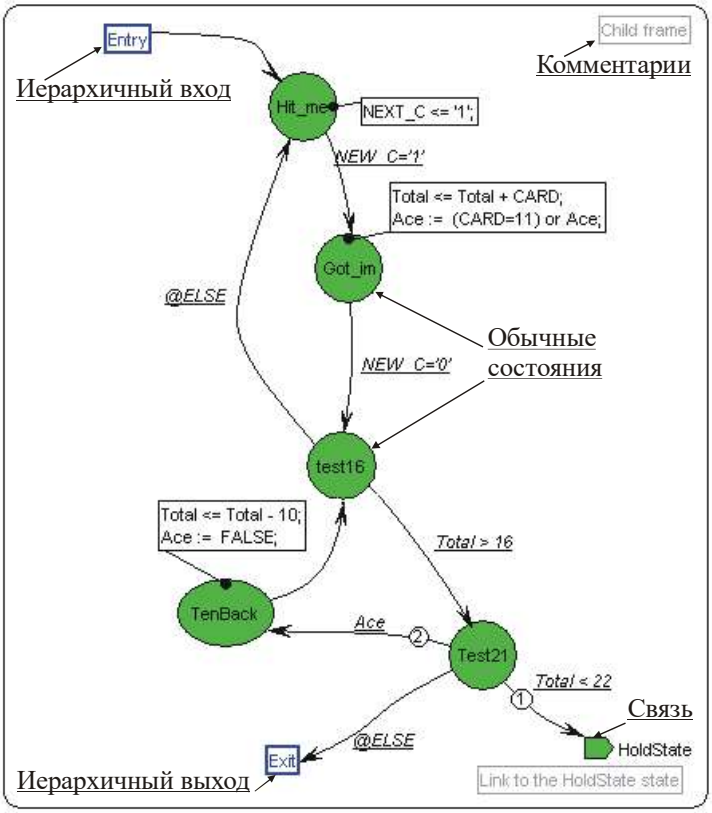
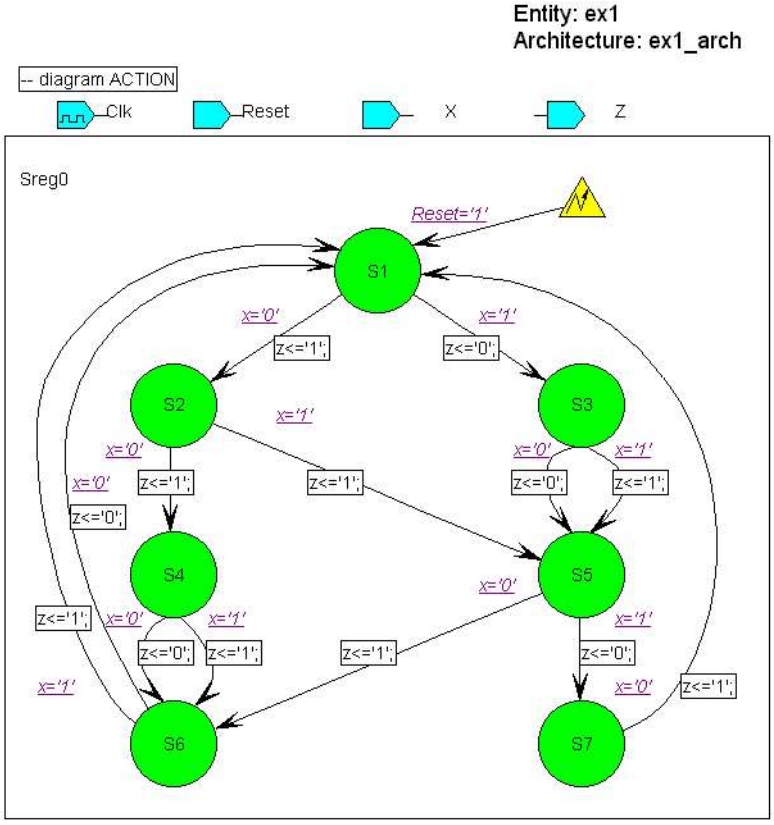


Рисунок 6.3. Граф, соответствующий иерархическому состоянию с рисунка 6.2



На рисунке 6.4 представлена диаграмма состояний для устройства с рисунка 2.15. На ее основе сгенерировано VHDL-описание модели устройства (рисунок 6.5).

Рисунок 6.4. Диаграмма состояний



Граф преобразуется в пару интерфейс – архитектура. Интерфейс содержит декларацию описанных на диаграмме портов Clk, Reset, X, Z. Сигнал, описывающий текущее состояние автомата, имеет одинаковое с ним имя – Sreg0, и тип Sreg0_type. Заданный в архитектуре тип перечисления Sreg0_type содержит символы S1, S2, S3, S4, которые соответствуют состояниям автомата.

Тело архитектуры включает моделирующий поведение автомата оператор процесса Sreg0_machine. В список его чувствительности входят сигналы CLK(синхровход) и reset(сброс). В процессе оператор if выполняет проверку сигнала сброса, а затем фронта синхросигнала:

```
elsif Clk'event and Clk = '1'.
```

Центральной частью процесса является оператор case, анализирующий сигнал Sreg0. Каждый when-вход оператора соответствует определенному состоянию автомата.

Для реализации действий, выполняемых автоматом, использован параллельный условный оператор назначения сигнала (метка z_assignment).

Рисунок 6.5. VHDL-код, соответствующий диаграмме состояний с рисунка 6.4

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ex1 is
   port (Clk: in STD_LOGIC;
         Reset: in STD_LOGIC;
         X: in STD_LOGIC;
         Z: out STD_LOGIC);
end entity;

architecture ex1_arch of ex1 is
   -- Символьное кодирование состояний: Sreg0
   type Sreg0_type is (S2, S3, S4, S6, S7, S1, S5);
   signal Sreg0: Sreg0_type;
begin
-- параллельное назначение сигналов
-------------------------------------------------------------------
-- Автомат: Sreg0
Sreg0_machine: process (Clk, reset)
begin
if Reset='1' then Sreg0 <= S1;
   -- сброс/установка значений для входных/выходных сигналов и
   -- для переменных
elsif Clk'event and Clk = '1' then
   -- установка значений для входных/выходных сигналов и
   -- для переменных
   case Sreg0 is
      when S2 =>    if x='1' then Sreg0 <= S5;
                       elsif x='0' then Sreg0 <= S4; end if;
      when S3 =>    if x='1' then Sreg0 <= S5;
                       elsif x='0' then Sreg0 <= S5; end if;
      when S4 =>    if x='1' then Sreg0 <= S6;
                       elsif x='0' then Sreg0 <= S6; end if;
      when S6 =>    if x='1' then Sreg0 <= S1;
                       elsif x='0' then Sreg0 <= S1; end if;
      when S7 =>    if x='0' then Sreg0 <= S1; end if;
      when S1 =>    if x='1' then Sreg0 <= S3;
                       elsif x='0' then Sreg0 <= S2; end if;
      when S5 =>    if x='0' then Sreg0 <= S6;
                       elsif x='1' then Sreg0 <= S7; end if;
      when others => null;
```

```
    end case;
end if;
end process;

-- оператор назначения для комбинационных выходов
z_assignment:
z <= '1' when (Sreg0 = S2 and x='1') else
     '1' when (Sreg0 = S2 and x='0') else
     '1' when (Sreg0 = S3 and x='1') else
     '0' when (Sreg0 = S3 and x='0') else
     '1' when (Sreg0 = S4 and x='1') else
     '0' when (Sreg0 = S4 and x='0') else
     '1' when (Sreg0 = S6 and x='1') else
     '0' when (Sreg0 = S6 and x='0') else
     '1' when (Sreg0 = S7 and x='0') else
     '0' when (Sreg0 = S1 and x='1') else
     '1' when (Sreg0 = S1 and x='0') else
     '1' when (Sreg0 = S5 and x='0') else
     '0' when (Sreg0 = S5 and x='1') else
     '0';
end ex1_arch;
```

Основные операции над диаграммами

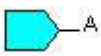
Создать новую диаграмму можно с помощью кнопки New  . Для этого следует щелкнуть по стрелке и выбрать пункт **State Diagram** из выпадающего меню. Чтобы создать диаграмму, можно также использовать мастер New Source File Wizard. Вызвать его можно командой File > New > State Diagram.

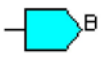
Для того чтобы выбрать HDL-язык, на котором будет записана модель текущей диаграммы, следует выполнить команду FSM > Set Target HDL, а затем установить флажок в нужную опцию: VHDL или Verilog. Но поскольку основным языком здесь является VHDL, далее будут рассмотрены средства для создания моделей на нем.

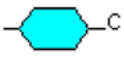
Команда FSM > Generate HDL Code и кнопка  предназначены для создания HDL-модели из текущей диаграммы. Сгенерированный HDL-файл записывается в папку Compile. Для отображения его на экране служит команда FSM>View HDL Code (кнопка ).

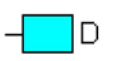
6.3. Порты

Порты (Port) используются для связи между автоматом и окружающей средой. С точки зрения направления передачи данных существует четыре типа портов: входной (input), выходной (output), двунаправленный (bidirectional) и буфер (buffer). Они соответствуют портам в VHDL: in, out, inout, buffer соответственно. Для их обозначения используются символы:

 – входной порт,

 – выходной,

 – двунаправленный,

 – буфер.

Выходные и двунаправленные порты можно описывать как комбинационные (combinatorial) и регистровые (registered). Последние реализуются с помощью дополнительных регистров и их значение может изменяться только по активному фронту синхронизации. Для регистровых портов используются специальные графические символы:

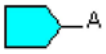
Чтобы создать порт, следует выбрать его тип из списка Port меню FSM или использовать соответствующую кнопку на панели инструментов:

После этого курсор мыши примет форму, которая определяется выбранным типом порта, например:

Новый порт можно разместить щелчком левой кнопки вверху диаграммы над полем графа автомата.

Сначала порт получает заданное по умолчанию имя и тип. Изменить их можно в окне свойств порта Port Properties(рисунок 6.6), которое вызывается командой Properties из контекстного меню правой кнопки мыши. Обратите внимание: эта команда вызывает различные окна свойств в зависимости от типа объекта, выделенного на рабочем поле диаграммы. Если же в момент выполнения команды ни один объект не был выбран, то будет открыто окно свойств графа автомата (курсор находится над его рабочим полем) или диаграммы состояний (курсор находится над полем диаграммы, но вне любого из полей графов автоматов).

Описанные на диаграмме порты преобразуются в порты интерфейса. Их свойства определяют имя порта, его направление и тип. Например,

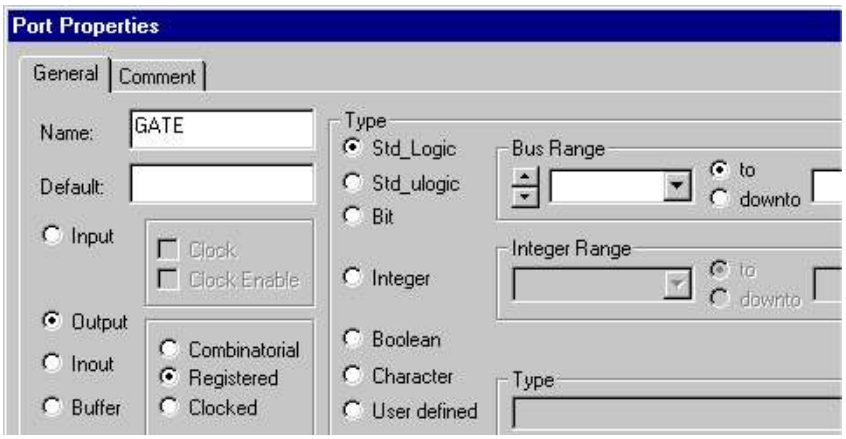



```
port A: in STD_LOGIC;
```

В окне свойств порта в поле Name можно отредактировать или ввести его новое имя, а в поле Default указать присваиваемое по умолчанию значение для портов, имеющих направление: output, inout или buffer. С помощью переключателей Input, Output, Inout и Buffer задается режим порта, а с помощью переключателей: STD_LOGIC, STD_ULOGIC, Bit,Integer, Boolean, Character, User defined – его тип. Поле Bus Range позволяет создавать шины с элементами типа STD_LOGIC, STD_ULOGIC или Bit. Для Integer можно указать его диапазон в поле Integer Range. Также имеется возможность использования определенных пользователем типов – User defined, описание которых задается в поле Type.

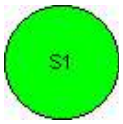
Один из входных портов может быть объявлен как синхровход (флаг Clock), активный фронт которого определяется в свойствах автомата. Флаг Clock Enable используется для создания сигнала разрешения синхронизации. Автомат может иметь только один такой порт. В свойствах автомата имеется возможность указать активный уровень сигнала разрешения синхронизации и состояния, зависящие от него.

Рисунок 6.6. Окно Port Properties



6.4. Состояния

Состояние (state) – внутреннее условие, определяющее выходы автомата. Любой автомат имеет, как минимум, два состояния. Вместе с переходами – это основные составляющие графа автомата. На схеме они обозначаются кружками:

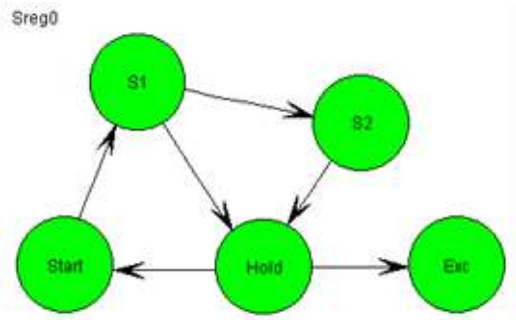


Состояния автомата имеют уникальные имена, которые назначает редактор или определяет сам проектировщик. По умолчанию, редактор присваивает состояниям имена S1, S2, S3, ...

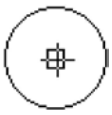
Текущее состояние автомата сохраняется во внутреннем сигнале, объявленном в теле архитектуры. Сигнал имеет имя, точно такое же, как и автомат. До объявления сигнала описывается тип перечисления, который включает все имена состояний. Для автомата, представленного на рисунке 6.7, тело архитектуры должно содержать следующие строки:

```
type Sreg0_type is (Exc, Hold, S1, S2, Start);
signal Sreg0: Sreg0_type;
```

Рисунок 6.7. Пример графа автомата

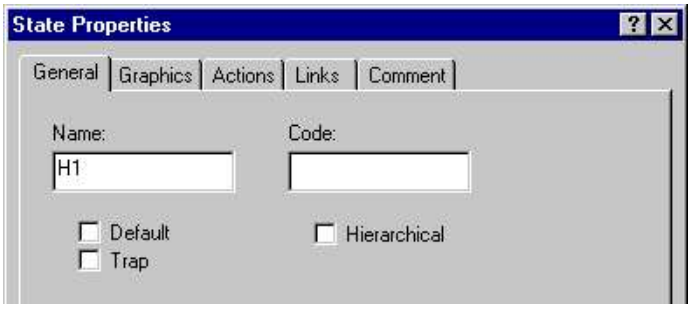


Добавить новое состояние к графу можно используя команду FSM > State или кнопку  . Курсор примет форму:



Следует разместить новое состояние внутри рабочего поля автомата и щелчком закрепить его. Оно будет иметь заданное по умолчанию имя. Изменить его можно в окне свойств состояния State Properties (рисунок 6.8) – поле Name, вкладка General.

Рисунок 6.8. Окно State Properties

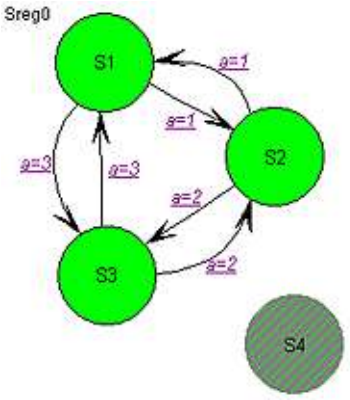


В окне свойств на вкладке General с помощью флагов Defaulf, Trap и Hierarchical можно определять состояния по умолчанию, ловушки, а также иерархические.

По умолчанию (default state) – это состояние, в которое автомат будет устанавливаться, если нет точно определенных переходов из текущего (все условия переходов имеют значение FALSE). Состояние по умолчанию полезно для отладки устройств. В графе автомата, приведенном на рисунке 6.9, S4 имеет статус состояния по умолчанию. Если по некоторым причинам значение сигнала a, имеющего тип integer **range** 0 **to** 3, изменяется в 0, то автомат из любого состояния по следующему синхросигналу перейдет в S4. Обратите внимание: оператор **case** создает модель автомата, а в конце каждого оператора **if** реализуется переход в состояние по умолчанию:

```
else Sreg0 <= S4; end if;
```

Рисунок 6.9. Пример графа автомата с состоянием по умолчанию




```
case Sreg0 is
   when S1 =>    if a=3 then  Sreg0 <= S3;
                 elsif a=1 then Sreg0 <= S2;
                 else Sreg0 <= S4;  end if;
   when S2 =>    if a=1 then Sreg0 <= S1;
                 elsif a=2 then Sreg0 <= S3;
                 else Sreg0 <= S4; end if;
   when S3 =>    if a=2 then Sreg0 <= S2;
                 elsif a=3 then Sreg0 <= S1;
                 else Sreg0 <= S4; end if;
   when S4 =>
   when others => null;
end case;
```

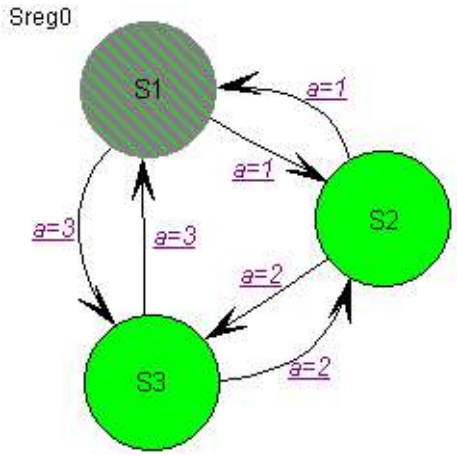
Любое состояние можно объявить состоянием-ловушкой (Trap State). Автомат будет переключаться в него при ближайшем активном фронте синхросигнала, если по некоторым причинам текущему состоянию не соответствует ни одно другое, явно определенное в диаграмме. На рисунке 6.10 изображен граф автомата и соответствующий ему VHDL-код, в котором символом S1 обозначено состояние-ловушка. Если по некоторым причинам автомат перейдет в состояние, отличное от S1, S2 или S3, то по следующему синхроимпульсу он вернется в S1. Обратите внимание, как вход **when others** оператора **case** моделирует переход в состояние-ловушку S1:

```
when others =>  Sreg0 <= S1;
```

Может возникнуть вопрос, как автомат может находиться в состоянии, отличном от S1, S2, S3, ведь сигналу Sreg0 разрешается присваивать только эти три значения. Конечно, такая ситуация не возникнет во время моделирования VHDL-кода, сгенерированного из графа автоматов. Тем не менее, если выполнить синтез схемы, то потребуется, как минимум, два бита для кодирования состояний. В таком случае для

устройства, применяющего 2-разрядный код, есть еще одно неиспользуемое состояние, в которое оно может быть случайно установлено. Если это произойдет, то состояние-ловушка позволит по следующему активному синхросигналу вернуть автомат в разрешенное состояние, в данном примере S1.

Рисунок 6.10. Пример графа автомата с состоянием-ловушкой

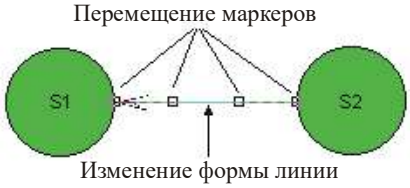



```
case Sreg0 is
   when S1 =>    if a=3 then Sreg0 <= S3;
                 elsif a=1 then Sreg0 <= S2; end if;
   when S2 =>    if a=1 then Sreg0 <= S1;
                 elsif a=2 then Sreg0 <= S3; end if;
   when S3 =>    if a=2 then Sreg0 <= S2;
                 elsif a=3 then Sreg0 <= S1; end if;
   when others => Sreg0 <= S1;
end case;
```

6.5. Переходы, условия и приоритеты

Переход (Transition) – это изменение состояния автомата по переднему фронту синхроимпульса. Обычно он имеет место при выполнении некоторого условия, называемого условием перехода (transition condition). Оно описывается булевым выражением, которое должно в этом случае иметь значение true. В качестве условия перехода можно также использовать строку @else, которая соответствует true, в ситуации когда нет другого разрешенного перехода. Если в один момент времени два или более условий перехода могут быть равными true, то для исключения конфликтов таким переходам следует назначать приоритеты.

Создать переход между состояниями можно с помощью команды FSM>Transition или кнопки . Курсор примет форму ┼. Необходимо выполнить первый щелчок внутри символа состояния, в котором переход начинается, а второй – внутри состояния, в котором переход заканчивается. Перемещая маркеры линии или линию перехода, можно отредактировать ее форму:



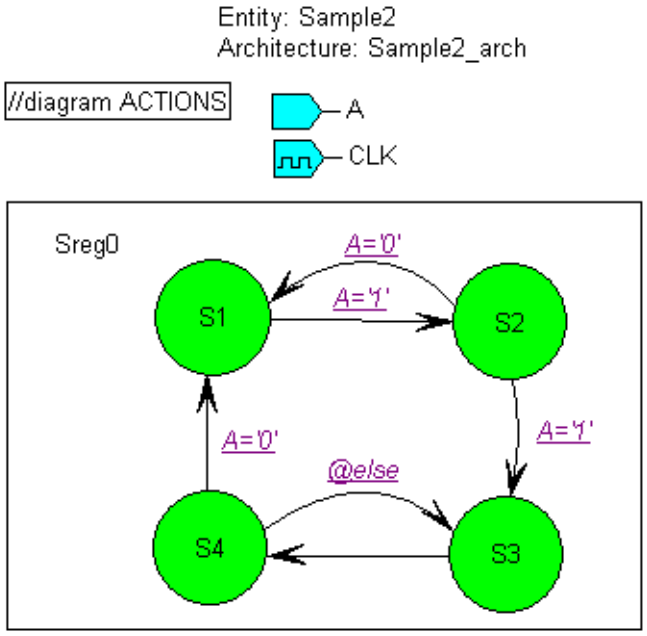
В редакторе State Diagram для определения условий переходов предназначена команда FSM > Condition и кнопка =?. После использования одного из этих инстру-

ментов курсор будет иметь вид ┼. Следует щелкнуть по линии перехода и ввести булево выражение условия перехода, которое должно соответствовать синтаксису VHDL и не иметь в конце точки с запятой. Также ввести условие для выполнения перехода или отредактировать уже существующее можно в окне свойств, используя поле Condition на его вкладке HDL.

В VHDL с помощью оператора if кодируются выходящие из одного состояния переходы. Их число определяет наличие и количество конструкций elsif в операторе if. Если существует переход с условием @else, то соответствующий оператор if содержит конструкцию else. Диаграмма, представленная на рисунке 6.11, иллюстрирует преобразование переходов в VHDL-код.

Обратите внимание: переход, выходящий из состояния S3, является безусловным, и в соответствующей ему конструкции when отсутствует оператор if. Если для автомата было определено состояние по умолчанию, то все операторы if заканчиваются конструкцией else и оператором назначения, задающим текущее S_i следующим состоянием автомата. Однако это правило не работает, если существует переход из состояния с условием @else.

Рисунок 6.11. Фрагмент графа автомата с приоритетами для переходов из состояния S1




```
case Sreg0 is
   when S1 =>    if A='1' then Sreg0 <= S2; end if;
   when S2 =>    if A='0' then Sreg0 <= S1;
                 elsif A='1' then Sreg0 <= S3; end if;
   when S3 =>    Sreg0 <= S4;
   when S4 =>    if A='0' then Sreg0 <= S1;
                 else Sreg0 <= S3; end if;
   when others => null;
end case;
```

На рисунке 6.12 представлена вкладка General окна свойств перехода. На ней указаны начальное (From) и конечное (To) состояния автомата для перехода, а также его приоритет (Priority), значение которого может изменяться от 0 до 9. По умолчанию, все переходы имеют самый высокий нулевой приоритет.

В фрагменте графа автомата, изображенном на рисунке 6.13, переходам из состояния S1 присвоены приоритеты с 0 по 2. Нулевой приоритет на диаграмме не обозначается. Этим переходам будет соответствовать следующее VHDL-описание:

```
if a='0' then Sreg0 <= S2;
elsif a='1' and b='1' then Sreg0 <= S4;
elsif a='1' then Sreg0 <= S3;
end if;
```

Приоритеты определяют порядок записи в операторе if конструкций, вычисляющих следующее значение сигнала Sreg0. Переход в состояние S2 имеет приоритет 0 и расположен первым, затем следует переход в состояние S4 с приоритетом 1, и последним записан переход в S3, имеющий приоритет 2.

Если автомат находится в состоянии S1 и условие (a='1' and b='1') имеет значение TRUE, то он перейдет в состояние S4, поскольку приоритет этого перехода выше, по сравнению с состоянием S3.

Рисунок 6 .12 Окно Transition Properties

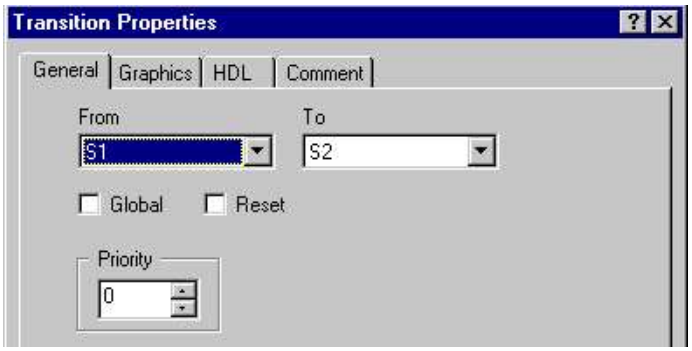
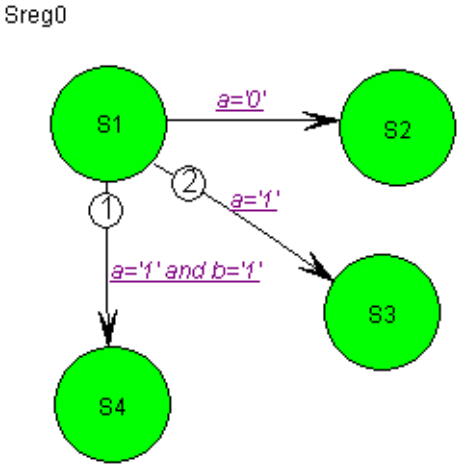


Рисунок 6.13. Фрагмент графа автомата с приоритетами для переходов



6.6. Действия

Действия (Actions) – это множество HDL-конструкций, задающих новые значения портам, сигналам и переменным. В действиях можно использовать операторы назначения для сигналов и присваивания для переменных.

Синтаксис выражений, определяющих действия, должен соответствовать правилам языка VHDL. Если действие состоит из несколько операторов, они должны быть расположены в разных строках и разделяться точкой с запятой.

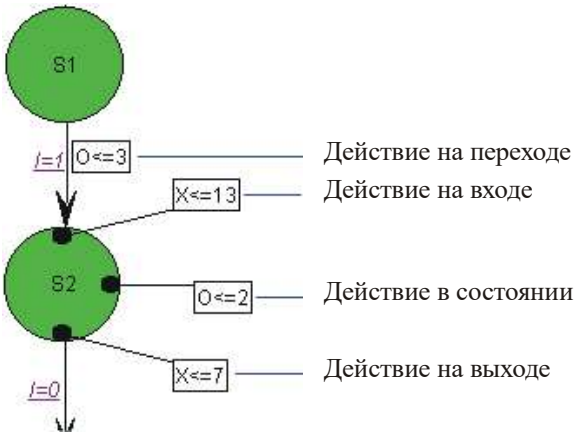
Существует два основных класса действий (рисунок 6.14):

1) *Действие в состоянии* (state action) определяется для состояний и может иметь один или несколько операторов назначения сигналов/переменных. Операторы назначения комбинационных сигналов и портов выполняются, когда автомат установится в состояние по активному фронту синхросигнала. Операторы назна-

чения регистровых сигналов и портов выполняются с задержкой на 1 такт по активному фронту синхроимпульса. Действия в состоянии используются при реализации автоматов Мура.

2) *Действие на переходе* (transition action) определяется для перехода и может состоять из одного или более операторов назначения сигналов или переменных. Момент выполнения действия зависит от типа объекта с левой стороны оператора назначения. Действие над комбинационными сигналами и портами выполняются в тот момент, когда условие перехода принимает значение TRUE. Обратите внимание, что такие действия могут выполняться асинхронно по отношению к синхронизации автомата. Выполнение действий над регистровыми портами/ сигналами и переменными происходят в момент, когда автомат выполняет переход по активному фронту синхроимпульса. Действия на переходе полезны при проектировании автоматов Мили.

Рисунок 6.14. Фрагмент графа автомата с действиями



Два следующих типа являются частным случаем действий на переходе и введены для упрощения графического отображения диаграмм:

1) Действие на входе (Entry action). Определяется для состояния и выполняется при каждом переходе в него.
2) Действие на выходе (Exit action). Определяется для состояния и выполняется при каждом выходе из него.

Эти два типа позволяют уменьшить число записей для действий на переходах, не ухудшая алгоритм функционирования автомата. Например, если в графе существует несколько переходов в некоторое состояние и при осуществлении каждого из них выполняется одинаковое действие, то его можно оформить как действие на входе. Тогда диаграмма будет иметь только одну запись этого действия, а не несколько, как было до этого.

В зависимости от типа действий, для их созданий применяются различные инструменты редактора. Чтобы описать действия в состоянии, следует использовать команду FSM >Action>State или кнопку  . Когда курсор примет форму  , необходимо щелком выбрать символ состояния и ввести операторы VHDL.

Аналогичным образом с помощью команд FSM >Action>Entry и FSM >Action>Exit можно описать действия на входе или выходе соответственно.

Для создания действий на переходе предназначена команда FSM >Action>Transition и кнопка  . После их использования щелчком курсора  выбирается переход и вводятся составляющие действие операторы.

Ввести или отредактировать все типы действий можно также в окне свойств состояний (рисунок 6.15) и переходов (рисунок 6.16). При этом на вкладке Actions окна

State Properties поле Entry соответствует действию на входе, State – в состоянии, а Exit – на выходе. Для ввода действий в окне Transition Properties служит поле Actions.

Рисунок 6.15. Вкладка Actions окна State properties

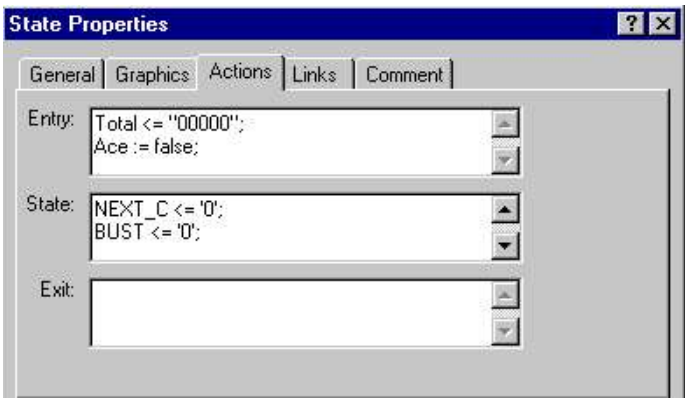
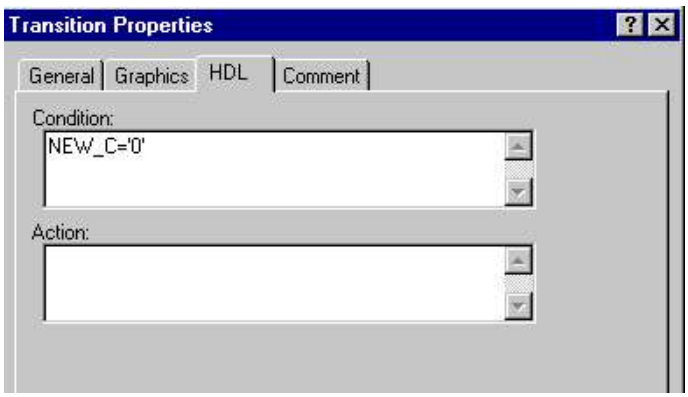
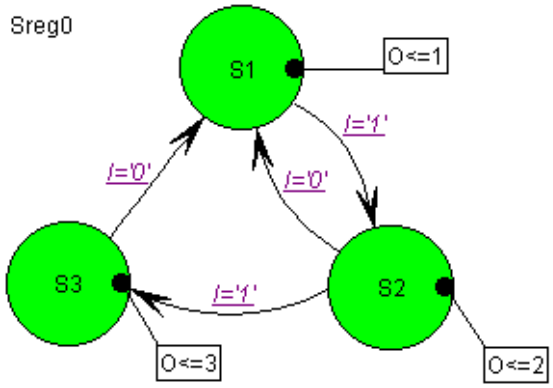


Рисунок 6.16. Вкладка HDL окна Transition properties



На примере автомата, представленного на рисунке 6.17, покажем VHDL-описание действия в состоянии для комбинационных сигналов и портов. Здесь оно используется для обновления сигнала **O** и в данном случае несущественно, является ли он внутренним сигналом или портом, поскольку для обоих применяется одинаковый способ кодирования.

Рисунок 6.17. Граф автомата Мура



Все назначения сигналу **O** реализуются одним простым параллельным оператором назначения, управляемым сигналом Sreg0 (рисунок 6.18, а). В данном случае значение сигнала **O** является функцией от текущего состояния автомата (Sreg0). Обратите внимание, что сигнал получает значения в отдельном операторе и не зависит от синхронизации процесса автомата. Для каждого комбинационного сигнала, который управляется действием в состоянии, в теле архитектуры создается свой параллельный оператор назначения сигнала. Регистровые сигналы кодируются иначе. Если

допустить, что сигнал O является регистровым, то его операторы назначения сигнала O размещаются в when входах, соответствующих состояниям автомата (рисунок 6.18, б). Каждая ветвь when содержит операторы действия, за которыми следует реализующий переход из состояния оператор if.

Рисунок 6.18. Кодирование действий в состоянии

а) в случае комбинационных сигналов

```
Sreg0_machine:
process (CLK)
begin
   if CLK'event and CLK = '1' then
      case Sreg0 is
         when S1 =>   if I='1' then Sreg0 <= S2; end if;
         when S2 =>   if I='1' then Sreg0 <= S3;
                      elsif I='0' then Sreg0 <= S1; end if;
         when S3 =>   if I='0' then Sreg0 <= S1; end if;
         when others => null;
      end case;
   end if;
end process;

O_assignment:
O <= 1 when (Sreg0 = S1) else
     2 when (Sreg0 = S2) else
     3 when (Sreg0 = S3) else
     3;
```

б) в случае регистровых сигналов

```
Sreg0_machine:
process (CLK)
begin
   if CLK'event and CLK = '1' then
      case Sreg0 is
         when S1 =>   O<=1;
                      if I='1' then Sreg0 <= S2; end if;
         when S2 =>   O<=2;
                      if I='1' then Sreg0 <= S3;
                      elsif I='0' then Sreg0 <= S1;
                      end if;
         when S3 =>   O<=3;
                      if I='0' then Sreg0 <= S1; end if;
         when others =>  null;
      end case;
   end if;
end process;
```

На рисунке 6.19 представлен граф автомата Мили, использующий действия на переходах для обновления сигнала O. Если O – это комбинационный внутренний сигнал или порт, то все назначения ему реализуются одним простым параллельным оператором (рисунок 6.20, а). Тогда значение сигнала является функцией от текущего состояния автомата Sreg0 и входов, которые появляются в действиях на переходе. В этом случае для сигнала, получающего значение в действии на переходе, в теле архитектуры создается отдельный параллельный оператор назначения. Если **O** – регистровый сигнал или порт, то кодирование автомата будет выглядеть, как это представлено на рисунке 6.20, б. Оператор назначения сигнала **O** размещается на when входах, соответствующих следующим друг за другом состояниям автомата. В отличие от действий в состоянии, операторы действий на переходе размещаются в операторе if, реализующем переход в следующее состояние.

Рисунок 6.19. Граф автомата Мили

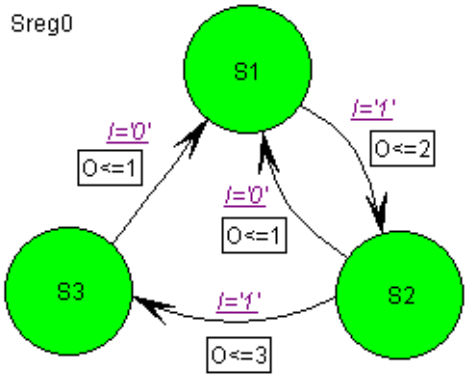


Рисунок 6.20. Кодирование действий на переходах

а) в случае комбинационных сигналов

```
Sreg0_machine:
process (CLK)
begin
   if CLK'event and CLK = '1' then
      case Sreg0 is
         when S1 =>   if I='1' then Sreg0 <= S2; end if;
         when S2 =>   if I='1' then Sreg0 <= S3;
                      elsif I='0' then Sreg0 <= S1; end if;
         when S3 =>   if I='0' then Sreg0 <= S1; end if;
         when others =>  null;
      end case;
   end if;
end process;

O_assignment:
O <= 2 when (Sreg0 = S1 and I='1') else
     3 when (Sreg0 = S2 and I='1') else
     1 when (Sreg0 = S2 and I='0') else
     1 when (Sreg0 = S3 and I='0') else
     1;
```

б) в случае регистровых сигналов

```
Sreg0_machine:
process (CLK)
   begin
      if CLK'event and CLK = '1' then
         case Sreg0 is
            when S1 => if I='1' then Sreg0 <= S2; O<=2; end if;
            when S2 => if I='1' then Sreg0 <= S3; O<=3;
                       elsif I='0' then Sreg0 <= S1; O<=1; end if;
            when S3 => if I='0' then Sreg0 <= S1; O<=1; end if;
         when others => null;
      end case;
   end if;
end process;
```

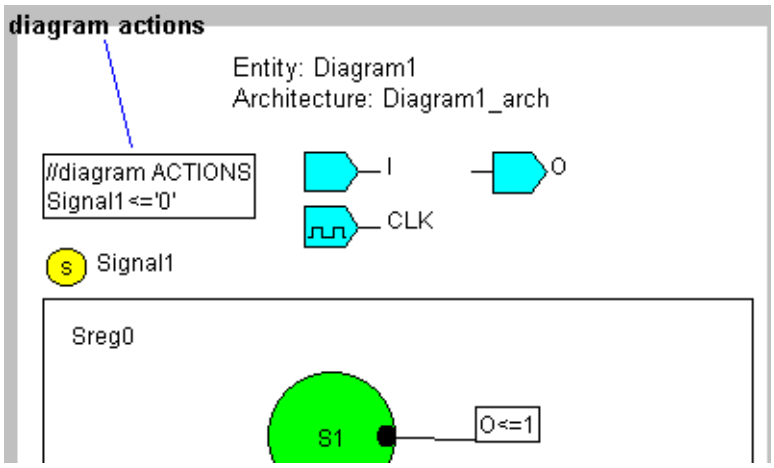
Поскольку переменные декларируются в операторе процесса, их оператор присваивания может находиться только в процессе, который размещается в том же месте, где и операторы назначения для регистровых сигналов.

Диаграммные действия

Диаграммные действия (Diagram Actions) – это вспомогательные, заданные пользователем, параллельные операторы. Как правило, они размещаются в самом начале операторной части архитектуры и позволяют ввести дополнительные функции в устройство.

Задать диаграммное действие можно командой FSM >Action>Diagram или используя кнопку A. Курсор мыши примет форму ⊣⊢. Следует щелкнуть им вверху диаграммы над границей рабочего поля графа автомата и ввести операторы в открывшееся окно ниже текста //diagram ACTIONS. По умолчанию диаграммные действия размещаются в левом верхнем углу поля диаграммы (рисунок 6.21).

Рисунок 6.21. Размещение диаграммных действий



6.7. Переменные и внутренние сигналы

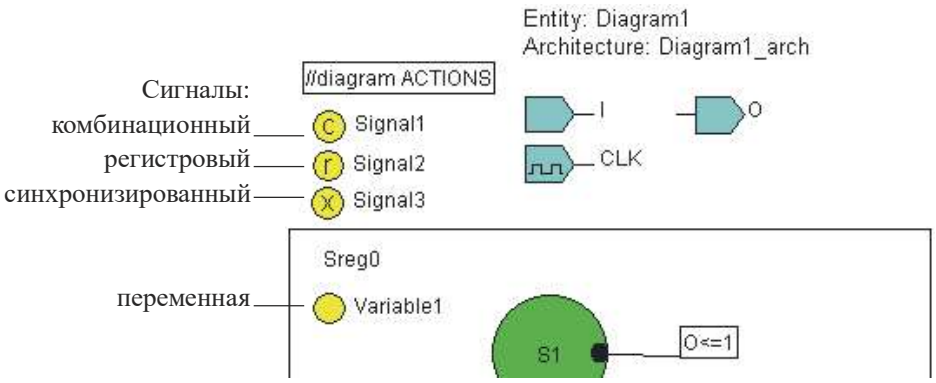
Редактор State Diagran Editor позволяет определять переменные и внутренние сигналы. Переменные описываются в операторе процесса, следовательно, они являются локальными для автомата. Если диаграмма содержит более одного автомата, переменные, определенные в одном автомате, не могут быть доступны в другом.

Сигналы декларируются в теле архитектуры и являются глобальными для всех автоматов диаграммы. Следовательно, они могут быть использованы для коммуникации между автоматами.

Для того чтобы создать переменную или сигнал, можно использовать команду FSM > Signal/Variable или кнопку ○. Курсор примет форму ⊕▭. После этого следует щелчком создать новый элемент. Переменные и сигналы обозначаются в диаграмме одним и тем же символом. Местоположение символа определяет, будет ли он сигналом или переменной. Если символ расположен в поле автомата, он рассматривается как переменная, если за полем – как сигнал (рисунок 6.22).

Сигнал или переменная получат заданное по умолчанию имя. Для того чтобы изменить его, можно щелкнуть правой кнопкой мыши по символу переменной или сигнала, затем выбрать команду Properties из контекстного меню и задать новое имя в открывшемся окне свойств. Также это окно позволяет задавать сигналу начальное значение, объявлять его комбинационным, регистровым или синхронизированным, указывать тип переменной или сигнала.

Рисунок 6.22. Размещение символа

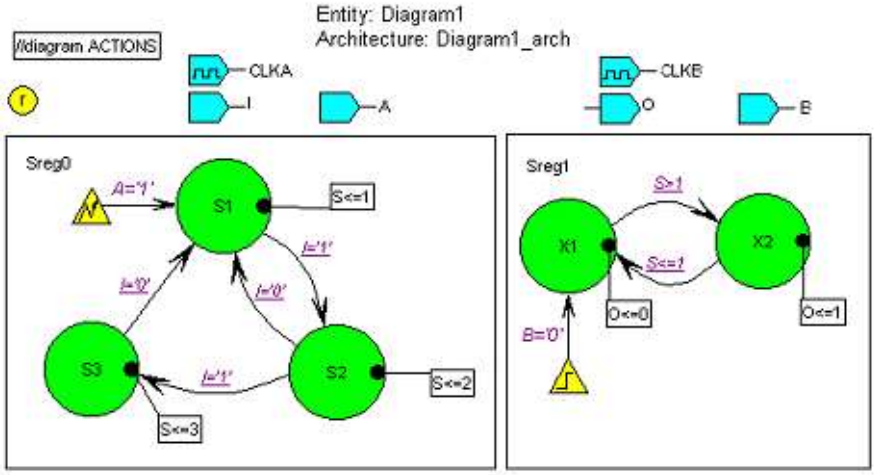


6.8. Автоматы

Обычно диаграммы содержат один автомат (Machines). Он представлен на диаграмме квадратной рамкой с состояниями внутри. В левом верхнем углу автомата находится его имя и декларации переменных, если такие имеются. Каждый автомат кодируется отдельным оператором процесса. Все автоматы, расположенные в одной диаграмме, имеют общий интерфейс (порты). Для каждого автомата можно индивидуально определять синхровход и сигнал сброса (рисунок 6.23).

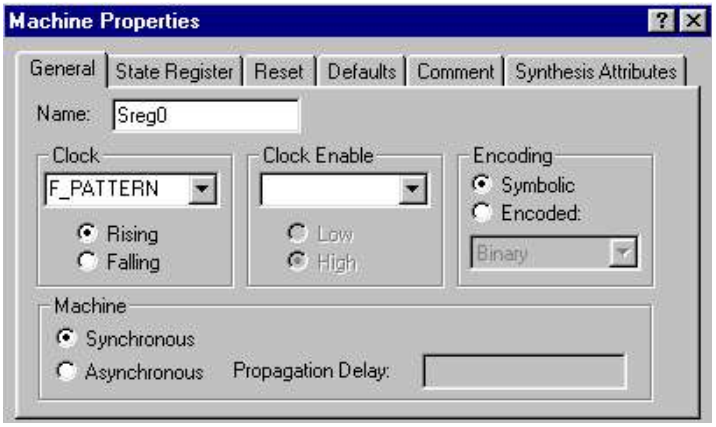
Создать в диаграмме состояний новый граф автомата можно командой **FSM >Add New Machine**. После этого будет сгенерировано новое рабочее поле графа. Если рамка существующего графа занимает все поле диаграммы, это помешает выделению места для нового автомата. Поэтому перед его созданием , чтобы освободить место, необходимо откорректировать рабочие поля существующих автоматов. Для этого следует выделить рамку, окружающую автомат, и изменить ее размер.

Рисунок 6.23. Диаграмма с двумя автоматами



Для управления свойствами автомата удобно использовать окно свойств. В его подокне General (рисунок 6.24) можно задавать: имя автомата (поле Name); сигнал синхронизации (поле Clock) и его активный фронт – передний (Rising) или задний (Faling); сигнал разрешения синхронизации (поле Clock Enable) и способ кодировки автоматов (флаги из группы Encording). Имеется также возможность определить: будет ли автомат использовать синхросигнал (флаг Synchronous из группы Machine), или он будет асинхронным (флаг Asynchronous из группы Machine).

Рисунок 6.24. Окно Machine Properties, вкладка General



Декларации диаграмм и автоматов

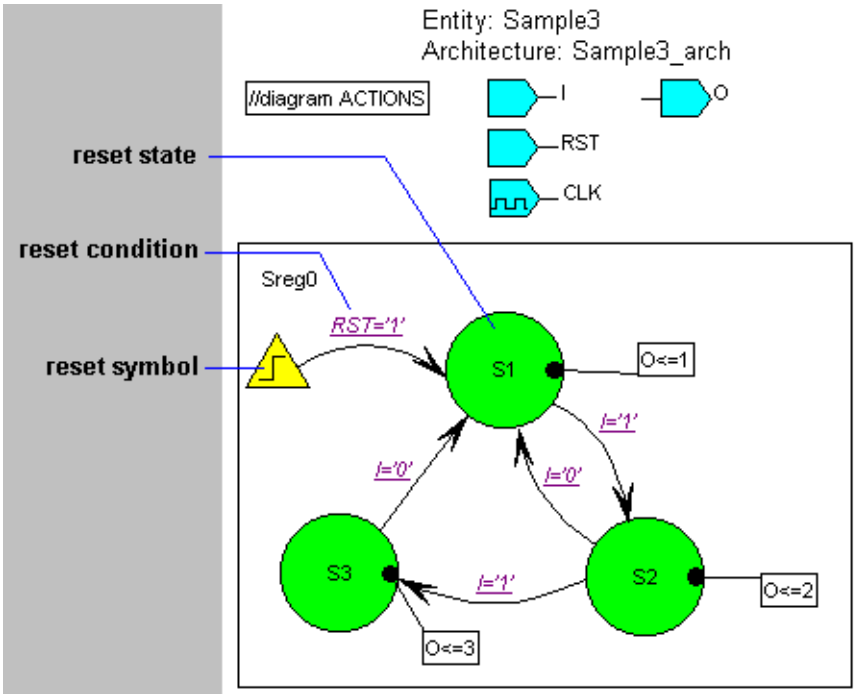
Редактор State Diagram Editor имеет две дополнительные опции: декларация диаграмм и автоматов. Они доступны по команде FSM > Declarations основного меню, позволяют размещать дополнительные декларативные поля.

Декларация диаграммы размещается в декларативной части архитектуры, а декларация автомата – в декларативной части процесса, соответствующего данному автомату.

6.9. Установка автомата в начальное состояние

Автомат переходит в состояние, определенное как начальное (reset state), если выполняется соответствующее условие сброса. Переход в начальное состояние может быть асинхронным или синхронным. Сброс автомата с асинхронным сигналом происходит в момент, когда условие сброса получает значение true, а синхронный автомат требует еще и наличия активного фронта синхросигнала. В качестве условия сброса может быть использовано любое булево выражение. На рисунке 6.25 показан автомат, который переходит в начальное состояние, когда сигнал RST равен '1'.

Рисунок 6.25. Перевод в начальное состояние

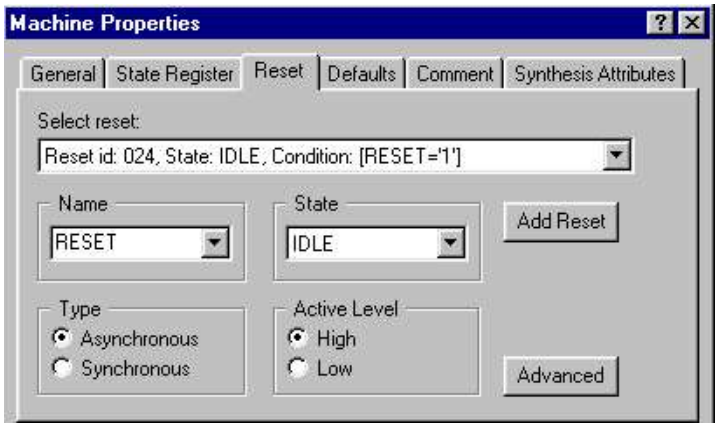


Для того чтобы описать установку автомата в начальное состояние, следует выполнить, команду FSM >Reset или использовать кнопку  . Курсор примет форму  . Первый раз щелкнуть рядом с состоянием, которое будет начальным, второй – на символе состояния. Затем необходимо задать условие сброса таким же образом, как и любое условие перехода.

Вкладка Reset окна свойств автомата (рисунок 6.26) предоставляет возможности управления сигналами его сброса: позволяет выбрать имя сигнала reset из описанных входных портов (поле Name); начальное состояние (поле State), в которое автомат установится после поступления сигнала сброса; его активный уровень (группа Active Level) и его тип (группа Type): асинхронный или синхронный.

Поле Select reset содержит список доступных сигналов сброса и основную информацию об их свойствах: имя, состояние для сброса, тип и активный уровень. Это поле полезно, если существует несколько сигналов сброса и необходимо эффективное средство для управления ими и их редактирования. В случае, когда автомат содержит несколько сигналов сброса, для исключения конфликтных ситуаций всем переходам, выходящим из объекта Reset, необходимо назначить приоритеты с помощью окна Transition Properties.

Рисунок 6.26. Окно Machine Properties, вкладка Reset



В автомате с синхронным сбросом оператор case, расположенный в конструкции if-then-else, моделирует поведение автомата (рисунок 6.27, а). Если условие сброса true, то имеет место переход устройства в начальное состояние (оператор назначения Sreg0 <= S1). Иначе – выполняется оператор case. Оператор if-then-else окружен внешним оператором if, который гарантирует, что любые изменения автомата будут выполняться при наличии активного фронта синхросигнала.

В автомате с асинхронным сигналом сброса оператор case, моделирующий его поведение, расположен в конструкции if-then-elsif (рисунок 6.27, б). Поскольку автомат использует асинхронный сброс, в список чувствительности процесса, кроме синхросигнала, входят сигналы, отвечающие за асинхронную установку устройства в начальное состояние. Сброс автомата происходит (оператор назначения Sreg0 <= S1), когда условие сброса принимает значение true. Иначе, после поступления активного фронта синхросигнала выполняется оператор case.

Рисунок 6.27. Кодирование автомата

а) с синхронным сбросом

```
Sreg0_machine:
process (CLK)
begin
if CLK'event and CLK = '1' then
   if RST='1' then Sreg0 <= S1;
   else
```

```
      case Sreg0 is
      ...
      end case;
   end if;
end if;
end process;
```

б) с асинхронным сбросом

```
Sreg0_machine:
process (CLK, RST)
begin
   if RST='1' then Sreg0 <= S1;
   elsif CLK'event and CLK = '1' then
      case Sreg0 is
      ...
      end case;
   end if;
end process;
```

6.10. Иерархические графы автоматов

Иерархическое состояние позволяет создавать многоуровневые графы автоматов. Существует две главные причины создания такого состояния. Первая, если необходимо разделить сложный граф автомата на несколько фрагментов, чтобы затем каждый из них описать отдельно. Это повышает читаемость схемы и упрощает ее верификацию в дальнейшем. Вторая, если требуется соединить несколько графов автоматов в один. Использование иерархических состояний гарантирует связь между несколькими состояниями и, что наиболее важно, сокращает число операций, необходимых для объединения графов обычным способом.

Для создания иерархического состояния допускается использование существующего состояния или нового символа. Объявить состояние иерархическим можно с помощью команды Hierarchical State или установив переключатель Hierarchical State на вкладке General окна State Properties. Иерархическое состояние отображается голубым цветом с двойным контуром:

Кроме разных графических символов, существует еще несколько различий между стандартным и иерархическим состоянием. Для последнего, также как и для стандартного, можно легко определить имя. Однако иерархическое состояние имеет специальное имя для представления в сгенерированном HDL-коде. Обозначение этого имени имеет следующий вид:

```
<hierarchical_state_name>_<state_name>,
```

где <hierarchical_state_name> – имя состояния в схеме верхнего уровня; <state_name> – имя состояния в схеме нижнего уровня.

Следующий пример (рисунок 6.28) представляет собой граф автомата, содержащий иерархическое состояние HierState (рисунок 6.29). На рисунке 6.30 приведен сгенерированный код, соответствующий графу автомата, изображенного на рисунках 6.28 и 6.29. Выделенные подчеркиванием строки описывают состояние HierState данного графа.

Рисунок 6.28. Граф автомата верхнего уровня

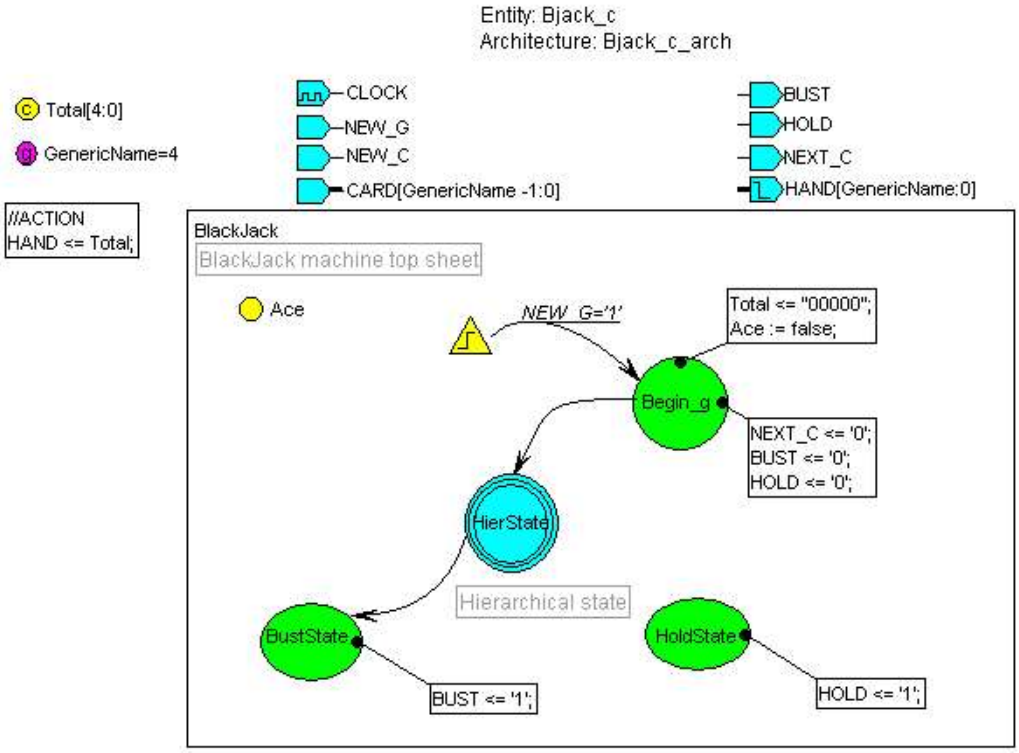
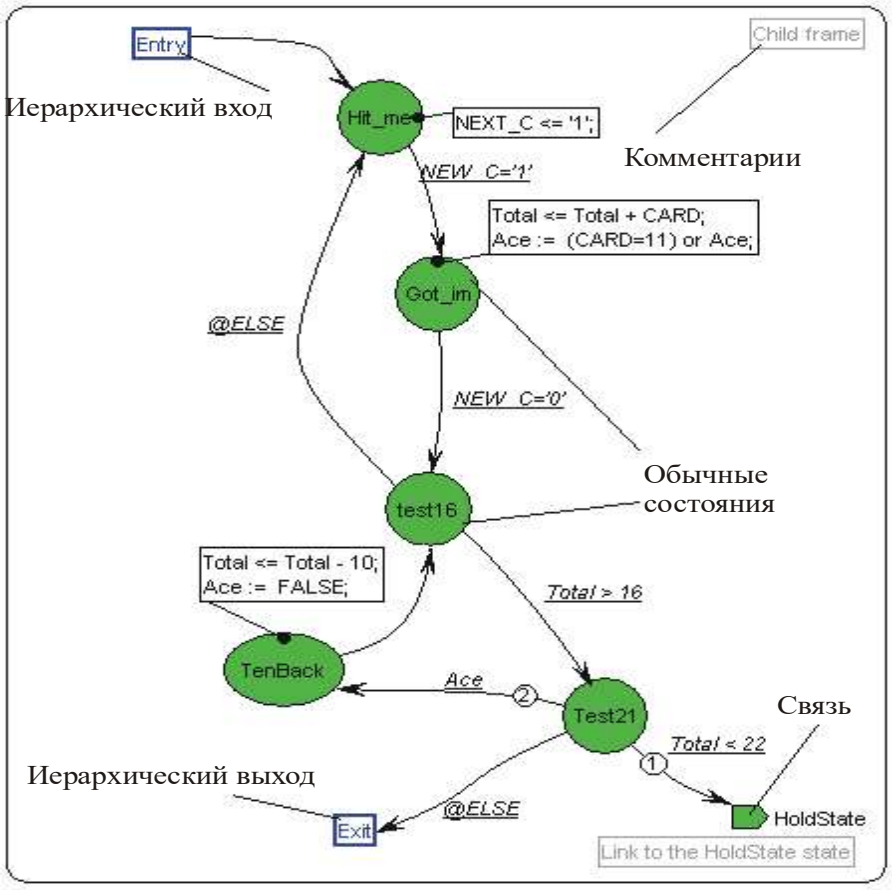


Рисунок 6.29. Граф, соответствующий состоянию HierState



Объект иерархический вход/выход (Hierarchy Entry/Exit) используется для соединения переходов, входящих и выходящих из состояния HierState, расположенного на верхнем уровне проекта. Он несет такую же информацию, как и переход, но может быть использован только на нижнем уровне иерархии.

Каждый переход, входящий в иерархическое состояние, соединяется с каждым входом объекта, расположенного на нижнем уровне иерархии. Аналогично, каждый иерархический выход соединяется со всеми иерархическими состояниями на диаграмме верхнего уровня. Для создания иерархического входа или выхода можно использовать команду Hierarchy Entry/Exit меню FSM или кнопки  /  .

Для описания перехода из состояния Test21, расположенного на нижнем уровне иерархии (см. рисунок 6.29), в состояние HoldState из другой части проекта (см. рисунок 6.28), используется объект связи (Link):

В общем случае чтобы соединить два состояния, расположенных в разных схемах и на разных уровнях иерархии, следует создать объект связи и указать имя состояния для перехода. Затем необходимо нарисовать переход из начального состояния к объекту связи. Такое соединение эквивалентно переходу из одного состояния в другое.

Создать новый объект связи можно с помощью кнопки  , а отредактировать имя состояния для перехода – в окне свойств связи (Link Properties).

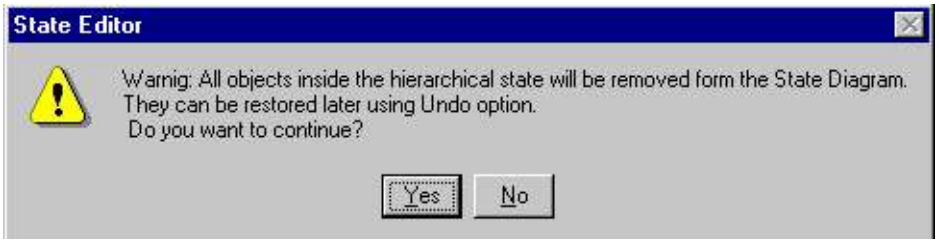
Рисунок 6.30. Код описания графа с иерархическими состояниями

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity Bjack_c is
generic (GenericName: INTEGER := 4 );
port (CARD: in STD_LOGIC_VECTOR (GenericName-1 downto 0);
      CLOCK: in STD_LOGIC;
      NEW_C: in STD_LOGIC;
      NEW_G: in STD_LOGIC;
      BUST: out STD_LOGIC;
      HAND: out STD_LOGIC_VECTOR (GenericName downto 0);
      HOLD: out STD_LOGIC;
      NEXT_C: out STD_LOGIC);
end;

architecture Bjack_c_arch of Bjack_c is
   signal Total: STD_LOGIC_VECTOR (4 downto 0);
   -- Символическое кодирование автомата BlackJack
   type BlackJack_type is (Begin_g, BustState, HoldState,
        HierState_Got_im, HierState_Hit_me, HierState_TenBack,
        HierState_test16, HierState_Test21);
   signal BlackJack: BlackJack_type;
begin
   -- Диаграммные действия
   HAND <= Total;
   -- Процесс, моделирующий автомат
   BlackJack_machine: process (CLOCK)
      variable Ace: BOOLEAN;
   begin
      if CLOCK'event and CLOCK = '1' then
      if NEW_G='1' then
         BlackJack <= Begin_g;
      -- Установка значений по умолчанию для регистровых
      -- сигналов и для переменных
            Total <= "00000";
```

```
            Ace := false;
         else
            case BlackJack is
               when Begin_g => BlackJack <= HierState_Hit_me;
               when BustState =>
               when HoldState =>
               when HierState_Got_im =>  if NEW_C='0' then
                                           BlackJack <= HierState_test16;
                                        end if;
               when HierState_Hit_me =>
                                    if NEW_C='1' then
                                       BlackJack <= HierState_Got_im;
                                       Total <= Total + CARD;
                                       Ace := (CARD=11) or Ace;
                                    end if;
               when HierState_TenBack => BlackJack <= HierState_test16;
               when HierState_test16 =>
                          if Total > 16 then
                             BlackJack <= HierState_Test21;
                          else BlackJack <= HierState_Hit_me; end if;
               when HierState_Test21 =>
                     if Total < 22   then    BlackJack <= HoldState;
                     elsif Ace  then BlackJack <= HierState_TenBack;
                                     Total <= Total-10; Ace := FALSE;
                     else    BlackJack <= BustState;
                     end if;
               when others =>  null;
            end case;
         end if;
         end if;
      end process;
   --  Операторы назначения для комбинационных сигналов
      NEXT_C_assignment:
         NEXT_C <= '1' when (BlackJack = HierState_Hit_me) else '0';
      BUST_assignment:
         BUST <= '1' when (BlackJack = BustState) else '0';
      HOLD_assignment:
         HOLD <= '1' when (BlackJack = HoldState) else '0';
   end Bjack_c_arch;
```

Если по некоторым причинам для иерархического состояния атрибут Hierarchical будет удален, то часть описания автомата будет утеряна, так как нижний уровень иерархии удаляется. Каждый раз при попытке отменить иерархичность состояния будет появляться сообщение, показанное на рисунке 6.31.

Рисунок 6.31. Код описания графа с иерархическими состояниями



Соединение состояний

Использование иерархического состояния дает возможность соединить несколько их вместе, позволяя сократить число требуемых для этого переходов, например, как это сделано на рисунке 6.32. В данном фрагменте состояния S1-S4 соединяются с S6-S10 через иерархическое состояние S5. Для того чтобы создать связь между вершинами, граф нижнего уровня содержит иерархический вход (Hierarchy Entry) и иерархический

выход (Hierarchy Exit), где они просто связаны вместе (рисунок 6.33). Каждый переход, входящий в иерархическое состояние, соединяется с аналогичным входом, размещенным на нижнем уровне иерархии. А исходящий из нее иерархический выход связывается со всеми переходами, подключенными к иерархическому состоянию на диаграмме верхнего уровня. Это значит, что такая поддиаграмма и один переход между иерархическим входом и выходом обеспечивают множество невидимых на верхнем уровне переходов, соединяющих состояния. Использование иерархического состояния в качестве перехода позволяет значительно сократить общее число переходов в диаграмме. Например, в данном случае потребовалось бы 16 переходов при обычном методе ее отображения, а с использованием иерархического состояния – только 8.

Рисунок 6.32. Соединение двух групп состояний вместе

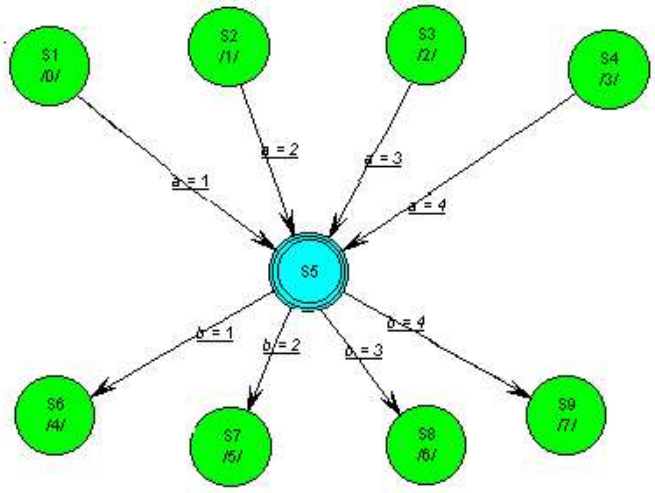


Рисунок 6.33. Содержание иерархического состояния (S5)

Связывание

Связывание (Junction) – это объект , который предназначен для упрощения представления графов путем сокращения в нем числа переходов. Использование его подобно применению иерархического состояния, но в отличие от последнего связывание не является иерархическим объектом.

На рисунке 6.34, а изображен граф, содержащий 9 переходов, соединяющих каждую вершину S1-S3 с каждой вершиной S4-S6. Применение связывания (рисунок 6.34, б) упрощает изображение, уменьшая количество переходов до 6, и облегчает его анализ.

Для создания объекта связывания можно использовать кнопку  на панели инструментов или команду Junction из меню FSM.

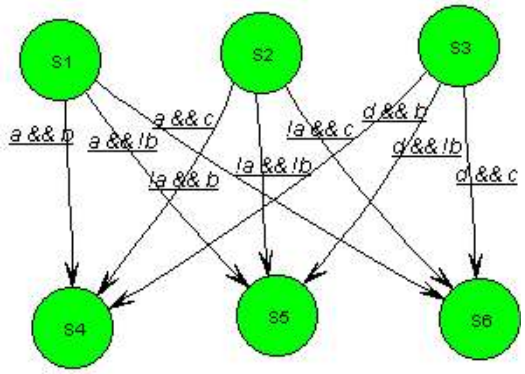
6.11. Стили записи автоматов

Редактор State Diagram предоставляет возможность использования нескольких стилей записи HDL-кода, сгенерированного из диаграммы состояний. Например, для вычисления следующего состояния может быть использован оператор case или if. Имеется возможность выбора между 1-, 2- и 3-процессорной VHDL-моделями автомата. Для изменения стиля записи генерируемого HDL-файла существует команда FSM > Code Generation Settings.

На примере диаграммы состояний с рисунка 6.35 проиллюстрируем различные стили кодировки VHDL-моделей устройств.

Рисунок 6.34. Использование связывания для построения графов автоматов

а) фрагмент графа без использования связывания



б) фрагмент графа с использования объекта связывания

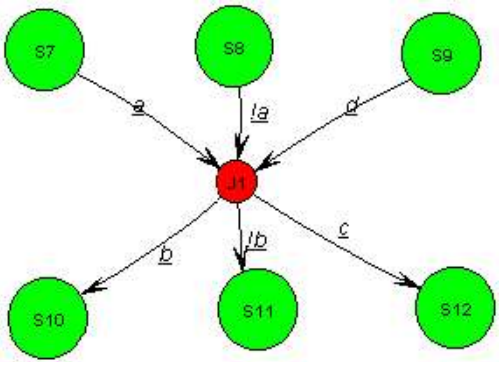
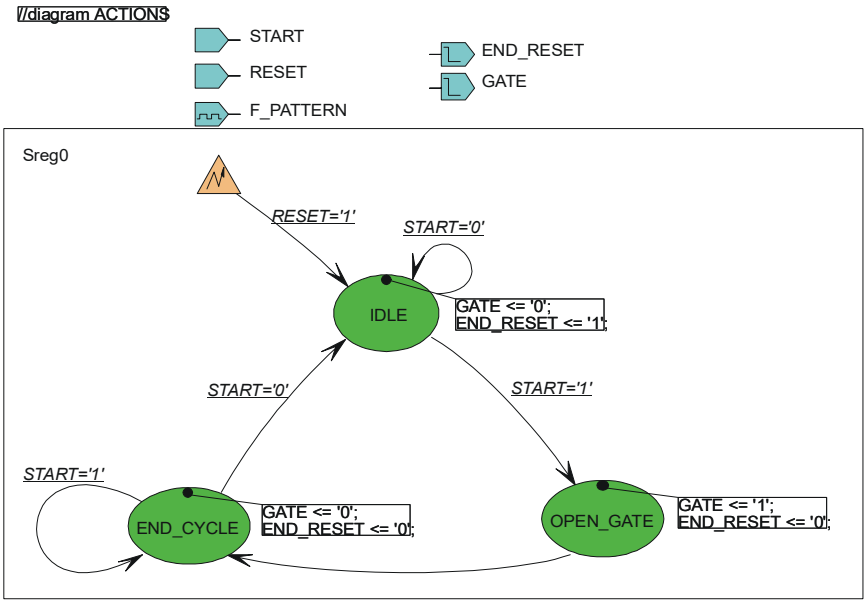


Рисунок 6.35. Граф автомата и соответствующий ему VHDL-код



Стиль CASE/One Process используется по умолчанию. В этом случае регистровые сигналы и регистр состояния автомата описываются в одном процессе Sreg0_machine. В нем после анализа сигнала сброса для моделирования регистра состояния автомата применяется оператор case. Комбинационные сигналы обновляются с помощью отдельных параллельных операторов назначения сигнала. На рисунке 6.36 представлен VDHL-код, сгенерированный для диаграммы (рисунок 6.35) с использованием стиля CASE/One Process.

Рисунок 6.36. Кодирование автомата с применением стиля CASE/One Process.

```
-- Имя автомата: Sreg0
Sreg0_machine: process (F_PATTERN, reset)
begin
if RESET='1' then
   Sreg0 <= IDLE;
   -- Установка инициализирующих значений для регистровых
   -- выходов/сигналов и для переменных
   GATE <= '0';
   END_RESET <= '1';
elsif F_PATTERN'event and F_PATTERN = '1' then
   -- Установка инициализирующих значений для регистровых
   -- выходов/сигналов и для переменных
   -- . . .
   case Sreg0 is
      when END_CYCLE => if START='0' then Sreg0 <= IDLE;
                              GATE <= '0'; END_RESET <= '1';
                        elsif START='1' then Sreg0 <= END_CYCLE;
                              GATE <= '0'; END_RESET <= '0';
                        end if;
      when IDLE => if START='1' then Sreg0 <= OPEN_GATE;
                              GATE <= '1'; END_RESET <= '0';
                   elsif START='0' then Sreg0 <= IDLE;
                              GATE <= '0'; END_RESET <= '1';
                   end if;
      when OPEN_GATE =>  Sreg0 <= END_CYCLE;
                         GATE <= '0'; END_RESET <= '0';
      when others => null;
   end case;
end if;
end process;
```

Модель, использующая кодирование IF/One Process, подобна сгенерированной с использованием стиля CASE/One Process. Единственное отличие заключается в том, что стиль IF/One Process для вычисления следующего состояния, вместо оператора case, применяет if. На рисунке 6.37 представлен пример кодирования автомата (см. рисунок 6.35) стилем IF/One Process.

Рисунок 6.37. Кодирование автомата с применением стиля IF/One Process

```
-- Имя автомата: Sreg0
Sreg0_machine: process (F_PATTERN, reset)
begin
if RESET='1' then
   Sreg0 <= IDLE;
   -- Установка инициализирующих значений для регистровых
   -- выходов/сигналов и для переменных
   GATE <= '0';
   END_RESET <= '1';
elsif F_PATTERN'event and F_PATTERN = '1' then
   -- Установка инициализирующих значений для регистровых
   -- выходов/сигналов и для переменных
   -- . . .
   if Sreg0 = END_CYCLE then
        if START='0' then
             Sreg0 <= IDLE; GATE <= '0'; END_RESET <= '1';
        elsif START='1' then
             Sreg0 <= END_CYCLE; GATE <= '0'; END_RESET <= '0';
        end if;
   elsif Sreg0 = IDLE then
        if START='1' then
             Sreg0 <= OPEN_GATE; GATE <= '1'; END_RESET <= '0';
        elsif START='0' then
```

```
            Sreg0 <= IDLE; GATE <= '0'; END_RESET <= '1';
        end if;
   elsif Sreg0 = OPEN_GATE then
            Sreg0 <= END_CYCLE; GATE <= '0'; END_RESET <= '0';
   else null;
   end if;
end if;
end process;
```

В стиле CASE/Two Processes регистровые сигналы и регистр состояний автомата реализуются с помощью двух различных процессов. Оператор case (или if) в процессе Sreg0_NextState используется для описания специальной комбинационной логики, вычисляющей необходимые значения сигналов, которые должны быть получены до перехода устройства в следующее состояние. Процесс Sreg0_CurrentState моделирует синхронизированный по фронту регистр состояний с помощью сигнала NextState_Sreg0, получающего значения в предыдущем процессе. Регистровые выходы описываются в процессе Sreg0_RegOutput, а комбинационные – непосредственно в Sreg0_NextState. Пример модели, использующей стиль кодирования CASE/Two Processes, приведен на рисунке 6.38.

Рисунок 6.38. Кодирование автомата с применением стиля CASE/Two Processes

```
type Sreg0_type is (IDLE, OPEN_GATE, END_CYCLE);
signal Sreg0, NextState_Sreg0: Sreg0_type;
-- Декларация внутренних сигналов
signal int_END_RESET, next_END_RESET: STD_LOGIC;
signal int_GATE, next_GATE: STD_LOGIC;
begin
-- Параллельные операторы
-- Имя автомата: Sreg0

-- Процесс для вычисления комбинационных сигналов и выходов
Sreg0_NextState: process (START, int_GATE, int_END_RESET, Sreg0)
   begin
      NextState_Sreg0 <= Sreg0;
      -- Установка значений по умолчанию выходов и сигналов
      next_GATE <= int_GATE; next_END_RESET <= int_END_RESET;
      case Sreg0 is
         when IDLE => if START='1' then NextState_Sreg0 <= OPEN_GATE;
                        next_GATE <= '1'; next_END_RESET <= '0';
                      elsif START='0' then NextState_Sreg0 <= IDLE;
                        next_GATE <= '0'; next_END_RESET <= '1';
                      end if;
         when OPEN_GATE => NextState_Sreg0 <= END_CYCLE;
                      next_GATE <= '0'; next_END_RESET <= '0';
         when END_CYCLE => if START='0' then NextState_Sreg0 <= IDLE;
                      next_GATE <= '0'; next_END_RESET <= '1';
                      elsif START='1' then
                           NextState_Sreg0 <= END_CYCLE;
                           next_GATE <= '0'; next_END_RESET <= '0';
                      end if;
         when others => null;
      end case;
   end process;

-- Последовательностная логика для вычисления следующего состояния
Sreg0_CurrentState: process (F_PATTERN, reset)
   begin
      if RESET='1' then Sreg0 <= IDLE;
      elsif F_PATTERN'event and F_PATTERN = '1' then
                                        Sreg0 <= NextState_Sreg0;
      end if;
   end process;
```

```
-- Последовательностный процесс для вычисления регистровых сигналов
Sreg0_RegOutput: process (F_PATTERN, reset)
   begin
      if RESET='1' then int_GATE <= '0'; int_END_RESET <= '1';
      elsif F_PATTERN'event and F_PATTERN = '1' then
            int_GATE <= next_GATE; int_END_RESET <= next_END_RESET;
      end if;
   end process;
   --Передача значений из временных сигналов на выходы
   GATE <= int_GATE;
   END_RESET <= int_END_RESET;
end control_arch;
```

Стиль CASE/Three Processes подобен CASE/Two Processes. Различие заключается в том, что в последнем для вычисления внутренних комбинационных сигналов и выходов используется один процесс Sreg0_NextState, а в стиле CASE/Three Processes – два: Sreg0_NextState и Sreg0_OutputBlock. Пример CASE/Three Processes стиля кодирования автомата (см. рисунок 6.35) приведен на рисунке 6.39

Рисунок 6.39. Кодирование автомата с применением стиля CASE/Three Processes

```
type Sreg0_type is (IDLE, OPEN_GATE, END_CYCLE);
signal Sreg0, NextState_Sreg0: Sreg0_type;
-- Декларация внутренних сигналов
signal int_END_RESET, next_END_RESET: STD_LOGIC;
signal int_GATE, next_GATE: STD_LOGIC;
begin
-- Параллельные операторы
-- Диаграммные действия
-- Имя автомата: Sreg0

-- Комбинационный процесс для вычисления следующего состояния
Sreg0_NextState: process (START, Sreg0)
   begin
      NextState_Sreg0 <= Sreg0;
      case Sreg0 is
         when IDLE => if START='1' then NextState_Sreg0 <= OPEN_GATE;
                      elsif START='0' then NextState_Sreg0 <= IDLE;
                      end if;
         when OPEN_GATE => NextState_Sreg0 <= END_CYCLE;
         when END_CYCLE =>
                 if START='0' then NextState_Sreg0 <= IDLE;
                 elsif START='1' then NextState_Sreg0 <= END_CYCLE;
                 end if;
         when others => null;
      end case;
   end process;

-- Процесс для вычисления комбинационных сигналов и выходов
Sreg0_OutputBlock: process (START, int_GATE, int_END_RESET, Sreg0)
   begin
      – Set default values for outputs and signals
      next_GATE <= int_GATE; next_END_RESET <= int_END_RESET;
      case Sreg0 is
         when IDLE => if START='1' then
                         next_GATE <= '1'; next_END_RESET <= '0';
                      elsif START='0' then
                         next_GATE <= '0'; next_END_RESET <= '1';
                      end if;
         when OPEN_GATE => next_GATE <= '0'; next_END_RESET <= '0';
         when END_CYCLE => if START='0' then
                             next_GATE <= '0'; next_END_RESET <= '1';
                           elsif START='1' then
                             next_GATE <= '0'; next_END_RESET <= '0';
```

```
                        end if;
         when others => null;
      end case;
   end process;

-- Последовательностная логика для вычисления состояния
Sreg0_CurrentState: process (F_PATTERN, reset)
   begin
      if RESET='1' then Sreg0 <= IDLE;
      elsif F_PATTERN'event and F_PATTERN = '1' then
         Sreg0 <= NextState_Sreg0;
      end if;
   end process;

-- Последовательностный процесс для вычисления регистровых сигналов
Sreg0_RegOutput: process (F_PATTERN, reset)
   begin
      if RESET='1' then int_GATE <= '0'; int_END_RESET <= '1';
      elsif F_PATTERN'event and F_PATTERN = '1' then
         int_GATE <= next_GATE;
         int_END_RESET <= next_END_RESET;
      end if;
   end process;
--Передача значений из временных сигналов на выходы
GATE <= int_GATE;
END_RESET <= int_END_RESET;
end control_arch;
```

6.12. Комментарии

Комментарии – это дополнительный текст, содержащий определенную пользователем информацию и добавляемый к каждому классу объектов диаграммы. Автоматы, состояния, переходы, порты, сигналы, константы и переменные могут содержать дополнительный текст в свойствах объекта в подокне Comment (комментарии), который будет использоваться при генерации HDL-кода.

Хотя комментарии могут применяться для любых объектов диаграмм, для некоторых из них комментарии не могут быть размещены в генерируемом коде, например, для иерархических состояний, переходов.

6.13. Контроль за объектами диаграмм

Редактор State Machine Editor имеет дополнительные инструменты, которые предназначены для управления объектами, используемыми во временных диаграммах. Окно View/Sort Objects позволяет просматривать следующие объекты диаграмм: автоматы (machine), порты, сигналы, константы, состояния, переменные.

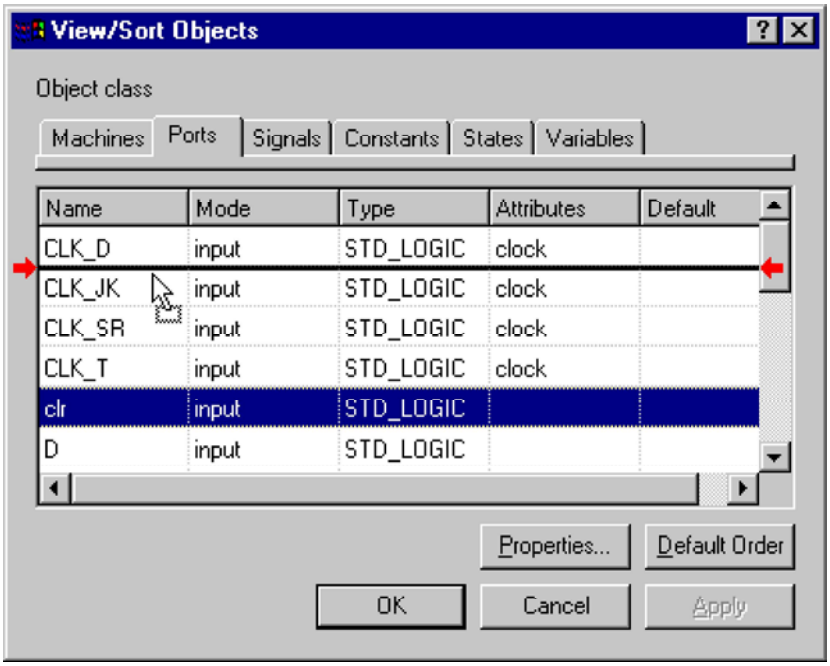
Каждая вкладка окна View/Sort Objects содержит информацию об одном из классов объектов. Имеется возможность одновременно для объекта открывать окно Properties (свойства) и непосредственно находить его на схеме. Список объектов, выведенный в окне, может быть изменен пользователем с применением технологии drag and drop. Можно также сортировать его по именам или другим параметрам. Результат сортировки объектов будет использоваться при генерации HDL-кода.

Чтобы просмотреть свойства объекта, следует выделить его в соответствующем подокне и нажать кнопку Properties (рисунок 6.40).

Если проект состоит из нескольких автоматов и содержит большое число портов, состояний и других объектов, то можно легко найти любой из них, используя команду поиска. Для этого нужно выполнить команду Find Object из выпадающего меню. Найденный объект выделяется подсветкой на рабочем поле (рисунок 6.41).

Рисунок 6.40. Просмотр свойств автоматов

Рисунок 6.41. Просмотр портов объекта



Если по некоторым причинам не подходит порядок отображения объектов, его можно изменить. Для этого объекты можно: отсортировать по имени, щелкнув по заголовку колонки Name; изменить порядок расположения объектов вручную, перемещая их с помощью мыши; разместить объекты по умолчанию, основываясь на внутреннем алгоритме FSM-модуля.

Важно помнить, что порядок объектов в окне View/Sort Objects может повлиять на форму сгенерированного HDL-кода. Например, на рисунке 6.35 представлена диаграмма, где для описания регистра состояний автомата был использован тип перечисления:

```
type Sreg0_type is (END_CYCLE, IDLE, OPEN_GATE);
```

Если теперь открыть окно View/Sort Objects, изменить порядок следования имен состояний в подокне States, как показано на рисунке 6.42, сохранить изменения и

сгенерировать VHDL-код, то тип Sreg0_type будет иметь следующий порядок следования элементов:

```
type Sreg0_type is (OPEN_GATE, IDLE, END_CYCLE);
```

Рисунок 6.42. Изменение порядка следования имен

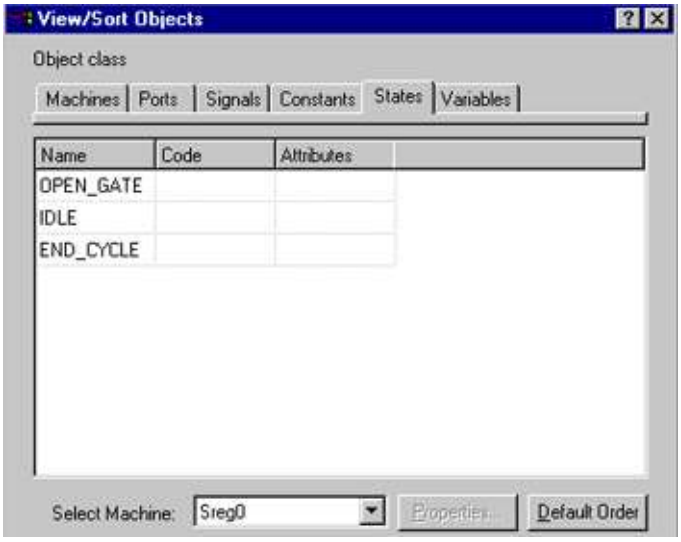


Диаграмма обычно содержит один граф автомата, однако сложные проекты могут включать в себя более одного автомата. Подокна States и Variables позволяют выбрать из списка Select Machine автомат, облегчая поиск информации (рисунок 6.43).

Рисунок 6.43. Выбор автомата из списка

Выводы. Описан редактор State Diagram, предназначенный для графического ввода последовательностных устройств. Поведение любого автомата может быть представлено в виде графа, а автоматизированная генерация HDL-кода на основе графа позволяет значительно упростить процесс создания и верификации моделей цифровых устройств. Рассмотрены основные элементы, составляющие диаграммы состояний, а также различные стили представления генерируемых VHDL-моделей.

6.14 Задачи

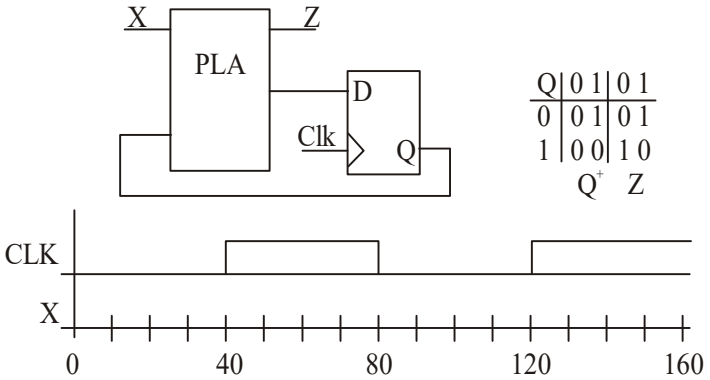
6.14.1. Создать диаграмму состояний для конечного автомата, имеющего вход X, выходы D и B. На вход X последовательно подается двоичное 4-битовое число N, начиная с младшего бита. D представляет 4-битовое число, равное N-2. Значение поступает на выход, начиная с младшего разряда. Во время приема четвертого бита выход B=1, если N-2 – отрицательное число, B=0 – в противном случае. Состояние автомата сбрасывается после поступления четвертого бита на вход X. Устройство использует синхронизацию по переднему фронту.

Сгенерировать однопроцессорную VHDL-модель. Откомпилировать код. Протестировать, используя следующую входную последовательность: X = 1011 0111 1000. Сигнал X изменяется в момент 1/4 периода после поступления заднего фронта синхронизации.

Сгенерировать двухпроцессорную VHDL-модель.

6.14.2. Создать диаграмму состояний и сгенерировать VHDL-описание для конечного автомата, представленного на рисунке 6.44.

Рисунок 6.44. Конечный автомат



Задержка автомата находится в пределах $5 \le t_c \le 20$ ns. Задержка между поступлением переднего фронта синхронизации и изменением выходного сигнала находится в пределах $5 \le t_c \le 10$ ns. Необходимое время установки и хранения данных для триггера t_{su} =10 ns и t_n=5 n_s. На диаграмме отмечены значения, в которых разрешены изменения сигнала.

6.14.3. Последовательностный автомат Мура с двумя входами (XI и X2) и одним выходом (Z) имеет следующую таблицу переходов и выходов:

	00	01	10	11	Z
1	1	2	2	1	0
2	2	1	2	1	1

Создать диаграмму состояний и сгенерировать VHDL-код. Изменения состояний в автомате происходят через 10 ns после поступления заднего фронта синхроимпульса, а изменения выхода – через 10 ns после изменения состояния.

ГЛАВА 7
ПРОЕКТИРОВАНИЕ АРИФМЕТИЧЕСКИХ УСТРОЙСТВ

Устройства двоичного умножения/деления рассматриваются в качестве примеров проектирования небольших цифровых систем. Описывается использование управляющих схем, контролирующих последовательность выполнения операций. В созданной поведенческой VHDL-модели тестируется правильность алгоритма. Затем определяются необходимые управляющие сигналы и действия, выполняемые при поступлении этих сигналов, создается и верифицируется VHDL-модель данного уровня. Для тестирования моделей также можно использовать язык VHDL

7.1. Проектирование последовательного сумматора с накоплением

Управляющее устройство представляет собой последовательностную схему, порождающую последовательность управляющих сигналов. Эти сигналы вызывают в необходимое время выполнение операций, например, таких как сложение или сдвиг. На рисунке 7.1 представлена структурная схема сумматора. Два сдвиговых регистра используются для хранения четырехбитных слагаемых, X и Y. Квадрат слева от каждого регистра изображает входы: Sh (сдвиг), SI (последовательный вход) и Clock (синхронизация). Если Sh=1, при поступлении активного синхроимпульса значение SI заносится в x_3 (или в y_3), а все содержимое регистра сдвигается вправо на одну позицию. Регистр X является аккумулятором, и после выполнения четвертой операции сдвига он содержит сумму X + Y. Подключение второго регистра формирует циклический сдвиговый регистр. После четвертой операции сдвига он возвращается в начальное состояние и значение Y не теряется. Последовательный сумматор состоит из полного сумматора и триггера переноса. В один момент времени выполняется сложение одной пары битов. Когда Sh = 1, по заднему фронту сихроимпульса бит суммы заносится в аккумулятор, а бит переноса сохраняется в триггере. Дополнительные связи, необходимые для начальной загрузки регистров (X и Y) и для очистки триггера переноса, на данном рисунке не показаны.

Рисунок 7.1. Последовательный сумматор с накоплением

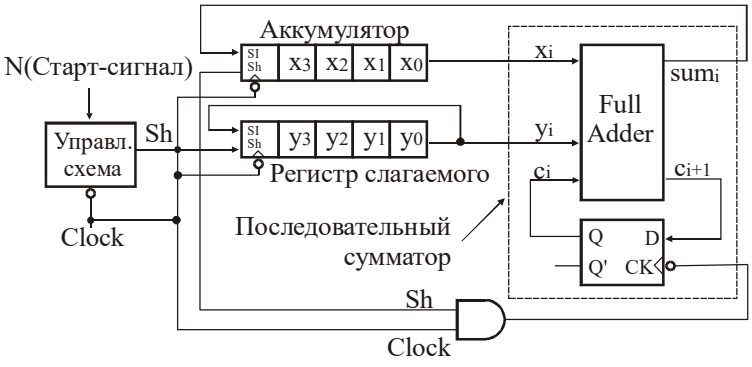


Таблица 7.1 описывает операции последовательного сумматора. Здесь представлены моменты времени: t_0 – до первого синхроимпульса, t_1 – после первого синхроимпульса, t_2 – после второго синхроимпульса и т.д. В начальный момент времени аккумулятор содержит число X, второй регистр содержит Y и триггер переноса находится в состоянии 0. Поскольку полный сумматор – это комбинационная схема, то сложение $x_0 = 1$, $y_0 = 1$ и $c_0 = 0$ будет выполнено после небольшой задержки, например 10 ns, с формированием результата: $sum_0 = 0$ и перенос $c_1 = 1$. После первого фронта синхросигнала сумма sum_0 заносится в аккумулятор и оставшееся в нем число

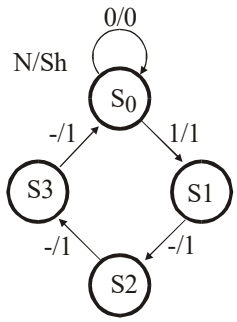
сдвигается вправо на одну позицию. Тогда же перенос c_1 сохраняется в триггере и выполняется сдвиг второго регистра вправо на 1 позицию. На входы полного сумматора поступает следующая пара битов: $x_1 = 0$ и $y_1 = 1$. Результат сложения: $sum_1 = 0$ и $c_1 = 1$. По второму синхроимпульсу сумма sum_1 поступает в аккумулятор, c_2 сохраняется в триггере переноса, выполняется циклический сдвиг второго регистра. Биты x_2 и y_2 подаются теперь на входы сумматора и процесс продолжается, пока не будет выполнено сложение всех пар битов. После четвертого синхроимпульса в аккумуляторе находится сумма X и Y, а второй регистр возвращается в начальное состояние.

Таблица 7.1. Операции последовательного сумматора

	X	Y	c_i	sum_i	c_{i+1}
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)

Далее необходимо спроектировать управляющую схему, которая после поступления старт-сигнала генерирует сигнал *Sh* = 1 на протяжении четырех тактов, а затем изменяет его в 0. На рисунке 7.2 представлены граф состояний и таблица переходов управляющего автомата. Вначале схема находится в состоянии S_0. После поступления старт-сигнала *N* генерируется сигнал *Sh* = 1 и схема переходит в состояние S_1. Затем на протяжении трех тактов *Sh* = 1, после чего схема возвращается в состояние S_0. Можно допустить, что старт-сигнал заканчивается до того, как схема возвращается в состояние S_0. Следовательно, никаких действий не будет выполнено до поступления следующего старт-сигнала. На графе черточка означает несущественность значения сигнала *N*.

Рисунок 7.2. Граф состояний и таблица переходов управляющей схемы



Present State	Next State N=0	Next State N=1	Present Output N=0	Present Output N=1
S_0	S_0	S_1	0	1
S_1	S_2	S_2	1	1
S_2	S_3	S_3	1	1
S_3	S_0	S_0	1	1

7.2. Граф состояний управляющей схемы

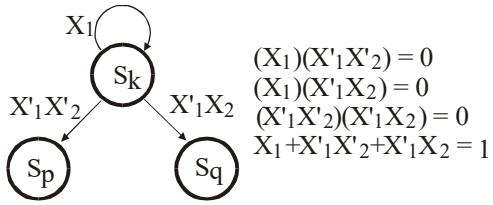
Прежде чем рассмотреть следующие примеры проектирования, необходимо принять обозначения, используемые в графах состояний. Граф автомата отмечается с помощью имен, вместо нулей и единиц. Это облегчает его анализ, особенно, если присутствует большое число входов и выходов. Если дуга графа автомата Мили имеет метку $X_iX_j/Z_p\ Z_q$, это означает, что переход в указанное состояние будет выполнен при наличии 1 на входах X_i и X_j (значения сигналов на других входах несущественны), при этом выходы Z_p и Z_q равны 1 (значение остальных выходных сигналов равно 0). Например, схема имеет 4 входа (X_1, X_2, X_3, X_4) и 4 выхода (Z_1, Z_2, Z_3, Z_4), метка $X_1X_4'/Z_2\ Z_3$ будет эквивалентна 1 - - 0/0 1 1 0. В общем случае, если метка дуги представляет собой выражение I, переход по дуге будет выполнен, когда I=1. Например, если

AB + C' является входной меткой, переход будет выполнен при наличии условия AB+C'= 1.

Для того чтобы правильно описать граф состояний, в котором следующее состояние однозначно определено для всех наборов входных комбинаций, для каждого состояния S_k метки должны удовлетворять следующим условиям:

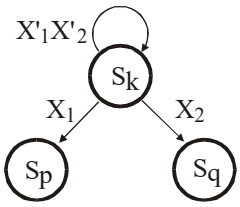
1. Если I_i и I_j – любая пара меток на дугах графа переходов, выходящих из состояния S_k, то $I_iI_j = 0$ для $i \neq j$.
2. Если из состояния S_k выходит n дуг, которые имеют метки $I_1, I_2 ..., I_n$ соответственно, то $I_1+I_2+...+I_n=1$.

Условие 1 гарантирует, что максимум одна метка будет равна 1 в любой момент времени, а условие 2 – что как минимум одна метка будет равна 1 в любой момент времени. Таким образом, только одна метка будет равна 1 и следующее состояние будет однозначно определено для любой входной комбинации. Рассмотрим следующий фрагмент графа, где $I_1=X_1$, $I_2=X_1'X_2'$ и $I_3=X_1'X_2$:



Условия 1 и 2 выполняются для S_k.

Неполностью описанный граф состояний должен всегда удовлетворять условию 2, а также условию 1 для всех комбинаций значений входных переменных, возможных в каждом состоянии. Таким образом, следующий фрагмент представляет правильный граф, если входная комбинация $X_1=X_2=1$ не может возникнуть в состоянии S_k:



Для трех переменных (X_1, X_2, X_3) предыдущему фрагменту графа соответствует следующая таблица переходов:

	000	001	010	011	100	101	110	111
S_k	S_k	S_k	S_q	S_q	S_p	S_p	-	-

7.3. Проектирование двоичного устройства умножения

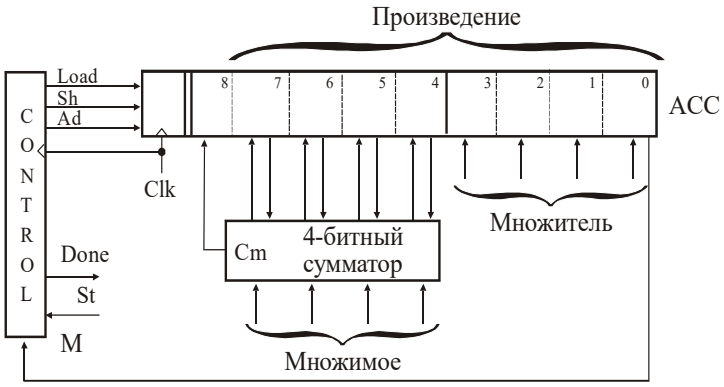
Рассмотрим пример проектирования устройства умножения для беззнаковых двоичных чисел. В выражении A × B первый оператор (A) называют множимое, второй (B) – множитель. Двоичное умножение требует только операций сложения и сдвига. В качестве примера выполняется умножение двоичных чисел: $13_{10} \times 11_{10}$.

```
Множимое ──────────► 1101 (13)
Множитель ─────────► 1011 (11)
                      1101
                     1101
Частичные            100111
произведения         0000
                     100111
                    1101
                   10001111 (143)
```

Следует обратить внимание, что каждое частичное произведение является или множимым (1101), сдвинутым на определенное число позиций, или 0. Вместо того чтобы формировать все частичные произведения и выполнять их сложение одновременно, прибавление каждого частичного произведения выполняется сразу после его формирования, другими словами, в один момент времени складывается не более двух двоичных чисел.

Для умножения двух 4-битных двоичных чисел требуются 4-битные регистры: множимого, множителя, 4-битный полный сумматор и 8-битный регистр для хранения результата. Регистр произведения представляет собой аккумулятор для хранения суммы частичных произведений. В случае, когда множимое сдвигается влево, каждый раз до его сложения с акумулятором, как показано в предыдущем примере, необходим 8-битный сумматор. Иначе, можно сдвигать содержимое регистра каждый раз вправо, как показано на рисунке 7.3. Такой тип утройства умножения часто называют последовательно-параллельным, так как умножение выполняется последовательно, но сложение выполняется параллельно. Как показано на рисунке 7.3, четыре бита из аккумулятора ACC и четыре бита из регистра множимого поступают на входы сумматора, четыре бита суммы и перенос возвращаются обратно в аккуммулятор. При поступлении сигнала Ad значения с выходов сумматора передаются в аккумулятор по следующему синхроимпульсу, что приводит к тому, что значение из аккумулятора будет сложено с множимым. Дополнительный левый бит используется для хранения переноса. При поступлении сигнала сдвига Sh все 9 битов из ACC сдвигаются вправо на 1 позицию по следующему синхросигналу.

Рисунок 7.3. Структурная схема устройства двоичного умножения



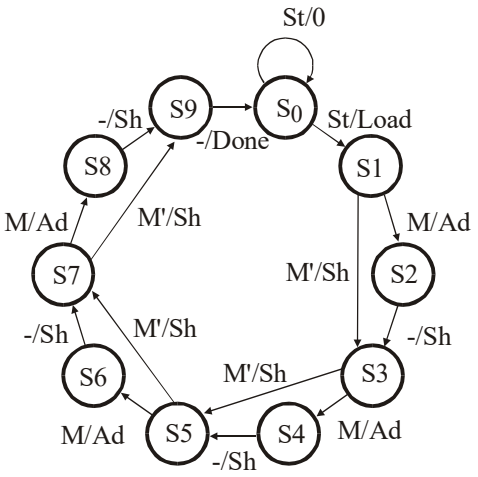
Поскольку вначале 4 младших бита регистра произведения не используются, их можно задействовать для хранения множителя, вместо выделения для него отдельного регистра. После того как каждый бит множителя был использован, он выталкивается из регистра, освобождая место для бита результата. Сигнал Sh вызывает сдвиг регистра произведения (включая множитель) вправо на одну позицию по следующему синхроимпульсу. Управляющая цифровая схема, после поступления на нее старт-сигнала St = 1, генерирует необходимую последовательность сигналов сложения и сдвига. Если текущий бит множителя M равен 1, множимое складывается с аккумулятором, после чего выполняется сдвиг вправо. Если бит множителя M = 0, операция

сложения пропускается и выполняется только сдвиг вправо. Ниже приведен пример умножения (13×11) с указанием значений битов в регистрах на каждом такте:

```
первоначальное содержание регистра произведения  00000|1011 ←M(11)
(сложение с множимым, так как M=1)                 1101      (13)
после сложения                                   01101|1011
после сдвига                                     001101|101 ←M
(сложение с множимым, так как M=1)                 1101
после сложения                                   100111|101
после сдвига                                     010011|110 ←M
(пропуск операции сложения, так как M=0)
после сдвига                                     001001|111 ←M
(сложение с множимым, так как M=1)                 1101
после сложения                                   100011|111
после сдвига(окончательный результат)            010001|111   (143)
                  Линия, разделяющая произведение и множимое
```

Целью функционирования управляющей схемы является создание правильной последовательности сигналов сложения и сдвига. На рисунке 7.4 представлен граф состояний управляющей схемы. Состояние S_0 – начальное. В нем схема находится пока не получит старт сигнал St = 1. После этого генерируется сигнал Load, по которому множитель загружается в 4 младших бита регистра результата, а остальные биты регистра сбрасываются в 0. В состоянии S_1 тестируется младший бит множителя M. Если M = 1, генерируется сигнал сложения, иначе генерируется сигнал сдвига. Аналогичным образом анализируется текущий бит множителя в состояниях S_3, S_5 и S_7. После сигнала сложения по следующему синхросигналу всегда генерируется сигнал сдвига (состояния S_2, S_4, S_6 и S_8). После выполнения четвертого сдвига управляющая схема переходит в состояние S_9 и формирует сигнал Done, после чего возвращается в начальное состояние S_0.

Рисунок 7.4. Граф состояний схемы управления двоичным умножением



На рисунке 7.5 приведена VHDL-модель, соответствующая графу состояний (см. рисунок 7.4). Поскольку граф содержит 10 состояний, они описываются сигналом типа integer с диапазоном от 0 до 9. Сигнал ACC соответствует 9-битному выходу аккумулятора. Оператор

```
alias M: bit is ACC(0);
```

позволяет использовать имя M вместо ACC(0). Обозначение 1|3|5|7 => означает, что в состояниях 1, 3, 5 или 7 будут выполняться указанные действия. Все операции над регистрами и изменения состояний происходят по переднему фронту синхроимпульса. Например, в состоянии 0, если St = 1, множитель загружается в аккумулятор в момент, когда синхросигнал изменяется в 1. Функция add4 вычисляет сумму двух 4-битных

векторов и переноса, возвращая 5-битный результат. Это соответствует выходам сумматора, значения с которых поступают на ACC, в этот же момент счетчик состояний увеличивается на 1. Выполняется сдвиг ACC вправо с занесением 0 в восьмой бит. Выражение '0'&ACC(8 downto 1) можно заменить на ACC **srl** 1.

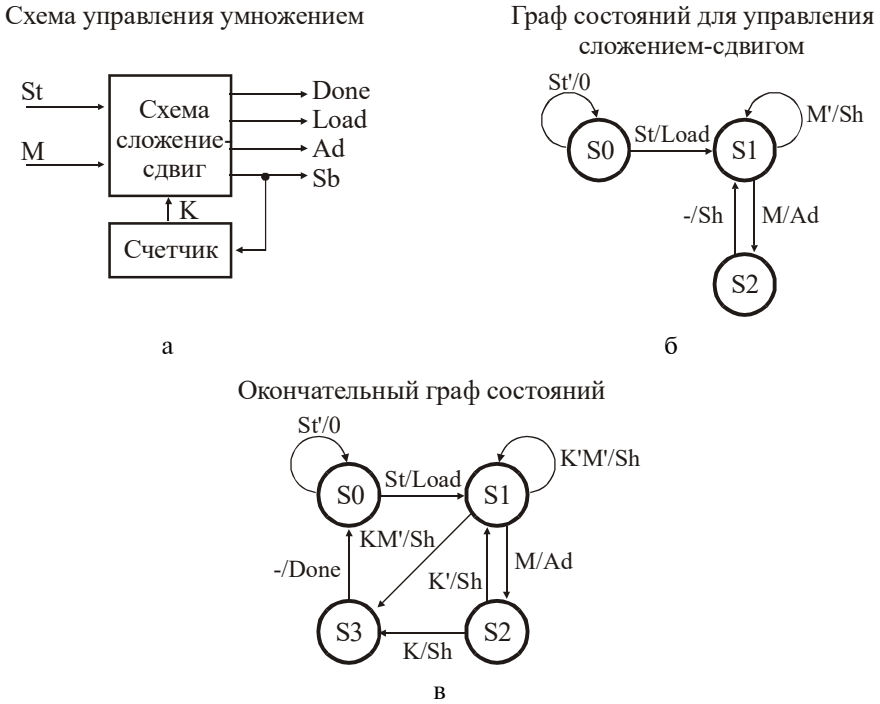
Сигнал Done должен генерироваться только в состоянии 9. Если записать **when** 9 => State <= 0; Done <= '1', сигнал Done будет установлен в тот момент, когда схема перейдет в состояние 0. Это слишком поздно, так как сигнал должен быть установлен, когда схема находится в состоянии 9. Поэтому вводится отдельный параллельный оператор назначения сигнала. Этот оператор находится вне процесса, следовательно, сигнал *Done* будет установлен, как только *State* изменится.

Рисунок 7.5. Поведенческая модель устройства двоичного умножения 4 x 4

```
-- Это поведенческая модель устройства умножения беззнаковых
-- двоичных чисел. Выполняет умножение 4-битового множимого
-- на 4-битовый множитель, генерируя 8-битовый результат.
-- Максимальное число тактов, необходимых для умножения, раняется 10

library BITLIB;
use BITLIB.BitPackage.all;

entity mult4X4 is
   port (Clk, St: in bit;
      Mplier,Mcand : in bit_vector(3 downto 0);
      Done: out bit) ;
end mult4X4;

architecture behavel of mult4X4 is
signal State: integer range 0 to 9;
signal ACC: bit_vector(8 downto 0);-- аккумулятор
alias M: bit is ACC(0);               -- M это нулевой бит ACC
begin
   process
   begin
      wait until Clk = '1';
                -- работает по переднему фронту синхроимпульса
      case State is
         when 0=>-- Начальное состояние
            if St='1' then
               ACC(8 downto 4) <= "00000";   -- начало цикла
               ACC(3 downto 0) <= Mplier;    -- Загрузка множителя
               State <= 1;
            end if;
         when 1|3|5|7 =>              -- состояние "сложение/сдвиг"
            if M = '1' then           -- добавление множимого
               ACC(8 downto 4) <=add4(ACC(7 downto 4),Mcand,'0');
               State <= State+1;
            else
               ACC <= '0' & ACC(8 downto 1); -- Сдвиг аккум. вправо
               State <= State + 2;
            end if;
         when 2 | 4 | 6 | 8 =>            -- Состояние "сдвиг"
            ACC <= '0' & ACC(8 downto 1);    -- Сдвиг вправо
            State <= State + 1;
         when 9 =>                           -- конец цикла
            State <= 0;
      end case;
   end process;
   Done <= '1' when State = 9 else '0' ;
end behavel;
```

Как видно из графа состояний (см. рисунок 7.4), управляющая схема выполняет две функции: генерацию сигналов сложения или сдвига и счет числа сдвигов. Для увеличения количества битов удобнее разделить управляющую схему на две:счетчик и сложение-сдвиг, как показано на рисунке 7.6, а. Вначале создается граф для управляющей схемы "сложение-сдвиг", которая тестирует сигналы St и M и выдает последовательность сигналов сдвига и сложения (см. рисунок 7.6, б). Затем добавляется к счетчику выходной сигнал завершения K, который останавливает умножение после выполнения определенного числа сдвигов. В состоянии S0 (рисунок 7.6, б), после поступления St = 1 генерируется сигнал загрузки и схема переходит в состояние S1. Затем, если M= 1, генерируется сигнал сложения и схема переходит в состояние S2; если M= 0, генерируется сигнал сдвига и схема остается в состянии S1. В S2 генерируется сигнал сдвига, следовательно, сдвиг всегда следует за сложением. Граф с рисунка 7.6, б генерирует правильную последовательность сигналов сложения и сдвига, но он не обеспечивает прекращение умножения.

Рисунок 7.6. Управляющая схема со счетчиком



Для того чтобы определить момент завершения умножения, при каждой генерации сигнала сдвига счетчик увеличивается на 1. Если множитель содержит n битов, то столько сдвигов и необходимо. Счетчик будет генерировать сигнал завершения после выполнения n-1 сдвига. После того, как поступил сигнал K = 1, схема выполнит, если необходимо, еще одно сложение и последнюю операцию сдвига. Управляющие операции на рисунке 7.6, в такие же, как и на рисунке 7.6, б, пока K=0. В состоянии S1, если K=1, как обычно тестируется бит M. Если M=0, формируется последний сигнал сдвига и выполняется переход в состояние S3. Если M=1, выполняется сложение перед сдвигом и переход в S2. В состоянии S2, если K=1, генерируется сигнал сдвига и переход в состояние S3. Последний сигнал сдвига переводит счетчик в 0 в момент, когда управляющая схема переходит в состояние S3.

В качестве примера рассмотрим устройство умножения с рисунка 7.3, но заменим управляющую часть схемой с рисунка 7.6, а. Так как n=4, для подсчета 4 сдвигов необходим 2-битный счетчик и сигнал K=1, когда счетчик находится в состоянии 3 (11_2). В таблице 7.2 представлены этапы умножения 1101 на 1011. S0, S1, S2 и S3 представляют состояния схемы управления (см. рисунок 7.6, в).Содержимое регистра состояния такое же, как и в рассмотренном выше примере (см. рисунок 7.5).

Таблица 7.2. Этапы умножения с использованием счетчика

Время	Состояние	Счетчик	Регистр произведения	St	M	K	Load	Ad	Sh	Done
t_0	S0	00	0 0 0 0 0 0 0 0 0 0	0	0	0	0	0	0	0
t_1	S0	00	0 0 0 0 0 0 0 0 0 0	1	0	0	1	0	0	0
t_2	S1	00	0 0 0 0 0 0 1 0 1 1	0	1	0	0	1	0	0
t_3	S2	00	0 1 1 0 1 1 0 1 1	0	1	0	0	0	1	0
t_4	S1	01	0 0 1 1 0 1 1 0 1	0	1	0	0	1	0	0
t_5	S2	01	1 0 0 1 1 1 1 0 1	0	1	0	0	0	1	0
t_6	S1	10	0 1 0 0 1 1 1 1 0	0	0	0	0	0	1	0
t_7	S1	11	0 0 1 0 0 1 1 1 1	0	1	1	0	1	0	0
t_8	S2	11	1 0 0 0 1 1 1 1 1	0	1	1	0	0	1	0
t_9	S3	00	0 1 0 0 0 1 1 1 1	0	1	0	0	0	0	1

В момент времени t_0 схема управления находится в начальном состоянии и ожидает старт-сигнала. В момент времени t_1 старт-сигнал St = 1 и генерируется сигнал загрузки Load. В момент времени t_2 M = 1, следовательно, генерируется сигнал сложения Ad. При поступлении следующего синхроимпульса значения с выходов сумматора загружаются в аккумулятор, а управляющая схема переходит в состояние S2. В момент времени t_3 генерируется сигнал сдвига Sh, значит, по следующему синхроимпульсу будет выполнен сдвиг и счетчик увеличится на 1. В момент времени t_4 M = 1, поэтому Ad = 1 и значение с сумматора загружается в аккумулятор по следующему синхроимпульсу. В моменты времени t_5 и t_6 выполняются сдвиги и увеличение счетчика. В момент t_7 производится третий сдвиг и счетчик переходит в состояние 11, следовательно, K=1. Так как M = 1, выполняется сложение и управляющая схема переходит в состояние S2. В момент t_8 Sh = K = 1, значит, по следующему синхроимпульсу будет выполнен последний сдвиг и счетчик вернется в состояние 00. В момент времени t_9 генерируется сигнал Done.

Спроектированное таким образом устройство умножения может быть легко расширено до 8, 16 и более битов с помощью увеличения размеров регистров и числа битов в счетчике. Управляющая схема "сложение-сдвиг" останется без изменения.

Далее проектируется устройство умножения, состоящее из массива И элементов и сумматоров. Эта схема будет иметь итеративную структуру без какой-либо последовательностной логики или регистров. Таблица 7.3 иллюстрирует умножение двух 4-битных беззнаковых чисел $X_3X_2X_1X_0$. и $Y_3Y_2Y_1Y_0$. Каждый X_iY_j результирующий бит будет генерироваться И элементом. Для добавления каждого частичного произведения к предыдущей сумме используется ряд сумматоров. $S_{13}S_{12}S_{11}S_{10}$ – это выходы с первого ряда сумматоров, формирующие первое частичное произведение, $C_{13}C_{12}C_{11}C_{10}$ – выходы переноса.Подобные результаты генерируются остальными двумя рядами сумматоров. Обозначения S_{ij} и C_{ij} используются для представления суммы и переноса из каждого ряда сумматоров. На рисунке 7.7 представлена соответствующая схема, состоящая из И элементов и сумматоров. Если сумматор имеет три входа, используется полный сумматор (FA). Если только два входа, используется полусумматор (HA). Полусумматор представляет собой полный сумматор, имеющий 0 на одном входе. Такое устройство умножения требует 16 И элементов, 8 полных сумматоров и 4 полусумматора. После поступления на входы X и Y перенос должен распространиться через каждый ряд, а сумма передаваться от ряда к ряду. Время, необходимое на выполнение операции умножения, зависит, главным образом, от задержек сумматоров. Самый длинный путь от входа до выхода состоит из 8

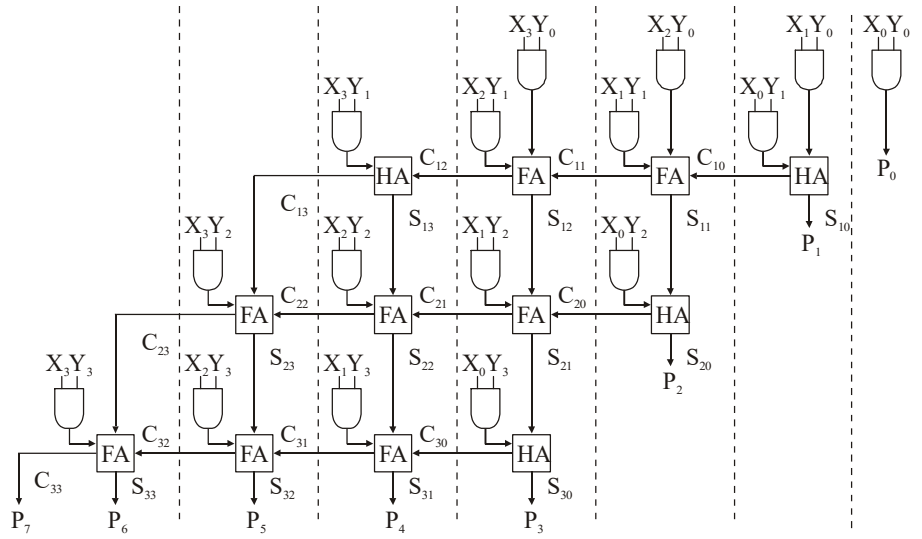
сумматоров. Если t_{ad} – это максимальная возможная задержка на сумматоре и t_g – максимальная задержка И вентиля, то максимальное время умножения равно $8t_{ad}+t_g$.

Таблица 7.3. Частичные произведения 4-битного умножения

			X_3	X_2	X_1	X_0		Множимое
			Y_3	Y_2	Y_1	Y_0		Множитель
			X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0		частичное произведение 0
		X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1			частичное произведение 1
		C_{12}	C_{11}	C_{10}				первая строка переносов
	C_{13}	S_{13}	S_{12}	S_{11}	S_{10}			первая строка сумм
	X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2				частичное произведение21
	C_{22}	C_{21}	C_{20}					вторая строка переносов
C_{23}	S_{23}	S_{22}	S_{21}	S_{20}				вторая строка сумм
X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3					частичное произведение31
C_{32}	C_{31}	C_{30}						третья строка переносов
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				третья строка сумм
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	окончательное произведение

В общем случае, умножение двух n-битных операндов требует n^2 И элементов, а также n(n - 2) полных сумматоров и n полусумматоров. Таким образом, увеличение числа элементов определяется квадратичной зависимостью. Для предыдущего последовательно-параллельного устройства умножения увеличение аппаратуры описывается линейной функцией от n.

Рисунок 7.7. Структурная схема матричного устройства умножения



Для $n \times n$ матричного устройства умножения самый длинный путь от входов до выходов проходит через 2n сумматоров и, соответственно, самое худшее время умножения равно $2nt_{ad} + t_g$. Спроектированное ранее последовательно-параллельное устройство умножения требует максимум 2n тактов для выполнения умножения, хотя можно еще сократить число тактов, используя метод, описанный далее. Минимальная продолжительность синхротакта зависит от задержки n-битного сумматора, а также от задержки и времени установки триггеров аккумулятора.

7.4. Умножение знаковых двоичных чисел

Существует несколько алгоритмов для умножения знаковых двоичных чисел. Следующая процедура представляет собой простой способ для реализации их умножения.

1. Получить дополнительный код множителя, если он отрицательный.

2. Получить дополнительный код множимого, если оно отрицательное.
3. Перемножить два положительных двоичных числа.
4. Получить дополнительный код произведения, если оно должно быть отрицательным.

Хотя этот метод является концептуально простым, он требует больше аппаратурных и вычислительных затрат, чем некоторые другие доступные методы.

Следующий метод требует только получения дополнительного кода множимого без дополнения множителя и произведения. Хотя алгоритм работает одинаково хорошо и с целыми числами, и с дробными, для его иллюстрации будут использованы дробные числа. Применяя дополнительный код двоичных чисел, двоичные дроби можно представить в следующе форме:

```
0.101   +5/8   1.011   -5/8
```

Число слева от двоичной точки является знаковым битом, 0 – для положительной дроби, 1 – для отрицательной. В общем случае дополнительный код двоичной дроби F – это F* = 2 – F. Таким образом, дополнительный код -5/8 будет равен 10.000 - 0.101 = 1.011. Этот метод определения дополнительного кода двоичных дробей подходит и для целых чисел ($N^* = 2^n - N$), так как перемещение на n - 1 позицию влево эквивалентно делению на 2^{n-1}. Дополнительный код для дроби может быть получен, начиная с правой части, дополнением всех цифр левее первой единицы, точно так же, как и для целых чисел. Обратный код 1.000... – это особый случай. Таким образом, обычно представляется число -1, так как знаковый бит отрицательный и двоичное дополнение 1.000 будет определяться как 2 – 1 = 1. Нельзя представить число +1 в форме дробного дополнительного кода, так как 0.111... – это самая большая положительная дробь.При умножении знаковых двоичных чисел необходимо рассмотреть четыре случая:

Множимое	Множитель
+	+
-	+
+	-
-	-

Если множимое и множитель положительны, используется стандартное двоичное умножение. Например,

```
        0.1 1 1    (+7/8) <-- Множимое
      x 0.1 0 1    (+5/8) <-- Множитель
(0. 0 0)0 1 1 1    (+7/64)<-- Примечание:правильное представление
(0.)0 1 1 1        (+7/16)<-- дробных частичных произведений требует
 0. 1 0 0 0 1 1    (+35/64)  расширения знакового бита за двоичной
                             точкой,как это показано в скобках. (Такое
                             расширение не является обязательным для
                             аппаратуры)
```

Когда множимое отрицательно, а множитель положителен, процедура умножения такая же, как и в первом случае, за исключением того, что необходимо определить знаковый бит множителя таким образом, чтобы частичные произведения и результирующее произведение имели отрицательный знак. Например,

```
        1.1 0 1    (-3/8)
      x 0.1 0 1    (+5/8)
(1. 1 1)1 1 0 1    (-3/64)<-- Примечание:расширение знакового бита
(1.)1 1 0 1        (-3/16)<-- обеспечивает правильное представление
 1. 1 1 0 0 0 1    (-15/64)  отрицательного произведения
```

Когда множитель отрицателен, а множимое положительно, необходимо внести небольшие изменения в процедуру умножения. Отрицательная дробь следующей формы имеет значение -1 + 0.*g;* например, 1.011 = –1+0.011 = –(1–0.011) = –0.101 = –5/8. Таким образом, при умножении на отрицательную дробь вида 1.*g* дробная часть рассматривается как положительная, а знаковый бит обрабатывается как -1. Поэтому для дробной части множителя умножение происходит обычным способом и сумма частичных произведений накапливается в аккумуляторе. Тем не менее при достижении знакового бита выполняется сложение частичных произведений с двоичным дополнением множимого, вместо самого множимого. Этот случай иллюстрирует следующий пример:

```
        0.1 0 1    (+5/8)
      x 1.1 0 1    (-3/8)
(0. 0 0)0 1 0 1    (+5/64)
(0.)0 1 0 1        (+5/16)
 0. 0 1 1 0 0 1
 1. 0 1 1          (-5/8) ←— Примечание: выполняется сложение
 1. 1 1 0 0 0 1    (-15/64)  с дополнительным кодом множимого
```

Когда множимое и множитель отрицательны, процедура умножения аналогична рассмотренной выше. На каждом шаге необходимо расширять знаковый бит частичных произведений для представления правильного отрицательного знака, на последнем шаге выполняется сложение с множимым в дополнительном коде, так как знаковый бит множителя отрицателен. Например,

```
        1.1 0 1    (-3/8)
      x 1.1 0 1    (-3/8)
(1. 1 1)1 1 0 1    (-3/64)
(1.)1 1 0 1        (-3/16)
 1. 1 1 0 0 0 1
 0. 0 1 1          (+3/8) ←— Примечание: выполняется сложение
 0. 0 0 1 0 0 1    (+9/64)  с дополнительным кодом множимого
```

Другими словами, процедура умножения знаковых двоичных дополнителных дробей такая же, как и для умножения двоичных положительных дробей, за исключением того, что необходимо сохранять знак частичных произведений на каждом шаге и нужно дополнять множимое перед добавлением его на последнем шаге. Схема состоит из тех же компонентов, что и для умножения положительных чисел, плюс дополняющая схема (complementer) для множимого.

На рисунке 7.8 представлена схема для умножения 4-битных дробей (включая знаковый бит). Используется 5-битный сумматор, чтобы знак суммы не был потерян из-за переноса в знаковый разряд. Вход M управляющей схемы — это активный бит множителя. Управляющий сигнал Sh вызывает сдвиг аккумулятора вправо на одну позицию, включая расширение знака. По сигналу Ad значения с выходов сумматора ADDER заносятся в пять левых битов аккумулятора. Перенос из последнего бита сумматора отбрасывается, поскольку сложение выполняется в дополнительном коде. По сигналу Cm перед загрузкой в сумматор множимое преобразуется в обратный код. Сигнал Cm также подается на вход переноса в сумматор. Таким образом, когда Cm = 1, в сумматоре выполняется сложение с обратным кодом и переносом, равным 1, что эквивалентно сложению с дополнительным кодом. На рисунке 7.9 представлен граф состояний управляющей схемы. Она тестирует каждый бит множителя M для опреде-

ления выполнения сложения и сдвига или просто сдвига. В состоянии S_7 M представляет собой знаковый бит, а если M = 1, дополнительный код множимого складывается с содержимым аккумулятора.

Рисунок 7.8. Структурная схема устройства умножения в дополнительном коде

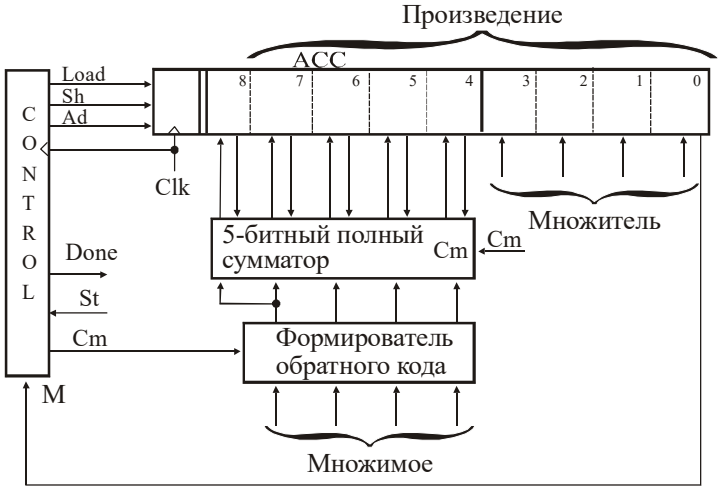
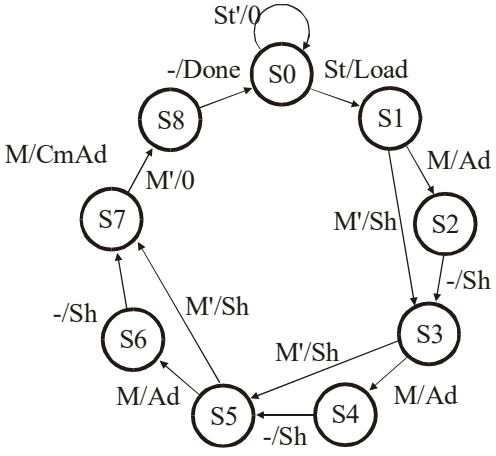


Рисунок 7.9. Граф состояний для умножения в дополнительном коде



При использовании схемы с рисунка 7.8 операции сложения и сдвига должны выполняться в двух различных синхротактах. Можно повысить скорость выполнения умножения, переместив выходы с сумматора на одну позицию вправо (рисунок 7.10). Таким образом, значения суммы будут уже сдвинуты вправо на одну позицию при загрузке в аккумулятор. В такой схеме операции сложения и сдвига могут выполняться в один и тот же синхротакт. Граф состояний управляющей схемы автомата представлен на рисунке 7.11. По окончании умножения результат, 6 битов и знак размещаются в трех младших битах регистра A и в регистре B. Двоичная точка, таким образом, находится в середине регистра A. Если необходимо разместить ее между двумя левыми позициями, следует выполнить сдвиг обоих регистров A и B влево на одну позицию.

Рисунок 7.10. Структурная схема быстрого устройства умножения

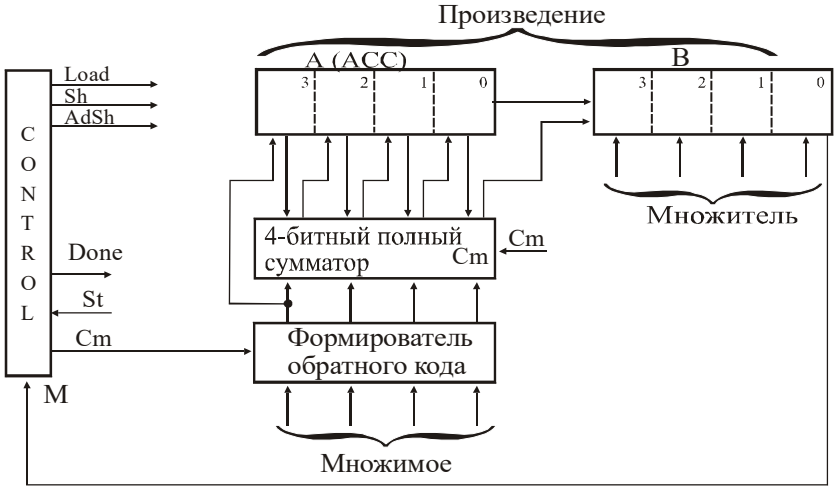
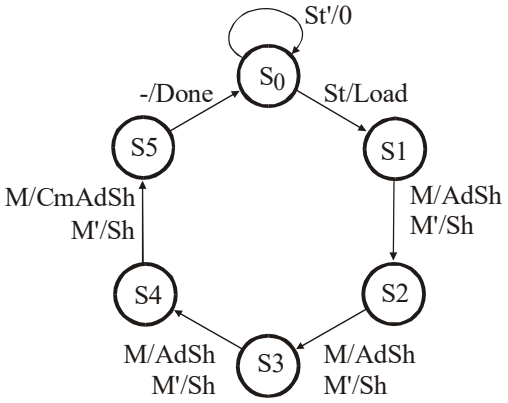


Рисунок 7.11. Граф состояний быстрого устройства умножения



На рисунке 7.12 представлена поведенческая VHDL-модель устройства умножения. Сдвиг вправо регистров A и B реализуется с помощью последовательных операторов

```
A <= A(3) & A(3 downto 1);
B <= A(0) & B(3 downto 1);
```

Хотя эти операторы выполняются последовательно, A и B получат новые значения в одно и то же время. Таким образом, для вычисления B используется старое значение A(0).

Переменная addout применяется для представления 5-битного выхода сумматора. В состояниях с 1 по 4, если текущий бит множитемя M равен 1, знаковый бит множимого и три бита переменной addout загружаются в A. В то же самое время младший бит A вместе с тремя старшими битами B загружаются в B. Сигнал Done устанавливается, когда схема переходит в состояние 5. Задержка **wait for** 0 ns необходима, чтобы регистры A и B обновили значения до выдачи результата.

Рисунок 7.12. VHDL-модель устройства умножения в дополнительном коде

```
library BITLIB;
use BITLIB.BitPackage.all;

entity mult2C is
   port (CLK, St: in bit;
      Mplier,Mcand : in bit_vector(3 downto 0);
      Product: out bit_vector (6 downto 0);
```

```
      Done: out bit) ;
end mult2C;

architecture behavel of mult2C is
   signal State : integer range 0 to 5;
   signal A, B: bit_vector(3 downto 0);
   alias M: bit is B(0);
begin
   process
   variable addout: bit_vector(4 downto 0);
   begin
      wait until CLK = '1' ;
      case State is
         when 0=>                              --начальное состояние
            if St='1' then
               A <= "0000";                    -- Начало цикла
               B <= Mplier;                    -- загрузка множителя
               State <= 1;
            end if;
         when 11213 =>                         -- состояние "add/shift"
            if M = '1' then
               -- Сложение множимого с A и сдвиг
               addout := add4(A,Mcand,'0');
               A <= Mcand(3) & addout(3 downto 1);
               B <= addout(0) & B(3 downto 1);
            else
               A <= A(3) & A(3 downto 1);    -- Арифм. сдвиг вправо
               B <= A(0) & B(3 downto 1);
            end if;
               State <= State + 1;
         when 4 =>       -- сложение с дополнительным кодом,
            if M = '1' then       -- если знак множителя равен 1
               addout := add4(A, not Mcand,'1');
               A <= not Mcand(3) & addout(3 downto 1);
               B <= addout(0) & B(3 downto 1);
            else
               A <= A(3) & A(3 downto 1);    -- Арифм. сдвиг вправо
               B <= A(0) & B(3 downto 1);
            end if;
            State <= 5;
            wait for 0 ns;
               Done <= '1' ;
               Product <= A (2 downto 0) & B;
         when 5 =>                      -- генерация значений выходов
            State <= 0;
            Done <= '0';
      end case;
   end process;
end behave1;
```

Перед тем как продолжить проектирование, необходимо протестировать VHDL-модель поведенческого уровня, чтобы убедиться в правильности алгоритма и его корректности в работе с моделями структурной схемы. На первом этапе тестирования алгоритм выполняется пошагово, для проверки внутренних операций умножения и облегчения отладки, если она необходима. Если предположить, что алгоритм умножения функционирует правильно, можно проверить результат умножения. Тестирование в большинстве случаев так обычно и выполняется.

На рисунке 7.13 представлен командный файл и результаты моделирования для случая умножения +5/8 на -3/8. Синхросигнал имеет период 20 ns. Сигнал *St* устанавливается в момент 2 ns и сбрасывается на один такт позже. Из анализа графа состояний видно, что умножение выполняется в течение шести тактов, поэтому задано время выполнения 120 ns. Результат соответствует примеру, приведенному выше.

Рисунок 7.13. Командный файл и результаты моделирования для реализации функции (+5/8 умножить на -3/8)

```
--  Командный файл для тестирования умножения
list CLK St State A B Done Product
force st 1 2 ns, 0 22 ns
force - repeat 20 ns clk 1 0 ns, 0 10 ns
--(5/8 * -3/8)
force Mcand 0101
force Mplier 1101
run 120 ns
```

ns	delta	CLK	St	State	A	B	Done	Product
0	+1	1	0	0	0000	0000	0	0000000
2	+0	1	1	0	0000	0000	0	0000000
10	+0	0	1	0	0000	0000	0	0000000
20	+1	1	1	1	0000	1101	0	0000000
22	+0	1	0	1	0000	1101	0	0000000
30	+0	0	0	1	0000	1101	0	0000000
40	+1	1	0	2	0010	1110	0	0000000
50	+0	0	0	2	0010	1110	0	0000000
60	+1	1	0	3	0001	0111	0	0000000
70	+0	0	0	3	0001	0111	0	0000000
80	+1	1	0	4	0011	0011	0	0000000
90	+0	0	0	4	0011	0011	0	0000000
100	+2	1	0	5	1111	0001	1	1110001
110	+0	0	0	5	1111	0001	1	1110001
120	+1	1	0	0	1111	0001	0	1110001

Для тщательного тестирования устройства умножения необходимо промоделировать не только четыре стандартных случая (+ +,+–,– +, – –), но и специальные, и крайние случаи. Значения множимого и множителя для теста должны включать 0, максимальную положительную дробь, минимальную отрицательную дробь и все 1. Необходимо написать VHDL testbench для тестирования устройства умножения. Такой testbench (рисунок 7.14) содержит последовательность значений для множимого и множителя. Он также генерирует сигнал синхронизации и старт-сигнал. Значения для множимого и множителя записаны в массивы констант. В цикле for выполняется чтение значений из массивов и устанавливается старт-сигнал в '1'. По следующему синхросигналу старт-сигнал сбрасывается в 0. Позже, когда сигнал Done переключится из 0 в 1, управляющая схема устройства умножения перейдет в сотояние S0. Процесс ждет заднего фронта сигнала Done для того, чтобы задать новые значения для Mcand и Mplier.

Рисунок 7.14. Testbench для знакового умножения

```
library BITLIB;
use BITLIB.BitPackage.all;

entity testmult is
end testmult;

architecture testi of testmult is
component mult2C
   port(CLK, St: in bit;
      Mplier,Mcand : in bit_vector(3 downto 0);
      Product: out bit_vector (6 downto0);
      Done: out bit) ;
end component;

   constant N: integer := 11;
   type arr is array(1 to N) of bit_vector(3 downto 0);
```

```
   constant Mcandarr: arr := ("0111", "1101", "0101", "1101",
                              "0111", "1000", "0111", "1000", "0000",
                              "1111", "1011");
   constant Mplierarr: arr := ("0101", "0101", "1101", "1101",
                              "0111", "0111", "1000", "1000", "1101",
                              "1111", "0000");
   signal CLK, St, Done: bit;
   signal Mplier, Mcand: bit_vector(3 downto 0);
   signal Product: bit_vector(6 downto 0);
begin
   CLK <= not CLK after 10 ns;
   process
   begin
      for i in 1 to N loop Mcand <= Mcandarr(i);
      Mplier <= Mplierarr(i);
      St <= '1' ;
      wait until rising_edge(CLK);
      St <= '0' ;
      wait until falling_edge(Done) ;
   end loop;
   end process;
   multl: mult2c port map(Clk, St, Mplier, Mcand, Product, Done) ;
end testi;
```

На рисунке 7.15 представлен командный файл и результат моделирования. В нем задаются сигналы для вывода результатов. Параметры: –NOtrigger и -Trigger задают режим, в котором значения будут отображаться только тогда, когда сигнал *Done* изменит свое значение. Без них значения сигналов отображаются всякий раз, когда хотя бы один из сигналов в списке изменяется. Все результаты произведения правильные, за исключением специального случая -1×-1 (1.000×1.000), который дал 1.000000 (–1) вместо +1. Это случилось потому, что невозможно представить +1 без введения дополнительного бита.

Рисунок 7.15. Командный файл и моделирование знакового устройства умножения

```
-- Командный файл для тестирования выполнения знакового умножения
list –NOtrigger Mplier Mcand product -Trigger done
run 1320
     ns  delta mplier  mcand product  done
      0   +1    0101   0111  0000000    0
     90   +2    0101   0111  0100011    1     5/8 * 7/8 = 35/64
    110   +2    0101   1101  0100011    0
    210   +2    0101   1101  1110001    1     5/8 * -3/8 = -15/64
    230   +2    1101   0101  1110001    0
    330   +2    1101   0101  1110001    1     -3/8 * 5/8 = -15/64
    350   +2    1101   1101  1110001    0
    450   +2    1101   1101  0001001    1     -3/8 * -3/8 = 9/64
    470   +2    0111   0111  0001001    0
    570   +2    0111   0111  0110001    1     7/8 * 7/8 = 49/64
    590   +2    0111   1000  0110001    0
    690   +2    0111   1000  1001000    1     7/8 * -1 = -7/8
    710   +2    1000   0111  1001000    0
    810   +2    1000   0111  1001000    1     -1 * 7/8 = -7/8
    830   +2    1000   1000  1001000    0
    930   +2    1000   1000  1000000    1     -1 * -1 = -1 (error)
    950   +2    1101   0000  1000000    0
   1050   +2    1101   0000  0000000    1     -3/8 * 0 = 0
   1070   +2    1111   1111  0000000    0
   1170   +2    1111   1111  0000001    1     -1/8 * -1/8 = 1/64
   1190   +2    0000   1011  0000001    0
   1290   +2    0000   1011  0000000    1     0 * -3/8 = 0
   1310   +2    0101   0111  0000000    0
```

На следующем шаге детализируется VHDL-модель для знакового устройства умножения. С этой целью определяются управляющие сигналы и действия, которые должны выполняться при поступлении каждого управляющего сигнала. Структура VHDL-модели с рисунка 7.16 подобна автомату Мили с рисунка 2.16. В первой части процесса следующее состояние Nextstate и выходные управляющие сигналы определяются в каждом текущем состоянии. При поступлении переднего фронта синхросигнала обновляются регистры и состояние State. Для тестирования VHDL-кода с рисунка 7.16 используется тот же самый тестовый файл, что и для предыдущей модели, а текущие результаты сравниваются с результатами, полученными ранее.

Рисунок 7.16. Описывающая управляющие сигналы модель устройства умножения

```
-- Это схема 4-битного устройства умножения чисел
-- в дополнительном коде с использованием управляющих сигналов.

architecture behave2 of mult2CS is
   signal State, Nextstate: integer range 0 to 5;
   signal A, B: bit_vector(3 downto 0);
   signal AdSh, Sh, Load, Cm, Done: bit;
   alias M: bit is B(0) ;
begin
   process
   variable addout: bit_vector(4 downto 0);
   begin
      Load <= '0'; AdSh <= '0'; Sh <= '0'; Cm <= '0'; Done <= '0';
      wait for 0 ns;
      case State is
         when 0=>                    -- начальное состояние
            if St='1' then Load <= '1'; Nextstate <= 1; end if;
         when 1|2|3 => —"add/shift" State
            if M = '1' then AdSh <= '1' ;
               else Sh <= '1' ;
            end if;
            Nextstate <= State + 1;
         when 4 =>                   -- сложение в дополнительном коде, если
            if M = '1' then          -- знаковый бит множителя 1
               Cm <= '1'; AdSh <= '1';
               else Sh <= '1';
            end if;
            nextstate <= 5;
         when 5 =>                   -- результирующее произведение
            done <= '1';
            nextstate <= 0;
      end case;
      wait until CLK = '1';          -- выполнение по переднему фронту
      if Cm = '0' then addout := add4(A.Mcand,'0') ;
         else addout := add4(A, not Mcand,'1') ;
      end if;
      if Load = '1' then             -- загрузка множителя
         A <= "0000";
         B <= Mplier;
      end if;
      if AdSh = '1' then          -- добавление множимого к A и сдвиг
         A <= (Mcand(3) xor Cm) & addout(3 downto 1);
         B <= addout(0) & B(3 downto 1);
      end if;
      if Sh = '1' then
         A <= A(3) & A(3 downto 1);
         B <= A(0) & B(3 downto 1);
      end if;
      if Done = '1' then
         Product <= A(2 downto 0) & B;
      end if;
```

```
      State <= Nextstate;
   end process;
end behave2 ;
```

Поскольку граф состояний схемы управления (см. рисунок 7.11) является циклическим, естественно спроектировать схему с применением счетчика. Будет использован 74163 счетчик и логические элементы, как это показано на рисунке 7.17. Выходы счетчика,$Q_3Q_2Q_1Q_0$, соответствуют состояниям управляющей схемы. Для представления шести состояний достаточно задействовать только три бита счетчика. Однако функции возбуждения будут проще, если использовать все 4 бита для кодирования состояний следующим образом:

```
S0-> 0000, S1-> 0100, S2-> 0101, S3-> 0110, S4-> 0111, S5->1000
```

При таком назначении должен выполняться сброс счетчика в состоянии S5, загрузка значением Din = 0100 – в S0 и счет до последнего состояния S5. Подробнее:

```
CLR1 = Q3', Done = Q3  (CLR1 = 0 и Done = 1 в состоянии 1000.)
Load = Q3'Q2'St  (Load = 1 в состоянии 0000, когда St = 1.)
Ldl  = Load'     (Загрузка счетчика в состоянии 0000, когда St=1.)
PI   = Q2        (Счет в состояниях 0100, 0101, 0110, 0111.)
Sh   = M'Q2      (Сдвиг в состояниях 0100, 0101, 0110,0111, если M=0.)
AdSh = MQ2       (Сложение/сдвиг в состояниях 0100, 0101, 0110, 0111,
                  если M = 1.)
Cm   = MQ1Q0
```

Для верификации этого проекта поведенческое описание управляющей схемы будет заменено на полученные уравнения. Триггеры состояний будут обновляться после поступления фронта синхросигнала (рисунок 7.18) точно так же, как и ранее. Определим явно выход в дополнительном коде как comp. Обратный код можно вычислить, применив операцию XOR Cm к каждому биту множимого. В выражении

```
comp <= Mcand xor Cm&Cm&Cm&Cm
```

используется оператор конкатенации для создания четырех копий Cm. Это соответствует реальной аппаратуре. Сигнал Cm будет связан со входами четырех элементов XOR. В карте портов счетчика зарезервированное слово open означает отсутствие связи с выходом переноса. Снова можно использовать тест-файл для проверки того, что значения на выходах остались такими же, как и после тестирования первой модели.

Рисунок 7.17. Реализация управляющей схемы устройства умножения

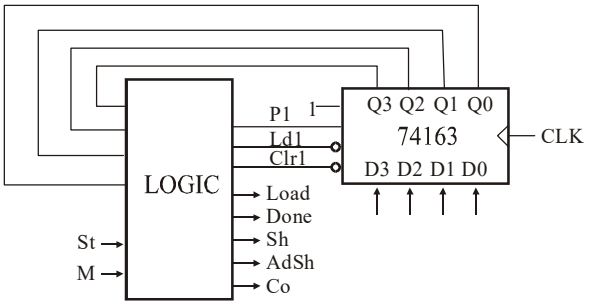


Рисунок 7.18. Модель устройства умножения с использованием логических уравнений

```
-- Модель 4-битного устройства умножения для чисел в дополнительном
-- коде, с реализацией управляющей схемы на счетчике с
-- использованием логических уравнений.
library BITLIB;
use BITLIB.BitPackage.all;
entity mult2CEQ is
  port(CLK, St: in bit;
```

```
        Mplier,Mcand: in bit_vector(3 downto 0);
        Product: out bit_vector(6 downto 0));
end mult2CEQ;
architecture m2ceq of mult2CEQ is
   signal A, B, Q, Comp: bit_vector(3 downto 0);
   signal addout: bit_vector(4 downto 0);
   signal AdSh, Sh, Load, Cm, Done, Ld1, CLR1, P1: bit;
   signal One: bit:='1';
   signal Din: bit_vector(3 downto 0) := "0100";
begin
   Counti: C74163 port map (Ld1, CLR1, P1, One, CLK, Din, open, Q);
   P1 <= Q(2);
   CLR1 <= not Q(3) ;
   Done <= Q(3) ;
   Sh <= not M and Q(2);
   AdSh <= M and Q(2) ;
   Cm <= Q(1) and Q(0) and M;
   Load <= not Q(3) and not Q(2) and St;
   Ld1 <= not Load;
   Comp <= Mcand xor (Cm & Cm & Cm & Cm);
                              -- обратный код Mcand, если Cm='1'
   addout <= add4(A,Comp,Cm);    -- сложение дополнительного кода с A
process
begin
   wait until CLK = '1';            -- выполнение по переднему фронту
   if Load = '1' then               -- загрузка множителя
      A <= "0000" ;
      B <= Mplier;
   end if;
   if AdSh = '1' then          -- сложение множимого с A и сдвиг
      A <= (Mcand(3) xor Cm) & addout(3 downto 1);
      B <= addout(O) & B(3 downto 1);
   end if;
   if Sh = '1' then            -- сдвиг вправо с расширением знака
      A <= A(3) & A(3 downto 1);
      B <= A(0) & B(3 downto 1);
   end if;
   if Done = '1' then
      Product <= A(2 downto 0) & B;
   end if;
end process;
end m2ceq;
```

7.5. Проектирование устройства двоичного деления

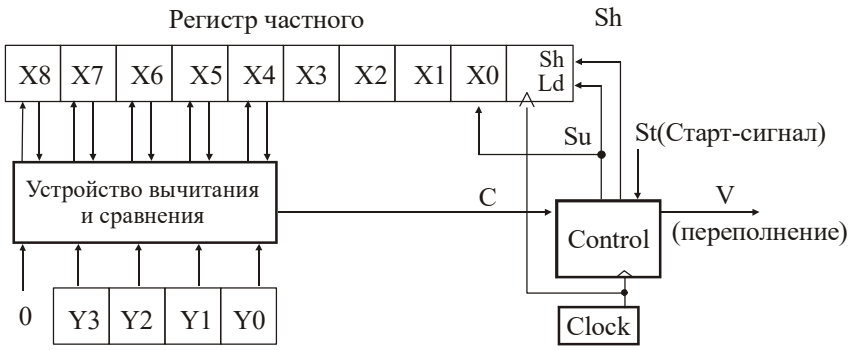
Рассмотрим проектирование устройства параллельного деления положительных двоичных чисел. В качестве примера будет спроектировано устройство для деления 8-битного делимого на 4-битный делитель. Следующий пример иллюстрирует процедуру деления двоичных чисел:

```
                          1010  частное
делитель  1101 /10000111  делимое
               1101
                0111
(135 ÷ 13 = 10  0000
 остаток 5)      1111
                 1101
                  0101
                  0000
                  0101 остаток
```

Аналогично двоичному умножению, которое может быть выполнено последовательностью операций сложения и сдвига, деление реализуется с помощью операций вычитания и сдвига. Для проектирования устройства деления используется 9-битный

регистр делимого и 9-битный регистр делителя, как это показано на рисунке 7.19. Во время операции деления вместо сдвига делителя вправо перед выполнением вычитания осуществляется сдвиг делимого влево. Обратите внимание, что используется дополнительный бит с левой стороны делимого. Таким образом, бит не будет теряться при сдвиге влево. Вместо использования отдельного регистра для сохранения частного делимое размещается бит за битом в правой части регистра делимого по мере того, как он сдвигается влево.

Рисунок 7.19. Структурная схема устройства параллельного двоичного деления



Выполним снова предыдущий пример деления (135 на 13), для того чтобы показать содержимое регистров на каждом синхротакте. На первом шаге осуществляется загрузка делимого и делителя, как это показано ниже:

0	1	0	0	0	0	1	1	1

1	0	0	0

Вычитание не может быть выполнено из-за отрицательного результата. Поэтому перед вычитанием осуществляется сдвиг. Вместо того, чтобы сдвигать делитель на одну позицию вправо, реализуется сдвиг делимого на одну позицию влево:

```
1 0 0 0 0 1 1 1 | 0   ← Линия, разделяющая делимое и частное
  1 1 0 1       |     ← Обратите внимание: после сдвига самая
                        правая позиция регистра делимого пуста
```

Теперь выполняется вычитание и первая цифра делимого 1 сохраняется в неиспользуемой позиции регистра делимого:

```
0 0 0 1 1 1 1 1 | 1  ← Первый бит частного
```

Затем осуществляется сдвиг делимого на одну позицию влево:

```
0 0 1 1 1 1 1 | 1 0
  1 1 0 1     |
```

Поскольку после вычитания был получен отрицательный результат, имеет место очередной сдвиг делимого влево и второй бит делимого становится 0:

```
0 1 1 1 1 1 | 1 0 0
  1 1 0 1   |
```

Выполняется вычитание и третий бит частного (1) сохраняется в неиспользуемой позиции регистра делимого:

```
0 0 0 1 0 1 | 1 0 1  ← Третий бит частного
```

Осуществляется последний сдвиг и четвертый бит делимого устанавливается в 0:

```
0 0 0 1 0 | 1 1 0 1
остаток   частное
```

Конечный результат соответствует результату, полученному в первом примере.

Если результат операции деления содержит больше битов, чем доступно для сохранения частного, это означает, что возникло переполнение. Для устройства деления с рисунка 7.19 переполнение произойдет, если частное больше 15, так как только 4 бита доступны для хранения частного. Выполняя деление, необходимо определять возможность возникновения переполнения. Например, если попытаться разделить 135 на 7, начальное содержание регистров будет равно 0100001110111.

Поскольку в результате вычитания можно получить положительный результат, следует выполниить деление и занести 1 в правую позицию регистра делимого. Тем не менее, этого делать нельзя, потому что самая правая позиция регистра делимого содержит младший бит делимого и ввод бита частного уничтожит его. Такое частное слишком большое, чтобы разместить его в отведенных для него 4 битах. Необходимо определить условие переполнения. В общем случае для рисунка 7.19, если изначально $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$ – левые пять битов регистра делимого больше или равны делителю, частное будет больше 15 и возникнет переполнение. Обратите внимание: если $X_8X_7X_6X_5X_4 \geq Y_3Y_2Y_1Y_0$, частное будет

$$\frac{X_8X_7X_6X_5X_4X_3X_2X_1X_0}{Y_3Y_2Y_1Y_0} \geq \frac{X_8X_7X_6X_5X_40000}{Y_3Y_2Y_1Y_0} = \frac{X_8X_7X_6X_5X_4X_3 \times 16}{Y_3Y_2Y_1Y_0} \geq 16.$$

Операция деления может быть описана в терминах структурной схемы с рисунка 7.19. Сигнал Sh выполняет сдвиг делимого влево на одну позицию.По сигналу Su делитель вычитается из пяти левых битов регистра делимого и устанавливается бит частного (самый правый бит регистра делимого) в 1. Если делитель больше значения, содержащегося в пяти левых битах делимого, выход компаратора C = 0, иначе C = 1. Управляющая схема генерирует необходимую последовательность сигналов сдвига и вычитания. Всякий раз, когда C = 0, результат вычитания будет отрицательным. В таком случае генерируется сигнал сдвига. Всякий раз, когда C=1, генерируется сигнал вычитания и бит частного устанавливается в 1.

На рисунке 7.20 представлен граф состояний управляющей схемы. После поступления старт-сигнала St 8-битное делимое и 4-битный делитель загружаются в соответствующие регистры. Если C = 1, то для частного может потребоваться как минимум пять битов. Поскольку для хранения делимого предоставляется только 4 бита, то возникает переполнение, поэтому деление прекращается и устанавливается выходной сигнал переполнения V. Если начальное значение C=0, на первом шаге выполняется сдвиг и управляющая схема переходит в состояние S2. Затем, если C=1, выполняется вычитание. После завершения операции вычитания C всегда будет равно 0. Таким образом, при поступлении следующего синхроимпульса будет выполнен сдвиг. Этот процесс продолжается, пока не будет выполнена четвертая операция сдвига и автомат не перейдет в состояние S5. Затем, если необходимо, выполняется последнее вычитание и схема переходит в stop-состояние. Предполагается, что для этого примера старт-сигнал (St) после поступления длится еще один такт и затем переключается в 0 до тех пор, пока схема не вернется в начальное состояние.

Рисунок 7. 20. Граф состояний для управляющей схемы устройства деления

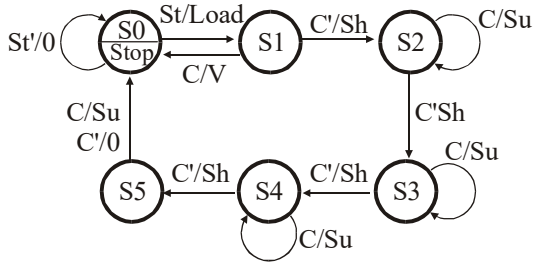


Таблица 7.4 представляет переходы и функции выходов управляющей схемы. Поскольку предполагается, что St = 0 в состояниях S1, S2, S3 и S4, для них не определны

значения следующего состояния и выходы для случая, когда St=1. Метки в таблице выходов обозначают выходы, значения которых равны 1. Например, метка Sh означает, что Sh=1, а остальные выходы равны 0.

Таблица 7.4. Таблица переходов и выходов управляющей схемы устройства деления

	StC				StC			
State	00	01	11	10	00	01	11	10
S0	S0	S0	S1	S1	0	0	Load	Load
S1	S2	S0	-	-	Sh	V	-	-
S2	S3	S2	-	-	Sh	Su	-	-
S3	S4	S3	-	-	Sh	Su	-	-
S4	S5	S4	-	-	Sh	Su	-	-
S5	S0	S0	-	-	0	Su	-	-

Этот пример иллюстрирует общий подход к проектированию устройств деления беззнаковых двоичных чисел. Проект может быть легко расширен до большего числа битов, например, до 16-битного делимого и 8-битного частного или 32-разрядного делимого и 16-разрядного делителя. Спроектируем устройство для знаковых двоичных чисел, которое выполняет деление 32-разрядного делимого на 16-битный делитель для получения 16-битного частного. Хотя существует алгоритм для непосредственного деления знаковых чисел, он достаточно сложен. Проще преобразовать делимое или делитель, если они отрицательные, в дополнитеьный код. После выполнения деления частное, если оно должо быть отрицательным, дополняется.

На рисунке 7.21 представлена структурная схема устройства деления. Для загрузки регистров используется 16-битная шина. Поскольку делимое имеет 32 бита, необходимо два такта для загрузки младшей и старшей половины делимого в регистр и один такт – для загрузки делителя. Дополнительный знаковый триггер используется для хранения знака делимого. Регистр делимого имеет встроенный преобразователь в дополнительный код. Устройство вычитания состоит из подсхемы сложения и устройства генерации обратного кода. Таким образом, вычитание может выполняться прибавлением делителя в дополнительном коде к делимому. Если делитель отрицателен, нет надобности получать его дополнительный код. В этом случае сложение выполняется с отрицательным делителем, а устройства генерации обратного кода отключается. Управляющая схема делится на две части: основную управляющую, которая определяет последовательность сигналов сдвига и вычитания, и счетчик, хранящий число сдвигов.Выходной сигнал счетчика после выполнения 15-го сдвига равен К=1. Управляющие сигналы определены следующим образом:

Ldu – загрузка с шины старшей половины делимого;

Ldl – загрузка с шины младшей половины делимого;

Lds – загрузка знака делимого в знаковый триггер;

S – знак делимого;

Cm1 – дополнение регистра делимого (дополнительный код);

Ldd – загрузка с шины делимого;

Su – подключение выхода сумматора к шине (Ena) и загрузка верхней половины делимого из шины;

Cm2 – разрешение преобразования в обратный код (Cm2 подключен к знаковому биту делимого таким образом, что положительное число преобразуется в дополнительный код, а отрицательное – нет);

Sh – сдвиг регистра делимого влево на 1 разряд и увеличение счетчика на 1;

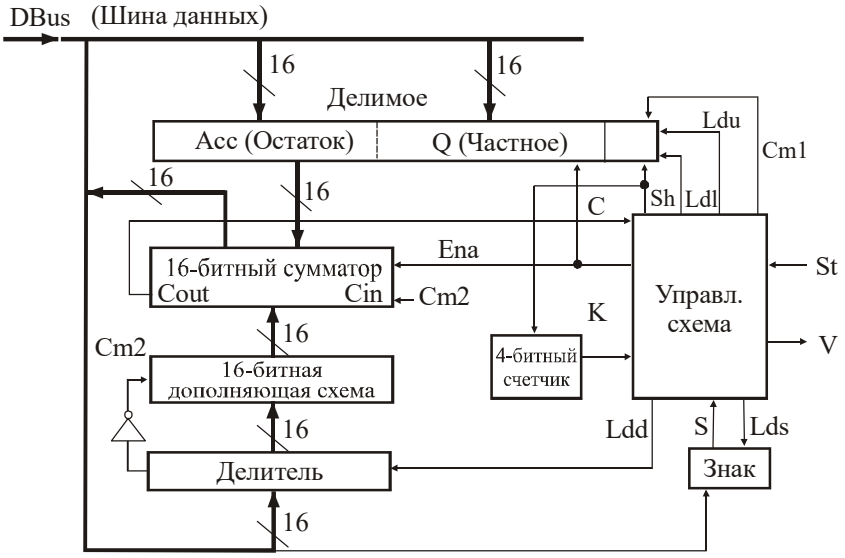
C – выход переноса сумматора (если C=1, делитель можно вычесть из верхней части делимого);

St – старт;

V – переполнение;

Qneg – частное будет отрицательным *(Qneg* = 1, когда знаки делимого и делителя различны).

Рисунок 7.21. Структурная схема устройства деления знаковых чисел



Деление знаковых чисел выполняется следующим образом:

1. Загрузка верхней половины делимого с шины и копирование знака делимого в знаковый триггер.
2. Загрузка младшей половины делимого с шины.
3. Загрузка делимого с шины.
4. Преобразование делимого в дополнительный код, если это необходимо.
5. Если выполняется условие переполнения, переход в начальное состояние.
6. Иначе выполнение деления с помощью операций сдвига и вычитания.
7. По окончании деления, если это необходимо, частное преобразуется в дополнительный код. Выполняется переход в начальное состояние.

Тестирование переполнения осуществляется немного сложнее, чем для случая беззнаковых чисел. Вначале рассмотрим все положительные числа. Поскольку делитель и частное занимают по 15 битов плюс знак, их максимальное значение равно 7FFFh. При условии, что остаток должен быть меньше делимого, его максимальное значение – 7FFEh. Отсюда максимальное делимое, которое не приведет к переполнению, равняется:

```
делитель × частное + остаток = 7FFFh × 7FFFh + 7FFEh = 3FFF7FFFh
```

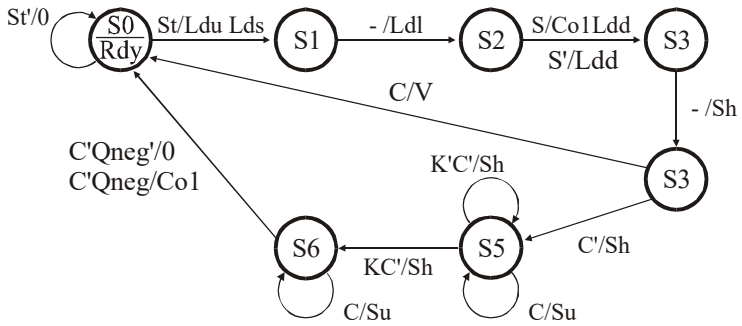
Если делимое будет только на 1 больше (3FFF8000h), деление на 7FFFh приведет к переполнению. Для проверки возможности переполнения делимое сдвигается на одну позицию влево и затем сравнивается старшая половина делимого divu с делителем. Если divu больше делителя, частное будет больше максимально возможного и возникнет переполнение. Например, сдвиг 3FFF8000h влево на одну позицию дает 7FFF0000h. Поскольку 7FFFh равно делителю, это означает возникновение переполнения. С другой стороны, сдвиг 3FFF7FFFh влево дает 7FFEFFFEh, и так как 7FFEh < 7FFFh, переполнение не возникнет при делении на 7FFFh.

Поэтому, чтобы проанализировать возможность возникновения переполнения, перед делением осуществляется сдвиг делимого влево на одну позицию. Если divu будет больше делителя, то можно выполнить вычитание и сгенерировать первый бит частного, равный 1. Но этот бит занял бы знаковую позицию частного и сделал его

отрицательным, что привело бы к ошибке. Поэтому после проверки на переполнение делимое сдвигается влево снова, чтобы освободить место для первого знака частного после знакового бита. Поскольку в этом методе отрицательное делимое и делитель преобразуются в дополнительный код, способ определения переполнения можно применять и к отрицательным числам, за исключением специального случая, когда делимое равно 80000000h (самое большое отрицательное число). Изменение проекта для проверки переполнения в данном случае можно сделать самостоятельно.

На рисунке 7.22 представлен граф состояний для управляющей схемы. Когда St=1, выполняется загрузка регистров. В состоянии S2, если знак делимого S=1, делимое преобразуется в дополнительный код. В S3 выполняется сдвиг делимого влево на одну позицию и тестирование переполнения в состоянии S4. Если C=1, результат вычитания не отрицателен, что означает возникновение переполнения, и схема переходит в начальное состояние. Иначе – делимое сдвигается влево. В состоянии S5 тестируется C. Если C = 1, генерируется сигнал Su = 1, который подразумевает сигналы Ldu и Ena: разрешается поступление значений с выходов сумматора на шину и загрузка их в старший регистр делимого для выполнения вычитания. Если C=0, сигнал Sh = 1 и регистры делимого сдвигаются влево. Это продолжается, пока не выполнится условие K=1, после чего реализуется последний сдвиг, и если C=0, схема переходит в состояние S6. Затем, если знак делителя и сохраненный знак делимого имеют различные значения, содержимое регистра делимого преобразуется в дополнительный код, чтобы частное имело правильный знак.

Рисунок 7.22. Граф состояний управляющей схемы деления знаковых чисел



VHDL-код для устройства деления знаковых чисел представлен на рисунке 7.23. Поскольку дополняющая схема и сумматор – комбинационные устройства, их функционирование моделируется с помощью параллельных операторов. Процедура ADDVEC выполняется в любой момент, когда ACC или compout изменяются, поэтому сумма Sum и перенос вычисляются сразу. Все сигналы, представляющие выходы регистра, обновляются по переднему фронту синхросигнала. Таким образом, эти сигналы обновляются в процессе, после того как CLK примет значение 1. Например, ADDVEC, реализованная в состояниях 2 и 6, заносит дополнительный код делимого обратно в его регистр. Счетчик моделируется сигналом типа integer – count. Для удобства чтения результатов моделирования введен сигнал готовности Rdy, который устанавливается в состояние S0 и означает, что деление выполнено.

Рисунок 7.23. VHDL-модель устройства деления 32-битных знаковых чисел

```
library BITLIB;
use BITLIB.BitPackage.all;

entity sdiv is
   port(Clk,St.: in bit;
         Dbus: in bit_vector(15 downto 0);
         Quotient: out bit_vector(15 downto 0);
         V, Rdy: out bit) ;
end sdiv;
```

```
architecture Signdiv of Sdiv is
   constant zero_vector: bit_vector(31 downto 0):=(others=>'0');
   signal State: integer range 0 to 6;
   signal Count : integer range 0 to 15;
   signal Sign,C,NC: bit;
   signal Divisor,Sum,Compout: bit_vector(15 downto 0);
   signal Dividend: bit_vector(31 downto 0);
   alias Q: bit_vector(15 downto 0) is Dividend(15 downto 0);
   alias Acc: bit_vector(15 downto 0) is Dividend(31 downto 16);
begin                                        -- параллельные операторы
      -- генерация обратного кода делителя
   compout<=divisor when divisor (15) = '1'
      else not divisor;
   Addvec(Acc,compout,not divisor(15),Sum,C,16);  --16-битный сумматор
   Quotient <= Q;
   Rdy <= '1' when State=0 else '0';
   process
   begin
      wait until Clk = '1';  --ожидание переднего фронта синхросигнала
      case State is
         when 0=>
            if St = '1' then
               Acc <= Dbus;     --загрузка старшего регистра делимого
               Sign <= Dbus(15);
               State <= 1;
               V <= '0';              -- начальные значения сигналов
               Count <= 0;            -- переполнения и счетчика
            end if;
         when 1=>
            Q <= Dbus;          -- загрузка младшего регистра делимого
            State <= 2;
         when 2=>
            Divisor <= Dbus;
            if Sign ='1'then
         -- если необходимо, генерация дополнительного кода делимого
               addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
            end if;
            State <= 3;
         when 3 = >
            Dividend <= Dividend(30 downto 0) & '0'; -- сдвиг влево
            Count <= Count+1;
            State <= 4;
         when 4 =>
            if C ='1' then                           -- C
               v <= '1';
               State <= 0;
            else                                     -- C'
               Dividend <= Dividend(30 downto 0) & '0'; -- сдвиг влево
               Count <= Count+1;
               State <= 5;
            end if;
         when 5 = >
            if C = '1' then                          -- C
               ACC <= Sum;                           -- вычитание
               Q(0)<= '1';
            else
               Dividend <= Dividend(30 downto 0) & '0'; -- сдвиг влево
               if Count = 15 then                    -- KC'
                  State <= 6; Count <= 0;
               else Count <= Count+1;
               end if;
            end if;
         when 6=>
            if C = '1' then                          -- C
```

```
            Acc <= Sum;                                  -- вычитание
            Q(0) <= 'I';
          else if (Sign xor Divisor(15))='1' then     -- C'Qneg
            addvec(not Dividend,zero_vector,'1',Dividend,NC,32);
          end if;                          -- дополнительный код делимого
          state <= 0;
        end if;
      end case;
    end process;
end signdiv;
```

Далее можно выполнить тестирование проекта устройства деления с помощью программы VHDL- моделирования. Для этого необходим полный тест, охватывающий различные ситуации, которые могут возникнуть в процессе деления. Вначале тестируются основные операции деления для всех различных комбинаций знаков делителя и делимого (++, +–, –+ и ––). Также необходимо промоделировать ситуации возникновения переполнения для этих четырех случаев. Еще должны присутствовать крайние режимы: самое большое частное, деление на 0. В данном случае удобно использовать VHDL testbench, поскольку тестовые данные должны подаваться в некоторой последовательности, а продолжительность выполнения деления зависит от тестовых данных. На рисунке 7.24 представлен testbench для устройства деления. Он содержит массив делимого и делителя для тестовых данных. Обозначение X" 07FF00BB" соответствует шестнадцатеричному представлению битовой строки. Процесс в testsdiv вначале устанавливает старт-сигнал и заносит старшую половину делимого в шину Dbus. Затем, после поступления нового синхроимпульса, младшая половина делимого передается на шину Dbus. После следующего синхросигнала на шину подается делитель. Затем используется оператор wait для ожидания сигнала окончания выполнения деления Rdy. Счетчик устанавливается равным индексу цикла, следовательно, изменения сигнала Count могут быть использованы для вывода результатов тестирования.

Рисунок 7.24. Testbench для устройства деления знаковых чисел

```
library BITLIB;
use BITLIB.BitPackage.all;

entity testsdiv is
end testsdiv;

architecture testi of testsdiv is
component sdiv
  port(Clk,St: in bit;
       Dbus: in bit_vector(15 downto 0);
       Quotient: out bit_vector(15 downto 0);
       V, Rdy: out bit) ;
end component;
constant N: integer := 12;          -- N тестовых векторов
type arri is array(1 to N) of bit_vector(31 downto 0);
type arr2 is array(1 to N) of bit_vector(15 downto 0);
constant dividendarr: arri := (X"0000006F", X"07FFOOBB",
     X"FFFFFE08", X"FF80030A", X"3FFF8000", X"3FFF7FFF",
     X"C0008000", X"C0008000",  X"C0008001", X"00000000",
     X"FFFFFFFF", X"FFFFFFFF");
constant divisorarr: arr2 := (X"0007", X"E005", X"001E", X"EFFA",
           X"7FFF", X"7FFF", X"7FFF", X"8000", X"7FFF", X"0001",
           X"7FFF", X"0000");
  signal CLK, St, V, Rdy: bit;
  signal Dbus, Quotient, divisor: bit_vector(15 downto 0);
  signal Dividend: bit_vector(31 downto 0);
  signal count: integer range 0 to N;
begin
```

```
   CLK <= not CLK after 10 ns;
   process
   begin
      for i in 1 to N loop
         St <= '1' ;
         Dbus <= dividendarr(i)(31 downto 16);
         wait until rising_edge(CLK);
         Dbus <= dividendarr(i)(15 downto 0);
         wait until rising_edge(CLK);
         Dbus <= divisorarr(i) ;
         St <= '0' ;
         -- сохранение делимого для вывода результатов
         dividend <= dividendarr(i)(31 downto 0);
         -- сохранение делителя для вывода результатов
         divisor <= divisorarr(i);
         wait until (Rdy = '1');
         count <= i;                    -- save index for triggering
      end loop;
   end process;
   sdivl: sdiv port map(Clk, St, Dbus, Quotient, V, Rdy);
end testi;
```

На рисунке 7.25 представлен командный файл и результаты моделирования. Их проверка показала, что деление выполняется правильно для всех случаев, кроме следующей ситуации:

```
C00080000h ÷ 7FFFh = -3FFF8000 ÷ 7FFFh = -8000h = 8000h
```

Здесь генерируется сигнал переполнения и деление не выполняется. Вообще устройство деления сообщает о переполнении в любом случае, когда частное равно 8000h (самое большое отрицательное число). Это возникает потому, что делитель изначально делит положительные числа и самое большое положительное число – 7FFFh. Если необходимо получать частное 8000h, процедура определения переполнения должна быть модифицирована, чтобы сигнал переполнения для этого случая не генерировался.

Рисунок 7.25. Результаты тестирования устройства деления знаковых чисел

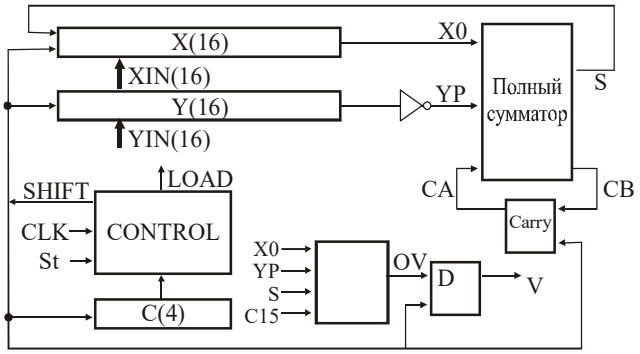
```
-- Командный файл для тестирования деления знаковых чисел
list -hex -NOtrigger dividend divisor Quotient V -Trigger count
run 5300 ns

   ns    delta dividenddivisor quotientV  count
      0   +0   00000000   0000    0000  0     0
    470   +3   0000006F   0007    OOOF  0     1
    910   +3   07FFOOBB   E005    BFFE  0     2
   1330   +3   FFFFFE08   001E    FFFO  0     3
   1910   +3   FF80030A   EFFA    07FC  0     4
   2010   +3   3FFF8000   7FFF    0000  1     5
   2710   +3   3FFF7FFF   7FFF    7FFF  0     6
   2810   +3   C0008000   7FFF    0000  1     7
   3510   +3   C0008000   8000    7FFF  0     8
   4210   +3   C0008001   7FFF    8001  0     9
   4610   +3   00000000   0001    0000  0     A
   5010   +3   FFFFFFFF   7FFF    0000  0     B
   5110   +3   FFFFFFFF   0000    0002  1     C
```

Выводы. Рассмотрены алгоритмы умножения и деления беззнаковых и знаковых двоичных чисел. Спроектированы цифровые системы, реализующие эти алгоритмы. После разработки структурных схем указанных систем и определения необходимых управляющих сигналов для описания управляющего автомата использовался граф состояний, который генерировал управляющие сигналы в необходимой последова-

тельности. Язык VHDL применялся для описания системы на различных уровнях для обеспечения возможности их моделирования и верификации.

7.6. Задачи

7.6.1. Приведена структурная схема для 16-битного устройства вычитания в дополнительном коде. Когда St = 1, данные загружаются в регистры и выполняется операция вычитания. Счетчик сдвигов C генерирует сигнал C15=1 после выполнения 15 сдвигов. Сигнал V устанавливается в 1 при возникновении переполнения. Триггер переноса должен переходить в 1 во время загрузки для формирования дополнительного кода. Допустим, что St равно 1 на протяжении одного синхротакта. Нарисовать граф состояний для управляющей схемы (два состояния). Написать VHDL-код для системы. Использовать два процесса. Первый применяется для определения следующего состояния и управляющих сигналов. Второй – для обновления регистров по переднему фронту синхросигнала:



7.6.2. Изначально 3-значное двоично-десятичное число размещается в A-регистре. При поступлении St-сигнала число преобразуется в двоичный код и сохраняется в регистре B. На каждом шаге преобразования двоично-десятичное число (как и двоичное) сдвигается на одну позицию вправо. Если результат в данной декаде больше или равен 1000, корректирующая схема вычитает из нее 0011. Иначе, содержимое декады не изменяется.Сдвиговый счетчик обеспечивает счет сдвигов. По окончании преобразования максимальное значение B может быть 999. Обратите внимание: длина B равна десяти битам. Нарисовать структурную схему преобразователя из двоично-десятичного кода в двоичный. Нарисовать граф состояний управляющей схемы (3 состояния), использующей следующие управляющие сигналы: S – начало преобразования; Sh – сдвиг вправо; Co – необходимость коррекции вычитанием; C9 – счетчик достиг состояния 9; C10 – счетчик находится в состоянии 10. Используйте C9 либо C10, но не оба вместе. Напишите VHDL-модель системы.

7.6.3. Структурная схема устройства умножения знаковых чисел представлена на рисунке 7.10. Определить содержимое регистров A и B после каждого синхроимпульса, если множимое = –1/8 и множитель = –3/8.

7.6.4. Нарисовать структурную схему 32-битного последовательного сумматора с аккумулятором. Для управляющей схемы используется 5-битный счетчик, который в состоянии 11111 выдает сигнал K = 1. При поступлении старт-сигнала N выполняется загрузка регистров. Пусть N=1 во время выполнения сложения. По окончании сложения управляющая схема переходит в состояние Stop и остается в нем, пока N не изменится на 0. Нарисовать граф состояний для управляющей схемы, включая счетчик. Написать VHDL-код для устройства и проверить корректность операций.

7.6.5. Нарисовать структурную схему устройства деления беззнаковых двоичных чисел, выполняющего деление 8-битного делимого на 3-битный делитель и формирующего 5-битное частное. Нарисовать граф состояний для управляющей схемы. Пусть старт-сигнал St присутствует один синхротакт. Написать VHDL-код верхнего уровня для устройства деления.

7.6.6. В подразд. 7.4 был рассмотрен алгоритм для умножения знаковых двоичных дробей, в котором отрицательное число представлялось в дополнительном коде. Проиллюстрировать алгоритм для умножения 1.0111 на 1.101. Нарисовать структурную схему устройства, реализующего алгоритм умножения для 4-битного множителя, включая знак, и 5-битного множимого, включая знак.

7.6.7. Создать VHDL-модуль, описывающий один бит полного сумматора с аккумулятором. Модуль должен иметь два управляющих входа, Ad и L. Если Ad = 1, вход Y и перенос добавляются к аккумулятору. Если L = 1, значение с входа Y загружается в аккумулятор. Используя модуль, написать код VHDL для 4-битного устройства вычитания с аккумулятором. Пусть отрицательное число представляется в обратном коде. Устройство должно иметь входы управления Su (вычитание) и Ld (загрузка).

7.6.8. Создать структурную схему для 16-битного последовательного сумматора, используя три 16-битных регистра, два 74163 счетчика, полный сумматор, D-триггер и необходимые вентили. Сдвиговые регистры и счетчик изменяют состояние по переднему фронту синхроимпульса. Нарисовать граф состояний для управляющей схемы. Реализовать схему на ПЛМ и 2 триггерах. Привести таблицу ПЛМ. Написать VHDL-код для устройства умножения, используя модель ПЛМ.

7.6.9. Написать VHDL-код для последовательного сумматора из задачи 7.6.8. Явно указать все управляющие сигналы. Не использовать операторы case или if вне модуля. Описать систему в терминах модулей и межсоединений.

7.6.10. Устройство деления беззнаковых двоичных чисел делит 16-битное делимое на 8-битный делитель и формирует 8-битное частное. Пусть старт-сигнал ST равен 1 на протяжении одного такта. Если частное требует более 8 битов, деление должно быть немедленно остановлено и сформирован сигнал V=1, свидетельствующий о переполнении. Использовать: 17-битный регистр делимого и сохранять частное в 8 младших битах этого регистра; 4-битный счетчик для счета числа сдвигов, вместе с управляющей схемой вычитание-сдвиг. Нарисовать структурную схему устройства деления, граф состояния для управляющей схемы вычитание – сдвиг (3 состояния). НаписатьVHDL-модель устройства деления.

7.6.11. Устройство выполняет двоичное деление 8-битного числа на 4-битный делитель, формирует 4-битное частное. Все числа – целые, положительные, без знака. Для повышения скорости выполнения операции деления схема спроектирована так, чтобы сдвиг и вычитание выполнялись за один синхротакт (вместо двух). Нарисовать структурную схему, включающую 8-битный регистр, 5-битное устройство вычитания и другие необходимые компоненты. Индикация переполнения не требуется. Описать управляющие сигналы и нарисовать граф состояний для схемы управления. Пусть старт-сигнал присутствует только один синхротакт и процесс деления прекращается в состоянии Done.

7.6.12. Устройство вычисляет двоичный корень от 8-битного беззнакового двоичного числа, используя метод вычитания нечетных целых. Для того чтобы найти квадратный корень числа N, из N вычитают 1, затем 3, затем 5 и т.д., пока результат вычитания не будет отрицательным. Число вычитаний равняется квадратному корню из N. Например, необходимо найти $\sqrt{27}$: 27-1 = 26; 26-3=23; 23-5=18; 18-7= 11; 11-9=2; 2-11. Вычитание было выполнено пять раз, $\sqrt{27}$ =5. Обратите внимание, что последнее нечетное число $11_{10} = 1011_2$ соответствует значению квадратного корня ($101_2 = 5_{10}$) с дополнительной 1 в конце. Нарисовать структурную схему устройства, вычисляющего квадратный корень , которая содержит регистр для хранения числа *N,* устройство вычитания, регистр для нечетных целых чисел и управляющую схему. Обозначить момент считывания результата – квадратного корня. Для этого ввести сигнал завершения операции. Определить управляющие сигналы, используемые в диаграмме.

Нарисовать граф состояний для управляющей схемы, имеющий минимальное число состояний. Число N должно загружаться в регистр после поступления старт-сигнала St = 1. Когда квадратный корень найден, управляющая схема генерирует сигнал завершения и ожидает значение сигнала St = 0 для выполнения сброса.

7.6.13. Спроектировать устройство умножения 16-битовых знаковых двоичных чисел, формирующее 32-битное произведение. Отрицательные числа представлены дополнительным кодом.Устройство реализует следующий метод. Сначала формируется дополнительный код множимого или множителя, если они отрицательны, и выполняется умножение как положительных чисел. Если необходимо, произведение преобразуется в дополнительный код. После загрузки регистров умножение должно быть выполнено не более чем за 16 тактов. Нарисовать структурную схему устройства умножения. Использовать 4-битный счетчик для подсчета числа сдвигов. Он должен формировать сигнал К=1 в состоянии 15. Определить все управляющие сигналы, используемые в схеме. Нарисовать граф для устройства умножения, имеющий минимальное (3) число состояний. После завершения процесса умножения управляющая схема генерирует сигнал Done и ожидает ST = 0 для возвращения в состояние S0. Создать поведенческое VHDL-описание устройства умножения без использования управляющих сигналов (в качестве примера см. рисунок 7.5) и протестировать его. Создать поведенческое VHDL-описание устройства умножения с использованием управляющих сигналов (в качестве примера см. рисунок 7.12). Протестировать его.

7.6.14. Реализовать структурную схему устройства умножения с рисунка 7.10, используя только микросхемы 22V10 PAL. Если возможно, применять только два устройства 22V10, если нет – три микросхемы. Показать соединения между микросхемами и привести логические уравнения для D-входа каждой макроячейки.

7.6.15. Создать и промоделировать VHDL-модель устройства умножения знаковых двоичных чисел по Booth-алгоритму. Отрицательные числа представляются в дополнительном коде. Пусть каждое число имеет размер n, включая знаковый бит. Для аккумулятора следует использовать (n+1)-битный регистр A, чтобы знаковый бит не был потерян при переполнении. Также использовать (n+1)-битный регистр B для хранения множителя и n-битный регистр C для хранения множимого. Проиллюстрировать функционировние Booth-алгоритма.

1. Очистить A (аккумулятор), загрузить множитель в старшие n битов регистра B, очистить B_0 и загрузить множимое в C.
2. Протестировать младшие два бита регистра B (B_1B_0).
   Если B_1B_0 = 01, прибавить C к A, C следует расширть по знаку до n+1 бит и сложить с A, используя (n+1)-битный сумматор.
   Если B_1B_0 = 10, тогда перевести C в дополнительный код и прибавить к A.
   Если B_1B_0 = 00 или 11, пропустить этот шаг.
3. Сдвинуть A и B на одну позицию вправо с расширением знака.
4. Повторить шаг 2 и 3, еще n-1 раз.
5. Произведение должно содержаться в регистрах A и B, за исключением B_0 .

Пример для n=5: Нужно умножить -9 на -13.

		A	B	B_1B_0	
1.	Загрузка регистров.	000000	100110	10	C= 10111
2.	Сложение C в	001001			
	дополнит. коде с A.	001001	100110		
3.	СдвигA&B.	000100	110011	11	
3.	СдвигA&B.	000010	011001	01	
2.	Прибавление C к A.	110111			
		111001	011001		

3.	СдвигA&B.	111100	101100	00
3.	СдвигA&B.	111110	010110	10
2.	Сложение C в	001001		
	дополнит. коде с A.	000111	010110	
3.	СдвигA&B.	000011	101011	

Результат: 0001110101 = +117

Нарисовать структурную схему устройства умножения для n = 8. Использовать 9-битные регистры A и B, 9-битный полный сумматор, 8-битное устройство для получения обратного кода, 3-битный счетчик и управляющую схему. Используйте счетчик для подсчета числа сдвигов. Нарисовать граф состояний для схемы управления. Когда счетчик находится в состоянии 111, происходит переход в начальное состояние вместе с выполнением последнего сдвига (3-х состояний будет достаточно). Написать поведенческий VHDL-код для устройства умножения (не использовать явно управляющие сигналы). Использовать процедуру addvec (из библиотеки BITLIB) для сложения двух n-битных векторов, переноса и генерации n-битной суммы и переноса. Промоделировать VHDL-проект, используя следующие входные значения (в каждой паре второе число — множитель):

```
01100110 × 00110011
10100110 × 01100110
01101011 × 10001110
11001100 × 10011001
```

7.6.16. Спроектировать устройство, реализующее алгоритм параллельного сложения-вычитания 8-битных чисел, оперирующий понятиями знака и модуля. Входы X ,Y и выход Z включают знаки и модули чисел. Внутренние вычисления выполняются в дополнительном или в обратном коде, но ввод данных не будет выполнен, если X и Y представляют обратный или дополнительный код. Если входной сигнал Sub = 1, тогда Z = X - Y, иначе Z = X + Y. Устройство должно работать для всех комбинаций положительных и отрицательных чисел, для сложения и вычитания. Допускается использование только следующих компонентов: 8-битный сумматор, устройство получения обратного кода (для входа Y), вторая схема для формирования дополнительного или обратного кода, комбинационная схема для генерации управляющих сигналов. Подсказка: -X + Y = - (X - Y). Также ввести сигнал переполнения, сообщающий, когда результат не может быть представлен 8 битами: знаком и модулем. Нарисовать структурную схему. Не разрешается использование в ней регистров, мультиплексоров или тристабильных буферов. Написать таблицу истинности для логической схемы, генерирующей необходимые управляющие сигналы. Входами в таблице должны быть Sub, Xs и Ys, где Xs – знак числа X и Ys – знак числа Y. Описать процедуру обнаружения переполнения и написать уравнения.

7.6.17. Используя полный сумматор (full adder), описанный в главе 2, создать VDHL-модель передачи данных для устройства матричного умножения 4-битных чисел.

7.6.18. Для матричного устройства умножения двухбитных чисел, формирующего 4-битный результат, нарисовать структурную схему. Сколько нужно логических элементов И, полных сумматоров и полусумматоров? Если максимальная задержка полного сумматора равна 15 ns, а задержка И вентиля равна 10 ns, то чему равно худшее время выполнения умножения? Какой должна быть синхронизация двоичного последовательно-параллельного устройства , подобного рисунку 7.3, чтобы скорость его функционирования была такой же, как и у матричного устройства умножения?

ГЛАВА 8
ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ СИСТЕМ НА ОСНОВЕ PLD И PGA

Как было показано ранее, PLD позволяют реализовывать последовательные схемы, но не пригодны для создания сложных цифровых систем. В этом случае используются более гибкие и многофункциональные программируемые вентильные матрицы (PGA) и сложные программируемые логические устройства, позволяющие реализовывать большие цифровые системы на одной микросхеме.

Типичная PGA – это микросхема, состоящая из массива идентичных логических ячеек с программируемыми соединениями. Пользователь может запрограммировать функции, реализуемые каждой логической ячейкой, и соединения между ними. Такие PGA часто называют FPGA, поскольку они являются программируемыми в условиях эксплуатации (field-programmable).

Ниже описывается внутренняя структура некоторых типичных FPGA, выпускаемых фирмой Xilinx. Рассматриваются методы их программирования для реализации цифровых логических схем. В общих чертах описываются основные этапы проектирования с использованием FPGA. Рассматриваются особенности применения Altera CPDL.

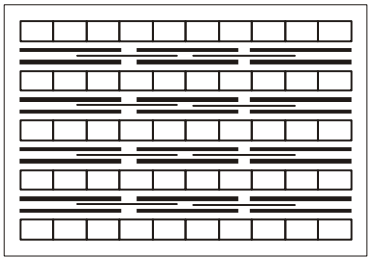
8.1. Основные типы FPGA

Существуют два основных класса FPGA: матричные (symmetrical array) и строковые (row-based). Матричные FPGA выпускаются фирмой Xilinx. Эти микросхемы представляют собой матрицы логических элементов, между строками и столбцами которых расположены каналы трассировки. К строковым FPGA (рисунок 8.1) принадлежат микросхемы фирмы Actel. В этих устройствах логические элементы расположены в виде строк, между которыми находятся каналы трассировки.

Кроме структурной организации, матричные и строковые FPGA различаются технологией изготовления. В матричных в качестве программируемых элементов используются элементы статического ОЗУ (SRAM), в строковых – одноразово пережигаемые перемычки (antifuse). Матричные FPGA, основанные на SRAM, допускают реконфигурацию, но требуют дополнительных устройств для хранения их конфигурации. Строковые программируются один раз и не требуют дополнительной аппаратуры. Преимущество строковых устройств в том, что их программируемые элементы значительно меньше, чем в матричных FPGA, поэтому они имеют лучшие характеристики при трассировке межсоединений, чем последние.

Размеры логических элементов строковых FPGA обычно меньше, чем матричных. Поэтому устройство, реализованное на строковой FPGA, требует больше конфигурируемых логических блоков и межсоединений, чем это было бы необходимо в случае матричной FPGA. Но поскольку логические элементы и межсоединения строковых FPGA работают значительно быстрее, это не отражается на производительности проектируемого устройства.

Рисунок 8.1. Архитектура строковой FPGA



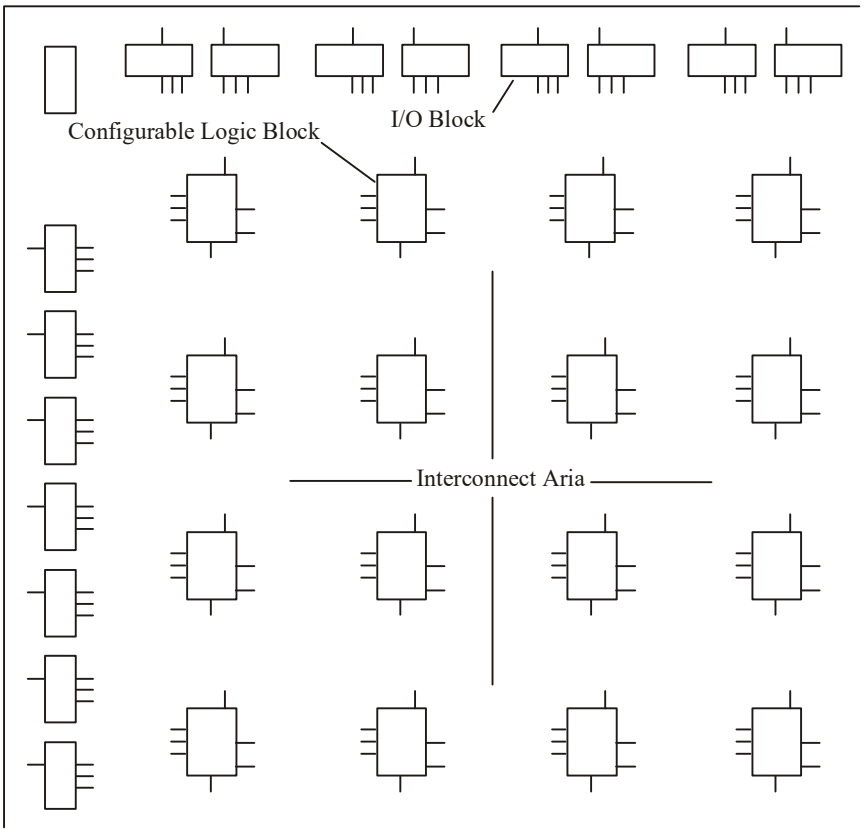
Современные микросхемы фирмы Actel имеют до 2 млн. эквивалентных вентилей и системную частоту 350 МГц. Размер микросхем фирмы Xilinx достигает 8 млн. эквивалентных вентилей.

8.2. Xilinx FPGA, серия 3000

На рисунке 8.2 показана основная структура микросхемы Xilinx XC3020, состоящая из внутреннего массива – 64 конфигурируемых логических блока (configurable logic blocks – CLBs), окруженные по периметру 64 вход-выходными интерфейсными блоками. Соединения между ними могут программироваться путем сохранения данных во внутренних конфигурирующих ячейках памяти.

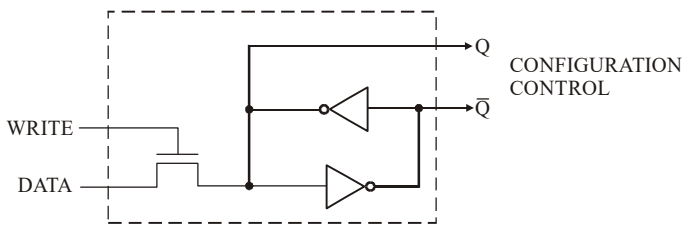
Каждый конфигурируемый логический блок состоит из комбинационной схемы и двух D-триггеров. Блок может быть запрограммирован на выполнение разнообразных логических функций.

Рисунок 8.2. Структурная схема матричной FPGA



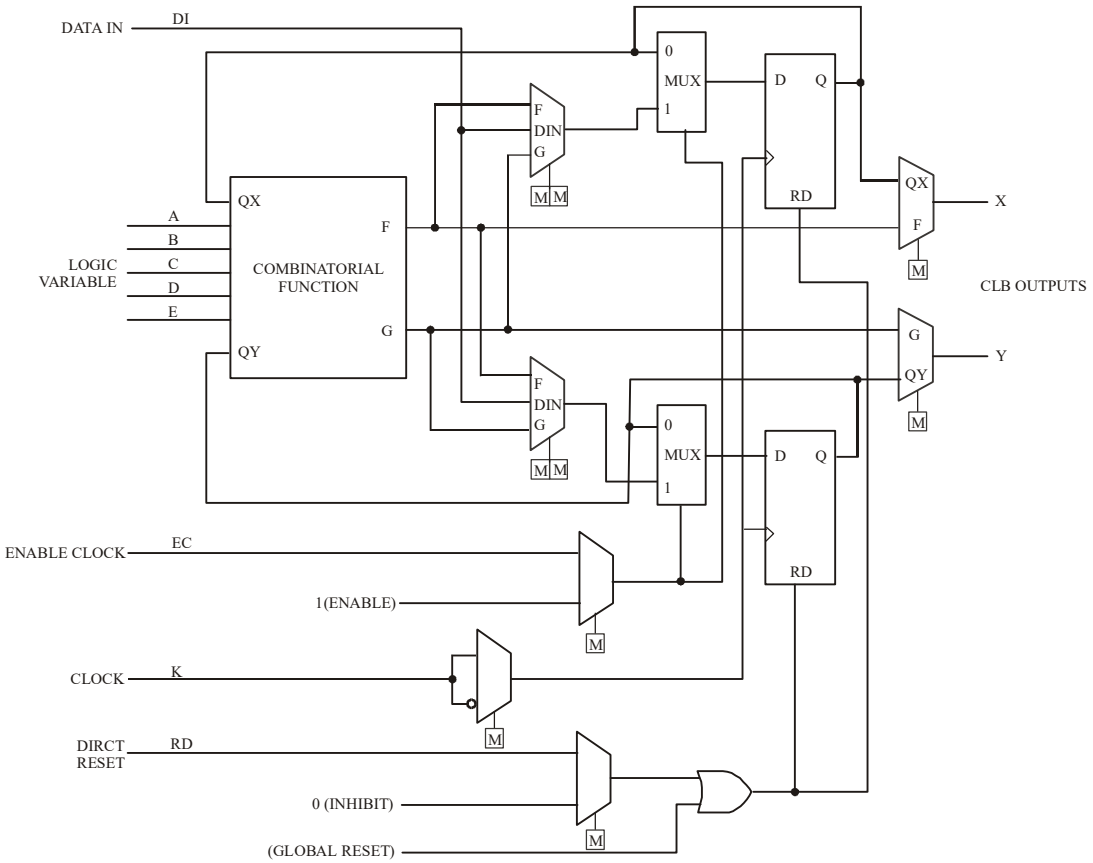
Конфигурируемые ячейки памяти программируются после подачи питания на FPGA, реализованные логические функции и межсоединения сохраняются до отключения напряжения. Во время конфигурации ячейки памяти (рисунок 8.3) выбираются по очереди. Когда сигнал WRITE поступает на передающий транзистор, значение DATA сохраняется в ячейке. Каждая точка соединения в матрице имеет связанную ячейку памяти. Данные, сохраненные в ней, определяют наличие соединения.

Рисунок 8.3. Конфигурирующая ячейка памяти



На рисунке 8.4 представлен конфигурируемый логический блок. Он имеет пять логических входов (A, B, C, D, E), вход данных (DI), синхровход (K), вход разрешения синхронизации (EC), непосредственный сброс (RD) и два выхода (X и Y). Трапециевидные блоки на схеме обозначают мультиплексоры, которые могут быть запрограммированы на выбор одного из входов. Например, значения на выход X могут поступать либо с верхнего триггера QX, либо с выхода F блока "комбинационная функция" (Combinatorial Function). Таким же образом значения на выход Y могут поступать с нижнего триггера QY или с выхода G. Знак M обозначает конфигурирующую ячейку памяти, данные в ячейке определяют вход используемого мультиплексора.

Рисунок 8.4. Конфигурируемый логический блок Xilinx серии 3000



Блок комбинационных функций состоит из ячеек RAM и может быть запрограммирован на реализацию любой функции пяти или двух – четырех переменных. Функции хранятся в виде таблицы истинности. Таким образом, число вентилей, необходимое для их реализации, не имеет значения. На рисунке 8.5 представлено три возможных операционных режима для функционального блока. Каждый трапециевидный блок обозначает мультиплексор, который программируется на выбор одного из входов. FG-режим генерирует две функции четырех переменных. Одна переменная A должна быть общей для обеих функций. Следующие две переменные могут быть выбраны из B, C, QX и QY. Последней переменной может быть D или E. Например, можно реализовать функции F = AB' + QX E и G = A'C + QYD. Если QX и QY не используются, то две функции четырех переменных должны иметь три общих переменных: A, B и C. Четвертая может быть D или E.

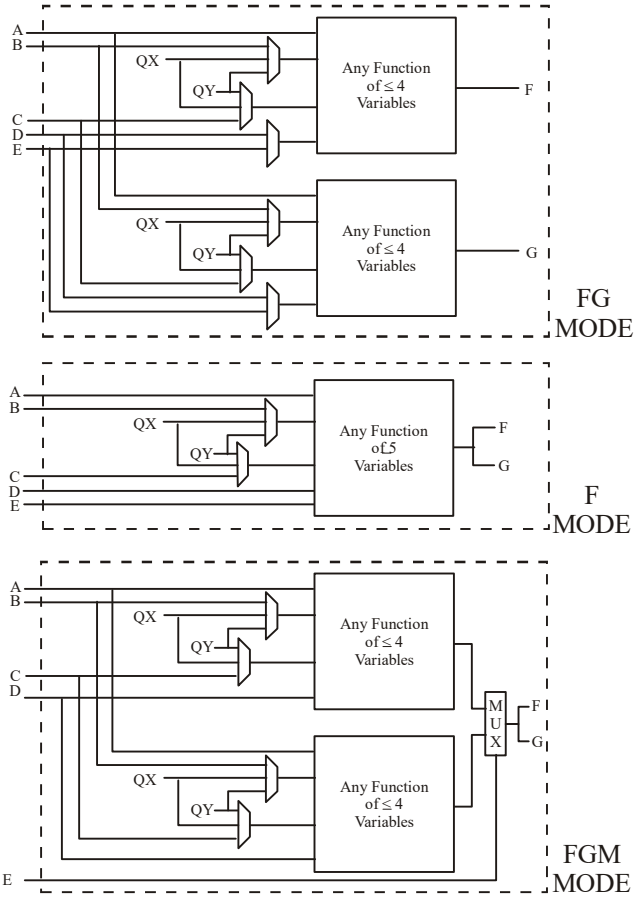
В режиме F генерируется одна функция пяти переменных: A, D, E и две переменные выбираются из B, C, QX и QY. Можно реализовать функцию по возрастанию сложности от простого AND вентиля: F = G = ABCDE до функции четности:

$$F = G = A \oplus B \oplus C \oplus D \oplus E\ ,$$

имеющей 16 термов, если разложить ее на дизъюнкцию конъюнкций.

Режим FGM использует мультиплексор с контрольным входом E для выбора одной из двух функций четырех переменных. Каждая функция использует входы A, D плюс два из четырех: B, C, QX, QY. В режиме FGM можно реализовать некоторые функции шести и семи переменных. Например, в этом режиме можно реализовать следующую функцию семи переменных: F=G=E(AB'+QXD) + E' (A'C+ QYD).

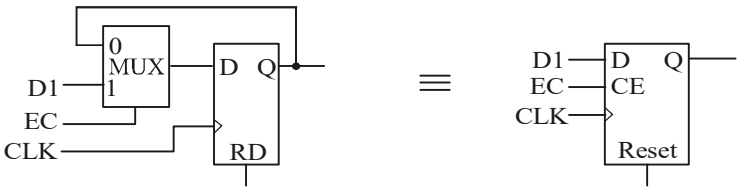
Рисунок 8.5. Режимы комбинационных схем



Вход D каждого триггера может быть запрограммирован на получение данных с F, G или DI входа. Оба имеют общий вход синхронизации. Мультиплексор MUX связан со входом CLOCK (K) и может быть запрограммирован на выбор K или K'. Поэтому имеется возможность выбора синхронизации D-триггера передним или задним фронтом синхросигнала. Синхроимпульс может быть или всегда разрешен, или управляться входом Enable Clock (EC). Мультиплексор MUX, связывающий D вход каждого триггера, используется для отключения синхронизации. Если EC = 0, сигнал с Q выхода поступает обратно на D вход. Таким образом, $Q^+ = Q$, и триггер никогда не изменит состояния, даже при наличии синхроимпульса. Если EC = 1, D вход соединяется с F, G и DIN, а состояние триггера изменяется по соответствующему фронту синхросигнала. Соединение D-триггера и мультиплексора эквивалентно D-триггеру со входом разрешения синхронизации EC, как показано на рисунке 8.6. Поскольку Q может изменять значение при EC = 1, следующее характеристическое уравнение описывает поведение триггера: $Q^+ = EC\ D+EC'Q$.

Использование триггеров такого типа устраняет необходимость в стробировании синхровхода с помощью управляющего сигнала. Поскольку синхроимпульс может непосредственно поступать на вход каждого триггера, легко получить правильные операции синхронизации. Триггеры имеют вход асинхронного сброса RD. Каждый из них сбрасывается в 0, если сброс не запрещен при поступлении 1 на линию. Общий сброс очищает триггеры во всех ячейках микросхемы.

Рисунок 8.6. Триггер с разрешением синхронизации



В качестве примера на XC3020 реализуется устройство параллельного сложения, вычитания с накоплением. Общая схема подобна схеме с рисунка 3.21, за исключением управляющих сигналов для сложения и вычитания. Если Ad = 1, значение с B входа будет добавлено к аккумулятору, если Su = 1 – будет вычитаться из аккумулятора. Вычитание выполняется сложением дополнительного кода B с аккумулятором. Если Ad = Su = 0, содержимое аккумулятора не изменится. Дополнительный код образуется из обратного кода B (инвертирование B) плюс 1 на входе переноса первого полного сумматора.

Поскольку каждая логическая ячейка имеет два триггера, можно было бы реализовать два бита аккумулятора на одной ячейке. Однако, если два бита будут реализованы в одной ячейке, необходимо иметь два триггера аккумулятора и выход переноса в следующую ячейку. Каждая ячейка имеет только два выхода, поэтому такая схема не будет работать. Следовательно, можно реализовать только один бит на ячейку.

На рисунке 8.7 показана типичная ячейка устройства параллельного сложения, вычитания. Логическими входами являются: b_i, c_i (перенос из предыдущей ячейки) и *Su.* Значение с выхода триггера аккумулятора a_i подается обратно в ячейку. Блок комбинационной функции реализует следующие уравнения:

$$F = sum = a_i^+ = a_i \oplus (b_i \oplus Su) \oplus c_i \; ; \quad G = c_{i+1} = carry\ \ out = a_i c_i + (a_i + c_i)(b_i \oplus Su) \; .$$

Если Su = 0, эти уравнения сокращаются до стандартных для полного сумматора (уравнения с рисунка 2.2) при x_i=a_i и y_i=b_i. Если Su = 1, выполняется вычитание A-B, которое реализуется путем прибавления к A дополнительного кода B. Для этого формируется обратный код b_i с помощью исключающего ИЛИ. Перенос в самый младший значащий бит соединен с Su, следовательно, он равен 1, когда Su = 1, образуя тем самым дополнительный код B. Так как F и G – функции от одних и тех же четырех переменных, они могут быть реализованы комбинационным функциональным блоком в режиме FG с рисунка 8.5. На рисунке 8.7 c_i и b_i подключены ко входам A и B. Внутренняя обратная связь с a_i триггера QX должна идти на третий вход блока. Оставшийся вход Su может быть связан с D или E входом блока. Так как аккумулятор должен изменяться, когда Ad = 1 или Su = 1, на вход разрешения синхронизации EC подается сигнал Ad + Su. Этот сигнал генерируется элементом ИЛИ из другой логической ячейки и используется всеми ячейками сложения, вычитания.

Пунктирной линией на рисунке 8.8 выделены пути сигналов в логической ячейке после ее программирования. Функция F связана с D входом триггера аккумулятора a_i, а функция G – с переносом в следующий разряд c_{i+1}.

Рисунок 8.7. Логическая ячейка устройства параллельного сложения-вычитания

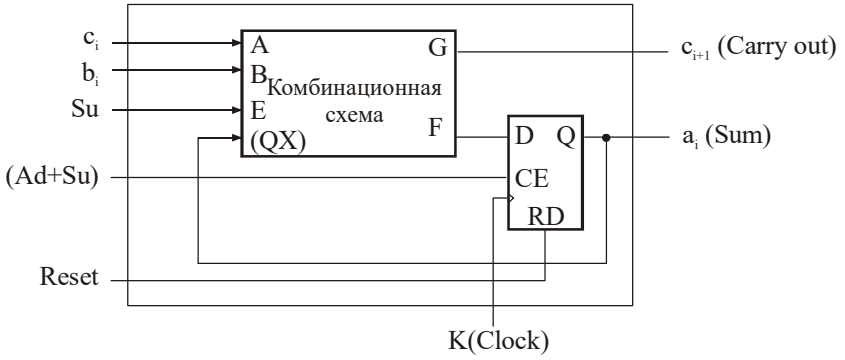
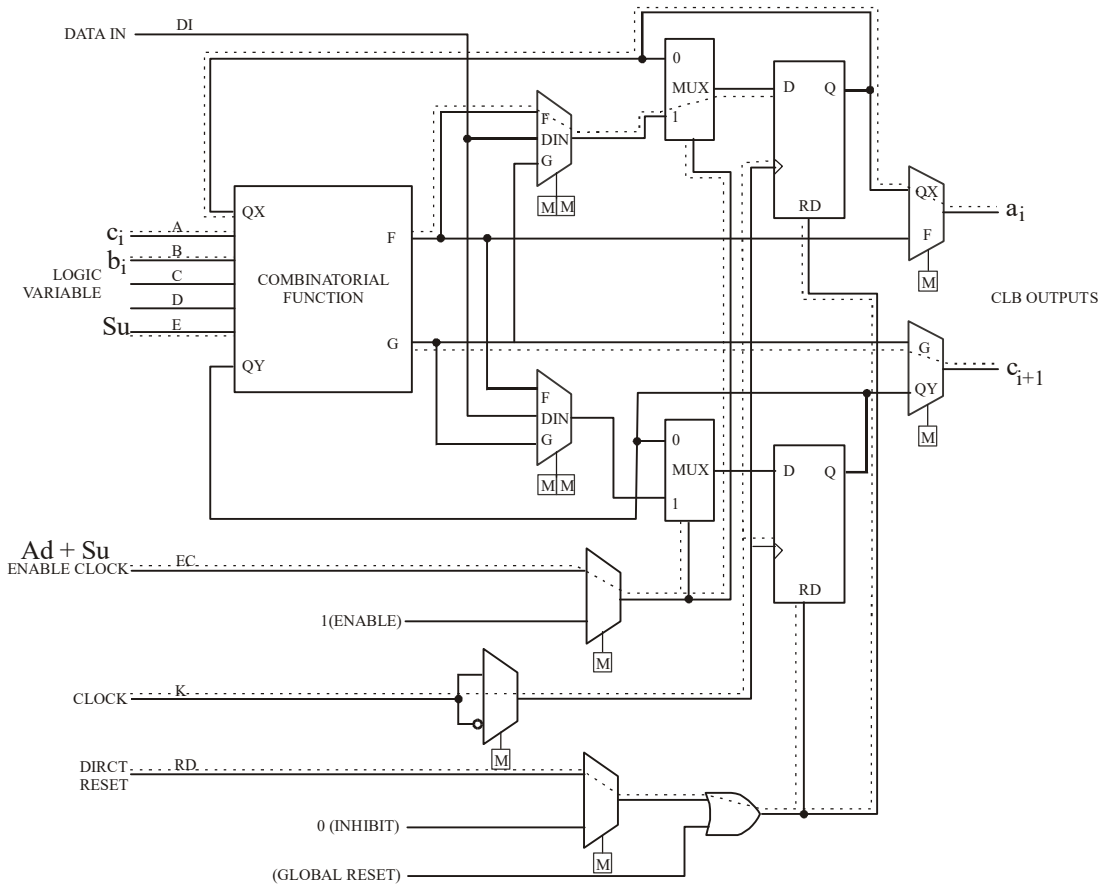


Рисунок 8.8. Пути сигналов в логической ячейке устройства сложения-вычитания



Вход-выходные блоки

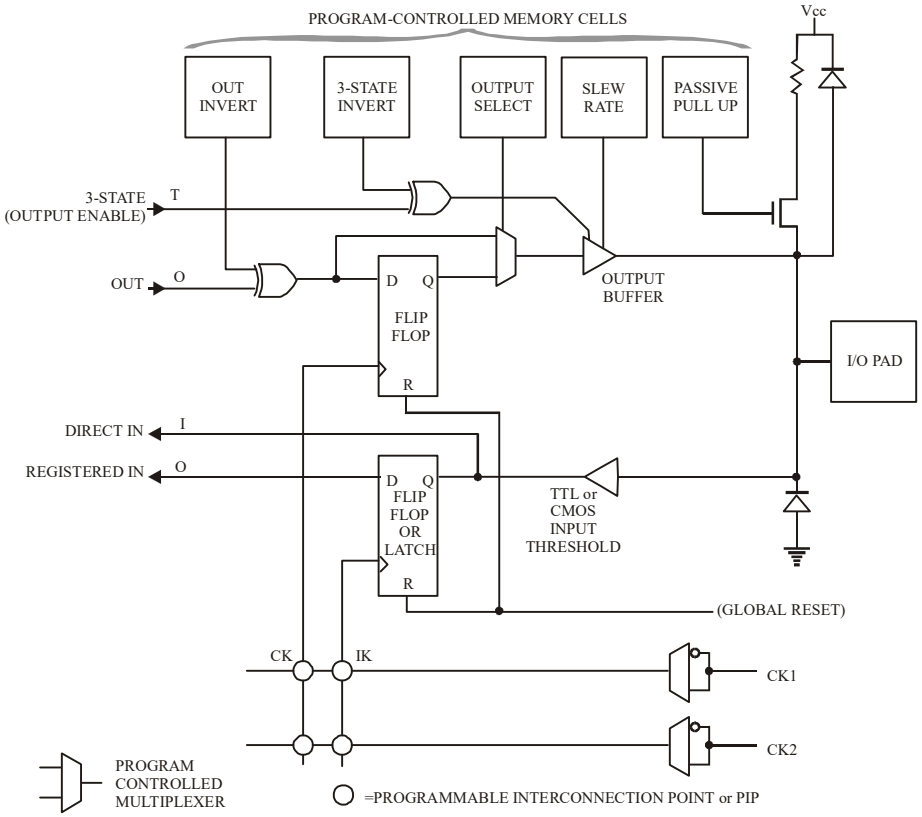
На рисунке 8.9 представлен реконфигурируемый вход-выходной блок (input-output block – IOB). Контакт I/O связан с одной из ножек корпуса микросхемы, которая может быть использована в качестве входа или выхода. Чтобы использовать ячейку как тристабильный выход (3-state), управляющий сигнал должен быть установлен в OUTPUT BUFFER.Чтобы использовать ячейку как вход, тристабильный буфер должен быть отключен. Триггеры позволяют хранить входные и выходные значения в блоке IOB. При необходимости их можно заблокировать для передачи сигнала непосредственно между внешним контактом и внутренней матрицей логических элементов. Две линии синхронизации CK1 и CK2 могут быть запрограммированы на соединение с любым из двух триггеров. Входной триггер может быть запрограммирован на работу как переключаемый по фронту D-триггер или как явный триггер-защелка. Даже если I/O контакт не используется, I/O триггеры тем не менее могут быть задействованы для хранения данных.

Выходной OUT сигнал, идущий из логического массива, вначале проходит через вентиль исключающего ИЛИ, где он инвертируется или нет, в зависимости от того, как запрограммирован OUT-INVERT бит. При необходимости выходной OUT сигнал может быть сохранен в триггере. Программирование OUTPUT-SELECT бита определяет: будет ли на выход подан непосредственно сигнал OUT или он пройдет вначале через триггер.

Если 3-STATE сигнал равен 1 и бит 3-STATE INVERT равен 0 (3-STATE сигнал равен 0 и 3-STATE INVERT бит – 1), выходной буфер имеет на выходе значение высокого импеданса. Иначе выходной сигнал поступает на I/O контакт. Когда I/O

контакт используется как вход, выходной буфер должен быть в состоянии высокого импеданса. Внешний сигнал, идущий с контакта I/O, проходит через буфер и затем поступает на D-триггер. Буфер входа подает сигнал DIRECT IN в логический массив. Иначе входной сигнал может быть вначале сохранен в D-триггере, из которого как сигнал REGISTERED IN будет передан в логический массив.

Рисунок 8.9. Входной-выходной блок Xilinx серии 3000



Каждый IOB имеет входные-выходные опции, которые выбираются конфигурацией ячеек памяти. Входная пороговая величина может быть запрограммирована на уровень сигналов TTL или CMOS. Бит SLEW RATE управляет скоростью изменения выходного сигнала. Когда последний управляет внешними устройствами, для уменьшения наведенных шумов, которые могут появляться при быстром изменении выходного сигнала, желательно снизить скорости нарастания выходного напряжения. Когда бит PASSIVE PULL-UP установлен, нагрузочный резистор соединяется с I/O pad. Внутренний нагрузочный регистр может быть использован для устранения плавающих входов.

Программируемые межсоединения

Программируемые соединения между конфигурируемыми логическими блоками и входными-выходными блоками бывают трех видов: универсальные (general-purpose interconnect), прямые межсоединения (direct interconnect), длинные линии (long line). Рисунок 8.10 иллюстрирует систему универсальных межсоединений. Сигналы между CLB или CLB и IOB направляются по горизонтальным и вертикальным межсоединительным линиям и маршрутизируются с помощью матриц-переключателей. Прямые соединения выполняются между соседними блоками CLB, как показано на рисунке 8.11. Длинные линии обеспечивают соединения CLB, расположенные далеко друг от друга. Все межсоединения программируются сохранением бита во внутренних конфигурирующих ячейках памяти в LCA.

Рисунок 8.10. Универсальные межсоединения (general-purpose interconnect)

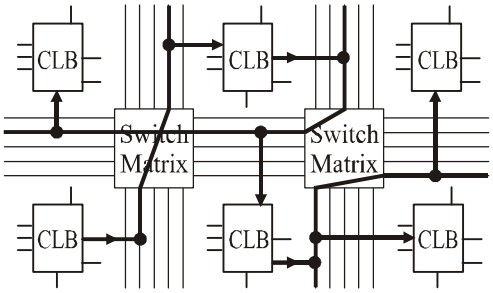
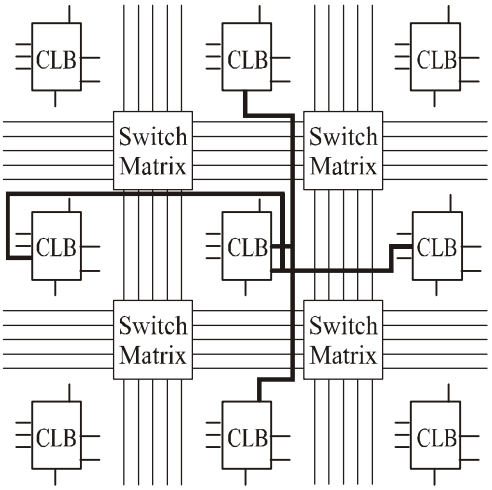
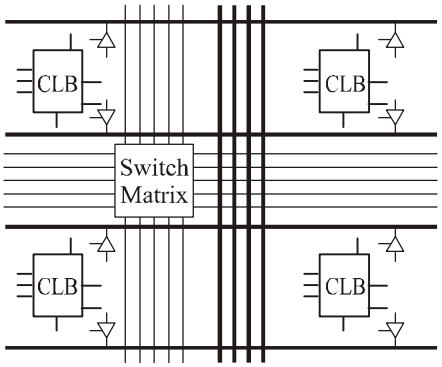


Рисунок 8.11. Прямые межсоединения между соседними блоками CLB



Длинные линии обеспечивают высокий коэффициент разветвления по выходу, низкое распределение расфазировки импульсов (low-skew distribution), которые должны проходить относительно большие расстояния. На рисунке 8.12 показаны четыре вертикальные длинные линии между двумя соседними рядами CLB. Две из них могут быть использованы для синхронизации. Также изображены две горизонтальные линии. Длинные линии охватывают полностью длину или ширину межсоединительной области логических блоков.

Рисунок 8.12. Вертикальные и горизонтальные длинные линии



Каждая логическая ячейка имеет два соседних тристабильных буфера, которые соединяются с горизонтальными длинными линиями. Проектировщики могут использовать эти длинные линии и буферы для реализации тристабильных шин. На рисунке 8.13,а показано, как тристабильные буферы применяются для мультиплексирования сигналов в горизонтальной длинной линии. Эти буферы имеют выходной разрешающий сигнал низкого уровня. Таким образом, когда A = 0, сигнал D_A поступает на линию. Weak keeper circuit в конце линии запоминает последнее значение, поступившее на линию, поэтому оно никогда не теряется. Внимание должно быть направлено на

устранение конфликтов шины, которые могут произойти, если значения 0 и 1 поступят на шину в одно и то же время. Тристабильные буферы могут быть также использованы для реализации функции монтажного И, как показано на рисунке 8.13, б. Когда один или больше входов D имеют 0, линия получает значение 0. Если все входы D имеют значение 1, все выходы буферов находятся в состоянии высокого импеданса и нагрузочный резистор выталкивает линию в состояние 1. На рисунке 8.14 представлен пример VHDL-кода, который после синтеза преобразуется в схему на тристабильных буферах.

Рисунок 8.13. Использование тристабильных буферов: а – реализация мультиплексора; б –реализация монтажного И

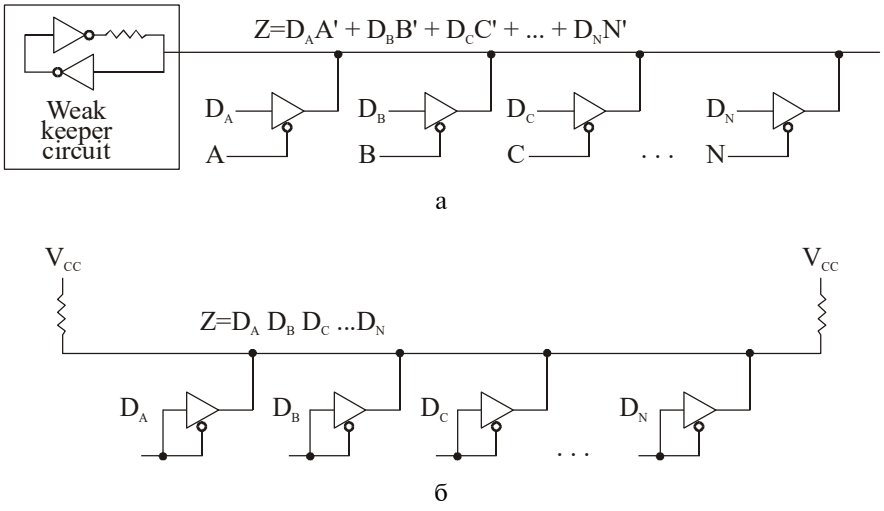
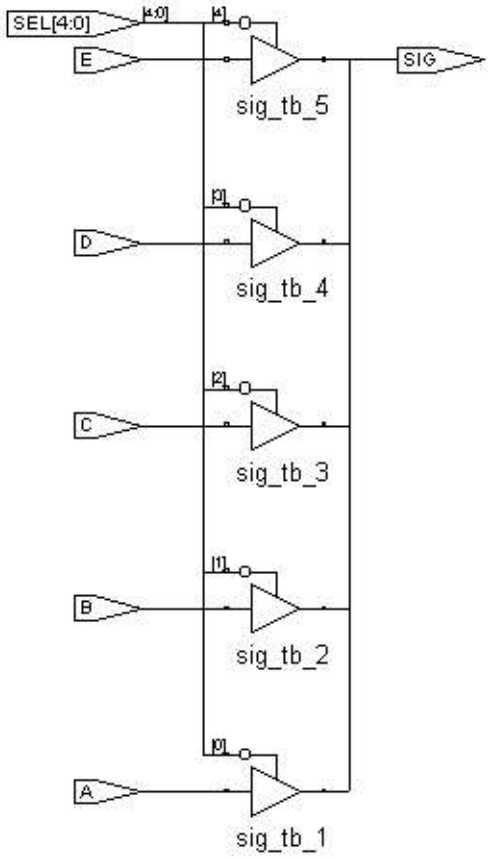


Рисунок 8.14. VHDL-код мультиплексора для реализации на тристабильных буферах

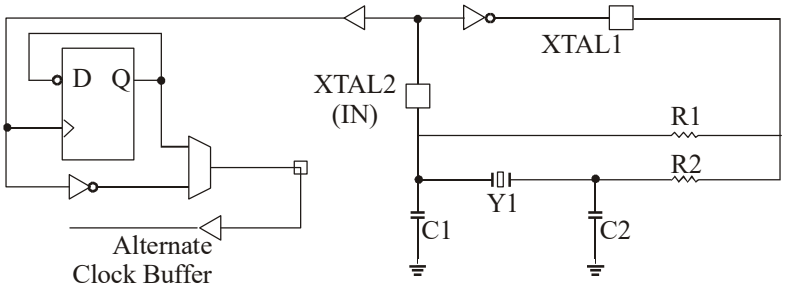


```
-- 5-to-1 Mux Implemented in 3-State Buffers
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity mux_tbuf is
port (SEL: in STD_LOGIC_VECTOR (4 downto 0);
A,B,C,D,E: in STD_LOGIC;
SIG: out STD_LOGIC);
end mux_tbuf;
architecture RTL of mux_tbuf is
begin
SIG <= A when (SEL(0)='0') else 'Z';
SIG <= B when (SEL(1)='0') else 'Z';
SIG <= C when (SEL(2)='0') else 'Z';
SIG <= D when (SEL(3)='0') else 'Z';
SIG <= E when (SEL(4)='0') else 'Z';
end RTL;
```

Можно реализовать кварцевый генератор, используя внутренний быстродействующий инвертирующий буфер вместе с внешним кристаллом Y1, резистором и конденсаторами, как показано на рисунке 8.15. Внешние компоненты соединяются с двумя контактами IOB, а выходной генератор – с альтернативным буфером синхронизации (alternate clock buffer). Он управляет горизонтальными длинными линиями, которые в свою очередь могут управлять вертикальными длинными линиями и входами синхронизации K (clock) логических блоков. Если используется внешняя синхронизация, ее выход соединяется с общим буфером синхронизации (global clock buffer). Он управляет всей схемой, что обеспечивается высоким коэффициентом разветвления по выходу, синхронизирующим общий вход для всех lOB и логических блоков. Если требуется симметричная синхронизация, к выходу генератора может быть подключен триггер, делящий частоту на 2.

Рассмотренная выше микросхема XC3020 FPGA имеет 64 CLB (8 x 8), 64 пользовательских входа-выхода, 256 триггеров (128 в CLB и 128 в lOB), 16 горизонтальных длинных линий и 14779 конфигурирующих битов данных. Другие микросхемы семейства XC3000 имеют до 484 CLB (22 x 22), 176 пользовательских I/O, 1320 триггеров, 44 горизонтальных длинных линий и 94984 конфигурирующих битов данных.

Рисунок 8.15. Кварцевый генератор



8.3. Проектирование устройств на основе FPGA

В настоящее время создано большое количество CAD-систем для проектирования устройств на FPGA. Обычно такое проектирование включает шаги:

1. Разработка модели устройства с применением языков описания аппаратуры, таких как VHDL и Verilog. Схему устройства можно также нарисовать, используя соответствующий схемный редактор.

2. Преобразование HDL-кода или схемы с помощью программ синтеза в *netlist* – описание блоков проекта и соединений между ними. Чаще всего такое описание записывается в формате EDIF.

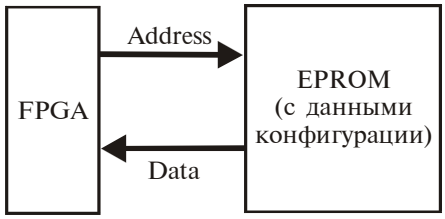
3. Использование программ имплементации для выполнения операции mapped. При этом вентили, образующие проект, разбиваются на группы, которые подходят определенным CLB. Размещение вентилей CLB и создание соединений между ними выполняется на этапе place & route.

4. После имплементации программа извлекает состояния ячеек памяти из матрицы маршрутизации (routing matrice) и генерирует битовый файл – *bitstream,* где единицы и нули определяют открытые или закрытые переключатели.

5. Bitstream-файл загружается в чип FPGA.

Когда конечная система построена, конфигурация битов для программирования FPGA обычно сохраняется в EPROM и автоматически загружается в FPGA при включении питания. EPROM соединяется с FPGA, как показано на рисунке 8.16. Сразу после подачи напряжения данные в FPGA сбрасываются. Затем считывается конфигурация из EPROM. Последовательность адресов на входы EPROM поступают с FPGA. Данные из EPROM сохраняются во внутренних конфигурируемых ячейках памяти FPGA.

Рисунок 8.16. Подключение EPROM для LCA инициализации



Реализация функций шести и более переменных

Хотя некоторые логические функции 6 переменных могут быть реализованы на одной или двух логических ячейках, в общем случае функция 6 переменных требует три ячейки. Общий метод реализации любой такой функции сначала использует ее разложение, как показано ниже:

$$Z(a,b,c,d,e,f) = a'Z(0,b,c,d,e,f) + aZ(1,b,c,d,e,f\} = a'Z_0 + aZ_1. \quad (8.1)$$

Это пример теоремы разложения Шеннона. Можно проверить, что уравнение (8.1) верно, установив а в 0, а затем в 1. Если оно верно для а=0 и для а=1, то оно всегда верно. Уравнение (8.1) приводит к схеме, в которой задействованы две ячейки для реализации Z_0 и Z_1 (рисунок 8.17, а). Половина третьей ячейки используется для реализации функции трех переменных, $Z = a'Z_0 + aZ_1$. Например, рассмотрим следующую функцию:

```
Z = abcd'ef'+ a'b'c'def' + b'cde'f
```

Установка а=0 дает Z_0 = 0 • bcd'ef'+ 1 • b'c'def '+ b'cde'f= b'c'def + b'cde'f , а установке а = 1 соответствует Z_1= 1•bcd'ef + 0•b'c'def'+ b'cde'f = bcd'ef'+ b'cde'f. Поскольку Z_0 и Z_1 являются функциями 5 переменных, каждая из них может быть реализована на одной ячейке.

Любая функция 7 переменных может быть реализована с помощью 6 или менее логических ячеек. Разложение для общей функции 7 переменных имеет вид:

$$Z(a,b,c,d,e,f,g) = a'b'Z(0,0,c,d,e,f,g) + a'bZ(0,1,c,d,e,f,g)+ +ab'Z(1,0,c,d,e,f,g)+abZ(1,1,c,d,e,f,g)=a'b'Y_0+a'bY_1+ab'Y_2+abY_3. \quad (8.2)$$

Уравнение (8.2) может быть получено применением теоремы разложения дважды: первое разложение по переменной а, затем – разложение по переменной b. Например, рассмотрим функцию 7 переменных:

```
Z = c'de'fg + bcd'e'fg' + a'c'def'g + a'b'd'ef'g' + ab'defg'.
```

– Подстановка a = b = 0 дает Y_0 = c'de'fg + c'def'g + d'ef'g'.
– Подстановка a = 0, b = 1 дает Y_1 = c'de'fg + cd'e'fg' + c'def'g.
– Подстановка a = 1, b = 0 дает Y_2 = c'de'fg + defg'.
– Подстановка a = b = 1 дает Y_3 = c'de'fg + cd'e'fg'.

Рисунок 8.17. Реализация функций 6 и 7 переменных: а – общая функция 6 переменных; б – общая функция 7 переменных

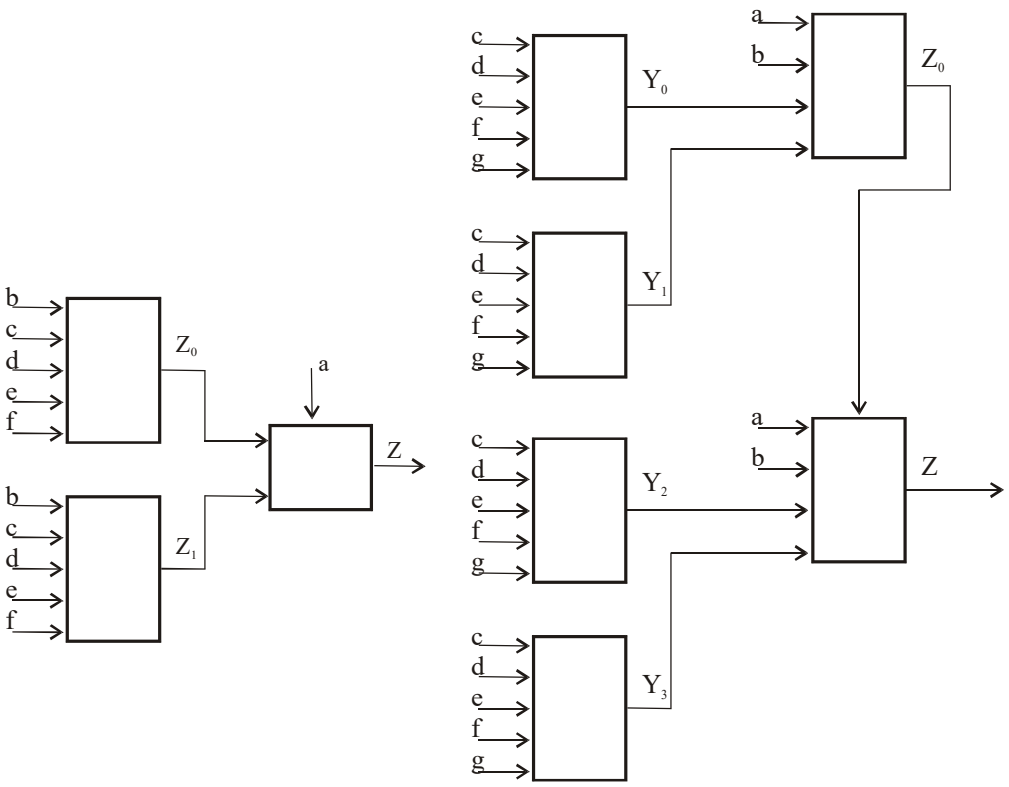


Схема на рисунке 8.17, б соответствует уравнению (8.2). Четыре ячейки реализуют функции 5 переменных: Y_0, Y_1, Y_2 и Y_3. Пятая ячейка реализует функцию 4 переменных, Z_0 = a'b'Y_0+ a'b'Y_1, и последняя ячейка реализует функцию 5 переменных, Z= Z_0+ ab'Y_2+ abY_3. При увеличении числа переменных n быстро возрастает максимальное количество логических ячеек, необходимых для реализации функции n переменных. По этой причине при большом числе входных переменных n использование CPLD может быть лучшим решением, чем FPGA.

8.4. Xilinx 4000 серии FPGA's

Xilinx FPGA серии 4000 сходны с устройствами серии 3000, но имеют большее число входов и выходов и многие другие дополнительные свойства. Рисунок 8.18 представляет упрощенную схему конфигурируемого логического блока XC4000, имеющего девять логических входов (F1, F2, F3, F4, G1, G2, G3, G4 и H1). Таким образом, можно сгенерировать две независимые функции четырех переменных:

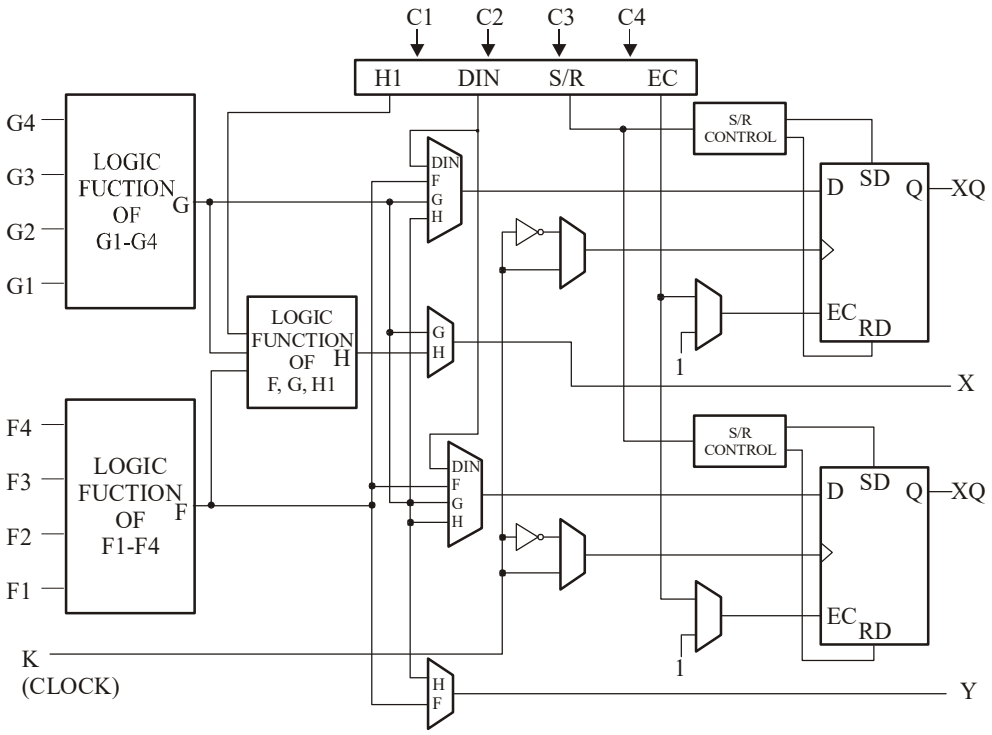
```
G(G1,G2,G3,G4) и F(F1,F2,F3,F4).
```

В отличие от устройств серии 3000, где входы генерируемых функций 4 переменных частично перекрываются, CLB серии 4000 может также генерировать функцию H,

которая определяется F, G и H1. Установив F1= G1l, F2 = G2, F3 = G3 и F4 = G4, можно сгенерировать любую функцию 5 переменных по формуле H = F(F1,F2,F3,F4) H1' + G(F1,F2,F3,F4) H1. Также возможно реализовывать некоторые функции 6, 7, 8 и 9 переменных.

CLB имеет два D-триггера с входом разрешения синхронизации EC, четыре выхода – два от триггеров и два от реализованных функций комбинационной логики. В отличие от CLB серии 3000, CLB серии 4000 не имеет внутренней обратной связи. Таким образом, когда она необходима, значения с выходов триггеров должны быть направлены назад к логическим входам с помощью внешней маршрутизации. CLB имеет S/R (set/reset – установка/сброс) вход, который может быть независимо сконфигурирован для соединения с SD или RD входом каждого триггера. Значит, один триггер может быть установлен в 1, а другой в 0 одним и тем же S/R сигналом. Синхронизирующий вход каждого триггера позволяет выполнять конфигурацию на срабатывание по переднему или заднему фронту K (clock) входа. Вход EC каждого триггера разрешает его переключение по сигналу со входа EC логического блока. D-вход каждого триггера может быть соединен с линиями DIN, F, G или H.

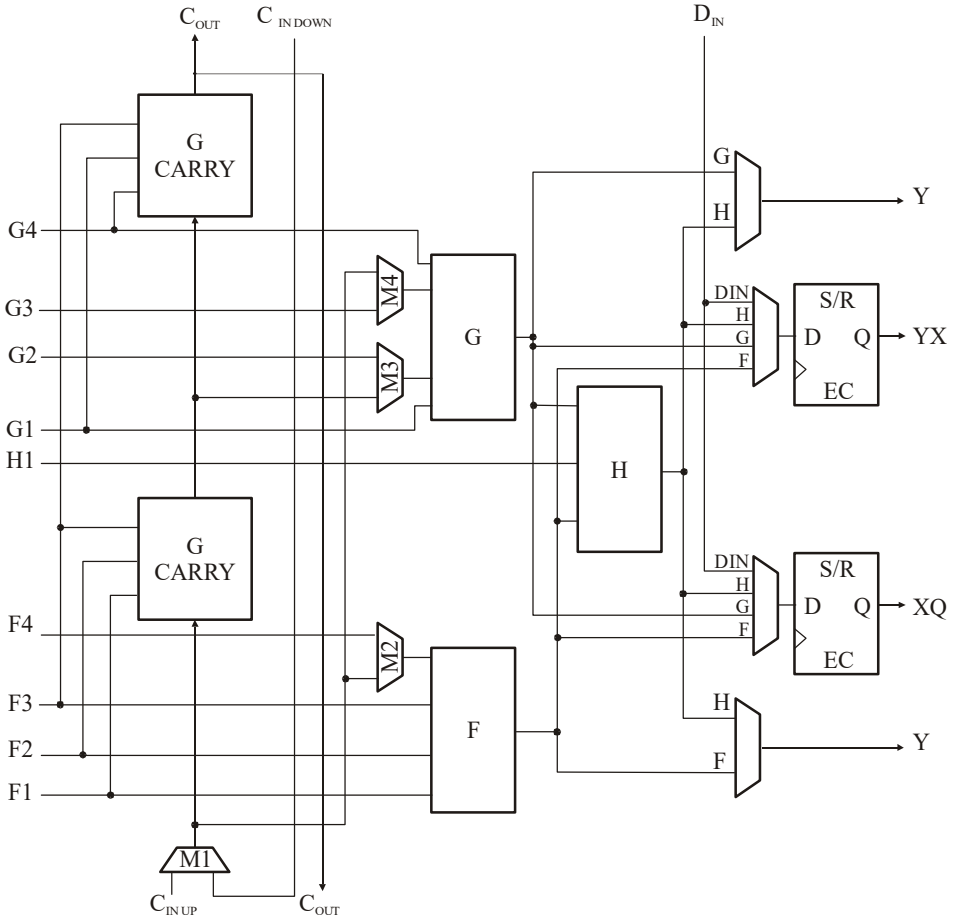
Рисунок 8.18. Упрощенная блок-схема для CLB серии 4000



Логическая ячейка устройств серии 4000 имеет специальную схему переноса (dedicated carry logic), как показано на рисунке 8.19. Каждая ячейка содержит логическую схему переноса 2-х битов, F и G функциональные преобразователи могут быть использованы для генерирования битов суммы. Таким образом, каждая ячейка может быть запрограммирована для одновременного вычисления суммы и переноса двух битов полного сумматора. Линии переноса между ячейками – фиксированные, а не программируемые, что обеспечивает быструю передачу переноса. Переносы могут передаваться вверх или вниз между ячейками и поступать на вход программируемого мультиплексора (метка M1 на рисунке 8.19), который выбирает направление передачи

сигнала. Мультиплексоры M2, M3 и M4 позволяют некоторым входам F и G генераторов функций получать значения с переносов вместо сигналов с обычных входов ячеек F и G. В добавок к сумматору схема переноса может быть запрограммирована на реализацию устройств вычитания, сложения/вычитания, увеличения/уменьшения (incrementer/decrementer), двоичного дополнения (2's complementer) и счетчиков.

Рисунок 8.19. Дополнительная логическая схема переноса XC4000



Если ячейка программируется на реализацию двух битов полного сумматора, логическая схема эквивалентна представленной на рисунке 8.20. В такой конфигурации A_i и B_i поступают на входы F1 и F2. Сигналы A_{i+1} и B_{i+1} подаются на входы G2 и G4. Вход C_i и выход C_{i+2} соединяются с линиями передачи фиксированного переноса. Рисунок 8.21 представляет соединения для 4-битного сумматора. Два средних CLB выполняют сложение 4 битов. Если перенос в последний значащий разряд необходим, C_0 должен быть направлен через дополнительную ячейку, поскольку прямое соединение фиксированной линии переноса не разрешено. Если необходим признак переполнения и внешний перенос из самого младшего разряда, требуется четвертая ячейка. В этом примере C_3 вместо C_4 направляется в четвертую ячейку и входы в ячейки A_3 и B_3 дублируются. C_4 вычисляется с помощью F функционального генератора, что обеспечивает внешний перенос из ячейки. C_4 также заново вычисляется, используя логическую схему переноса в ячейку. Переполнение вычисляется функциональным G-генератором как $V=C_3+C_4$. Этот 4-битный сумматор может быть легко расширен до 8- или 16-битного добавлением 2 или 6 дополнительных ячеек. Такие модули сумматора есть в библиотеке Xilinx.

Когда ячейка программируется для реализации двух битов устройства сложения/вычитания, сигнал $Add/\overline{Sub}$ должен быть соединен с F3 и G3. Когда $Add/\overline{Sub}=0$, входы B_i и B_{i+1} инвертируются внутри логической схемы переноса, а функциональные генераторы, таким образом, выполняют вычитание прибавлением двоичного дополнения.

Рисунок 8.20. Схема концептуального представления 2 битов полного сумматора

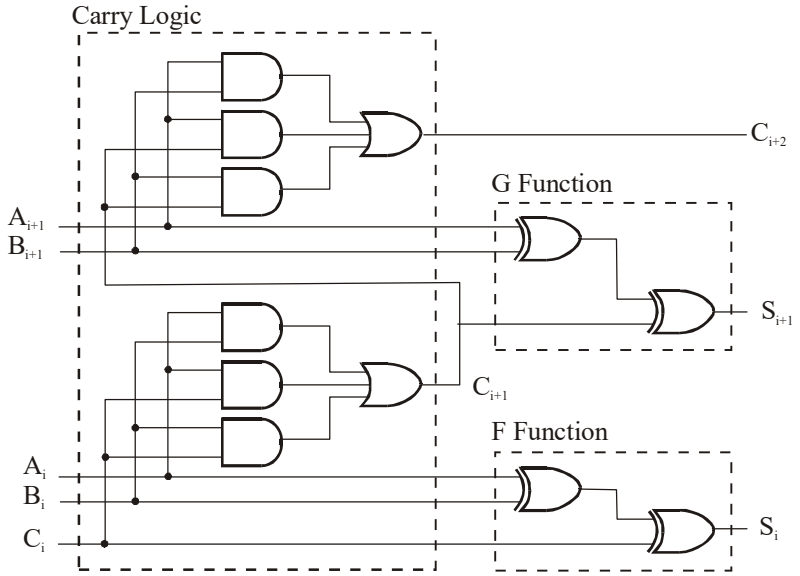
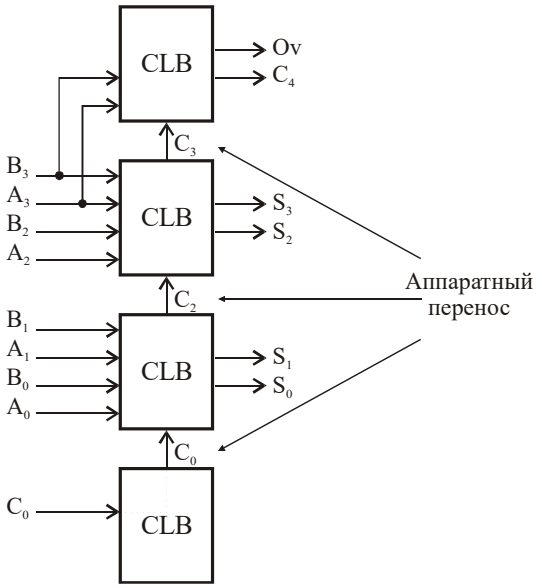


Рисунок 8.21. Соединения для 4-битового сумматора



Функциональные генераторы *F* и G могут быть запрограммированы для работы в качестве RAM памяти (рисунок 8.22). Каждая ячейка может быть сконфигурирована как 16-словная 2-битная RAM. Входы F и G функциональных генераторов (F1 = G1, F2 = G2, F3 = G3, и F4 = G4) обеспечивают 4-битную адресацию, C1 используется как линия разрешения записи WE, C2 и C3 служат для ввода данных D1 и D0. Содержимое адресуемых ячеек памяти доступно по выходам F и G функциональных генераторов. Кроме того, ячейка может быть сконфигурирована как 32x1-битовая RAM. Тогда C2 используется как пятый адресный бит, а C3 – как вход данных. Конфигурирующие биты WRITE G и WRITE F должны быть установлены на разрешение записи в F и G функциональные генераторы соответственно.

На рисунке 8.23 показан входной-выходной (I/O) блок серии 4000, который похож по возможностям на входной-выходной блок серии 3000. Метки M на диаграмме представляют конфигурирующие биты памяти. Как и IOB 3000, IOB 4000 может быть запрограммирован инвертировать входы, выходы, тристабильный контрольный буфер и вход синхронизаии. В дополнение общая S/R (GLOBAL S/R) линия может быть ориентирована на установку или сброс каждого триггера. Входной-выходной

блок 4000 имеет два pull-up и pull-down согласующих резистора, связанных с входным-выходным контактом (I/O pad). Входной-выходной блок 4000 также содержит дополнительную логику boundary scan, которая будет обсуждаться в главе 11.

Рисунок 8.22. CLB как читающая/записывающая ячейка памяти

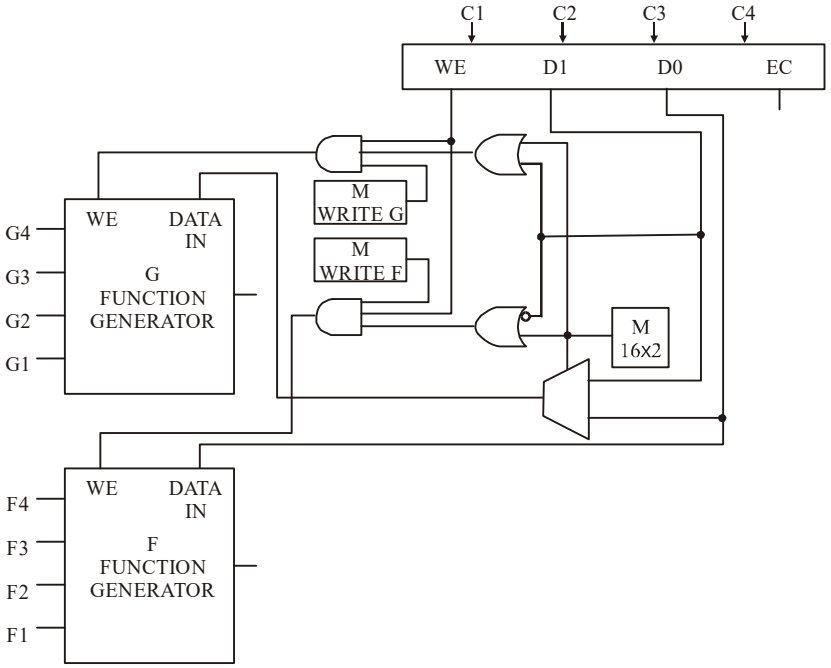
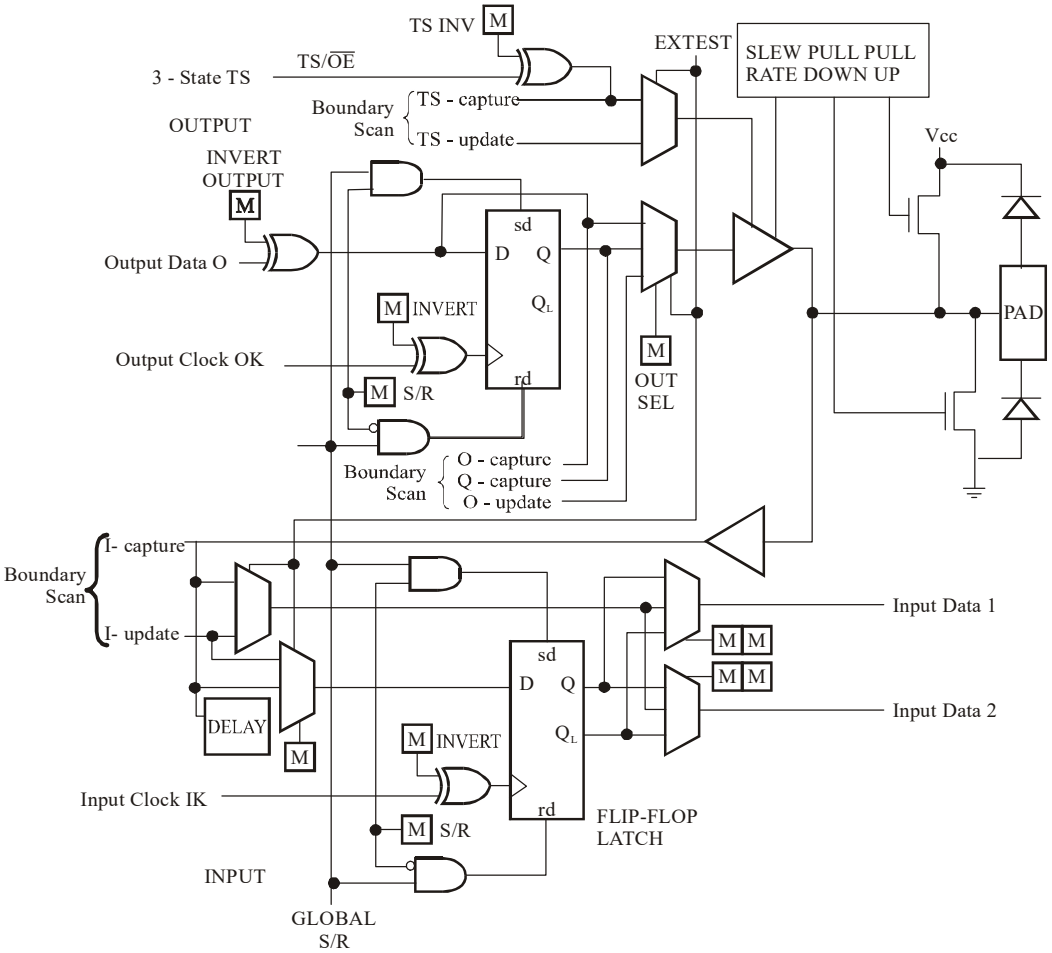


Рисунок 8.23. Входной-выходной блок серии 4000



XC4003 FPGA имеет 100 CLB (10 x 10), около 3000 вентилей, 80 пользовательских I/O, 360 триггеров (200 в CLB и 160 в lOB), 45636 конфигурирующих битов данных. Другие представители XC4000 семейства имеют до 2304 CLB (48 x 48), 384 пользовательских I/O, 5376 триггеров и более одного миллиона конфигрирующих битов данных.

VHDL-модель для CLB серии 4000

Ниже разрабатывается VHDL-модель для CLB серии 4000, включающая функциональные генераторы и триггеры, но без логической схемы переноса и RAM функции памяти. В декларации портов (рисунок 8.24) бит-вектор MEM_BITS содержит конфигурирующие биты памяти, которые определяют функции, генерируемые ячейкой, и связи с ней. G_IN представляет входы G4, G3, G2 и G1; F_IN – F4, F3, F2 и F1; C_IN – C4, C3, C2 и C1. Выходы – Y, X; двух-битовый вектор Q представляет выходы триггера (YQ и XQ).

Рисунок 8.24. Поведенческая модель XC4000 CLB

```
-- Функциональное описание XC4000 CLB
library BITLIB;
use BITLIB.BIT_PACK.ALL;

entity XC4000CLB is
   port (MEM_BITS : in bit_vector(0 to 51);
      G_IN, F_IN, C_IN : in bit_vector(4 downto 1);
      K : in bit;
      Y,X : out bit;
      Q : out bit_vector (1 downto 0));
end XC4000CLB;

architecture behavior of XC4000CLB is

   alias G_FUNC : bit_vector(0 to 15) is MEM_BITS(0 to 15);
   alias F_PUNC : bit_vector(0 to 15) is MEM_BITS(16 to 31);
   alias H_FUNC : bit_vector(0 to 7) is MEM_BITS(32 to 39);
   type bv2D is array (1 downto 0) of bit_vector(1 downto 0);
   constant FF_SEL: bv2D:= (MEM_BITS(40 to 41), MEM_BITS(42 to 43));
   alias Y_SEL : bit is MEM_BITS(44);
   alias X_SEL : bit is MEM_BITS(45);
   alias EDGE_SEL: bit_vector(1 downto 0) is MEM_BITS(46 to 47);
   alias EC_SEL : bit_vector(1 downto 0) is MEM_BITS(48 to 49);
   alias SR_SEL : bit_vector(1 downto 0) is MEM_BITS(50 to 51);

   alias H1 : bit is C_IN(1);
   alias DIN : bit is C_IN(2);
   alias SR : bit is C_IN(3);
   alias EC : bit is C_IN(4);

-- Пример временных соотношений для XC4000,
-- класс скорости (Speed Grade) -4
   constant Tiho : TIME := 6 ns;-- F/G входы к X/Y выходам через H
   constant Tilo : TIME := 4 ns;  -- F/G входы к X/Y выходам
   constant Tcko : TIME := 3 ns;  -- Синхровход K к выходам Q
   constant Trio : TIME := 7 ns;  -- S/R к выходам Q

   signal G,F,H : bit;

begin
   G <= G_FUNC (vec2int(G_IN));
   F <= F_FUNC (vec2int(F_IN));
   H <= H_FUNC (vec2int(Hl&G&F)) after (Tiho-Tilo);
   X <=(X_SEL and H) or (not X_SEL and G) after Tilo;
   Y <=(Y_SEL and H) or (not Y_SEL and F) after Tilo;
process (K, SR)                          -- обновление выходов FF
```

```
variable DFF_EC,D : bit_vector(1 downto 0);
begin
   for i in 0 to 1 loop
      DFF_EC(i) := EC or EC_SEL(i);
      case FF_SEL(i) is
         when "00" => D(i) := DIN;
         when "01" => D(i) := F;
         when "10" => D(i) := G;
         when "11" => D(i) := H;
      end case;
-- если установлен SR, триггеры установятся в 1 или сбросятся в 0
      if (SR='1') then
         Q(i)<=SR_SEL(i) after Trio;
      else
         if (DFF_EC(i)='1') then
-- Если синхронизация разрешена и если
-- присутствует фронт синхронизации – обновить значения триггеров
         if ((EDGE_SEL(i)='1' and rising_edge(K)) or
         (EDGE_SEL(i)='0' and falling_edge(K))) then
            Q(i)<=D(i) after Tcko;
         end if;
         end if;
      end if;
   end loop;
end process;
end behavior;
```

Чтобы сделать VHDL-код более понятным, использовались aliase для разделения MEM_B1TS на составляющие части. В ячейке функции, вычисляемые функциональными генераторами, сохраняются в виде таблицы истинности. G_FUNC описывает колонку выходов в таблице истинности для функции G(G4,G3,G2,G1). Поскольку G является функцией четырех переменных, она требует 16 битов. F_FUNC подобна G_FUNC, но H_FUNC требует только 8 битов, потому что H – функция трех переменных: F, G и H1. Y_SEL выбирает выход Y (1 выбирает H и 0 выбирает G). X_SEL выбирает выход X (1 выбирает H и 0 выбирает F). EDGE_SEL, EC_SEL и SR_SEL выбирают для двух триггеров фронт синхронизации, разрешение синхронизации и установку или сброс соответственно. Поскольку использование псевдонимов с массива бит-векторов на бит-вектор не разрешается, можно применять константное объявление для определения FF_SEL. FF_SEL(1) выбирает вход D для триггера YQ, а FF_SEL(0) – для триггера XQ.

Модель также содержит информацию о синхронизации. Константа Tilo – максимальное распространение задержки между изменением F/G входов и X/Y выходов. Когда выход получает значение с H функционального генератора, максимальная задержка распространения равна Tiho. Задержка от входа синхронизации K до выходов Q – Tcko, задержка с SIR (C3) до Q – Trio.

Тело архитектуры начинается с параллельного оператора, который обновляет G, F, H, X и V. Описание задержек для отдельных функциональных генераторов не дается в справочных данных, но эти задержки включаются в Tilo, когда обновляется X или Y. Если используется функциональный генератор H, дополнительная задержка равна (Tiho – Tilo).

Цикл for в процессе обновляет выходы триггера при любом изменении K или SR. Синхронизация D-триггера доступна, если EC_SEL(i) ='1' или вход EC (C4) = '1'. Оператор case представляет мультиплексор, который выбирает D(i) вход триггера, как DIN, F, G или H, в зависимости от значения FF_SEL(i). Когда SR изменится на '1', Q(i) установится или сбросится по истечению задержки Trio. Иначе, Q(i) обновит значение после задержки Tcko, если синхронизация доступна и встретился выбранный фронт синхронизации.

Чтобы проиллюстрировать использование модели VHDL CLB, реализуем управляющий граф состояний устройства умножения (рисунок 7.6) с помощью логических блоков XC4000. Его логические уравнения выглядят следующим образом:

```
Q1+   =  K Q1'Q0 + M Q1'Q0 + K Ql Q0'                    (8.3 а)
Q0+   =  StQ0'+ M'Q1'Q0 + QlQ0'                          (8.3 б)
Done  =  Q1Q0                                            (8.3 в)
Load  =  StQ1'Q0'                                        (8.3 г)
Ad     = MQ1'Q0                                          (8.3 д)
Sh     = M'Q1'Q0+Q1 Q0'                                  (8.3 е)
```

Поскольку каждое уравнение требует четыре или менее переменных, уравнения могут быть реализованы с помощью трех логических блоков.

На рисунке 8.25 представлен VHDL-код, который иллюстрирует пример трех копий компонентов XC4000 CLB и соединений между ними, реализующих уравнения (8.3). Задержки межсоединений между ячейками не учитываются, поскольку они не известны до размещения ячеек и создания связей между ними. Константы MEM1, MEM2 и MEM3 описывают конфигурирующие биты памяти для логических блоков. Эти биты описываются как константы, раз они загружаются в CLB после включения напряжения и затем не изменяются. Входы G блока CLB1, сигнал G_IN1 в VHDL-модели соответствуют G4=K, G3=M, G2=Q1 и G1=Q0. Таблица 8.1 представляет таблицу истинности для функционального генератора G, вычисляющего $Q1^+$, уравнение (8.3 а). Из этой таблицы первые 16 битов MEM1 равны "0000010001100110". Функциональный генератор F реализует уравнение (8.3 б), и следующие 16 битов MEM1 соответствуют таблице истинности для $F = Q0^+$ (см. таблицу 8.1). Поскольку H-функция не используется, следующие 8 битов равны 0. Оставшиеся 8 битов MEM1 описывают конфигурацию ячейки.

Таблица 8.1. Таблицы истинности для G и F функциональных генераторов

(a)

G4	G3	G2	G1	G
K	M	Q1	QO	Q1+
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

(b)

F4	F3	F2	F1	F
St	M	Q1	QO	$Q0^+$
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Рисунок 8.25. XC4000 реализация устройства управления умножением

```
entity Fig_4_6 is
   port (St, K, M, CLK : in bit;
      Ad, Sh, Load, Done : out bit);
end Fig_4_6;

architecture CLBs of Fig_4_6 is
   component XC4000CLB
```

```
      port(MEM_BITS : in bit_vector(0 to 51);
         G_IN, F_IN, C_IN : in bit_vector(4 downto 1);
         K : in bit;
         Y,X : out bit;
         Q : out bit_vector (1 downto 0));
   end component;

constant MEM1 : bit_vector (0 to 51) :=
     "000001000110011001100010111010100000000100100110000";
constant MEM2 : bit_vector (0 to 51) :=
     "000100010001000100000001000100000000000000000110000";
constant MEM3 : bit_vector (0 to 51) :=
     "000000000100010001100110001000100000000000000110000";

signal Q : bit_vector (1 downto 0) ;
signal G_IN1,G_IN2,G_IN3,F_IN1,F_IN2,F_IN3: bit_vector (3 downto 0;

begin
   G_IN1<=K&M&Q; F_INl<=St&M&Q;
   G_IN2<="00"&Q; F_IN2<=St&'0'&Q;
   G_IN3<=M&'0'&Q; F_IN3<=M&'O'&Q;

   CLBl: XC4000CLB port map (MEMl,G_IN1,F_IN1,"1000",CLK,open,open,Q);
   CLB2: XC4000CLB port map
(MEM2,G_IN2,F_IN2,"1000",CLK,Done,Load,open);
   CLB3: XC4000CLB port map (MEM3,G_IN3,F_IN3,"1000",CLK,Ad,Sh,open);
end CLBs;
```

8.5. Использование унарного распределения состояний

При создании проектов для PGA следует помнить, что каждая логическая ячейка содержит два триггера. Это значит, что минимизация количества используемых триггеров не имеет значения. Вместо этого нужно попытаться сократить общее число используемых логических ячеек и межсоединений между ними. Для того чтобы проектировать более быстрые схемы, следует уменьшить число ячеек, необходимых для реализации каждого уравнения. Применение *унарного распределения состояний* часто помогает этому.

Унарное распределение использует один триггер для каждого состояния, таким образом, автомат с N состояниями требует N триггеров. Только один триггер устанавливается в 1 для каждого состояния. Например, система с четырьмя состояниями (T0, T1, T2 и T3) может использовать четыре триггера (Q0, Q1, Q2 и Q3) со следующим распределением состояний:

```
TO: QO Ql Q2 Q3 = 1000, Tl: 0100,  T2: 0010, T3:0001.          (8.4)
```

Остальные 12 комбинаций не применяются.

Можно записать уравнения состояний и выходов, рассматривая граф состояний. Рассмотрим неполный граф, представленный на рисунке 8.26. Уравнение следующего состояния для триггера Q3 может быть записано так:

```
Q3+ = X1Q0Q1'Q2'Q3' + X2Q0'Q1Q2'Q3' +X3Q0'Q1'Q2Q3' + X4Q0'Ql'Q2'Q3.
```

Тем не менее, то, что Q0 = 1, означает Q1=Q2=Q3=0, терм Q1'Q2'Q3' – избыточный, его можно исключить. Подобным образом все переменные автомата с отрицанием могут быть исключены из других термов. Таким образом, уравнение следующего состояния сократится до

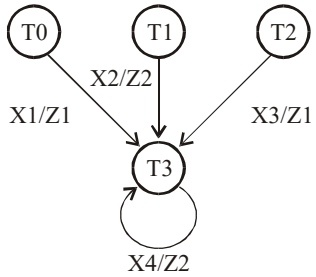
```
Q3+ = X1Q0 + X2Q1 + X3Q2 + X4Q3.
```

Теперь каждый терм содержит точно одну переменную состояния. Точно так же каждый терм уравнения выходов содержит точно одну переменную состояния:

```
Z1= X1Q0 + X3 Q2,   Z2=X2 Q1 + X4 Q3.
```

При использовании унарного назначения уравнение следующего состояния для триггера будет содержать один терм для каждой дуги, ведущей в соответствующее состояние, или для каждого пути, ведущего в состояние. В общем случае, каждый терм в уравнении следующего состояния и в каждом уравнении выходов содержит точно одну переменную состояния.

Рисунок 8.26. Неполный граф состояний



Когда используется унарное кодирование состояний, сброс системы требует, чтобы один триггер был установлен в 1 вместо сброса всех триггеров в 0. Если триггеры не имеют входов предварительной установки, как в случае с Xilinx серии 3000, тогда можно модифицировать унарное кодирование, заменив везде Q0 на Q0'. Для предыдущих уравнений такая модификация выглядит следующим образом:

T0: Q0 Ql Q2 Q3 = 0000, T1:1100, T2: 1010, T3: 1001 (8.5)

и измененные уравнения:

$Q3^{+}$ = XI Q0' + X2 Ql + X3 Q2 + X4 Q3;

Zl = XI Q0' + X3 Q2, Z2 =X2Q1+ X4 Q3.

Другой способ решения проблемы установки без изменения унарного кодирования есть включение дополнительного терма в уравнение для триггера, который должен быть равен 1 для начального состояния. Если система сбрасывается в состояние 0000 после включения питания, можно добавить терм Q0'Q1'Q2'Q3' в уравнение для $Q0^{+}$. Затем, после первого импульса синхронизации, состояние изменится с 0000 на 1000 (T0), которое является корректным начальным состоянием автомата.

В общем случае для любого способа кодирования следует попытаться найти решение проекта, требующее меньшего числа ячеек. Иначе, если скорость операций важна, нужно выбрать вариант проекта, чья логическая схема быстрее. При использовании прямого присвоения требуется более одного уравнения следующего состояния. В общем случае уравнения состояний и выходов, взятые вместе, содержат меньше переменных. Для реализации уравнения с меньшим числом переменных обычно используется небольшое число логических ячеек. Уравнения с пятью и меньшим числом переменных требуют всего одну логическую ячейку. Как показано на рисунке 8.17, уравнение с шестью переменными может потребовать каскад из двух ячеек, уравнения с семью переменными – каскад из трех ячеек. Чем больше число ячеек в каскаде, тем больше задержка распространения и медленнее операции.

8.6. Virtex

Дальнейшее развитие FPGA XC4000X привело к созданию новой серии микросхем Virtex, а также упрощенной версии Virtex – Spartan. СерияVirtex FPGA предоставляет высокоинтегрированные и многофункциональные микросхемы. Мощность и гибкость этих микросхем делает их альтернативой масочным матрицам логики. Серия Virtex насчитывает 9 типов микросхем, некоторые из них приведены в таблице 8.2. Их

основные характеристики: высокая степень интеграции; плотность от 50К до 1Мб системных вентилей; частотные возможности до 200МГц; совместимость с 66МГц PCI. Микросхемы серии Virtex-II содержат до 8 Мб системных вентилей.

Таблица 8.2. Микросхемы серии Virtex

Device	System Gates	CLB Array	Logic Cells	Maximum Avaible I/O	BlockRAM Bits	Maximum SelectRAM+™Bits
XCV50	57 906	16x24	1 728	180	32 768	24 576
XCV100	108 904	20 x 30	2 700	180	40 960	38 400
XCV800	888 439	56 x 84	21 168	512	114 688	301 056
XCV1000	1 124 022	64 x 96	27 648	512	131 072	393 216

Архитектура Virtex

Общая схема чипа Virtex FPGA изображена на рисунке 8.27. Микросхема состоит из двух основных элементов: конфигурируемых логических блоков (configurable logic block – CLB) и входных/выходных блоков (input/output block – lOB). CLB обеспечивают функциоальные элементы для конструирования устройств. IOB предоставляют интерфейс между контактами микросхемы и CLB.

Связь между CLB выполняется с помощью общих матриц маршрутизации (general routing matrix – GRM). GRM представляют собой матрицы переключателей, расположенных на пересечениях вертикальных и горизонтальных каналов трассировки. Каждый CLB в свою очередь вмонтирован в VersaBlock™, позволяющий выполнять местную маршрутизацию для связи CLB с GRM.

VersaRing™ – входной-выходной интерфейс предостовляет дополнтельные ресурсы для маршрутизации по периметру микросхемы, которые улучшают возможности маршрутизации входов/выходов и облегчают связь с внешними контактами.

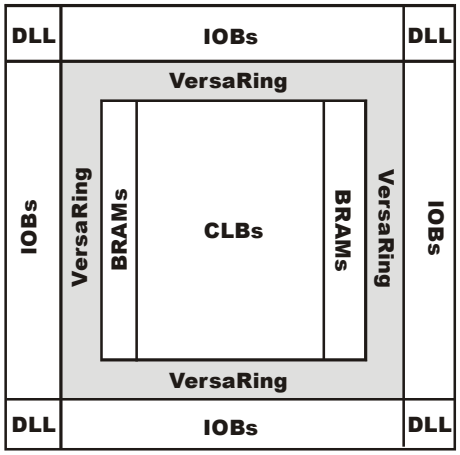
Virtex-архитектура включает также следующие элементы, доступ к которым можно осуществлять через матрицы маршрутизации (GRM):

– дополнительные блоки памяти по 4096 битов в каждом;

– синхронизацию DLL для компенсации задержки распространения синхросигнала и управления синхронизацией или длительностью синхроимпульса;

– тристабильные буферы, связанные с каждым CLB, которые управляют выделенными сегментируемыми горизонтальными ресурсами трассировки.

Рисунок 8.27. Общая схема Virtex-архитектуры



Данные, хранящиеся в статических ячейках памяти, управляют конфигурируемыми логическими элементами и ресурсами трассировки. Они заносятся в ячейки памяти при подаче напряжения и могут быть выгружены в случае необходимости изменения функции устройства.

Конфигурируемый логический блок (Configurable Logic Block)

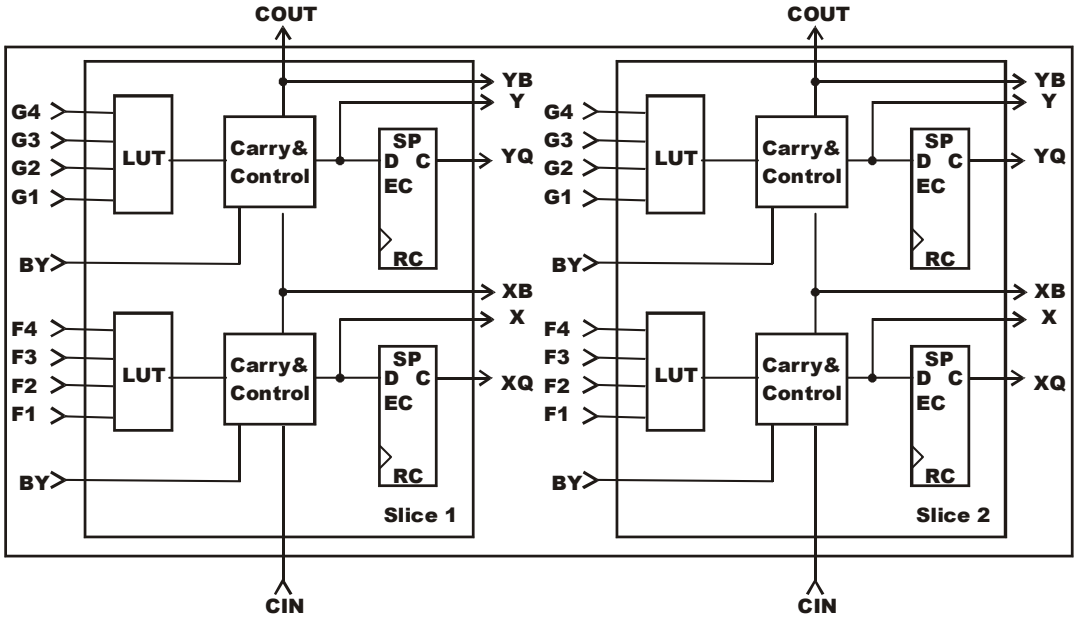
Основным элементом в Virtex CLB является логическая ячейка (logic cell – LC). Она состоит из генератора функции 4-х переменных, схемы переноса (carry logic) и логического элемента. Выход генератора функций может быть связан с любым из выходов CLB или входом D-триггера. Каждый Virtex CLB содержит четыре LC, организованных в виде двух одинаковых секторов, как это показано на рисунке 8.28. На рисунке 8.29 представлена более детальная схема одного сектора.

В дополнение к четырем основным LC Virtex CLB содержит логику, объединяющую функциональные генераторы для реализации функций пяти или шести переменных. Поэтому, когда оценивают число вентилей, предоставляемых данным устройством, каждый CLB оценивается как 4,5 LC.

Look-Up Table

Virtex функциональный генератор организован как 4-входовой (look-up table) LUT. В дополнение к генерации функции каждый LUT может функционировать как 16x1-бит синхронное ОЗУ (RAM). Более того, два LUT из одного сектора могут быть объединены для создания 16x2 или 32x1-бит синхронных ОЗУ, или 16x1-бит двухканальных синхронных ОЗУ. Virtex LUT может также работать как 16-битовый сдвиговый регистр, который идеален для высокоскоростных или пакетных данных.

Рисунок 8.28. Конфигурируемый логический блок микросхемы Virtex

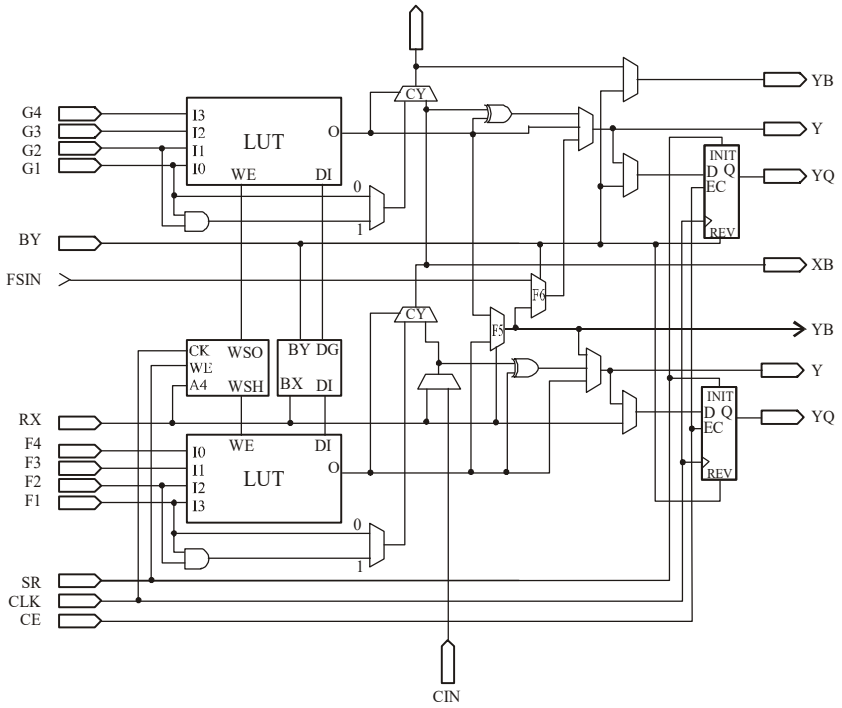


Триггеры в секторе CLB могут быть сконфигурированы как D-триггеры с синхронизацией по фронту или по уровню. Значения на вход триггера подаются с выхода функционального генератора или непосредственно со входов сектора, в обход функциональных генераторов.

Каждый триггер, в дополнение к сигналам синхронизации и разрешения синхронизации, имеет синхронный сброс SR и установку BY. SR устанавливает элемент памяти в начальное состояние, как оно описано для него в конфигурации. BY устанавливает элемент памяти в противоположное состояние. Допускается настройка этих сигналов для работы в асинхронном режиме.

Все управляющие сигналы могут быть независимо проинвертированы и являются общими для обоих триггеров сектора.

Рисунок 8.29. Подробная схема сектора Virtex FPGA



Дополнительная логика (Additional Logic)

Мультиплексор F5 в каждом секторе объединяет выходы функциональных генераторов. Таким образом, реализуется одна функция 5 переменных, мультиплексор 4 на 1 или некоторые частные случаи функций до 9 переменных.

Аналогичным образом мультиплексор F6 объединяет выходы всех четырех функциональных генераторов в CLB, выбирая один из выходов мультиплексоров F5. Это позволяет реализовать любую функцию 6 переменных, мультиплексор 8 на 1 или некоторые функции до 19 переменных.

Каждый CLB имеет четыре непосредственных сквозных пути, по одному на LC. Они предоставляют дополнительные входные линии для данных или маршрутизации дополнительной логики, позволяя не расходовать ресурсы схемы.

Арифметическая логика (Arithmetic Logic)

Специальная схема переноса выполняет быстрое вычисление переноса, давая возможность реализовывать высокоскоростные схемы переноса. Virtex CLB поддерживает две различные цепи переноса, по одной на каждый сектор. Высота цепи – два бита на CLB.

Арифметическая логика содержит один элемент XOR, позволяя реализовывать один бит полного сумматора на схеме. К тому же выделенный И элемент позволяет осуществить эффективную реализацию умножения. Схема переноса может использоваться и для каскадирования функциональных генераторов при реализации функций большого числа переменных.

Каждый Virtex CLB содержит два тристабильных драйвера (BUFTs), которые могут управлять шинами в микросхеме. Каждый Virtex BUFT имеет независимый 3-стабильный управляющий контакт и независимый входной контакт.

Блоки ОЗУ (Block RAM)

Virtex FPGA содержат несколько больших выделенных блоков ОЗУ BlockSelectRAM+, в дополнение к SelectRAM+ LUTRAM, рассредоточенных по CLB и формирующих теневую ОЗУ структуру, реализованную в конфигурируемых логических блоках.

Блоки памяти BlockSelectRAM+ организованы в столбцы. Все устройства Virtex содержат два таких столбца, распространяющихся на всю высоту чипа, по одному вдоль каждой из вертикальных сторон. Каждый блок памяти имеет высоту, равную 4 CLB. Соответственно Virtex-устройство высотой в 64 CLB содержит 16 блоков памяти в столбце и 32 блока памяти в микросхеме. В таблице 8.3 представлено количество блоков памяти Block SelectRAM+ в некоторых микросхемах Virtex.

Таблица 8.3. Количество Block SelectRAM+ в микросхемах серии Virtex

Device	# of Blocks	BlockRAM Bits
XCV50	8	32 768
XCV100	10	40 960
XCV800	28	114 688
XCV1000	32	131 072

Каждая ячейка Block SelectRAM+, как показано на рисунке 8.30, является полностью синхронизированным двухпортовым 4096-битным блоком ОЗУ с независимым управлением сигналами для каждого порта. Ширина данных может быть независимо сконфигурирована для каждого порта, что обеспечивается встроенным преобразователем ширины. В таблице 8.4 описана взаимосвязь между шириной шины и остальными параметрами Block SelectRAM+. Блоки ОЗУ в Virtex включают также дополнительную трассировку,обеспечивающую эффективный интерфейс между блоками CLB и другими модулями ОЗУ.

Рисунок 8.30. Двухпортовый Block SelectRam+

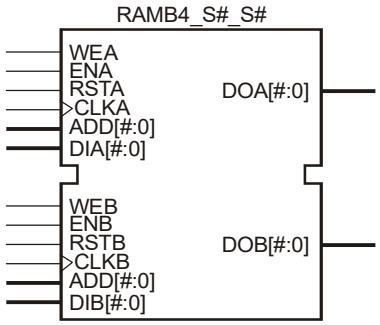


Таблица 8.4. Зависимость параметров ОЗУ Block SelectRAM+ от ширины шины

Width	Depth	ADDR Bus	Data Bus
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

Входной-выходной блок

На рисунке 8.31 изображен входной-выходной блок (IOB)Virtex. Он предоставляет возможность конфигурации входов и выходов, поддерживает большое количество входных и выходных стандартов (таблица 8.5). Эти высокоскоростные входы и выходы поддерживают PCI интерфейс до 66 МГц.

Три элемента памяти в IOB могут функционировать как триггеры с синхронизацией по фронту или по уровню. Каждый блок имеет сигнал синхронизации, общий для всех триггеров, и независимые сигналы разрешения синхронизации для каждого триггера. Вдобавок к управляющим сигналам CLK и CE триггеры имеют общий сигнал сброса/установки SR. Для каждого триггера этот сигнал может быть независимо сконфигурирован как синхронный или асинхронный сброс или установка. Входные и выходные буферы и все IOB управляющие сигналы имеют независимый контроль за направлением.

Рисунок 8.31. Входной-выходной блок Virtex FPGA

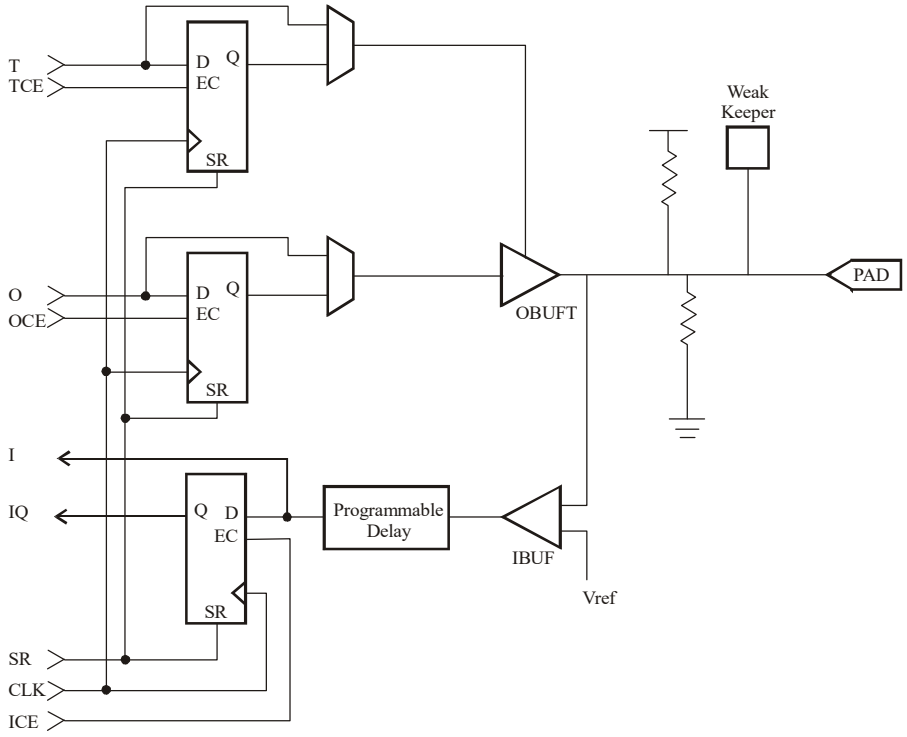


Таблица 8.5. Поддерживаемые входные-выходные стандарты

I/O Standart	Input Reference Voltage (Vref)	Output Source Voltage (Vcco)	Board Termination Voltage (Vtt)
LVTTL 2-24 mA	N/A	3.3	N/A
LVCMOS2	N/A	2.5	N/A
PCI	N/A	3.3	N/A
GTL	0.8	N/A	1.2
GTL+	1.0	N/A	1.5
HSTL Class I	0.75	1.5	1.5
HSTL Class III	0.75	1.5	1.5
HSTL Class IV	0.75	1.5	1.5
SSTL3 Class I and II	1.5	3.3	1.5
SSTL2 Class I and II	1.25	2.5	1.25
CTT	1.5	3.3	1.5
AGP	1.32	3.3	N/A

Все контакты имеют защиту микросхемы от повреждений из-за скачков напряжения. Предлагается две формы для защиты от высокого напряжения. Одна предоставляет согласование с 5В (compliance), другая нет. В первом случае подобно туннельному пробою выполняется соединение с землей, когда напряжение становится

выше, приблизительно 6,5В. Когда нет необходимости в согласовании с 5В, к выходу подключается обыкновенный ограничительный диод, поддерживающий напряжение V_{CCO}. Тип защиты может быть выбран независимо для каждого контакта.

Каждый контакт имеет повышающий (pull-up) и понижающий (pull-down) резисторы и weak-keeper схему. До программирования микросхемы все ее выходы считаются не конфигурированными и принудительно устанавливаются в состояние высокого импеданса. Понижающие резисторы и weak-keeper схемы не активны, но на входы может быть подан высокий уровень напряжения.

До конфигурации активизация повышающего резистора контакта управляется глобальной логикой. Если они не активны, напряжение на контактах плавает. Поэтому следует использовать внешние повышающие или понижающие резисторы для контактов, требующих наличия на них до конфигурации определенного логического уровня напряжения. Все Virtex IOB поддерживают IEEE 1149.1стандарт граничного сканирования (boundary scan).

Если блок Virtex IOB используется как входной, сигнал с внешнего контакта через буфер IBUF передается прямо или через триггер на внутреннюю логику. Необязательный программируемый элемент задержки на входе D-триггера позволяет компенсировать задержку на линии между контактами (pad-to-pad hold time). Величина задержки согласуется с задержкой внутренней синхронизации FPGA, и в этом случае считается, что упомянутый интервал (pad-to-pad hold time) равен 0.

Выходной путь содержит тристабильный буфер, который управляет поступлением сигнала на выход. Сигнал может подаваться непосредственно на внешний контакт или через IOB выходной триггер. Тристабильный выходной буфер может управляться непосредственно из внутренней логики микросхемы или через триггер, который позволяет синхронизировать сигнал запрещения и разрешения.

Каждый контакт может быть запрограммирован на работу в режиме любого из поддерживаемых электрических стандартов. Некоторые из них используют определяемое пользователем пороговое напряжение V_{REF}. Допускается конфигурация входов необязательными повышающими и понижающими резисторами 50-150 kОм. Каждый выход может поставлять до 24 mA и принимать до 48 mA. Управление мощностью и скоростью нарастания выходного напряжения позволяет уменьшить переходные процессы на шине. Во многих стандартах выходное напряжение высокого уровня зависит от подаваемого внешне напряжения Vcco.

Необязательная схема weak-keeper, подсоединяемая к каждому выходу, следит и мягко управляет максимальным (High) или минимальным (Low) напряжением выходного сигнала, сравнивая его со входным. Если контакт подсоединен к многоисточниковому сигналу, схема будет сохранять последнее значение сигнала, когда выходы всех его источников находятся в состоянии высокого импеданса. Это позволяет избежать возникновения дребезга на шине.

Схема weak-keeper используется во входных буферах (IOB), для наблюдения за входным уровнем и обеспечения соответствующего напряжения V_{REF}, если этого требует стандарт сигнала.

Банки входов/выходов (I/O Bank)

Как было показано выше, некоторые из входных-выходных стандартов требуют использования напряжений V_{CCO} и V_{REF}, которые подаются извне и подсоединяются к контактам, обслуживающим группы IOB блоков. Такие группы называются банками. Существуют ограничения на комбинацию входных-выходных стандартов в каждом банке.

Восемь входных-выходных банков являются результатом деления каждой стороны FPGA на два банка, как это показано на рисунке 8.32. Каждый их них имеет несколько Vcco контактов. Все они должны быть подключены к одному и тому же источнику напряжения, которое определяется используемым выходным стандартом.

Внутри одного банка разные стандарты могут быть использованы, если они имеют одинаковое Vcco напряжение. В таблице 8.6. описана возможность совмещения различных стандартов. GTL и GTL+ могут применяться с любым из трех напряжений, поскольку их выходы с открытым стоком (open-drain outputs) не зависят от Vcco.

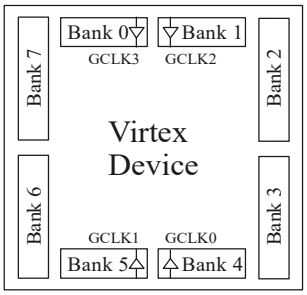
Таблица 8.6. Совместимость выходных стандартов

Vcco	Совместимые стандарты
3.3 V	PCI, LVTTL, SSTL3 I, SSTL3 II, CTT, AGP, GTL, GTL+
2.5 V	SSTL2 I, SSTL2 II, LVCMOS2, GTL, GTL+
1.5 V	HSTL I, HSTL III, HSTL IV, GTL, GTL +

Некоторые входные стандарты требуют определенного пользователем порогового напряжения Vref. В этом случае отдельные пользовательские входные/выходные контакты автоматически конфигурируются как входы для Vref напряжения. Для этого может быть использован, приблизительно каждый шестой контакт в банке.

Все Vref контакты имеют внутреннее соединение в пределах одного банка, следовательно, в одном банке может быть использовано только одно Vref напряжение.

Рисунок 8.32. Virtex I/O банки



Программируемые матрицы трассировки (Programmable Routing Matrix)

Задержка самого длинного пути определяет скорость любого проекта. Оптимизация, которая минимизирует задержки длинных путей, позволит создать устройство с лучшими системными возможностями. Она также сократит время компиляции проекта, поскольку архитектура становится дружелюбной к программному обеспечению, что соответственно уменьшит и общую продолжительность циклов проектирования.

Местная трассировка (Local Routing)

VersaBlock обеспечивает локальные ресурсы трассировки и, как показано на рисунке 8.33, предоставляет следующие типы связи:

– соединения между LUT, триггерами и GRM;

– обратная связь для CLB, обеспечивающая высокоскоростное соединение LUT внутри одного CLB и объединяющая их в цепь с минимальной задержкой на линиях трассировки;

– прямое соединение, которое обеспечивает высокоскоростное соединение между соседними горизонтальными CLB, исключая задержку на GRM.

Трассировка общего назначения (General Purpose Routing)

Большинство Virtex-сигналов направляются через трассировку общего назначения, соответственно, большинство соединений ассоциируются с этим уровнем иерархии трассировки. Ресурсы трассировки общего назначения располагаются в горизонталь-

ных и вертикальных каналах, связанных со строками и столбцами CLB. Эти ресурсы перечислены ниже:

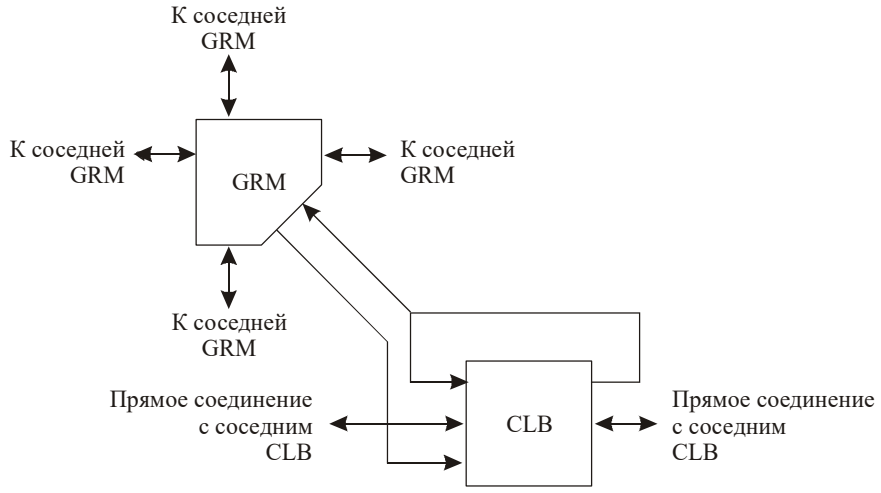
– Соседней к каждому CLB является матрица общей трассировки (General Routing Matrix (GRM)). GRM – это матрица переключателей, соединяющая горизонтальные и вертикальные ресурсы и предоставляющая для CLB доступ к ресурсам трассировки общего назначения.

– 24 линии трассировки единичной длины передают сигналы к соседним GRM в каждом из 4 направлений.

– 72 буферизированных Hex lines передают сигналы в каждом из 4 направлений. Организованные в шахматном порядке Hex lines могут управляться только в их конечных точках.Сигналы с них доступны либо в конечных, либо в средних точках (три блока от источника). Каждая третья линия – двунаправленная, остальные – однонаправленные.

– 12 длинных линий (Longlines) являются буферизированными, двунапрвленными, передающими сигнал через устройство быстро и эффективно.Они имеют длину всего чипа, в высоту или ширину.

Рисунок 8.33. Virtex локальная маршрутизация



Входная-выходная трассировка (I/O Routing)

Устройства Virtex имеют дополнительные ресурсы трассировки на его границах, формирующие интерфейс между матрицей CLB и IOB блоками. Такая дополнительная трассировка, называемая VersaRing, облегчает pin-свопинг и pin-соединение, так что измененный проект может быть адаптирован к существующей PCB (Printed Circuit Board – печатная плата) планировке. Это позволяет сократить время выхода на рынок, поскольку PCB и другие компоненты системы могут уже выпускаться, в то время как проект находится еще на стадии разработки.

Выделенная трассировка (Dedicated Routing)

Некоторые виды сигналов требуют выделенных ресурсов трассировки. В Virtex-архитектуре такие ресурсы введены для двух классов сигналов:

– Горизонтальные ресурсы трассировки представляют собой внутренние тристабильные шины. Каждому ряду CLB предоставляются четыре шинные линии, позволяя создавать сложные шины, как это показано на рисунке 8.34.

– Две дополнительные линии на CLB передают сигналы переноса вертикально между соседними CLB.

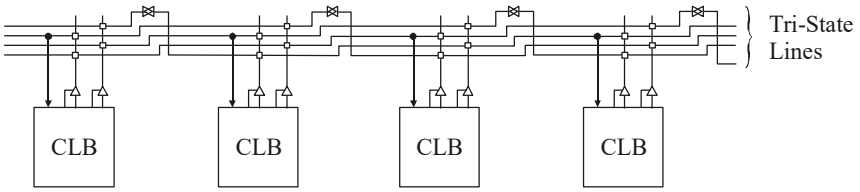
Глобальная трассировка (Global Routing)

Глобальные ресурсы трассировки распределяют синхросигналы и другие сигналы, имеющие высокое разветвление по устройству. Устройства Virtex включают два типа глобальных ресурсов трассировки: первичные и вторичные.

К первичным глобальным ресурсам маршрутизации относят четыре выделенные глобальные линии с выделенными входными контактами, которые спроектированы для распространения с минимальной расфазировкой высокоразветвленных синхросигналов. Каждая линия глобальной синхронизации может управлять всеми синхровходами блоков CLB, IOB и RAM. Первичные глобальные линии управляются только глобальными буферами. Существует четыре глобальных буфера, по одному на каждую линию.

Вторичные глобальные ресурсы трассировки состоят из 24 магистральных линий (backbone lines), 12 вдоль верха микросхемы и 12 внизу.От этих линий могут быть распространены до 12 отдельных сигналов в столбце, через 12 длинных линий. Вторичные ресурсы являются более гибкими, чем первичные, и их использование не ограничивается только сигналами синхронизации.

Рисунок 8.34. Подключение BUFT к выделенным горизонтальным линиям

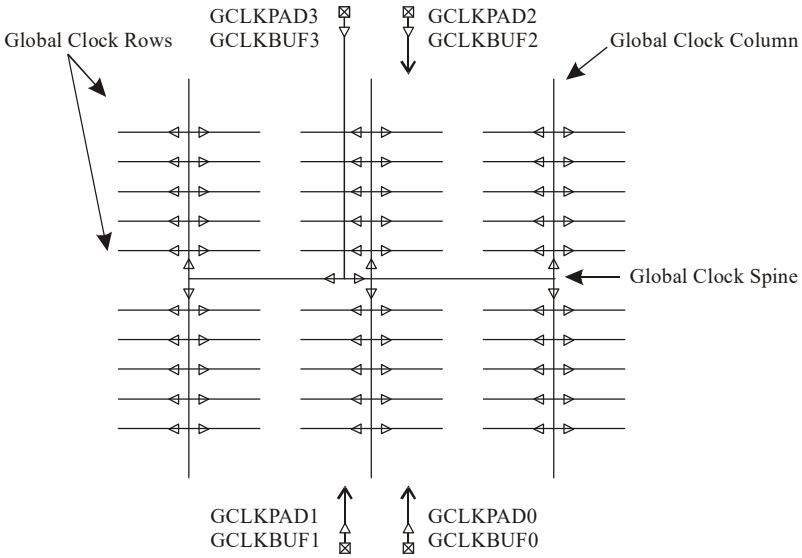


Распространение синхросигналов (Clock Distribution)

Virtex обеспечивает высокоскоростное, с низкой задержкой, распространение синхросигналов через первичные глобальные ресурсы маршрутизации, описанные выше. Типичное распространение сигналов представлено на рисунке 8.35.

Имеются 4 глобальных буфера: два вверху и два внизу, по центру устройства. Они управляют 4 первичными глобальными линиями, которые, в свою очередь, управляют любым синхровходом. Существуют четыре выделенных синхроконтакта, по одному на каждый глобальный буфер. Вход глобального буфера выбирается только через эти контакты или через сигналы трассировки общего назначения.

Рисунок 8.35. Схема распределения глобальной синхронизации



Система автоматической подстройки по задержке

Связанный с каждым глобальным синхросигналом входной буфер является полностью цифровой системой автоматической подстройки по задержке (Delay-Locked Loop – DLL), которая может исключать сдвиг сигнала между входным контактом синхросигнала и внутренним синхровходом. Каждая DLL может управлять двумя глобальными синхролиниями. DLL наблюдает входной и распространяемый синхросигналы и автоматически регулирует элемент, задерживающий синхросигнал. Дополнительная задержка вводится таким образом, чтобы фронт на внутренние триггеры поступал точно с задержкой на один период синхроимпульса. Эта система имеет обратную связь и эффективно исключает задержку распространения сигнала, гарантируя, что он будет подан на внутренние триггеры синхронно с поступлением фронта на вход.

В добавок к исключению задержки распространения сигнала DLL обеспечивает дополнительное управление длительностью синхроимпульса. DLL предлагает 4 квадратурных фазы исходного синхроимпульса и может удваивать его частоту или делить ее на 1.5, 2, 2.5, 3,4, 5, 8, 16. Он имеет 6 выходов.

DLL также функционирует как зеркало для синхросигнала. Управляя выходом с микросхемы и получая его обратно, DLL может быть использована для устранения искажений между несколькими устройствами Virtex.

8.7. Сложные программируемые логические устройства фирмы ALTERA

Сложные программируемые логические устройства (Complex Programmable Logic Devices – CPLD) являются расширением концепции PAL. В общем случае CPLD представляет собой микросхему, которая содержит определенное количество PAL-подобных логических блоков с программируемой матрицей межсоединений. Каждый PAL-блок имеет программируемую AND матрицу, которая питает макроячейки. Выходы этих макроячеек могут быть проведены ко входам других логических блоков в той же самой микросхеме. Большинство CPLD – электрически стираемые и репрограммируемые, их иногда относят к EPLD (стираемое программируемое логическое устройство).

Altera MAX серии 7000 является семейством высокоэффективных CMOS CPLD (комплементарный металло-оксидный полупроводник – КМОП). В противоположность Xilinx FPGA, Altera серии 7000 использует основанные на EEP ROM конфигурирующие ячейки памяти. Таким образом, одна запрограммированная конфигурация будет сохраняться пока ее не сотрут. На рисунке 8.36 показана основная архитектура серии 7000, состоящая из некоторого числа блоков логических матриц (Logic Array Blocks – LABs), входных-выходных контрольных блоков (I/O Control Block) и программируемой матрицы межсоединений (Programmable Interconnect Matrix –PIA). Каждый LAB содержит 16 макроячеек, каждая из которых имеет комбинационную логику и триггер. Каждый LAB имеет 36 входов с PIA и 16 выходов в PIA. От 8 до 16 выходов из каждого LAB могут быть направлены к входному-выходному контакту через аналогичный контрольный блок. От 8 до 16 входов с входных-выходных контактов могут быть направлены к PIA через входной-выходной контрольный блок. Вход глобальной синхронизации (GCLK) и вход общего сброса (GCLRn) связывают все макроячейки. Два выхода сигналов разрешения (OEIn и OE2n) соединяют все входные-выходные контрольные блоки.

Каждая макроячейка (рисунок 8.37) включает логическую матрицу (Logic Array), матрицу выбора конъюнктивного терма (Product Term Select Matrix), питающего OR вентиль, и программируемый регистр (Programmable Register). Вертикальные линии в логических матрицах, являющиеся общими для всех макроячеек в LAB, управляются программируемыми сигналами межсоединений от PIA и от совместно используемых расширителей логики. Конъюнктивные термы формируются в логических матрицах

точно так же, как и в PAL. Каждая макроячейка обеспечивает пять конъюнктивных термов, которые размещаются по матрицам их выбора. Эти термы могут быть использованы как входы вентиля OR, XOR, расширителя логики, для предустановки, очистки триггера, как синхровход или вход разрешения синхронизации.

Рисунок 8.36. Архитектура для устройств EPM7032, 7064 и 7096 Altera серии 7000

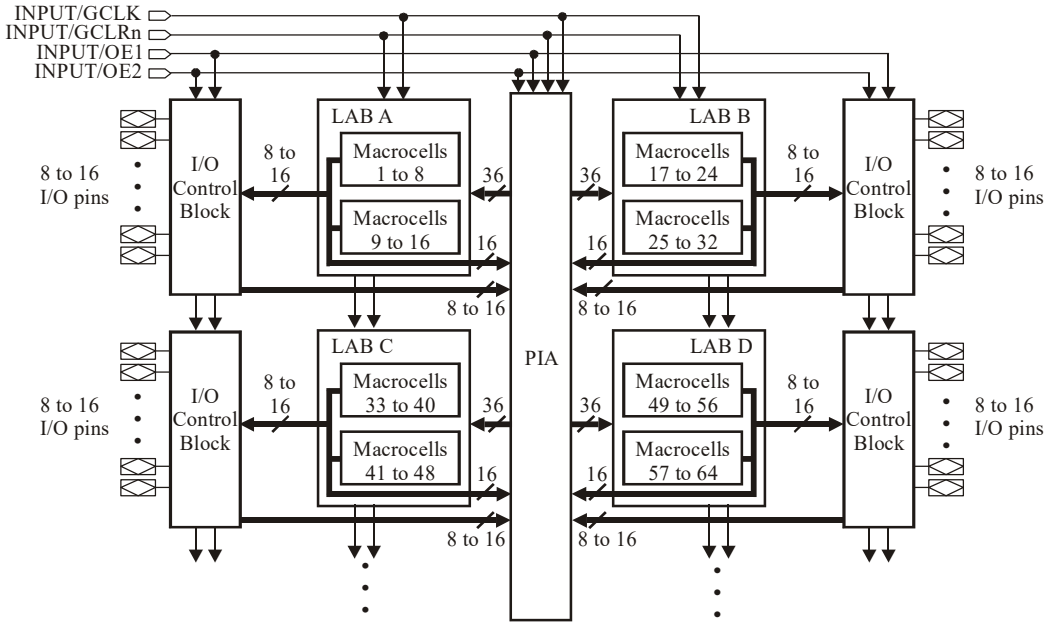
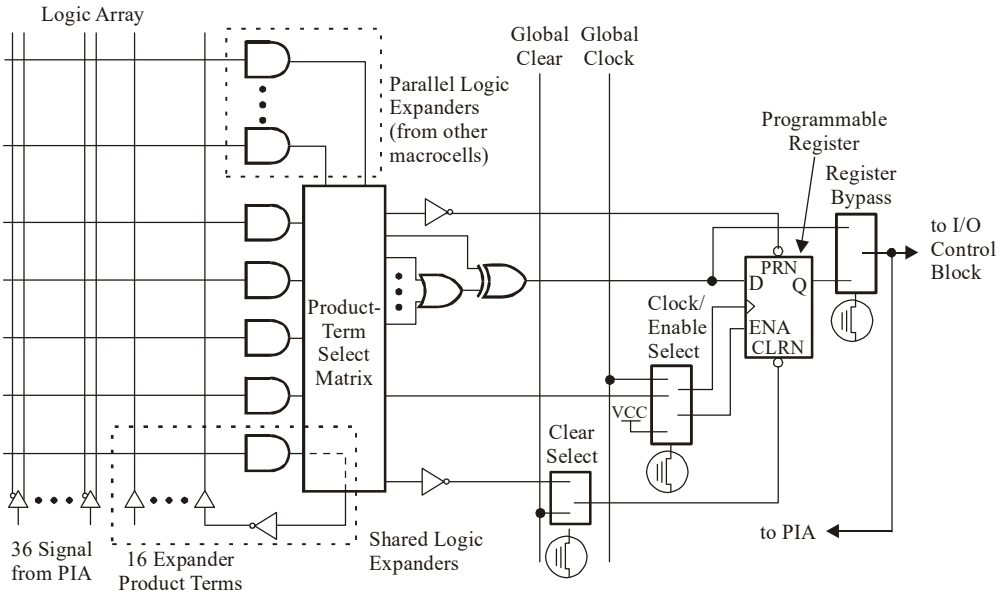


Рисунок 8.37. Макроячейка для устройств EPM7032, 7064 и 7096



Элемент памяти в каждой макроячейке представлен D-триггером с входами разрешения синхронизации, асинхронной установки и сброса. Синхровход может управляться общим входом синхронизации или конъюнктивным термом, вход разрешения синхронизации – конъюнктивным термом или иметь значение Vcc – всегда разрешенный, а вход сброс управляется общим сбросом или конъюнктивным термом. Вход установки управляется конъюнктивным термом. Вход D всегда связан с выходом вентиля XOR. Мультиплексор Register Bypass может выбирать Q выход триггера или выход с XOR. Выбранный выход идет к PIA или на входной-выходной контрольный блок. D-триггер может быть преобразован в T-триггер с помощью вентиля XOR. Поскольку характеристическое уравнение для T-триггера $Q^+ = Q + T$,

можно соединить один вход XOR вентиля с Q и использовать другой вход в качестве T. Применение T-триггера для реализации счетчика или сумматора часто требует меньше вентилей, чем использование D-триггера.

Каждая ячейка имеет пять конъюнктивных термов, а более сложные функции можно реализовать, применяя неиспользуемые конъюнктивные термы из других макроячеек. Возможны два типа расширения конъюнктивных термов – общий и параллельный расширители логики. В каждой ячейке один конъюнктивный терм может использоваться как общий расширитель (рисунок 8.38). Выбранный конъюнктивный терм подается обратно на логическую матрицу через инвертор. Инвертированный конъюнктивный терм используется как вход любого AND вентиля макроячейки. Применение общего расширителя эквивалентно трехуровневой схеме NAND-AND-OR. Выражение AND-OR с более чем пятью термами может быть разложено для использования общего расширителя из других макроячеек. Например,

P =AB+B'C+C'D+E'F+E'G+E'H+F'I+F'J=AB+ B'C + C'D + E'(F'G'H')' + F'(I'J')'

применяет общий расширитель для генерации (F'G'H')' и (I'J')'. Вентиль XOR в ячейке может использоваться для дополнения функции, так как $F = F' \oplus 1$. Иногда дополнительная функция F' требует меньше термов, чем оригинальная F. В этом случае экономично реализовать F' и дополнить ее, используя вентиль XOR.

Параллельный расширитель (рисунок 8.39) позволяет неиспользуемым конъюнктивным термам быть задействованными в соседних ячейках. Конъюнктивные термы параллельного расширителя могут соединять одну ячейку с соседней внутри двух групп – макроячейки с 8 до 1 и с 16 до 9. Когда параллельный расширитель используется без общего расширения, максимальное число конъюнктивных термов в любой логической функции равно 20, пять термов в самой макроячейке и три дополнительные группы из пяти термов, подключенных из соседних макроячеек.

На рисунке 8.40 изображен входной-выходной контрольный блок для входных-выходных контактов. Блок позволяет каждому из них быть сконфигурированным в виде входа или двунаправленного контакта. Мультиплексор OE control программируется для выбора Vcc, Gnd или одного из общих выходов сигнала разрешения. Если выбирается Vcc, выход макроячейки соединяется с входным-выходным контактом. Если выбран Gnd, буфер будет отключен и входной-выходной контакт используется как вход. Иначе, буфер может быть управляемым OE1n или OE2n сигналами.

Рисунок 8.38. Общие расширители

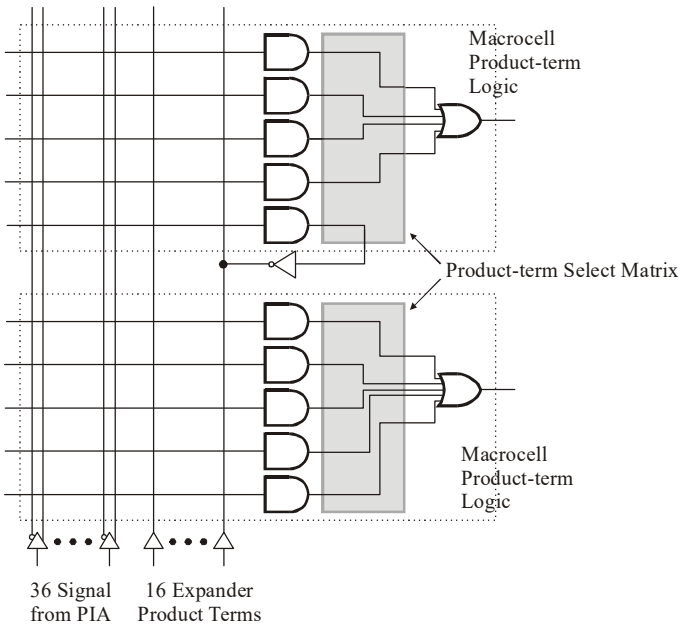


Рисунок 8.39. Параллельные расширители

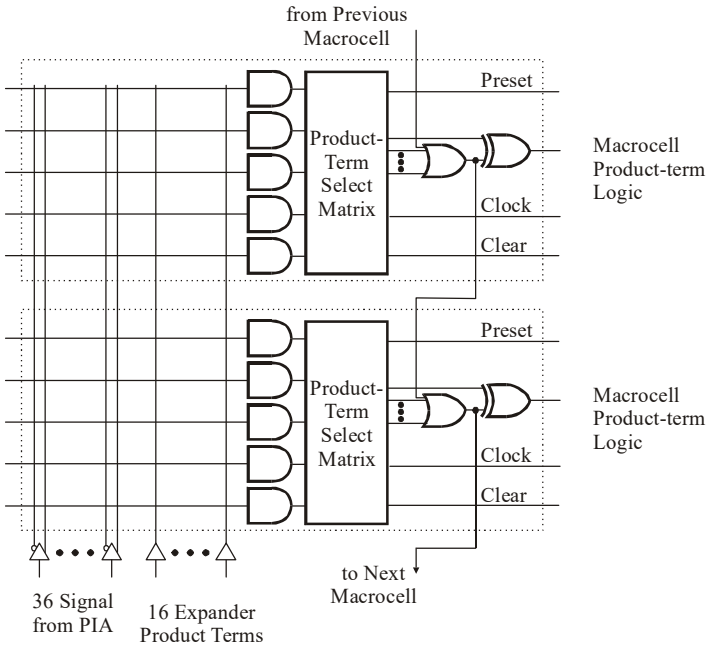
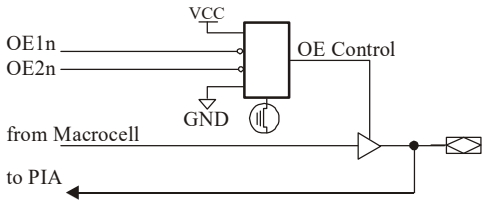


Рисунок 8.40. Входной-выходной контрольный блок для EPM 7032, 7064, и 7096



Программное обеспечение фирмы Altera может быть использовано для оптимизации и разделения проекта, для размещения его в логических ячейках и создания межсоединений между ними. Например, если применить программное обеспечение Altera для реализации двух битов полного сумматора с рисунка 4.21, используя его логические уравнения в качестве описания, то программа сначала определит, Т-триггер требует меньше вентилей, чем D-триггер, затем изменит уравнения для использования общих расширителей. Результирующие уравнения имеют следующий вид:

```
C3 = Al A2 X01 + Bl Cl X02 + Al B2 X01 + A2 B2,
   где X01 = B1+C1 и X02 = A2+B2 – выходы общих расширителей;
T2 = Ad Al B2'C1 + Ad Bl B2' X03 + Ad Bl'B2 X04 + Ad Al' B2 Cl',
   где X03 = A1+C1 и X04 = Al'+C1' – выходы общих расширителей;
Tl = Ad Bl Cl' + Ad Bl'Cl.
```

Каждое логическое уравнение имеет меньше или ровно пять термов. Таким образом, их можно реализовать на одной логической ячейке. Для всех уравнений необходимо иметь три логические ячейки и четыре общих расширителя.

Altera производит несколько других серий CPLD. Серия MAX 7000S подобна серии MAX 7000, за исключением того, что она встроенно-программируемая. Серия MAX 9000 является расширенной версией серии MAX 7000S, имеет более высокую плотность и дополнительные ресурсы маршрутизации. Серии FLEX 8000 и FLEX 10K используют ячейки с RAM конфигурирующей памятью вместо EEPROM.

8.8. Altera CPDL серии FLEX 10K

Altera FLEX 10K – семейство логических микросхем со встроенным программированием, имеющее высокую плотность и выделенную память RAM. Логика и межсоединения программируются с помощью конфигурирующих RAM-ячеек, как и в Xilinx FPGA. На рисунке 8.41 представлена блок-схема устройства FLEX 10K. Каждый ряд в логической матрице содержит несколько блоков логических матриц (LAB) и встроенный блок матриц (EAB). Каждый LAB содержит восемь логических элементов и локальный соединительный канал. EAB содержит 2048 битов памяти RAM. Блоки LAB и EAB могут соединяться через быстрые горизонтальные и вертикальные соединительные каналы, относящиеся к FastTrack Interconnect. Каждый входной-выходной элемент (IOE) допускает использование его в качестве входа, выхода или двунаправленного контакта. Каждый IOE содержит двунаправленный буфер и триггер, который может быть применен для сохранения входных и выходных данных. Одно устройство FLEX 10K содержит от 72 до 624 LAB, от 3 до 12 EAB и до 406 IOE. Это соответствует 10000 – 100000 эквивалентных вентилей при обычной реализации.

На рисунке 8.42 изображена схема блока логических матриц LAB FLEX 10K. Блок логических матриц содержит 8 логических элементов (LE). Локальные соединительные каналы имеют 22 или более входов горизонтальных соединений и 8 входов, на которые подаются значения с выходов логических элементов. Каждый LE имеет 4 входа данных с локальных соединительных каналов, а также дополнительные входы управления. Выходы LE могут быть соединены с горизонтальными и вертикальными соединительными каналами. Также есть возможность выполнить связи между горизонтальными и вертикальными соединительными каналами.

Каждый логический элемент (рисунок 8.43) содержит функциональный генератор, который может реализовывать любую функцию четырех переменных с помощью таблицы преобразования (look-up table – LUT). Каскадное соединение (cascade chain) обеспечивает связь соседних LE. Таким образом, реализуются функции более четырех переменных. Каскадное соединение используется для AND или OR конфигураций, как это показано на рисунке 8.44.

Рисунок 8.41. Блок-схема устройства FLEX 10K

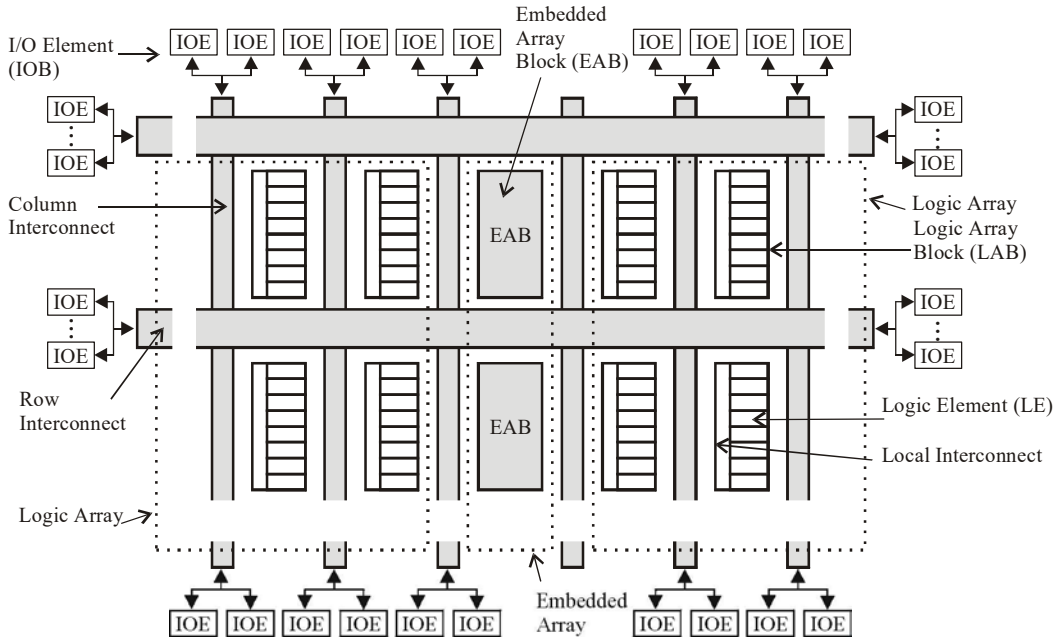
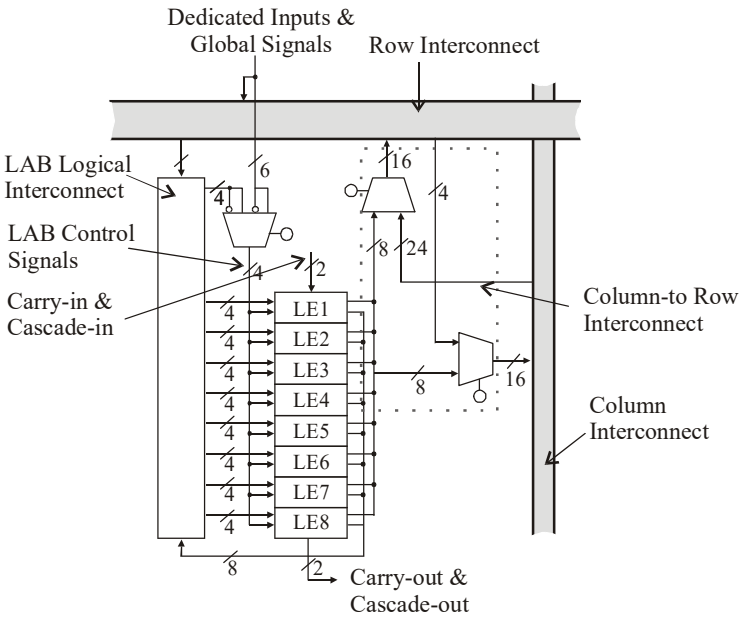


Рисунок 8.42. Блок логической матрицы FLEX 10K



В арифметическом режиме LE может реализовывать сумму и перенос для одного бита полного сумматора. Цепь переноса обеспечивает его передачу между соседними ячейками. Каждый LE содержит один D-триггер с входами разрешения синхронизации, асинхронного сброса и установки. Значения на выход LE поступают с триггера или непосредственно с комбинационной логической схемы.

Функции более чем четырех переменных требуют для реализации несколько LE. Например, для функции шести переменных Z(a, b, c, d, e, f) требуется шесть LE. Применяя теорему разложения, получаем: $Z(a,b,c,d,e,f) = a'b'Z_0(c,d,e,f) + a'bZ_1(c,d,e,f) + ab'Z_2(c,d,e,f) + abZ_3(c,d,e,f)$. Каждая функция Z_0, Z_1, Z_2 и Z_3 может быть реализована на одном LE. Выходы этих LE соединяются со входами других LE через локальное соединение. Каждая из функций 4 переменных $Y_0 = a'b'Z_0 + a'bZ_1$ и $Y_1 = ab'Z_2 + ab'Z_3$ требует еще LE. Y_0 и Y_1 могут быть соединены через OR с помощью каскадного соединения. Таким образом, здесь уже не требуется дополнительных LE.

Рисунок 8.43. Логический элемент FLEX 10K

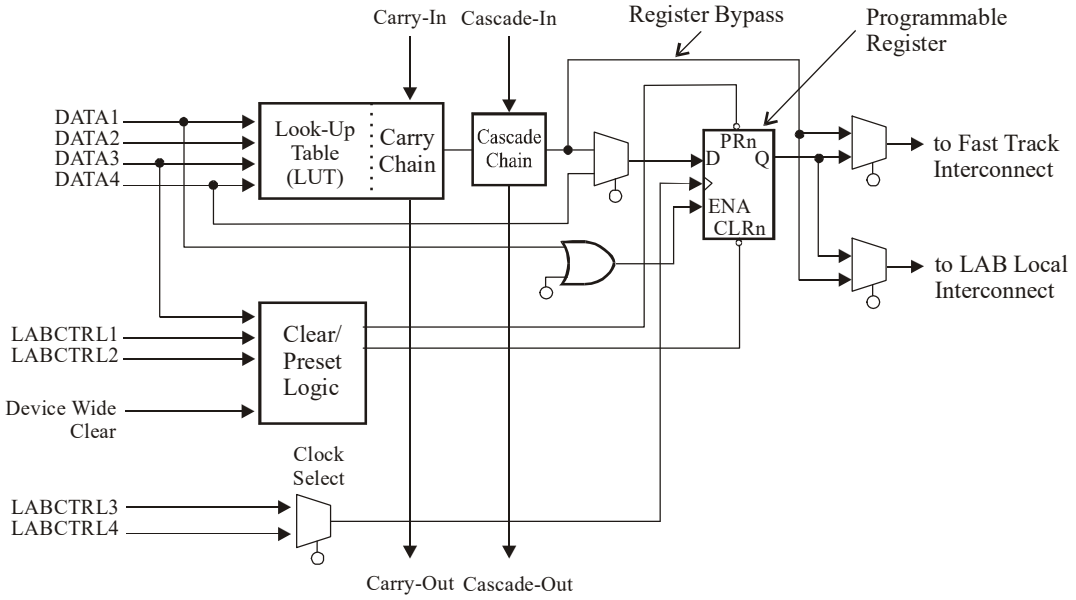


Рисунок 8.44. Операция каскадного соединения

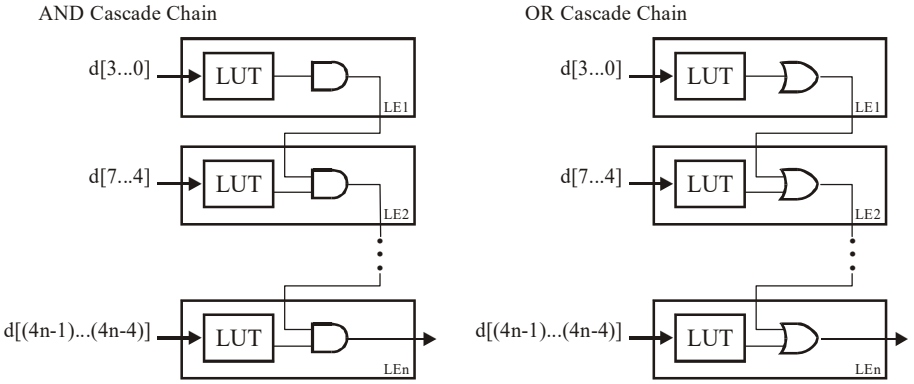
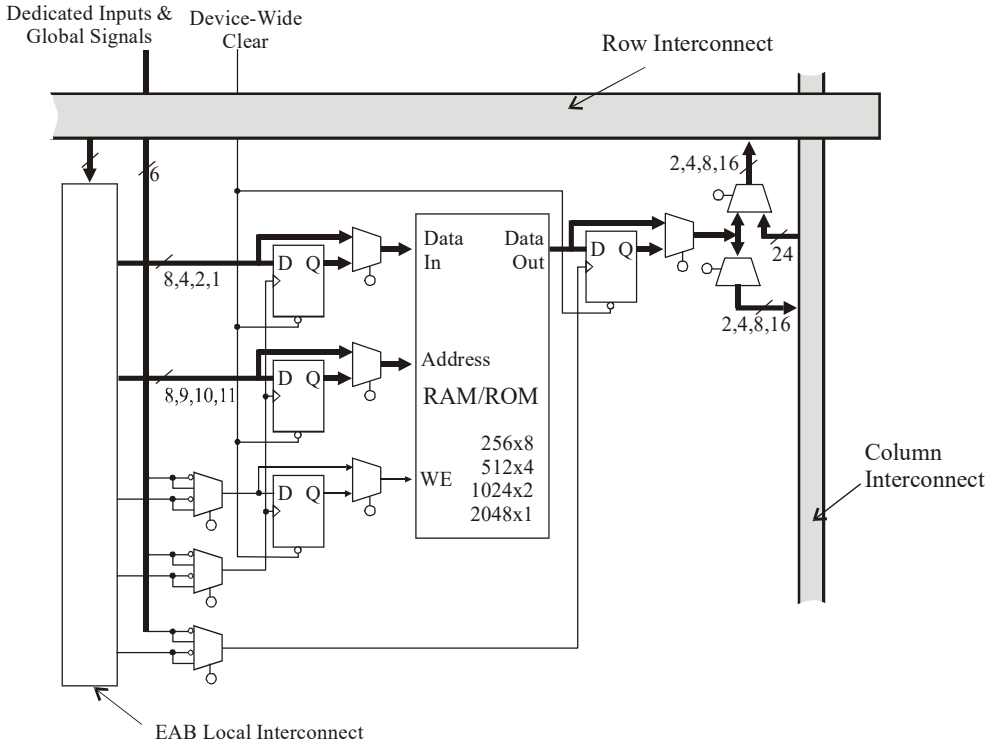


Рисунок 8.45 представляет встроенный блок матриц. Входы от горизонтальных соединений идут через EAB локальные соединения и могут быть использованы как входы данных или адресные входы для EAB. Массив внутренней памяти позволяет реализовывать RAM или ROM размерами 256 × 8, 512 × 4, 1024 × 2 или 2048 × 1. Несколько EAB способны вместе формировать большую по размеру память. Выходы данных памяти могут быть направлены через горизонтальные или вертикальные соединения. Все входы и выходы памяти соединяются с регистрами, таким образом, память становится управляемой в синхронном режиме. Иначе, регистры могут быть блокированы и память функционирует как асинхронная.

Использование CPLD таких, как серия FLEX 10K, позволяет выполнять сложные цифровые системы на одной микросхеме.

Рисунок 8.45. Встроенный блок массива FLEX 10K



Выводы. Описано несколько типов FPGA и CPLD, процедуры проектирования с использованием этих устройств. Применение соответствующих программных средств CAD облегчает декомпозицию проекта по логическим блокам, размещение их в логические матрицы и трассировки соединений между блоками. Далее будет описано использование синтезирующих инструментов, позволяющих реализовать цифровую схему из VHDL-описания устройства, подходящую для выбранной FPGA или CPLD.

8.9. Задачи

8.9.1. 8-битный сдвигающий вправо с параллельной загрузкой регистр реализован с использованием Xilinx 3000 логических ячеек. Триггеры отмечены $X_7X_6X_5X_4X_3X_2X_1X_0$. Управляющие сигналы N и S действуют следующим образом: NS = 11 – сдвиг вправо; NS = 10 – загрузка, N=0 – режим ожидания. S1 – последовательный вход для сдвига вправо.

1) Сколько логических ячеек необходимо? 2) Показать необходимые соединения для крайней справа ячейки на копии рисунка 8.4. 3) Дать F и G уравнения для этой ячейки.

8.9.2. Можно ли реализовать два JK-триггера на одной логической ячейке серии 3000? Если нет, объясните, почему. Если да, укажите необходимые соединения на копии рисунка 6.3, напишите уравнения для F и G. Отметьте входы ячейки J1, K1, J2, K2, выходы Q1 и Q2. Выделите жирными линиями все внутренние соединения.

Уравнение состояний для JK-триггера: Q=JQ'+K'Q.

8.9.3. Реализуйте двухразрядный двоичный счетчик, используя одну логическую ячейку серии 3000. Qx – последний значащий бит, Qy – старший значащий бит счетчика. Счетчик имеет асинхронный сброс AR и синхронную загрузку Ld. Счетчик работает следующим образом: En = 0 – сохранение состояния; En = 1, Ld = 1 – загрузка Qx и Qy с внешних входов U и V по переднему фронту синхроимпульса; En = 1, Ld = 0 – счет на увеличение по переднему фронту синхроимпульса.

1) Напишите уравнения следующего состояния для Qx и Qy.

2) Укажите все необходимые входы и соединения на копии рисунка 8.4. Выделите соединения жирными линиями. Отметьте входы на диаграмме FG режима (рисунок 8.4) и укажите пути соединения.

8.9.4. Спроектируйте 4-разрядный сдвигающий вправо регистр, используя 3020 FPGA. При поступлении синхроимпульса, если Ld = 1 и En = 1, в регистр загружаются значения; если Ld = 0 и En = 1 – сдвиг вправо, если En = 0 – сохранение состояния. Si и So – сдвиговые вход и выход регистра. $D_{3\text{-}0}$ и $Q_{3\text{-}0}$ – параллельные входы и выходы. Уравнение сохранения состояния для самого левого триггера: $Q_3^+ = En'Q_3 + En\,(Ld\,D_3 + Ld'\,Si)$.

1) Напишите уравнения следующего состояния для остальных трех триггеров.

2) Определите минимальное число логических ячеек серии 3000, необходимых для реализации сдвигового регистра.

3) Для самой левой ячейки укажите соединения входов и внутренних путей на копии рисунка 8.4. Также запишите F и G функции.

8.9.5. Покажите, как реализовать следующую комбинационную функцию:

$F=X_1'X_2X_3'X_6+X_2'X_3'X_4X_6'+X_2X_3'X_4'+X_2X_3X_4'X_6+X_3'X_4X_5X_6'+X_7$, используя две логические ячейки серии 3000. Покажите соединения ячеек на копии рисунка 6.3 и напишите функции F и G для обеих ячеек.

8.9.6. Реализуйте следующие уравнения состояний:

Q^+=UQV'W + U'Q'VX'Y' + UQX'Y + U'Q'V'Y + U'Q'XY+ UQVW' + U'Q'V'X, используя 3020 FPGA. Минимизируйте число необходимых логических ячеек. Нарисуйте диаграмму, представляющую соединения логических ячеек (рисунок 8.4), и напишите минимизированные уравнения для функций F и G для каждой ячейки. Отдельные вентили указывать не надо.

8.9.7. Используя унарное кодирование состояний, определите коды состояний для графа с рисунка 7.22. Получите уравнения функций переходов и выходов.

Оцените число логических блоков, необходимых для реализации графа состояний, при использовании FPGA серии 3000. Сделайте то же самое для FPGA серии 4000.

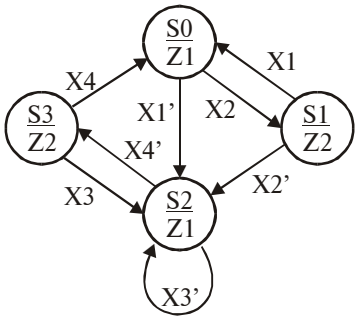
8.9.8. Сколько I/O блоков серии 3000, не использующих I/O контакты, понадобится для создания 8-разрядного последовательного на входах и параллельного на выходах регистра? Когда IN равен 1, данные поступают в регистр по синхроимпульсам и параллельные входы недоступны. Нарисуйте соединения для первого I/O блока.

8.9.9. Какое минимальное число логических блоков серии 4000 необходимо для реализации следующей функции:

X = X1'X2'X3'X4'X5 + X1 X2 X3 X4 X5 + X5'X6 X7'X8'X9 + X5'X6'X7 X8 X9'.

Если один, то отметьте необходимые соединения входов на копии рисунка 8.24. Если больше, чем один, нарисуйте блок-схему, показывая входы ячеек и соединения между ними. В любом случае укажите функции для реализации каждого F, G и H блока.

8.9.10. Для данного графа переходов получить упрощенные уравнения фунций переходов и выходов. Использовать следующее унарное кодирование состояний для триггеров $Q_0Q_1Q_2Q_3$: S0, 1000; S1, 0100; S2, 0010; S3, 0001. Сколько CLB серии 4000 потребуется для реализации этих уравнений? Рассмотреть функциональную VHDL-модель XC4000 CLB с рисунка 8.24. Для CLB, реализующего Q_0 и Q_0, описать следующие массивы: G_FUNC, FF_SEL, SR_SEL. Чему равна максимальная тактовая частота синхронизации, полагая, что максимальная задержка межсоединений между CLB равна 2 ns? Установленный период для входов D-триггера включает задержку функционального генератора 6 ns и задержку распространения от синхровхода до выхода Q 5 ns:



8.9.11. Может ли мультиплексор из 4 в 1 быть реализован с использованием только одной логической ячейки серии 4000? Если это так, покажите конфигурацию 4000 CLB (рисунок 8.24) и запишите уравнения для F, G и H. Если нет, покажите и объясните, почему этого нельзя сделать. Как это может быть сделано для мультиплексора 3 в 1?

8.9.12. Дано: Z(T, U, V, W, X,Y) = VW'X + U'V'WY + TV'WY'. Покажите, как можно реализовать Z, используя одну логическую ячейку серии 4000. Покажите входы ячейки на копии рисунка 8.24, выделите внутренние межсоединения и опишите функции F, G и H. Используйте V для входа H1. Покажите, как можно реализовать Z, используя две логических ячейки серии 3000. Нарисуйте схему, показывающую входы каждой ячейки, межсоединения между ними. Приведите F и G функции каждой ячейки.

8.9.13. Выполните следующую VHDL-модель на логической ячейке XC3000 (рисунок 8.4). Пусть доступна процедура FG_GEN(A,B,C,D,E,QX,QY,F,G), возвращающая значения F и G. Задержка распространения игнорируется. Для каждого программируемого 2 в l MUX верхний вход выбирается, когда M = 0. Для каждого программируемого 3 в l MUX F выбирается, когда MM = 00, DIN – когда MM = 01, и G – когда MM = 10. M – битовый массив, используемый для программирования логической ячейки. Укажите управляющие входы вверху F-DIN-G MUX как M(0) и Af(l). Аналогичным образом отметьте другие биты в массиве M:

```
entity XC3000 is
   port(M: in bit_vector(0 to 8);
```

```
        A, B, C, D, E, DI, EC, K, DR, GR: in bit;
        X, Y: out bit) ;
end XC3000;
```

8.9.14. Устройство умножения массивов 4 x 4 (рисунок 7.7) было реализовано на XC4003 FPGA. Без использования встроенной логики переноса разделите логику для реализации на минимальном числе логических ячеек. Обведите каждое множество компонентов, которое подходит для одной логической ячейки. Определите общее число необходимых F и G функциональных генераторов. Выполните задание только с использованием встроенной логики переноса.

8.9.15. Реализуйте мультиплексор 8 в 1, применяя устройство Altera серии 7000. Напишите логические уравнения и определите требуемое число макроячек при использовании параллельных расширителей. Повторите задание, используя только общие расширители вместо параллельных. Реализуйте MUX, применяя FLEX 10K. Сколько необходимо логических элементов?

8.9.16. Реализуйте 6-разрядный счетчик, используя устройство Altera серии 7000. Дайте логические уравнения и определите требуемое число макроячеек. Примените вентиль XOR для создания Т-триггера. Реализуйте счетчик с помощью устройства FLEX 10K. Примените цепь переноса так, чтобы каждый логический элемент мог реализовать сумму и перенос.

ГЛАВА 9
ДОПОЛНЕНИЕ VHDL

Рассмотрены дополнительные возможности VHDL, которые иллюстрируют мощность и гибкость этого языка. Описан процесс автоматического синтеза схем на основе VHDL-кода для их реализации на программируемых вентильных матрицах – PGA, сложных программируемых логических устройствах – CPLD или ASIC – специализированных интегральных микросхемах.

9.1. Атрибуты

Важным свойством языка VHDL являются атрибуты. В таблице 9.1 приведены некоторые из них, которые могут быть связаны с сигналом. Здесь S – это имя сигнала, апостроф отделяет его от имени атрибута. В VHDL событие сигнала означает его изменение. Таким образом, S'Event возвращает true, если сигнал S только что изменил значение. Если S изменится в момент времени T, тогда S'Event вернет значение true в момент времени T и false – в момент времени T+Δ. Транзакция в данном случае означает каждое новое вычисление сигнала, даже если его значение при этом не изменилось. Рассмотрим параллельный оператор A<= B and C. Если B=0, то транзакция сигнала A будет иметь место всякий раз при изменении сигнала C. Другими словами, A будет вычисляться всякий раз при изменении сигнала C. Если B=1, событие и транзакция сигнала A будут иметь место, когда C изменяет свое значение. S'Active возвращает значение true всякий раз, когда значение сигнала S будет вычислено заново, даже если оно при этом не изменится

Таблица 9.1. Атрибуты сигналов, возвращающие значение

Атрибут	Результат
S'Event	True, если событие произошло в текущем дельта-цикле, false - иначе
S'Active	True, если транзакция произошла в текущем дельта-цикле, false - иначе
S'Last_event	Время, прошедшее после последнего события сигнала S , если событие не произошло, возвращается Time'High
S'Last_value	Предыдущее значение S, т.е. значение S до последнего события
S'Lastactive	Время, прошедшее после последней транзакции сигнала S, если ее не было, возвращается Time'High
S'Driving	True, если текущий процесс содержит драйвер сигнала S (или любой элемент сложного сигнала S)
S'Driving_value	Значение, занесенное в драйвер для S в текущем процессе

Таблица 9.2 содержит формирующие сигнал атрибуты. Квадратные скобки вокруг переменной time обозначают ее необязательное использование. Если параметр времени (time) явно не указан, по умолчанию он равен дельта. Атрибут S'Delay (time) создает сигнал, идентичный S, но с задержкой на время time. Пример с рисунка 9.1 иллюстрирует использование атрибутов из таблицы 9.2. Диаграмма сигнала C_delayed5 имеет форму сигнала C с задержкой на 5 ns. Сигнал A_trans переключается всякий раз, когда изменяются B или C. Это означает транзакцию сигнала A. Первое вычисление оператора A <= B **and** C реализует транзакцию A в момент времени Δ. Таким образом, A_trans изменяется в '1' в этот момент времени. Сигнал A'Stable(time) равен true, если A не изменился на протяжении интервала времени, равного time. Значит, A_stable5 равен false на протяжении 5 ns после изменения A, true – в противном случае. Сигнал

A'quiet(time) будет равен true, если A не имел транзакций на протяжении предыдущего интервала времени time. Поэтому A_quiet5 равен false на протяжении 5 ns после транзакции сигнала A. S'Event и not S'Stable равны true, если событие произошло в текущий момент времени дельта. Тем не менее, они не могут быть полностью взаимозаменяемыми, поскольку первый формирует значение, а второй – сигнал.

Таблица 9.2. Атрибуты, формирующие сигналы

Атрибут	Результат
S'Delayed[(time)]*	Сигнал, эквивалентный сигналу S, но с задержкой на указанное время
S'Stable[(time)]*	Булевый сигнал, равный true, если сигнал S не имел событий на протяжении описанного момента времени
S'Quiet[(time)]*	Булевый сигнал, равный true, если S не имел транзакций на протяжении описанного момента времени
S'Transaction	Сигнал типа bit, который изменяет значение при каждой транзакции S

Рисунок 9.1. Пример использования атрибутов сигнала

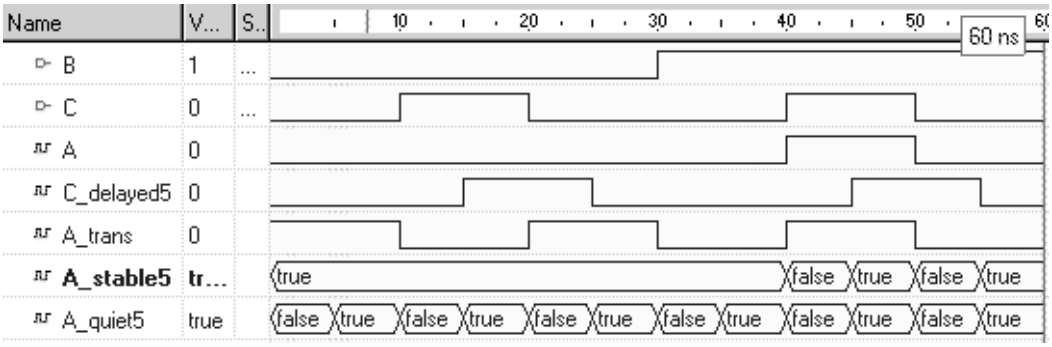
а) VHDL-код, использующий атрибуты

```
entity attr_ex is
   port (B, C : in bit);
end attr_ex;

architecture test of attr_ex is
   signal A, C_delayed5, A_trans: bit;
   signal A_stable5, A_quiet5: boolean;
begin
   A <= B and C;
   C_delayed5 <= C'delayed(5 ns);
   A_trans <= A'transaction;
   A_stable5 <= A'stable(5 ns);
   A_quiet5 <= A'quiet(5 ns);
end test;
```

б) Макрокоманды, использующиеся для моделирования

```
asim attr_ex
wave
wave B C A C_delayed5 A_trans A_stable5 A_quiet5
force b 0 0 ns, 1 30 ns
force c 0 0 ns, 1 10 ns, 0 20 ns, 1 40 ns, 0 50 ns
run 60 ns
```

в) Временные диаграммы результатов моделирования



В таблице 9.3 приведены атрибуты массивов. Здесь A – имя массива или его типа. Например, ROM1 – это двумерный массив, для которого первый индекс принадлежит диапазону 0 **to** 15, а второй – 7 **downto** 0. ROM1'Left(2) равно 7, поскольку левая граница второго индекса равна 7. Хотя ROM1 объявлен как сигнал, атрибуты массивов могут так же использоваться с переменной или константой типа массив. Для векторов или одномерных массивов параметр N равен 1 и может быть опущен. Если A есть bit_vector размерностью 2 **to** 9, то A'LEFT = 2 и A'LENGTH=8.

Таблица 9.3. Атрибуты массивов

```
Type ROM is array (0 to 15, 7 downto 0) of bit;
Signal ROM1 : ROM;
```

Атрибут	Результат	Примеры
A'Left(N)	Левая граница N-го индексного диапазона	ROM1'Left(1) =0 ROM1'Left(2) = 7
A'Right(N)	Правая граница N-го индексного диапазона	ROM1'Right(1) = 15 ROM1'Right(2) = 0
A'High(N)	Наибольшее значение N-го индексного диапазона	ROM1'High(1) = 15 ROM1'High(2) = 7
A'Low(N)	Минимальное значение N-го индексного диапазона	ROM1'Low(1) =0 ROM1'Low(2) =0
A'Range(N)	N-й индексный диапазон	ROM1'Range(1) = 0 to 15 ROM1'Range(2) =7 downto 0
A'Reverse_range(N)	N-й индексный диапазон в обратном порядке	ROM1'Reverse_range(1) = 15 **downto** 0 ROM1'Reverse_range(2) = 0 **to** 7
A'Length (N)	Размер N-го индексного диапазона	ROM1'Length (1) =16 ROM1'Length (2) =8
A'Ascending(n)	Возвращает значение истина, если N-й индексный диапазон возрастает, и false - иначе	ROM1'Ascending(1)=true ROM1'Ascending(2)=false

Атрибуты часто используются с оператором assert для обнаружения ошибок. Оператор assert проверяет выполнение некоторого условия и в зависимости от результата выводит или не выводит сообщение об ошибке. Приведенный ниже процесс, проверяющий время установки (setup_time) и время хранения (hold_time) для D-триггера, является примером такого применения атрибутов:

```
check: process
begin
   wait until rising_edge(Clk);
   assert (D'stable(setup_time))
      report ("Setup time violation")
      severity error;
   wait for hold_time;
   assert (D'stable(hold_time))
      report ("Hold time violation")
      severity error;
end process check;
```

В процессе check после поступления активного фронта синхроимпульса D вход проверяется на стабильность в течение описанного времени установки setup_time. Если условие не выполняется, выдается сообщение о нарушении времени установки

(Setup time violation). Затем, после задержки, равной времени хранения, D проверяется на стабильность на протяжении данного периода.

Процедура Addvec (см. рисунок 2.29) требует, чтобы оба вектора имели диапазон N - 1 **downto** 0 и чтобы N был включен в вызов процедуры. Используя атрибуты, можно написать подобную процедуру, не имеющую других ограничений для диапазона векторов, кроме их одинаковой длины. Во время выполнения процедуры Addvec2 (рисунок 9.2) создается временная переменная C для внутреннего переноса, инициализируемая значением Cin. Затем создаются альтернативные идентификаторы (alias) nl, n2 и S, имеющие такую же длину, как Add1, Add2 и Sum соответственно. Их диапазон будет length - 1 **downto** 0, даже если диапазон векторов Add 1, Add2 и Sum нисходящий (**downto**) или восходящий (**to**) и не содержит 0. Диапазон альтернативных идентификаторов определен для облегчения вычислений. Если входные векторы и Sum имеют разную длину, будет выдано сообщение об ошибке. Сумма и перенос вычисляются побитно в цикле. Он начинается с i = 0, и диапазон i является обратным по отношению к диапазону S. В конце вычисления выход переноса Cout получает значение временной переменной C.

Рисунок 9.2. Процедура для сложения двух битовых векторов

```
-- Эта процедура выполняет сложение двух векторов типа bit_vector
-- и переноса, возвращает сумму и перенос. Все векторы должны иметь
-- одинаковую длину.
procedure Addvec2
   (Add1,Add2: in bit_vector;
   Cin: in bit;
   signal Sum: out bit_vector;
   signal Cout: out bit) is

   variable C: bit := Cin;
   alias nl : bit_vector(Add1'length-1 downto 0) is Add1;
   alias n2 : bit_vector(Add2'length-1 downto 0) is Add2;
   alias S : bit_vector (Sum'length-1 downto 0) is Sum;
begin
   assert ((nl'length = n2'length) and (nl'length = S'length))
      report "Vector lengths must be equal!"
      severity error;
   for i in s'reverse_range loop
      S(i) <= nl(i) xor n2(i) xor C;
      C := (nl(i) and n2(i)) or (nl(i) and C) or (n2(i) and C) ;
   end loop;
   Cout <= C;
end Addvec2;
```

Здесь определена группа атрибутов скалярных типов, дающих информацию о значениях типа (таблица 9.4). Например, для описания

```
type New_Range is range 1 to 10;
```

New_Range'Ascending = TRUE означает возрастающий диапазон.

Атрибут T'Base возвращает базовый тип, может быть применим к любому типу или подтипу и используется только как приставка перед другим атрибутом. Например, Table_New'Base'Left.

Таблица 9.4. Атрибуты скалярных типов

Атрибут	Результат
T'Left	Крайнее левое значение типа T
T'Right	Крайнее правое значение типа T
TLow	Наименьшее значение типа T
THigh	Наибольшее значение типа T
T Ascending	True, если T - возрастающий диапазон, и false - иначе
T'Image(x)	Текстовое представление(тип string) значения x типа T
T'Value(s)	Значение типа T, представленное строкой s

Таблица 9.5 содержит атрибуты дискретных или физических типов, или подтипов. Пример использования атрибута из этой группы:

```
type New_Values is (Low, High, Middle);
```

New_Values'Pred(High)соответствует значению Low.

Таблица 9.5. Атрибуты дискретных или физических типов, или подтипов

Атрибут	Результат
T'Pos(s)	Номер позиции(integer) элемента s в типе T
T'Val(x)	Элемент с позиции x типа T
T'Succ(x)	Элемент с позиции x+1 типа T
T'Pred(x)	Элемент с позиции x-1 типа T
T'Leftof(x)	Элемент слева от позиции x
T'Rightof(x)	Элемент справа от позиции x

9.2. Перегрузка функций и процедур

VHDL позволяет создавать подпрограммы с одинаковым именем, различающиеся числом или типами их параметров. При вызове таких подпрограмм число или тип фактических параметров определяют подпрограмму, которая будет выполнена. Рассмотрим несколько примеров, иллюстрирующих ситуации, которые могут возникнуть при использовании перегрузки (overloading) процедур и функций. Пусть существуют три процедуры, имеющие заголовок:

```
procedure increment(a:inout integer; n:in integer:=1) is ...
procedure increment(a:inout bit_vector;n:in bit_vector:=B"1") is ...
procedure increment(a: inout bit_vector; n: in integer:=1)  is ...
```

Пусть некоторые переменные определены следующим образом:

```
variable count_int: integer:=2;
variable count_bv: bit_vector (15  downto 0):=X"0002";
```

Если написать вызов процедуры, используя переменную count_int, то будет вызвана первая процедура, даже если применяется только один параметр:

```
increment(count_int, 2);
increment(count_int);
```

Для операторов

```
increment(count_bv, X"0002");
increment(count_bv, 1);
```

первый вызывает вторую функцию, поскольку оба фактических оператора имеют тип bit_vector; второй – третью: первый оператор bit_vector, а второй integer.

Проблемы возникнут, если попытаться сделать вызов: "increment (count_bv);". Поскольку такое описание может соответствовать и второй, и третьей функции, его нельзя использовать.

9.3. Перегрузка операторов

В VHDL арифметические операции "+" и "-" определены для целых типов и не могут быть использованы с битовыми векторами. Выше были рассмотрены примеры функций и процедур, выполняющих сложение двух векторов. Используя перегрузку операторов, можно расширить определение оператора "+", чтобы применять его для сложения двух векторов. Когда компилятор встречает функцию, у которой имя представляет собой оператор, заключенный в двойные кавычки, он рассматривает ее как функцию перегрузки оператора. Пакет, представленный на рисунке 9.3, определяет две функции "+". Первая выполняет сложение двух битовых векторов и возвращает сумму типа bit-vector. Вторая функция складывает значения типа integer и bit-vector и возвращает bit-vector.

При вычислении оператора "+" компилятор автоматически проверяет тип операндов и вызывает соответствующую функцию. Пусть для вычисления оператора

```
A <= B + C + 3;
```

используется пакет bit_overload (см. рисунок 9.3). Если A, B и C имеют тип integer, реализуется арифметика для целых чисел. Если A, B и C имеют тип bit_vector, используется первая функция из пакета для сложения B и C, а затем вторая – чтобы прибавить 3 к сумме B и C. Оператор

```
A <= 3 + B + C;
```

приведет к ошибке во время компиляции, поскольку в пакете не определена функция "+", в которой первый оператор имеет тип integer, а второй – тип bit-vector.

Рисунок 9.3. VHDL-пакет с перезагружаемыми операторами для векторов типа bit_vector

```
--Данный пакет определяет две перегружаемые функции для оператора
сложения
package bit_overload is
   function "+" (Add1, Add2: bit_vector) return bit_vector;
   function "+" (Add1: bit_vector; Add2: integer) return bit_vector;
end bit_overload;

library BITLIB;
use BITLIB.bit_pack.all;

package body bit_overload is
-- Эта функция возвращает значение типа bit_vector,
-- соответствующее сумме двух операндов типа bit_vector
--Сложение выполняется побитно с внутренним переносом.
   function "+" (Add1, Add2: bit_vector) return bit_vector is
      variable sum: bit_vector (Add1'length-1 downto  0);
      variable  c: bit:='0';
      alias n1: bit_vector (Add1'length-1 downto  0) is Add1;
      alias n2: bit_vector (Add2'length-1 downto  0) is Add2;
   begin
      for i in sum'reverse_range  loop
         sum(i):=n1(i) xor n2(i) xor c;
```

```
        c:= (n1(i) and  n2(i) or (n1(i)  and  c)  or (n2(i)  and c);
      end loop;
      return  (sum);
   end "+";

-- Эта функция возвращает сумму типа bit_vector для операндов:
-- bit_vector и integer. В ней используется первая функция
-- после преобразования операнда типа integer
   function "+" (Add1: bit_vector; Add2: integer) return bit_vector
is
   begin
      return (Add1 + int2vec(Add2, Add1'length));
   end "+";
end bit_overload;
```

9.4. Многозначная логика и сигналы разрешения

В предыдущих главах в VHDL-коде в основном использовалась двухзначная битовая логика. Для описания тристабильных буферов и шин необходимо как минимум еще одно значение 'Z', соответствующее высокому импедансу. Также можно ввести четвертое значение 'X', которое будет представлять неизвестное состояние. Такая ситуация может возникать, если начальное значение сигнала неизвестно или он может иметь два конфликтующих значения '0' и '1'. Если на вход вентиля подается 'Z', значение на его выходе может рассматриваться как 'X'.

На рисунке 9.4 представлены два тристабильных буфера с объединенными выходами. Все сигналы имеют тип X01Z и могут принимать значения: 'X", '0', '1', 'Z'. Когда b=1 и d=0, то f=a; когда b=0 и d=1, то f = c; и когда b=d=0, тогда выход f принимает значение высокого импеданса Z. Ситуация b=d=1 может привести к непредсказуемым значениям на выходе. На рисунке 9.5 представлены две архитектуры. В первой используется два параллельных оператора, во второй – два процесса. В обоих случаях f управляется двумя источниками и VHDL использует функцию разрешения для определения фактических значений на выходе. Например, если a = c = d = 1 и b = 0, то f получает значение 'Z' от одного параллельного оператора и '1' – от другого. Для определения правильного значения f, равного '1', автоматически вызывается функция разрешения. Она же установит сигнал f в значение 'X', если ему будет назначено '0' и '1' одновременно.

Рисунок 9.4. Тристабильные буферы

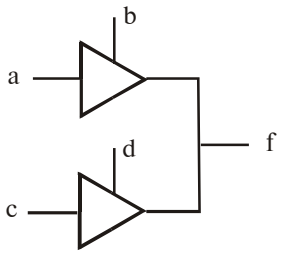


Рисунок 9.5. VHDL-код для тристабильного буфера

```
use WORK.fourpack.all;
entity t_buff_exmpl is
    port (a,b,c,d : in X01Z; -- сигналы могут иметь четыре значения
         f: out X01Z);
end t_buff_exmpl;

architecture t_buff_conc of t_buff_exmpl is
begin
   f <= a when b = '1' else 'Z';
   f <= c when d = '1' else 'Z';
end t_buff_conc;
```

```
architecture t_buff_bhv of t_buff_exmpl is
begin
   buff1: process (a,b)
   begin
      if (b='1') then f<=a;
      else f<='Z';
-- задает для выхода значение Z, когда отсутствует сигнал разрешения
      end if;
   end process buff1;

   buff2: process (c,d) begin
      if (d='1') then f<=c;
      else f<='Z';
   end if;
      end process buff2;
end t_buff_bhv;
```

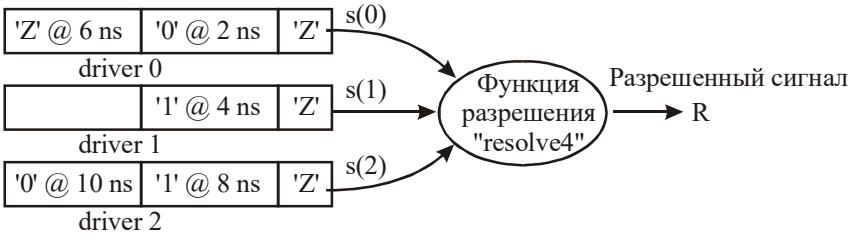
VHDL-сигналы могут быть разрешенными или неразрешенными, другими словами, могут иметь или не иметь функцию разрешения. Во всех предыдущих примерах, за исключением рисунков 9.5 и 9.6, использовались неразрешенные сигналы. Если сигналу битового типа будут назначены значения в двух параллельных операторах, компилятор сообщит об ошибке.

Пусть имеются три параллельных оператора, где R – разрешенный сигнал типа X01Z:

```
R <= transport '0' after 2 ns, 'Z' after 6 ns;
R <= transport '1' after 4 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;
```

Пусть 'Z' – это начальное значение R. Для R создаются три драйвера s(0), s(l) и s(2), как показано на рисунке 9.6. Каждый раз при изменении неразрешенного сигнала s(0), s(l) или s(2) функция разрешения определяет значение сигнала R.

Рисунок 9.6. Драйверы разрешенного сигнала



На рисунке 9.7 показано, как определяется функция разрешения для типа X01Z в пакете fourpack. Сначала определяется неразрешенный тип u_X01Z вместе с безразмерным массивом u_X01Z_vector. Затем декларируется функция разрешения, названная resolve4. Разрешенный тип X01Z является подтипом u_X01Z. Описание подтипа включает имя функции разрешения resolve4. Это означает, что при вычислении сигнала типа X01Z вызывается функция resolve4.

Следующая таблица содержит функцию разрешения для тристабильной шины:

	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'

Функция resolve4 имеет один аргумент s, который представляет собой вектор с одним или более значениями сигнала. Если длина вектора 1, функция возвращает первый и единственный элемент вектора. Иначе, возвращаемое значение вычисляется последо-

вательно по таблице, начиная со значения result = 'Z' и перебирая по одному элементу вектора s. В примере с рисунка 9.6 вектор s имеет три элемента. Функция resolve4 вызывается для вычисления R в моменты времени 0, 2, 4, 6, 8 и 10 ns. Следующая таблица содержит результат моделирования:

Time	s(0)	s(l)	s(2)	R
0	'Z'	'Z'	'Z'	'Z'
2	'0'	'Z'	'Z'	'0'
4	'0'	'1'	'Z'	'X'
6	'Z'	'1'	'Z'	'1'
8	'Z'	'1'	'1'	'1'
10	'Z'	'1'	'0'	'X'

Рисунок 9.7. Функция разрешения для X01Z логики

```
package fourpack is
   type u_x01z is ( 'X', '0', '1', 'Z');-- u_x01z неразрешенный тип
   type u_x01z_vector is array (natural range <>) of u_x01z;
   function resolve4 (s:u_x01z_vector) return u_x01z;
   subtype x01z is resolve4 u_x01z;  -- x01z - разрешенный подтип,
                       -- использующий функцию разрешения resolve4
   type x01z_vector is array (natural range <>) of x01z;
end fourpack;

package body fourpack is
   type x01z_table is array (u_x01z,u_x01z) of u_x01z;
   constant resolution_table : x01z_table := (
         ('X','X','X','X'),
         ('X','0','X','0'),
         ('X','X','1','1'),
         ('X','0','1','Z'));
   function resolve4 (s:u_x01z_vector) return u_x01z is
      variable result : u_x01z := 'Z';
   begin
      if (s'length = 1) then return s(s'low);
      else for i in s'range loop
               result := resolution_table(result,s(i));
           end loop;
      end if;
      return result;
   end resolve4;
end fourpack;
```

Для того чтобы применять X01Z логику в VHDL-коде, необходимо определить операции для этого типа. Например, операции AND и OR могут быть определены следующим образом:

AND	'X'	'0'	'1'	'Z'
'X'	'X'	'0'	'X'	'X'
'0'	'0'	'0'	'0'	'0'
'1'	'X'	'0'	'1'	'X'
'Z'	'X'	'0'	'X'	'X'

OR	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'1'	'X'
'0'	'X'	'0'	'1'	'X'
'1'	'1'	'1'	'1'	'1'
'Z'	'X'	'X'	'1'	'X'

AND и OR функции, основанные на этих таблицах, могут быть добавлены в пакет fourpack и использоваться для перегрузки операторов AND и OR.

9.5. Стандартная логика IEEE-1164

IEEE-1164 стандарт задает 9-значную логику для использования в VHDL. Как уже говорилось в главе 2, эти девять значений имеют следующий вид:

```
'U'     Неинициализированный
'X'     Сильное неизвестное
'0'     Сильный 0
'1'     Сильная 1
'Z'     Высокий импеданс
'W'     Слабое неизвестное
'L'     Слабый 0
'H'     Слабая 1
'-'     Don't Care
```

Сигналы "неизвестное", 0 и 1 в зависимости от напряжения могут быть сильными и слабыми. Если сильный и слабый сигналы используются одновременно, доминирует сильный сигнал. Например, если соединить сигналы 0' и 'H', результат будет '0'. Выход повышающего резистора (pull-up resistor) может быть представлен значением 'H'. Девятизначная логика может быть полезной для моделировании внутренних операций некоторых типов микросхем. Далее будет использоваться подмножество IEEE значений – 'X','0','1' и 'Z'.

Стандарт IEEE-1164 определяет AND, OR, NOT, XOR и другие функции для 9-значной логики. Он также описывает несколько подтипов 9-значной логики, таких как X01Z подтип, который уже использовался выше. Таблица 9.6 представляет функцию разрешения для IEEE 9-значной логики. Индекс строки описан как комментарий. Таблица функции разрешения X01Z логики, являющаяся ее подмножеством, выделена рамкой.

Таблица 9.6. Таблица функции разрешения для IEEE 9-значной логики

```
CONSTANT resolution_table : stdlogic_table := (

--      ---------------------------------------------
--  | U   X   0   1   Z   W   L   H   -       |   |
--      ---------------------------------------------
    ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
    ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
    ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
    ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
    ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- |W |
    ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
    ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
    ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | - |
);
```

Таблица 9.7 соответствует and-операции стандартной логики. Функция and с рисунка 9.8 использует эту таблицу. Она обеспечивает перегрузку оператора. Это значит, что если в выражении применяется and-оператор, компилятор автоматически вызовет соответствующую функцию в зависимости от типа операндов. Если функция and используется с операндами типа bit, то вызывается обычная функция and, но если операнды имеют тип std_logic, то используется функция and из пакета IEEE.Std_logic_1164. Перезагрузка операторов требует применения соответствующей and-функции к векторам. Когда функция and применяется к векторам типа bit_vector, она выполняется побитно, аналогично выполняется and-операция для векторов типа std_logic. Первая функция с рисунка 9.8 вычисляет функцию and по таблице, где l – левый операнд, r – правый. Вторая функция and работает с векторами типа std_logic. Альтернативные идентификаторы используются для того, чтобы гарантировать одинаковое направление индексных диапазонов операндов. Если векторы имеют разную длину, оператор assert выдает соответствующее сообщение.

Таблица 9.7. Таблица функции and для IEEE 9-значной логики

```
CONSTANT and_table : stdlogic_table := (

        --        ----------------------------------------
        --   | U   X   0   1   Z   W   L   H   -       |   |
        --        ----------------------------------------
            ('U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
            ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
            ('0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
            ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
            ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
            ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
            ('0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
            ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
            ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | - |
        );
```

Рисунок 9.8. And-функция для векторов std_logic_vector

```
function "and" ( l : std_ulogic; r : std_ulogic) return UX01 is
begin
   return (and_table(l, r) ) ;
end "and";

function "and" ( l,r : std_logic_vector ) return std_logic_vector is
   alias lv : std_logic_vector ( 1 to l'LENGTH ) is l;
   alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
   variable result : std_logic_vector ( 1 to l'LENGTH );
begin
   if ( l'LENGTH /= r'LENGTH ) then
      assert FALSE
      report "arguments of overloaded 'and' operator are not of the
same length"
      severity FAILURE;
   else
      for i in result'RANGE loop
         result(i) := and_table (lv(i), rv(i));
      end loop;
   end if;
   return result;
end "and" ;
```

Функция rising_edge стандартной логики имеет вид:

```
function rising_edge (signal s : std_ulogic) return BOOLEAN is
begin
   return (s'EVENT and (To_X01(s) = '1') and (To_X01(s'LAST_VALUE)
='0'));
end;
```

Функция To_X01 преобразует s к трем значениям 'X', '0' и '1'. Только изменение сигнала из '0' в '1' рассматривается как передний фронт.

В VHDL-коде с рисунка 4.11 для вычисления выходов ПЛМ использовалась функция PLAout. При вызове функции явно не указывались размеры входных и выходных векторов. Вместо этого они определялись в самой функции с помощью атрибутов. Функция PLAout приведена на рисунке 9.9. Input'length определяет длину входного вектора Input. PLA'length(2) определяет ширину таблицы, описывающей ПЛМ. Тогда максимальный индекс выходного вектора Output будет (PLA'length(2) -

Input'length - 1). Внешний цикл (LP1) обрабатывает построчно таблицу PLA. Поскольку диапазон входной колонки Input возрастает слева направо, необходимо гарантировать, чтобы обработка массива PLA выполнялась в таком же порядке. Диапазон колонки PLA может быть возрастающим или убывающим, поэтому параметр step может быть равен -1 или 1 соответственно. В цикле LP2 текущий ряд PLA таблицы копируется в переменную PLArow, индексный диапазон которой убывает от максимального значения до 0. Переменной PLAcol вначале присваивается значение левой границы индексного диапазона колонки таблицы PLA, которое затем увеличивается или уменьшается, в зависимости от значения параметра step. После выхода из цикла LP2 входная часть строки PLA копируется в PLAinp. Цикл LP3 проверяет входную часть каждой строки на совпадение со входами ПЛМ. При совпадении выполняется векторная операция ИЛИ между выходным вектором Output и выходной частью таблицы PLA.

Рисунок 9.9. Функция, вычисляющая выход ПЛМ

```
-- Функция формирует выходные значения ПЛМ (PLA), определяемые
-- на основе описания ПЛМ и входной последовательности.
-- Алгоритм функционирования функции:
-- 1) Начинается работа с первой строки PLAmtrx
-- 2) Определяется, является ли входная последовательность входной
-- последовательностью текущей строки.
-- 3) Если вход совпадает с текущей строкой PLA, текущие значения
-- выходов добавляются по OR к выходам ПЛМ
-- 4) Повторить шаги (2) и (3) для всех строк ПЛМ
type PLAmtrx is array (integer range <>, integer range <>)
                                                    of std_logic;
function PLAout (PLA: PLAmtrx; Input: std_logic_vector)
     return std_logic_vector is
  alias In1: std_logic_vector(Input'length-1 downto 0) is Input;
  variable match: std_logic;
  variable PLAcol, step: integer;
  variable PLArow: std_logic_vector(PLA'length(2)-1 downto 0);
  variable PLAinp: std_logic_vector(Input'length-1 downto 0);
  variable Output:
       std_logic_vector((PLA'length(2)-Input'length-1) downto 0);
begin
  Output := (others=>'0');     -- Сброс всех выходных значений в 0
  if PLA'left(2) > PLA'right(2) then step := -1; else step := 1;
  end if;
     -- Построчно сканируется таблица PLA
     LP1: for row in PLA'range loop
     match := '1';                -- Assume match for now
     PLAcol := PLA'left(2);
        -- Копируется строка таблицы PLA
     LP2: for col in PLArow'range loop
          PLArow(col) := PLA(row,PLAcol);
          PLAcol := PLAcol + step;
     end loop LP2 ;
     PLAinp:=PLArow(PLArow'high downto PLArow'high-Input'length+1);
        -- Сканирование выходных колонок
     LP3: for col in In1'range loop
             if IN1(col) /= PLAinp(col) and PLAinp(col) /= 'X' then
                  match := '0'; exit;     --  несовпадение строк
             end if;
     end loop LP3 ;

     if (match = '1') then
          Output := Output or PLArow(Output'range);
     end if;
  end loop LP1;
```

```
  return Output;
end PLAout;
```

Тип PLAmtrx и функция PLAout размещены в библиотеке MVLLIB. VHDL-код (рисунок 9.10) тестирует PLAout, использующий описание ПЛМ из таблицы 4.2. Когда функция PLAout вызывается со значением ABC =" 110", выполняются следующие вычисления:

```
INl = "110", Output = "0000", 6 > 0 таким образом step = -1
LPl: row = 0, PLAcol = 6
LP2: PLArow = "00X1010"; PLAinp = "OOX"
LP3: col = 2, INl(2) = '1', PLAinp(2) = '0', match = '0'
(LPl) row = 1, match = '1', PLAcol = 6
LP2: PLArow = "1X01100"; PLAinp = "1X0"
LP3: col = 2, INl(2) = '1', PALinp(2) = '1'
     col = 1, INl(l) = '1', PALinp(l) = 'X'
     col = 0, INl(O) = '0', PALinp(O) = 'O'  -- совпадение строк
  Output = "0000" or "1100" = "1100"
(LPl) row = 2, и т.д.
```

Рисунок 9.10. Тестирование функции PLAout

```
library ieee;
use ieee.std_logic_1164.all;

library MVLLIB;
use MVLLIB.mvl_pack.all;
entity PLAtest is
  port(ABC: in std_logic_vector(2 downto 0);
       F: out std_logic_vector(3 downto 0));
end PLAtest;

architecture PLA1 of PLAtest is
constant PLA3_2: PLAmtrx(0 to 4, 6 downto 0) :=
  ("OOX1010", "1X01100", "X1X0101", "X100010", "1X10001");
begin
  F <= PLAout (PLA3_2, ABC);
end PLA1;
```

9.6. Оператор generate

В примере с рисунка 2.4 для создания модели 4-битного сумматора (см. рисунок 2.3) использовались четыре реализации компонента full-adder. Описание карт портов для каждой из них было бы очень утомительно, если бы сумматор выполнял сложение чисел, имеющих более 8 разрядов. Оператор **generate** позволяет значительно облегчить эту задачу. Рисунок 9.11 иллюстрирует генерирование операторов для создания модели 4-разрядного сумматора с рисунка 2.4. Обозначение сигналов не изменилось, за исключением 5-битного вектора, изображающего перенос. Cin соответствует C(0), а Cout – C(4). Цикл for генерирует четыре копии полного сумматора full adder. Каждая из них со своей картой портов описывает межсоединения между сумматорами.

Рисунок 9.11. Модель сумматора Adder4, использующая генерацию операторов

```
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit;   -- Входы
        S: out bit_vector(3 downto 0); Co: out bit);   -- Выходы
end Adder4 ;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit;                             -- Входы
        Cout, Sum: out bit);                           -- Выходы
```

```
end component ;

signal C: bit_vector(4 downto 0);
begin
   C(0) <= Ci;
      --генерирование четырех копий FullAdder
   FullAdd4: for i in 0 to 3 generate
   begin
      FAx: FullAdder port map (A(i), B(i), C(i), C (i+1), S(i));
   end generate FullAdd4;
    Co <= C(4) ;
 end Structure;
```

Синтаксис оператора generate, используемый в примере, имеет вид:

```
generate_label: for identifier in range generate
[begin]
   concurrent statement(s)
end generate [generate_label] ;
```

В момент компиляции для каждого значения идентификатора из указанного диапазона генерируется множество параллельных операторов. На рисунке 9.11 используется один параллельный оператор реализации компонента. Оператор generate сам является параллельным и может быть вложенным.

Одинарный оператор generate можно использовать с конструкцией if, образуя условное генерирование группы параллельных операторов. Такой тип generate имеет форму

```
generate_label: if condition generate
[begin]
     concurrent statement(s)
end generate [generate_label] ;
```

В данном случае операторы будут генерироваться в момент выполнения компиляции, если условие (condition) будет иметь значение true.

9.7. Тип Record

Ранее был описан сложный тип данных – массив. В VHDL определен еще один сложный тип – запись (record). Запись состоит из элементов различных типов. Каждый такой элемент идентифицируется своим собственным именем. Синтаксис для определения типа запись следующий:

```
record
   identifier {,...}: subtype_indication;ы
   {...}
end record [identifier]
```

Каждое имя в списке идентификаторов определяется типом или подтипом. Следующий пример содержит определение типа запись и декларацию переменной этого типа:

```
type time_stamp is record
     seconds:integer range 0 to 59;
     minutes: integer range 0 to 59;
     hours: integer range 0 to 23;
  end record time_stamp;
variable sample_time, current_time: time_stamp;
```

Целое значение типа запись можно присвоить объекту с помощью оператора назначения, например sample_time:=current_time. Если необходимо обратиться к от-

дельному элементу, следует указать его индивидуальное имя через точку после имени переменной:

```
sample_hour:=sample_time.hour;
```

На рисунке 9.12 приведена поведенческая модель системного уровня CPU и памяти, использующая тип запись для представления команд и данных. Тип записи Instruction представляет собой информацию, которая содержится в каждой команде: код команды, адрес источника и результата, смещение. Тип записи word описывает слово, сохраняемое в памяти. Поскольку оно представляет собой код и данные, то тип включает оба возможных элемента. В отличие от большинства обычных языков, VHDL не поддерживает варианты в значениях запись. Тип word иллюстрирует, как сложные типы данных могут включать элементы, которые в свою очередь являются сложными типами данных. Сигналы в модели используются для представления адресных, информационных и управляющих линий между CPU и памятью.

В процессе CPU переменная instr_reg представляет собой командный регистр, содержащий инструкцию для выполнения. Процесс захватывает слово из памяти и записывает код команды, являющийся элементом записи, в командный регистр instr_reg := read_word.instr. Затем поле opcode типа instruction используется для определения правила выполнения команды.

Процесс memory содержит переменные, соответствующие массиву в типе word, предназначенному для сохранения в памяти. Переменная инициализируется программой и данными. Слово, представляющее команду, содержит команду и бит-вектор, который игнорируется. Аналогично, слово, представляющее данные, содержит игнорируемый код инструкции и бит-вектор с данными.

Рисунок 9.12. Фрагмент поведенческой модели CPU и памяти

```
architecture system_level of computer is

   type opcodes is (add, sub, addu, subu, jmp, breq, brne, Id,
                    st,...);
   type reg_number is range 0 to 31;
   constant r0 : reg_number := 0; constant r1 : reg_number := 1; ...

   type instruction is record
      opcode: opcodes;
      source_reg1, source_reg2, dest_reg : reg_number;
      displacement: integer;
   end record instruction;

   type word is record
        instr: instruction;
        data : bit_vector(31 downto 0);
   end record word;

   signal address: natural;
   signal read_word, write_word : word;
   signal mem_read, mem_write : bit := '0';
   signal mem_ready : bit := '0';

begin
   cpu: process is
        variable instr_reg: instruction;
        variable PC: natural;
        . . . -- другие декларации регистров файла и т.д.
     begin
        address <= PC;
        mem_read <= T;
        wait until mem_ready == '1';
```

```
        instr_reg := read_word.instr;
        mem_read <='0';
        PC := PC + 4;
        case instr_reg.opcode is      -- выполнение команды
        . . .
        end case;
      end process cpu;
    memory: process is
        type memory_array Is array (0 to 2**14-1) of word;
        variable store : memory_array :=
      (0 => ((Id, rO, r0, r2, 40), X"00000000"),
       1 => ((breq, r2, r0, r0, 5), X"00000000"),
      . . .
       40=> ((nop, r0, r0, r0, 0), X"FFFFFFFE"),
       others => ((nop, r0, r0, r0, 0), X"00000000"));
    begin
    ...
    end process memory;
  end architecture system_level;
```

При начальной инициализации объектов типа запись можно использовать именную или позиционную ассоциацию значений. Например, задать начальные значения константе, применяя позиционную ассоциацию, можно следующим образом:

```
constant midday: time_stamp:=(0, 0, 0);
```

При именной ассоциации необходимо явно указать имя каждого элемента записи:

constant midday: time_stamp:=(hours=>12, minutes=>0, second=>0);

Можно смешивать позиционную и именную ассоциации. При этом все именные элементы должны следовать за позиционными. Можно также использовать символы "|" и **others**. Например:

```
constant nop_instr: instruction :=
        (opcode => addu,
         source_reg1 | source_reg2 | dest_reg => 0,
         displacement => 0);
variable latest_event: time_stamp := (others => 0);
```

9.8. Тип доступа

Скалярные и сложные типы соответствуют одиночным и регулярным последовательностям данных. Однако могут возникнуть ситуации, когда необходимо сохранить объект, размер которого заранее неизвестен, или описать сложные взаимоотношения между отдельными структурами данных. В этом случае, вместо перечисленных выше типов, удобнее использовать данные типа доступ (access). Они аналогичны указателям (pointer) в процедурных языках программирования. Тип access используется на верхнем поведенческом уровне описания моделей и очень редко для описания моделей нижнего уровня. Применение типа access ограничивается переменными. Другими словами, только переменные, могут иметь этот тип данных.

Синтаксическое правило декларации типа доступа:

```
access subtype_indication.
```

Параметр subtype_indication может соответствовать любому типу объекта, на который будет выполняться ссылка: скалярный, сложный или другой тип доступа, – кроме разрешенного типа данных. Другими словами, не может иметь функцию разрешения.

Следующая декларация объявляет новый тип natural_ptr, который используется для создания переменной count, содержащей ссылку на объект типа natural:

```
type natural_ptr is access natural;
variable count: natural_ptr;
```

Изначально переменная count содержит значение **null**. Затем создается новый объект данных типа natural, а ссылка на него записывается в переменную count. Это выполняется с помощью функции распределения памяти (allocator) **new**. Ее синтаксис:

```
new subtype_indification | new qualified_expression.
```

Таким образом, выражение

```
count := new natural;
```

создает в памяти объект данных типа natural, инициализирует его 0 и записывает ссылку на него в переменную count. После этого переменная count будет содержать адрес объекта. Для доступа к его значению после имени переменной типа access употребляется ключевое слово all. Например, оператор

```
count.all := 10;
```

присваивает объекту, на который ссылается переменная count, значение 10. Значение такого объекта можно использовать также в выражениях:

```
if count.all = 0 then
. . .
end if;
```

Следующий оператор создает объект типа bit_vector размером в три элемента и присваивает ссылку на него переменной z:

```
z:= new BIT_VECTOR(1 to 3);
```

Начальное значение объекта можно задать в момент его инициализации. Для этого используется второй тип оператора new: **new** qualified_expression. Тогда вместо двух операторов

```
count := new natural;
count.all := 10;
```

можно записать один:

```
count := new natural'(10);.
```

Аналогичным образом создаются ссылки на массивы и записи, например для следующего типа записи описан тип доступа stimulus_ptr и переменная bus_stimulus:

```
type stimulus_record is record
   stimulus_time :time;
   stimulus_value : bit_vector(0 to 3);
end record stimulus_record;
type stimulus_ptr is access stimulusj'ecord;

variable bus_stimulus: stimulus_ptr;
```

Записанный ниже оператор создает объект типа stimulus_record и задает его начальные значения:

```
bus_stimulus := new stimulus_record'(20 ns, B"0011");
```

Переменная bus_stimulus будет содержать ссылку на объект типа stimulus_record со значениями bus_stimulus.stimulus_time =20 ns, bus_stimulus.stimulus_time ="0011".

Для правильной работы с памятью необходима процедура, обратная функции new, которая освобождала бы ее от созданных объектов. В VHDL всякий раз, при определении типа access, создается процедура deallocate, которая освобождает выделенную для объекта память и присваивает переменной доступа значение null. Например, для типа

```
type T_ptr is access T;
```

может быть использован вызов функции вида

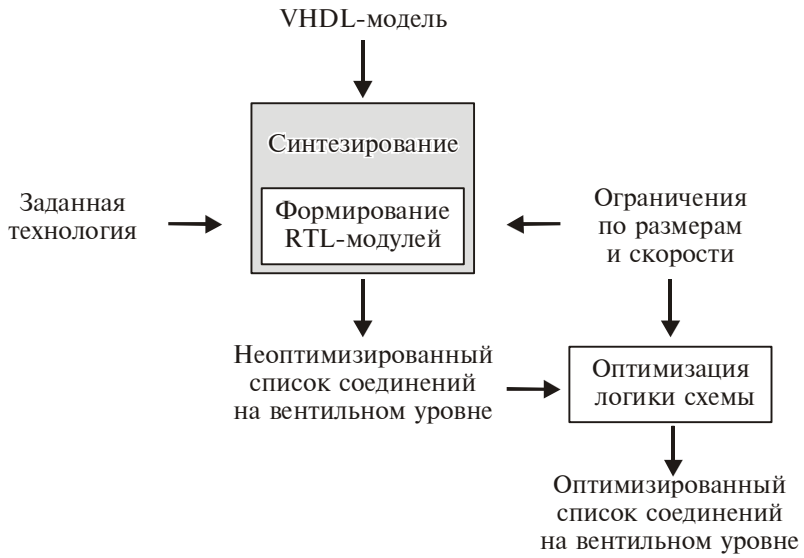
```
procedure deallocate (P: inout T_ptr);
```

Несмотря на то, что тип access очень полезен для создания моделей больших структур, таких как память или FIFO, он не поддерживается средствами синтезирования.

9.9. Синтез схем на основе VHDL-кода

Синтезом называется автоматический процесс построения списка соединений вентильного уровня из VHDL-описания схемы. Программы синтеза могут также создавать списки соединений RTL-уровня, состоящие из блоков уровня регистровых передач: триггеров, арифметико-логических устройств и мультиплексоров. В таком случае необходима вторая программа – компоновщик модулей. Она предназначена для построения или запроса из библиотеки предопределенных компонентов RTL-блоков в заданной пользователем технологии. На рисунке 9.13 представлен такой процесс.

Рисунок 9.13. Процесс синтеза



После создания списка соединений вентильного уровня выполняется оптимизация схемы с учетом заданной пользователем технологии и временных ограничений. Эти параметры могут использоваться для выбора или создания блоков RTL.

В настоящее время существуют инструменты проектирования, которые автоматически синтезируют из VHDL-кода описание моделей RTL и вентильного уровня. Примером таких программ являются FPGA Express фирмы Synopsys и Synplify фирмы Synplicity.

Не все операторы и конструкции языка VHDL могут быть использованы в моделях, предназначенных для автоматического синтеза. Поэтому даже если VHDL-код откомпилирован и правильно функционирует, это не означает, что он может быть просинтезирован. И даже если из VHDL-модели удалось получить реализацию цифрового устройства, она не всегда эффективна. В общем случае для синтеза можно

использовать только подмножество конструкций языка VHDL. Изменения кода должны быть сделаны таким образом, чтобы синтезирующее средство однозначно понимало намерение проектировщика. Дальнейшее редактирование может потребоваться для получения более оптимальной реализации цифрового устройства. Сегодня не существует единого стандарта для синтезируемого подмножества VHDL. Поэтому некоторые конструкции языка, в зависимости от версии синтезирующего инструмента, могут обрабатываться различным образом.

VHDL-сигнал может соответствовать выходу триггера (регистра) или комбинационного блока. Тип сигнала программа синтеза будет пытаться определить из контекста. Например, в параллельном операторе

```
A <= B and C;
```

сигнал A будет реализован с помощью комбинационной логики. Иначе будут обработаны расположенные в процессе последовательные операторы:

```
wait for clock'event and clock = '1';
A <= B and C;
```

В данном случае A соответствует регистру или триггеру, изменяющему состояние по переднему фронту синхроимпульса.

При использовании сигнала типа integer большое значение имеет описание его диапазона. Если он не указан, после синтеза такой сигнал может быть реализован 32-битным регистром, если его размер нельзя определить из контекста. Когда же диапазон для integer явно задан, большинство синтезаторов будет реализовывать сложение и вычитание целых чисел с помощью двоичного сумматора с соответствующим числом битов.

В общем случае, когда VHDL-сигналу присваивается значение, он будет сохранять его, пока не получит новое. Поэтому некоторые инструменты синтеза, если нет дополнительных уточнений проектировщика, реализуют такой сигнал триггером с синхронизацией по уровню. На рисунке 9.14, а, б представлен пример оператора case, который создает непреднамеренный триггер с синхронизацией по уровню, с асинхронными R и S входами сброса и установки. Здесь и далее, если это специально не оговорено, синтез выполняется с помощью программы Synplify 6.0 фирмы Synplicity. Поскольку значение b не описано для случая, когда a не равняется 0, 1 или 2, синтезатор полагает, что в этой ситуации значение b должно сохраняться в триггере. На рисунке 9.14, в представлена схема, сгенерированная программой FPGA Compiler фирмы Synopsys. В данном случае оператор case реализуется с помощью D-триггера с синхронизацией по уровню. Если a равно 0, 1 или 2, $b = a_0'$, тогда $D = a_0'$. Когда a = 3, предыдущее значение b должно сохраняться. Таким образом, G должно быть 0, когда a = 3. Другими словами, $G = (a_1 a_0)'$. В обоих случаях появления триггера можно избежать, заменив оператор null на b <= '0' .

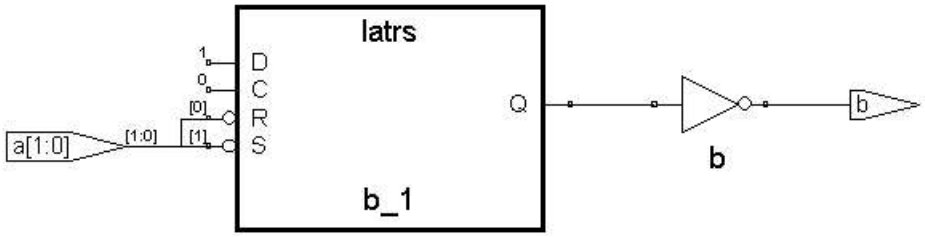
Рисунок 9.14. Пример создания непреднамеренного триггера

а) VHDL-код, синтезируемый в триггер

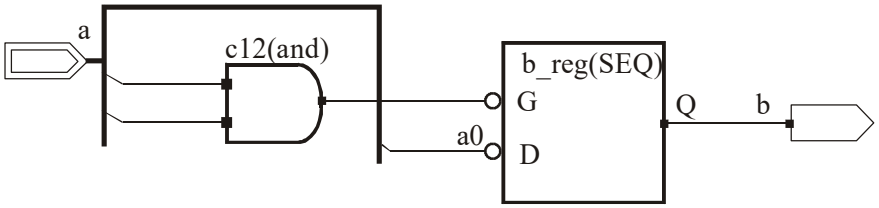
```
entity latch_example is
port(a: in integer range 0 to 3;
     b: out bit) ;
end latch_example;
architecture test1 of latch_example is
begin
  process(a)
  begin
     case a is
        when 0 => b <= '1';
        when 1 => b <= '0';
```

```
          when 2 => b <= '1';
          when others => null;
       end case;
    end process;
end test1;
```

б) Схема, синтезированная программой Synplify



в) Схема, синтезированная программой FPGA Compiler



Когда используется оператор **if**, следует описывать значения для каждой ветви. Например, запись:

```
if A = '1' then Nextstate <= 3;
end if;
```

подразумевает, что Nextstate сохранит предыдущее значение при A≠'1' и код будет моделироваться правильно. Тем не менее программа синтеза может интерпретировать такой код так: если A≠'1', то Nextstate неизвестно (X), и результат синтеза может быть некорректен. По этой причине всегда лучше включать **else** ветвь в каждый оператор **if**. Например, следующий код является точно определенным:

```
if A = '1' then Nextstate <= 3;
else Nextstate <= 2;
end if;
```

Большинство программ синтеза выполняет построчное преобразование VHDL-кода в вентили, регистры, мультиплексоры и другие основные компоненты с некоторой предварительной оптимизацией. Затем результирующий проект оптимизируется и отображается в конкретной технологии реализации, такой как PGA или CPLD. Некоторые программы синтеза, например FPGA Compiler позволяют оптимизировать проект по скорости, по площади кристалла или по некоторому компромиссу между максимальной скоростью и минимальной площадью. При оптимизации по скорости число компонентов может быть увеличено для сокращения длины пути с максимальной задержкой распространения. Например, преобразование трехуровневой вентильной схемы в двухуровневую уменьшает задержку распространения сигнала, увеличивая количество логических элементов. При оптимизации по площади число компонентов обычно сокращается, что в свою очередь уменьшает требуемую площадь кристалла. Во время начальной трансляции VHDL-кода и во время фазы оптимизации синтезирующее средство будет выбирать компоненты из своей библиотеки. Предусмотрены несколько библиотек для обеспечения возможности реализации с использованием различных технологий.

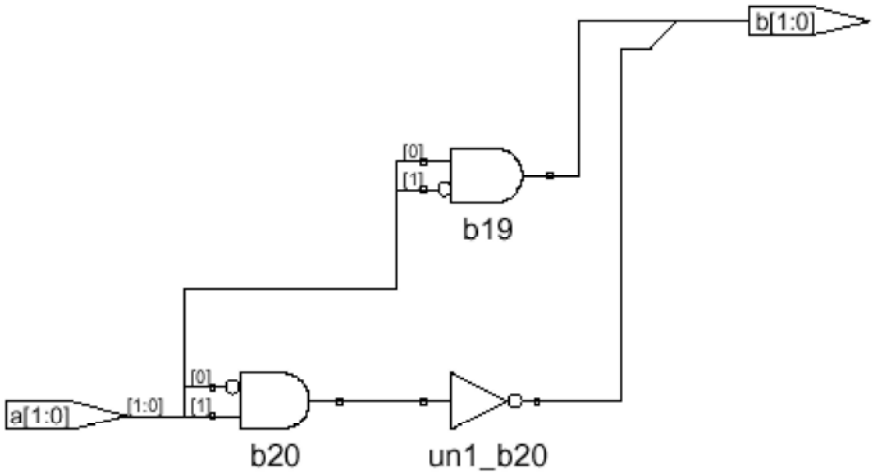
Пример на рисунке 9.15 показывает, как реализуется оператор **case** без использования триггера. Сигналы целого типа a и b представляются 2-битным двоичным числом. Оператор case преобразуется в мультиплексор. Три набора входных данных, подаваемых на него, формируются кодированием двух битов a и равны 01, 11 и 00. Но поскольку входы мультиплексора являются константами, он исключается из схемы в ходе ее оптимизации (рисунок 9.15, б). Конечные уравнения имеют вид $b_1=a_1'a_0$ и $b_0=(a_1a'_0)'$.

Рисунок 9.15. Синтезирование case оператора

a) VHDL-код для примера с case

```
entity case_example is
  port(a: in integer range 0 to 3;
       b: out integer range 0 to 3);
end case_example;

architecture testi of case_example is
begin
  process(a)
  begin
     case a is
        when 0 => b <= 1;
        when 1 => b <= 3;
        when 2 => b <= 0;
        when 3 => b <= 1;
     end case;
  end process;
end testi;
```

b) Синтезированная схема



Пример на рисунке 9.16 иллюстрирует реализацию оператора **if-then-elsif-else** с помощью мультиплексора и вентилей. Поскольку каждый сигнал C, D, E и Z имеет размер 3 бита, используется 3-битный мультиплексор. Он имеет три раздельных входа управления для выбора C, D или E. Сигнал C выбирается, если A=1; D – если A=0 или B=0; E – если A=0 и B=1.

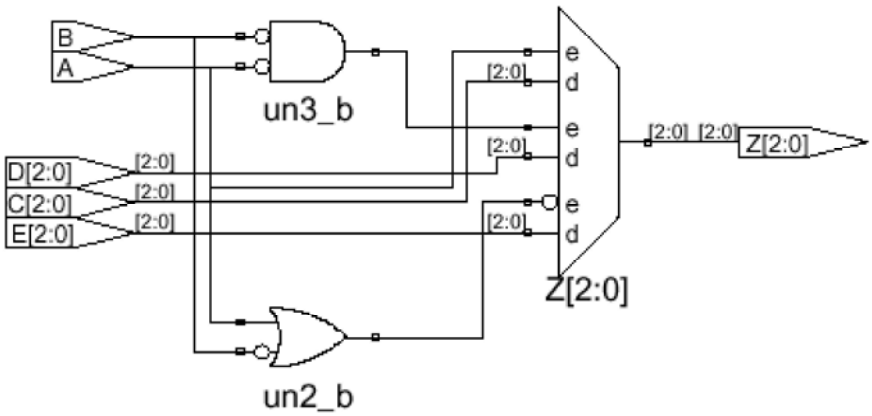
Рисунок 9.17. Синтез if оператора

a) VHDL-код

```
entity if_example is
  port(A,B: in bit;
       C,D,E: in bit_vector(2 downto 0);
       Z: out bit_vector(2 downto 0));
end if_example;
```

```
architecture testi of if_example is
begin
   process(A,B)
   begin
      if A = '1' then Z <= C;
      elsif B = '0' then Z <= D;
      else Z <= E;
      end if;
   end process;
end testi;
```

б) Синтезированная схема



Стандартные синтезируемые пакеты VHDL

Поскольку стандартный VHDL не поддерживает арифметических операций над бит-векторами, для выполнения их сложения использовались функции и процедуры, такие как Add4 и Addvec. Каждое VHDL синтезирующее средство предусматривает свой собственный функциональный пакет для таких операций. IEEE создает стандартный синтезируемый пакет, который включает функции для арифметических операций над векторами типов bit_vector и std_logic.

Пакет numeric_bit определяет арифметические операции для векторов типа bit_vector, а также два типа массивов с неограниченным диапазоном для представления беззнаковых и знаковых битовых значений:

```
type unsigned is array (natural range <>) of bit;
type signed is array (natural range <>) of bit;
```

Числа со знаками представляются в двоичном дополнительном коде. Пакет содержит перегружаемые версии VHDL-операторов: арифметических, логических, отношения, сдвига и функции преобразования типов. Пакет numeric_std определяет аналогичные операции для std_logic векторов. Беззнаковые и знаковые типы описываются как массивы std_logic векторов вместо массивов битов.

В пакетах numeric_bit и numeric_std определены перезагружаемые операторы:

- унарные**:** abs;
- арифметические: +, -, *, /, rem, mod;
- отношения: **>, <, >==, <=, =, /=**;
- логические: not, and, or, nand, nor, xor, xnor;
- сдвига: shift_left, shift_right, rotate_left, rotate_right, sll, sri, rol, ror.

Унарные операторы требуют одного знакового операнда. Арифметические, отношения и логические операторы, за исключением not, требуют два операнда: левый и правый. Для операторов арифметических и отношений допустимы следующие пары операндов: знаковый и знаковый, знаковый и целый, целый и знаковый, беззнаковый и беззнаковый, беззнаковый и натуральный, натуральный и беззнаковый (signed и signed, signed и integer, integer и signed, unsigned и unsigned, unsigned и natural, natural и unsigned). Для логических операторов левый и правый операнды должны быть знаковые либо беззнаковые.

Если левый и правый знаковые операнды имеют разную длину, более короткий операнд будет знаково расширен до необходимого размера. Для беззнаковых операндов более короткий операнд будет расширен дополнительными нулями слева от операнда. Например:

```
signed: "01101" + "1011" becomes "01101" + "11011" = "01000"
unsigned:"01101" + "1011" becomes "01101" + "01011" = "11000".
```

Когда выполняется сложение беззнаковых и знаковых операндов, последний перенос отбрасывается и переполнение игнорируется. Если перенос необходим, дополнительный бит может быть добавлен к одному из операндов. Например:

```
constant A: unsigned(3 downto 0) := "1101";
constant B: signed(3 downto 0) := "1011";
variable Sumu: unsigned(4 downto 0);
variable Sums: signed(4 downto 0);
variable Overflow: boolean;
- - - -
Sumu := '0' & A + unsigned'("0101");
          -- результат "10010"  (sum = 2, carry = 1)
Sums := B(3)& B + signed'("1101");
          -- результат "11000"  (sum = -8, carry = 1)
Overflow := Sums(4) /= Sums(3); -- Проверка переполнения
```

В этом примере запись unsigned' ("0101") является уточнением типа, которое присваивает оператор unsigned типу объект для типа bit_vector, равного "0101".

Операторы сдвига допускают знаковые или беззнаковые операнды вместе с указанием числа разрядов, на которые выполняется сдвиг. Сдвигаемый вправо беззнаковый операнд заполняется слева нулем, а знаковый – значением знака. Например,

```
A = "1001"
unsigned: shift_right(A, 2) = "0010"
signed:   shift_right(A, 2) = "1110"
```

Функция To_Integer конвертирует объекты типа Signed или Unsigned в тип Integer. Обратные функции: To_Unsigned преобразует Integer в Unsigned и To_Signed – Integer в Signed.

CAD инструменты синтеза имеют библиотеки проектов, которые включают компоненты для реализации операций, определенных в арифметических пакетах. В примере на рисунке 9.17 используется стандартный пакет std_logic_arith. Когда этот код синтезируется, результат содержит библиотеку компонентов, которые реализуют 4-битный компаратор, 4-битный двоичный сумматор с 4-битным накапливающим регистром и 4-битным счетчиком. Некоторые инструменты могут реализовывать счетчик с 4-битным сумматором со входом "0001" и затем оптимизировать результат, удаляя ненужные вентили.

Рисунок 9.17. Пример VHDL-кода для синтеза

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_arith.all;

entity examples is
   port (signal clock: in bit;
   signal A, B: in signed(3 downto 0) ;
   signal ge: out boolean;
   signal ace: inout signed(3 downto 0) := "0000";
   signal count: inout unsigned(3 downto 0) := "0000");
end examples;

architecture xl of examples is
begin
   ge <= (A >= B);       -- 4-битный компаратор
   process
   begin
      wait until clock'event and clock = '1';
      ace <= ace + B;    -- 4-битный регистр и 4-битный сумматор
      count <= count +1;   -- 4-битный счетчик
   end process;
end;
```

9.10. Пример синтеза схем цифровых устройств

Синтезируется автомат с рисунка 2.15. Его VHDL-код приведен на рисунке 2.18. Соответствующий синтезируемый код представлен на рисунке 9.18. Необходимо сделать некоторые изменения в оригинальном коде для выполнения синтеза и формирования эффективного кода-результата. Сначала нужно указать диапазон для сигналов целого типа, представляющих State и Nextstate. Для описания присваивания значений состояниям их имена типа integer заменяются константами S0, S1,..., S6. Затем описываются константы для согласования кодировки состояний, приведенных в таблице 4.1. Каждая пара if операторов для X = '0' и X = '1' заменяется оператором if-then-else. Даже если VHDL-код корректно моделируется, программа синтеза требует замены clock = '1' на clock = '1' **and** clock'event.

С применением VHDL-кода, представленного на рисунке 9.18, синтезируется схема, изображенная на рисунке 9.19. При использовании библиотеки Xilinx и выборе XC4000 FPGA для реализации устройства потребуется два CLB.

Рисунок 9.18. VHDL-код для синтеза управляющего автомата

```
entity SM1_2 is
   port(X, CLK: in bit;
      Z: out bit) ;
end SM1_2;

architecture Table of SM1_2 is
   subtype s_type is integer range 0 to 7;
   signal State, Nextstate: s_type;
   constant S0: s_type := 0;
   constant S1: s_type := 4;
   constant S2: s_type := 5;
   constant S3: s_type := 7;
   constant S4: s_type := 6;
   constant S5: s_type := 3;
   constant s6: s_type := 2;
begin
   process(State,X)
   begin
      Z <= '0'; Nextstate <= S0;
      case State is
         when S0 =>
            if X='0' then Z<='1'; Nextstate<=S1;
            else Z<='0'; Nextstate<=S2;  end if;
```

```
            when S1 =>
               if X='0' then Z<='1'; Nextstate<=S3;
               else Z<='0'; Nextstate<=S4; end if;
            when S2 =>
               if X='0' then Z<='0'; Nextstate<=S4;
               else Z<='1'; Nextstate<=S4; end if;
            when S3 =>
               if X='0' then Z<='0'; Nextstate<=S5;
               else Z<='1'; Nextstate<=S5;  end if;
            when S4 =>
               if X='0' then Z<='1'; Nextstate<=S5;
               else Z<='0'; Nextstate<=S6; end if;
            when S5 =>
               if X='0' then Z<='0'; Nextstate<=S0;
               else Z<='1'; Nextstate<=S0; end if;
            when S6 =>
               if X='0' then Z<='1'; Nextstate<=S0; end if;
            when others => null;
         end case ;
      end process;

      process(CLK)       -- State Register
         begin
            if CLK='1' and CLK'event then   -- rising edge 0f cl0ck
               State <= Nextstate;
         end if;
      end process;
   end Table;
```

9.11. Файлы и стандартный пакет TEXTIO

File – это класс объектов VHDL, предназначенный для хранения данных. VHDL обеспечивает последовательный доступ к информации в файлах с помощью таких операторов: "open", "close", "read" и "write".

Файлы часто используются при построении testbench для хранения тестовых последовательностей и результатов тестирования. В VHDL существует стандартный пакет, который используется для чтения/записи строк текста из/в файл.

Вначале файл необходимо описать, применяя синтаксическое правило:

```
file file-name: file-type [open mode] is "file-pathname";
```

Например,

```
file test_data: text open read_mode is "c:\testl\test.dat"
```

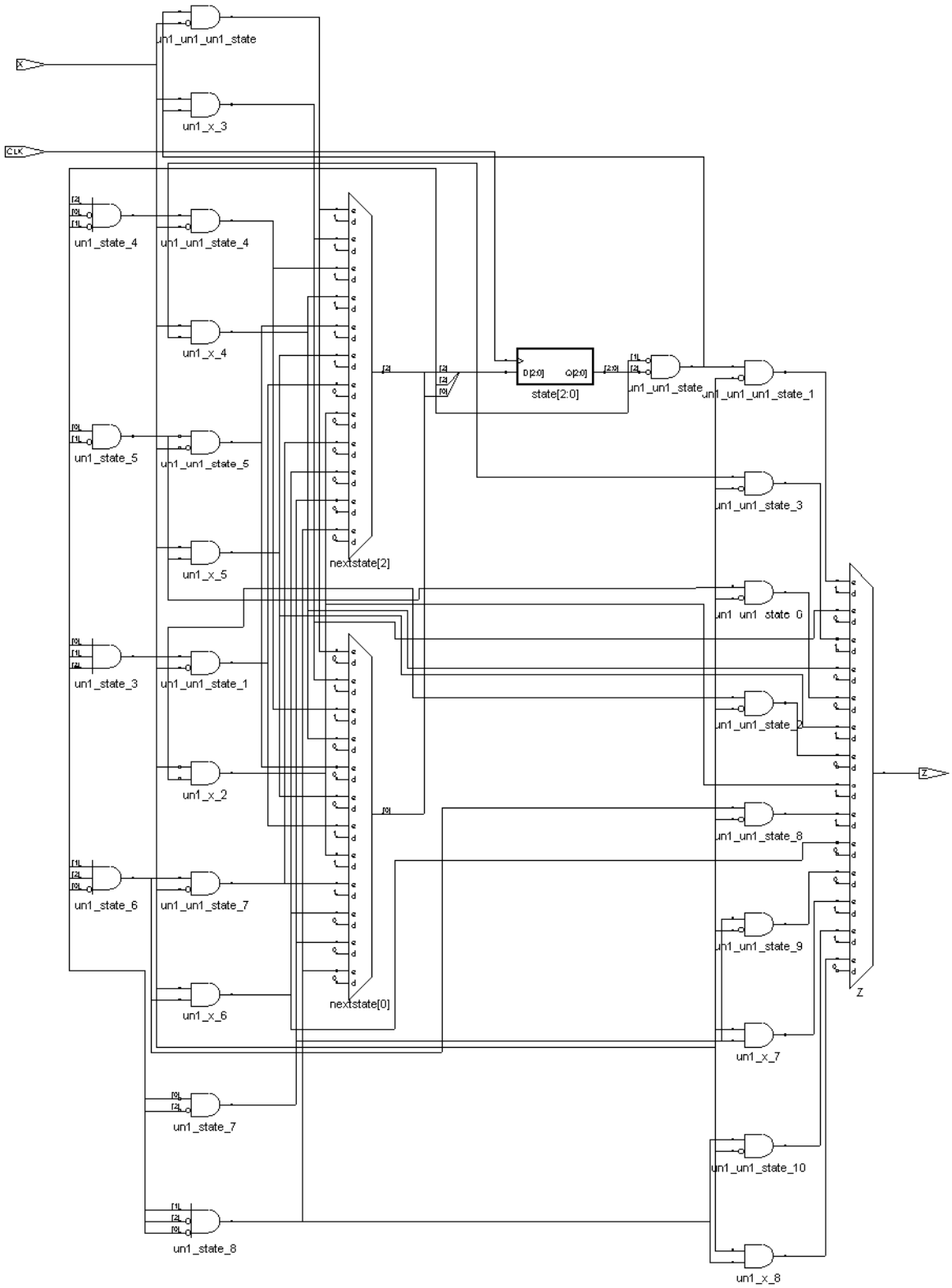
описывает текстовый файл типа text с именем test_data, который открывается в режиме read для чтения. Физически файл расположен в директории test1 на диске C.

Файл может быть открыт в режимах read_mode, write_mode или append_mode. В режиме read_mode элементы могут быть последовательно считаны с помощью процедуры чтения read. Когда файл открывается в режиме write_mode, на диске создается новый пустой файл, в который могут быть записаны данные с помощью процедуры write. Для того чтобы выполнять запись данных в существующий файл, он должен быть открыт в режиме append_mode.

Файлы состоят из объектов одного типа. Этим типом может быть integers, bit-vectors или text strings. Например, следующая декларация:

```
type bv_file is file of bit_vector;
```

Рисунок 9.19. Синтезированный управляющий автомат с рисунка 9.18



создает файловый тип bv_file, состоящий из бит-векторов. Каждый файл имеет связанную с ним неявную функцию, определяющую конец файла:

```
endfile(file_name);
```

и возвращающую TRUE, если указатель файла достиг его конца.

Поставляемый с VHDL стандартный пакет TEXTIO содержит декларации и процедуры для работы с файлами, состоящими из строк текста. В нем определен тип текстового файла с именем text:

```
type text is file of string;
```

Реализованы процедуры, выполняющие чтение и запись текстовых строк из/в файл.

Процедура readline считывает строки текста из файла и размещает их в буфере, на который ссылается указатель. В пакете TEXTIO тип указателя на буфер определен следующим образом:

```
type line is access string;
```

Переменная типа line является указателем на строку. Следующий VHDL-код:

```
variable buff: line;
...
readline (buff, test_data);
```

читает строки текста из файла test_data и размещает их в буфере, адрес которого записан в переменной buff. После занесения строк в буфер для извлечения их оттуда используется процедура read. Пакет TEXTIO имеет перегружаемые процедуры read для чтения из буфера данных типа bit, bit_vector, boolean, character, integer, real, string и time. Например, пусть bv4 – это bit_vector длиной в 4 разряда, тогда оператор

```
read(buff, bv4);
```

извлечет 4-битный вектор из буфера, занесет его в bv4 и изменит указатель, для того чтобы он указывал на следующие данные. Очередное выполнение процедуры read считает следующие по порядку 4 бита.

Вызов процедуры read может иметь одну из двух форм:

```
read (pointer, value);
read (pointer, value, good);
```

где pointer – указатель типа line, value – переменная, в которую считываются данные. Во второй форме вызова параметр good имеет тип boolean и возвращает значение TRUE, если операция чтения прошла успешно, и FALSE – в противном случае. Размер и тип параметра value определяют: какая именно процедура пакета TEXTIO будет вызвана. Например, если value – это строка из пяти символов, из буфера будут считаны пять элементов. Если value имеет целый тип integer, то процедура чтения пропустит все пробелы и будет считывать все десятичные цифры, пока не встретит символ пробела или другой нецифровой символ. Полученная таким образом строка преобразуется в число. Символы, строки и бит-векторы в файле типа text не разделяются кавычками.

Чтобы записать строку текста в файл, нужно выполнить один или несколько раз процедуру write, которая сформирует данные в буфере; затем вызвать процедуру writeline, чтобы передать строку в файл. Стандартный пакет TEXTIO предоставляет перезагружаемые процедуры write для записи в буфер данных типа bit, bit_vector, boolean, character, integer, real, string и time. Например, VHDL-код

```
variable buffw: line;
variable inti: integer;
variable bv8: bit_vector(7 downto 0);
write (buffw, inti, right, 6);
write (buffw, bv8, right, 10);
writeline (buffw, output_file);
```

преобразует inti в текстовую строку, запишет строку в line буфер, заданный указателем buffw, и откорректирует значение указателя. Текстовая строка будет выровнена вправо на поле шириной в шесть символов. Второй вызов процедуры write поместит bit_vector bv8 в буфер и также изменит значение указателя buffw. Вектор из 8 битов будет выровнен по правому краю поля шириной в 10 символов. Затем процедура writeline запишет содержимое буфера в файл output_file. Каждый вызов процедуры write имеет четыре параметра: 1) указатель на буфер типа line, 2) значение любого допустимого типа, 3) выравнивание влево или вправо (left или right), которое описывает расположение текста в выходном поле, 4) field_width, параметр типа integer, который задает размер текстового поля в символах.

В качестве примера приведена процедура, считывающая данные из файла и сохраняющая их в массиве памяти. Эта процедура может быть использована для записи команд в память модуля компьютерной системы, что позволит затем, выполняя команды, хранящиеся в памяти, осуществить тестирование системы. Данные в файле должны иметь следующий формат:

```
address N comments
bytel byte2 byte3 ... byteN comments
```

Адрес состоит из четырех десятичных цифр. N – это целое число, определяющее количество байтов в коде следующей строки. Каждый байт кода состоит из двух шестнадцатеричных цифр и отделен от следующего пробелом. После последнего байта также должен стоять пробел. Все, что следует за последним пробелом, не будет прочитано и рассматривается как комментарий. Первый байт будет сохранен в памяти по заданному адресу, второй – по следующему. Например, пусть существует файл:

```
12AC 7 (7 hex bytes follow)
AE 03 B6 91 C7 00 0C (LDX imm, LDA dir, STA ext)
005B 2 (2 hex bytes follow)
01 FC<space>
```

Когда процедура fill_memory прочитает этот файл, AE сохранится по адресу 12AC, 03 – 12AD, B6 – 12AE, 91 – 12AF.

На рисунке 9.20 приведен VHDL-код, в котором используется процедура fill_memory для чтения данных из файла и сохранения их в массиве с именем mem. Поскольку TEXTIO не содержит процедуры для чтения шестнадцатеричных чисел, процедура fill_memory считывает каждое число как строку символов, а затем преобразует ее в integer. Преобразование шестнадцатеричной цифры в целый тип выполняется с помощью таблицы соответствий. Константа lookup – это массив целых чисел, индексируемых символами в диапазоне от '0' до 'F'. В этот промежуток входят 23 ASCII символа: '0', '1', '2',..., '9',':',';', '<', '=', '>','?', '@', 'A', 'B', 'C', 'D', 'E', 'F'. Они соответствуют значениям: 0,1,2,..., 9, -1, -1, -1, -1, -1, -1, -1, 10,11,12, 13,14, 15. Вместо числа -1 можно записать любое другое целое значение, потому что эти семь специальных символов индексного диапазона никогда не будут встречаться на практике. Таким образом, lookup('2') соответствует числу 2, lookup ('C') – 12.

Для чтения строк текста, содержащих шестнадцатеричный адрес и целое значение, процедура fill_memory вызывает процедуру readline. Затем с помощью двух обращений к процедуре read считываются из буфера строка с адресом и целое число, означающее: сколько байтов нужно считывать из следующей строки. Используя таблицу соответствий, каждый символ адреса переводится из шестнадцатеричного в значение типа integer. В буфер считывается следующая строка текста. Затем в цикле извлекаются байты данных из строки. Поскольку data_s равняется 3, каждый вызов процедуры read приводит к считыванию двух шестнадцатеричных символов и пробела. Шестнадцатеричные символы преобразуются в целые значения, а затем в данные

типа std_logic_vector, которые сохраняются в памяти. Адрес увеличивается перед чтением и сохранением каждого следующего байта. По достижению конца файла процедура завершается. Другой пример использования пакета TEXTIO приведен на рисунке 10.28.

Рисунок 9.20. VHDL-код для заполнения массива памяти из файла

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;   -- CONV_STD_LOGIC_VECTOR(int, size)
use std.textio.all;

entity testfill is
end test;

architecture fillmem of testfill is
   type RAMtype is array (0 to 2047) of std_logic_vector (7 downto 0);
   signal mem: RAMtype:= (others=>(others=> '0'));

procedure fill_memory(signal mem: inout RAMType)  is
type HexTable is array(character range <>) of integer;
-- шестн. символы : 0, 1, ... A, B, C, D, E, F (только верхний регистр)
constant lookup : HexTable('0' to 'F'):= (0, 1, 2, 3, 4, 5, 6, 7, 8,
9, -1, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
-- файл открывается для чтения
file infile: text open read_mode is "meml.txt";

variable buff: line;
variable addr_s: string(4 downto 1);
variable data_s: string(3 downto 1); -- data_s(l) содержит пробел
variable addr, addr1, byte_cnt: integer;
variable data: integer range 255 downto 0;

begin
   while (not endfile(infile)) loop
      readline (infile, buff);
      read (buff, addr_s);    -- прочитать шестн. адрес
      read(buff, byte_cnt); -- прочитать число байтов для считывания
      addr:= lookup(addr_s(4))*4096 + lookup(addr_s(3))*256
         + lookup(addr_s(2))*16 + lookup(addr_s(1));
      readline (infile, buff);
      for i in 1 to byte_cnt loop
         read (buff, data_s); -- read 2 digit hex data and a space
         data:= lookup(data_s(3))*16 + lookup(data_s(2));
         mem(addr) <= CONv_STD_LOGIC_VECTOR(data, 8) ;
         addr:= addr + 1;
      end loop;
   end loop;
end fill_memory;

begin
   testbench: process begin
      fill_memory(mem);
   end process;
end fillmem;
```

9.12. Определяемые пользователем атрибуты

Атрибуты , определяемые пользователем, предоставляют ему инструмент для добавления дополнительной информации к элементам разрабатываемых моделей. Их можно использовать для описания такой физической информации как размещение стандартных ячеек; описание констант планировки: задержки линий и внутренняя расфазировка; задание параметров для синтеза: кодирование типов перечисления и указа-

ния по распределению ресурсов. В общем случае информация, не описывающая структуру или поведение моделей, может быть добавлена к их коду с помощью атрибутов.

Декларация атрибутов

Сначала необходимо описать имя и тип атрибутов с помощью декларации атрибута:

```
Декларация атрибута <= attribute identifier: type_mark;
```

Декларация просто задает идентификатор identifier для определяемого пользователем атрибута, который может получать значения указанного типа type_mark. Допускается использование любого VHDL-типа, кроме файлового, доступа или любого сложного типа, элементы которого имеют тип файловый или доступа. Примеры декларации атрибутов имеют вид

```
attribute cell_name: string;
attribute pin_number: positive;
attribute max_wire_delay: delay_length;
attribute encoding: bit_vector;
```

Тип атрибутов необязательно должен быть скалярным. Например, описать тип атрибута, определяющий местоположение ячейки, можно следующим образом:

```
type length is range 0 to integer'high
     units
        nm;
        um = 1000nm;
        mm = 1000 um;
        mil =25400 nm;
     end units length;
type coordinate is record
        x, y: length;
  end record coordinate;
attribute cell_position: coordinate;
```

Описание атрибутов

После декларации атрибута, чтобы использовать его в проекте, необходимо создать описание атрибута, задающее объекты, с которыми он будет применяться, и его значение:

```
Описание атрибута <=
attribute identifier of name_list: class is expression;
name_list <=
   ((identifier|character_literal|operator_symbol) [signature]) {,...}
   | others
   | all
class <=
   entity     | architecture     | configuration    | package
| procedure  | function         | type             | subtype
| constant   | signal           | variable         | file
| component  | label            | literal          | units
| qroup
```

Identifier в описании атрибута соответствует его имени, указанному в декларации. Затем идет name_list – список описываемых атрибутом элементов. Список класов представляет элементы, для которых используется этот атрибут. Очевидно, любой пункт проекта может быть описан с применением атрибутов. В заключение следует отметить, что реальное значение атрибута задается с помощью выражения expression, включенного в описание атрибутов:

```
attribute cell_name of std_cell: architecture is "DFF_SR_QQNN";
attribute pinjiumber of enable: signal is 14;
attribute max_wire_delay of clk: signal is 50 ps;
attribute encoding of idle_state: literal is b"0000";
attribute cell_position of thejpu: label is (540 urn, 1200 urn );
```

Для большинства классов элементов описание их атрибутов должно появляться в том же месте, где выполняется декларация этого элемента. Однако первые четыре класса, показанные в синтаксическом правиле, являются модулями проекта, размещаемыми в его библиотеке. Для них атрибуты размещаются в декларативной части самих модулей проекта. Например, для архитектуры std_cell атрибут cell_name может быть описан следующим образом:

```
architecture std__cell of flipflop is
  attribute cell_name of std_cell: architecture is "DFF_SR_QQNN";
  . . . -- другие декларации
begin
  . . .
end architecture std_cell;
```

Для пакетов декларация и описание атрибутов должны быть включены в декларацию пакета, но не в его тело. Например, для пакета model_utilities определение атрибута optimize может иметь вид

```
package model_utifities is
  attribute optimize: string;
  attribute optimize of model_utilities: package is "level_4";
  . . .
end package model_utifities;
```

Описание атрибутов для типов, подтипов и объектов данных (констант, переменных, сигналов и файлов) располагается после декларации последних. При этом декларация и описание атрибута должны располагаться в одной и той же декларативной части. Например, для разрешенного подтипа resolved_mvl:

```
type mvl is ('X','O', '1', 'Z');
type mvl_vector is array (integer range <>) of mvl;
function resolve_mwl (drivers: mvl_vector) return mvl;
subtype resolved_mvl is resolve_mvl mvl;
```

создается атрибут**:**

```
type builtin_types is (builtin_bit, builtin_mvl, builtin_integer);
attribute builtin: builtin_types;
attribute builtin of resolved_mvl: subtype is builtin_mvl;
```

Для generic-констант и портов атрибуты описываются в декларативной части интерфейса. Пусть в пакете physical_attributes содержится декларация следующих атрибутов:

```
attribute layout_ignore: boolean;
attribute pin_number: positive;
```

С их помощью generic-константы и порты описываются в виде, как это показано на рисунке 9.21.

Рисунок 9.21. Пример создания описания атрибутов для generic-констант и портов

```
library ieee;
use ieee.std_logic_1164.all;
use work.physical_attributes.all;
entity \74x138\ is
```

```
  generic (Tpd: Mime);
  port (en1, en2a_n, en2b_n: in std_logic;
        s0, s1, s2: in std_logic;
        y0, y1, y2, y3, y4, y5, y6, y7: out std_logic);
    attribute layout_ignore of Tpd: constant is true;
  attribute pin_number of s0: signal is 1;
  attribute pin_number of s1: signal is 2;
  attribute pin_number of s2: signal is 3;
  attribute pin_number of en2a_n: signal is 4;
  . . .
end entity \74x138\;
```

Несмотря на то, что в предыдущих примерах для описания элементов использовалось только одно значение для данного атрибута имени, один элемент может описываться с помощью различных атрибутов. Например, для копии компонента с меткой mult может быть задано несколько атрибутов:

```
attribute cell_allocation of mult: label is "wallace_tree_multiplier";
attribute cell_position of mult: label is (1200 um, 4500 um);
attribute cell_orientation of mult: label is down;
```

Если элемент проекта описывается с помощью определенного пользователем атрибута, обращаться к нему можно так же, как и к предопределенным атрибутам. Синтаксическое правило имеет вид

```
attribute_name <= name [signature]'identifier
```

Например:

```
std_cell'cell_name
enable'pin_number
clk'max_wire_delay
idle_state'encoding
the_fpu'cell_position
```

Хотя в VHDL допустимо обращение к таким значениям в выражениях, использование атрибутов для воздействия на поведение или структуру модели не считается хорошим стилем. Лучше применять специально предназначенные для этого конструкции языка. Атрибуты же следует использовать для объявления информации, которая предназначена другими программными средствами. Они обычно имеют документацию, описывающую необходимые атрибуты и их использование.

Выводы. Описаны новые важные свойства VHDL. Атрибуты сигналов позволяют получать время их установки, хранения и другие временные характеристики. Атрибуты, связанные с массивами, позволяют писать процедуры, независящие от направления индексации первых. Перегрузка операторов дает возможность расширить стандартные VHDL-операторы таким образом, что их можно использовать с операндами различных типов. Многозначная логика с функциями разрешения позволяет моделировать тристабильные шины и другие системы, в которых сигнал получает значения от нескольких источников. Стандарт IEEE.1164 определяет 9-значную логику, которая широко используется в VHDL. Оператор generate предоставляет эффективный способ описания итеративных структур. Пакет TEXTIO предоставляет удобный путь для передачи данных между файлами. Некоторые из описанных конструкций VHDL будут использованы для создания моделей памяти и шин.

Проведено ознакомление с автоматическим синтезом цифровых систем из VHDL-описания. Для этого VHDL-код должен быть синтезируемым – его конструкции необходимо поддерживать программами, выполняющими синтез. Различный способ написания моделей может приводить к разным результатам после синтеза при одинаковом их функционировании. Использование стандартных пакетов для знаковой и беззнаковой арифметики облегчает написание синтезируемого кода.

9.13. Задачи

9.13.1. Записать VHDL-функцию, вычисляющую скалярное произведение (dot product) двух векторов A и B по формуле $C = \sum a_i \cdot b_i$. Например:

```
A(3 downto 1)=(1, 2, 3), B(3 downto 1) = (4, 5,6), C = 3 * 6 + 2 * 5
+ 1 * 4 = 32
```

Вызов функции должен иметь вид DOT(A,B), где A и B – векторы с элементами типа integer. Не нужно заранее устанавливать верхнюю и нижнюю границы их индексных диапазонов. Использовать атрибуты для определения длины и индексного диапазона векторов. Функция должна выдавать сообщение об ошибке, если векторы A и B имеют различные индексные диапазоны.

9.13.2. Разработать VHDL-модель дешифратора адреса. Его вход – 8-разрядный битовый вектор. Он имеет индексный диапазон, равный 8 элементам, например bit_vector addr(8 to 15). Второй вход – вектор check: x01z_vector (5 downto 0). Выход дешифратора адреса Sel равен '1', если шесть старших битов адреса совпадают с битами вектора check. Например, если addr= "10001010" и check= "1000XX", тогда Sel = '1'. Сравниваются только шесть левых битов адреса addr, остальные игнорируются. Значение 'X' в векторе check рассматривается как несущественное.

9.13.3. Интерфейс VHDL-модели имеет входы A и B, выходы C и D типа bit. Изначально A и B инициализируются верхним уровнем. Когда сигнал A станет равным 0, C примет значение 1 спустя 5 ns. Если A изменится снова, C переключится через 5 ns. D изменится, если A не будет иметь транзакций в течение 5 ns.

1) Написать VHDL-архитектуру с процессом, вычисляющим выходы C и D.

2) Описать процесс, определяющий, что B не изменялся в течение 2 ns до и 1 ns после переключения A в 1. Процесс должен также выдавать сообщение об ошибке, если B изменится в 0 в течение промежутка времени, меньшего 10 ns.

9.13.4. Разработать перегружаемую функцию для "<" оператора с операндами типа bit_vector. Функция возвращает значение true, если A меньше B, и false – в противном случае. Выдает сообщение об ошибке, если векторы имеют различную длину.

9.13.5. Для шины с открытым коллектором повышающий резистор может быть промоделирован с использованием IEEE стандартной логики. Все повышающие резисторы работают как слабая '1'. Создать VHDL-модель буфера для управления шиной с открытым коллектором. Входные и выходные сигналы буфера имеют тип std_logic.

9.13.6. Для следующих параллельных VHDL-операторов:

```
A <= transport '1' after 5 ns, '0' after 10 ns, 'Z' after 15 ns;
B <= transport 'X' after 8 ns, '0' after 4 ns, '1' after 12 ns,
       'Z' after 10 ns;
C <= A after 6 ns;
C <= transport A after 5 ns;
C <= reject 3 ns inertial B after 4 ns;
```

1) нарисовать драйверы для сигналов A и B;

2) нарисовать три драйвера s_0, s_1 и s_2 для сигнала C и временные диаграммы для каждого из них;

3) написать для C список значений, вычисленных функцией разрешения. Нарисовать временные диаграммы для C.

9.13.7. Создать VHDL-модель для одного триггера из 74HC374. Восьмеричный триггер типа D с тристабильными выходами. Использовать пакет IEEE. Возможные логические значения – 'X', '0', '1' и 'Z'. Проверку соответствия техническим параметрам

времени установки, хранения и ширины импульса выполнять с помощью оператора assert. Если выход не равен 'Z', он должен быть равен 'X', когда CLK или OC имеет значение 'X', или когда 'X' был сохранен в триггере.

9.13.8. Написать VHDL-функцию, сравнивающую векторы типа std_logic_vector. Функция должна выдавать сообщение об ошибке, если значения элементов векторов отличаются от '0', '1' или '-', или если они имеют различную длину. Функция возвращает значение TRUE, если векторы равны, и FALSE – иначе. При сравнении двух векторов полагают, что '0' = '-' и '1' = '-'. Не указывать заранее индексные диапазоны векторов. Например, один вектор может иметь диапазон 1 to 7, а другой – 8 downto 0.

9.13.9. Включить в следующий интерфейс описание generic-констант Tpw_clk_h и Tpw_clk_l, которые задают минимальную длину синхроимпульса. По умолчанию, обе константы получают значение 3 ns:

```
entity flipflop is
  port (clk, d: in bit; q, q_n: out bit);
end entity  flipflop;
```

9.13.10. Написать оператор реализации компонента, который описывает использование в структурной схеме следующего элемента:

```
entity clock_generator is
  generic (period: delay_length);
  port (clk: out std_logic);
end entity  clock_generator;
```

Реальное значение константы generic – 10 ns, сигнал clk должен быть связан с сигналом master_clk.

9.13.11. Следующее незаконченное описание generic-констант используется для указания размера входных или выходных портов типа std_logic_vector. Дополнить интерфейс, задав тип портов:

```
entity adder is
  generic (data_length: positive);
  port (a, b: in ...; sum: out ...;
end entity  adder;
```

9.13.12. Система содержит восьмибитную шину, определенную как:

signal data_out: bit_vector(7 **downto** 0);

Записать оператор реализации компонента reg для создания четырехбитного управляющего регистра.

9.13.13. Создать VHDL-модель для N-битного компаратора, используя итеративную схему. В интерфейсе для описания длины входного вектора использовать generic-константу N. Выходы компаратора должны быть равны EQ = '1', если A = B, и GT= '1', если A > B. Для побитового сравнения векторов, начиная со старшего бита, использовать цикл for. Несмотря на то, что сравнение выполняется побитно, окончательные значения EQ и GT относятся к целым векторам A и B.

9.13.14. Используя generate, написать VHDL-модель для N-битного двунаправленного сдвигового регистра. По умолчанию, длина регистра = 8. Определить компонент, представляющий один бит сдвигового регистра. Компонент должен иметь порты

```
port (L, R, CLR, CLK, Pin, Lin, Rin: in bit; Q: out bit);
```

где L и R – управляющие входы. Если LR = 00 – регистр сохраняет состояние. LR = 01 – сдвиг вправо. LR = 10 – сдвиг влево. LR = 11 – параллельная загрузка данных в регистр. CLR – прямой сброс. Pin – параллельный вход. Lin – последовательный вход для сдвига влево. Rin – последовательный вход для сдвига вправо.

9.13.15. Создать VHDL-модель D-триггера с асинхронным сбросом. Для описания временных параметров используются generic-константы: t_{plh}, t_{phl}, t_{su}, t_h и t_{cmin}. Минимальный допустимый период синхронизации – t_{cmin}. При нарушении временных параметров должно выдаваться сообщение об ошибке. Написать testbench для тестирования модели. Включить тесты для каждого условия возникновения ошибки.

9.13.16. Сделать необходимые изменения VHDL-кода устройства управления светофором таким образом, чтобы его можно было синтезировать. Реализовать схему для подходящей FPGA или CPLD.

Используя ту же программу синтеза и целевое устройство, найти более эффективную реализацию устройства с меньшим числом ячеек. Можно, например, изменить кодировку состояний или написать VHDL-код, используя логические уравнения.

9.13.17. На рисунке 7.23 приведен код модели 32-битового знакового устройства деления. Сделать необходимые изменения, чтобы он стал синтезируемым. Вместо использования процедуры Addvec применить соответствующий арифметический пакет. Синтезировать код, применяя подходящую FPGA или CPLD в качестве целевого устройства.

9.13.18. Записать декларацию типа record для тестовых последовательностей, включающую трехразрядный bit_vector с тестом, задержку и восьмиразрядный bit_vector – эталонную реакцию на тест.

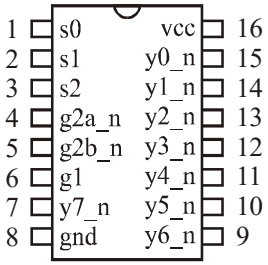
9.13.19. Разработать модель дешифратора 3 в 8 и testbench для его проверки. Использовать разработанный в задаче 9.13.18 тип record. Константа данного типа должна содержать информацию для тестирования. Она используется для инициализации входных наборов и сравнения результатов моделирования с эталонными, хранящимися в константе.

9.13.20. Дано описание физического типа, продекларированного следующим образом:

```
type capacitance is range 0 to integer'high
  units pF;
  end units capacitance;
```

Написать декларацию атрибута, представляющего емкость нагрузки, и описание атрибута для сигнала d_in, задающего значение 3 pF.

9.13.21. Написать декларацию интерфейса для дешифратора из 3 в 8. Включить в нее декларацию атрибута и описание атрибутов, задающих информацию о номерах ножек микросхемы:



ГЛАВА 10
VHDL-МОДЕЛИ ПАМЯТИ И ШИН

Рассматривается функционирование статической памяти ОЗУ (RAM). Создается VHDL-модель для описания операций и временных параметров памяти. Разбираются операции шинного интерфейса микропроцессора и разрабатывается временная VHDL-модель. Создается проект интерфейса между памятью и микропроцессорной шиной. VHDL-модели памяти и шины используются для тестирования их временных характеристик.

10.1. Статическое ОЗУ (RAM)

Ранее было описано постоянное запоминающее устройство (ПЗУ – ROM), теперь рассмотрим оперативное запоминающее устройство (ОЗУ). Английский термин random-access memory (RAM) дословно означает память с произвольной выборкой. Такой тип организации памяти характерен и для ПЗУ, но термин RAM применяется для тех устройств, которые в русскоязычной литературе называются ОЗУ. На рисунке 10.1 представлена схема ОЗУ, имеющего n адресных линий, m данных и три управляющих линии. В такой памяти можно хранить 2^n m-битных слов. Линии данных являются двунаправленными, для того чтобы сократить число контактов и, соответственно, размер корпуса микросхемы памяти. Во время операции чтения из памяти линии данных будут функционировать как выходы, записи в память – как входы. Управляющие линии функционируют следующим образом:

$\overline{CS}$ – выбор микросхемы (Chip Select). Когда на вход поступает логический 0, микросхема памяти становится активной, разрешается выполнение операций чтения и записи.

$\overline{OE}$ – разрешение чтения (Output Enable). Когда на вход поступает логический 0, разрешается передача данных из памяти на шину.

$\overline{WE}$ – разрешение записи (Write Enable). Когда на вход поступает логический 0, разрешается запись данных в память.

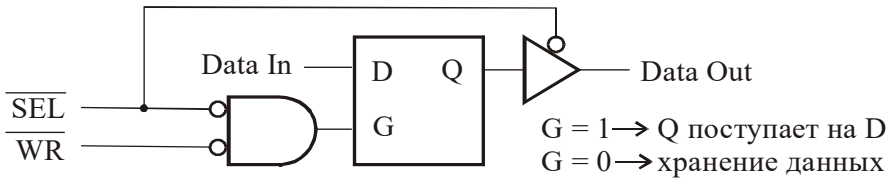
Рисунок 10.1. Структурная схема статического ОЗУ

Термин статическое ОЗУ означает, что однажды записанные данные будут сохраняться в памяти до отключения питания. В этом его отличие от динамического, требующего периодического обновления данных для предотвращения их потери. Более подробное обсуждение динамической памяти выходит за рамки данной книги.

ОЗУ состоит из дешифратора адреса и массива памяти. Адресные входы ОЗУ преобразуют данные для выбора ячейки памяти. На рисунке 10.2 представлен функциональный эквивалент ячейки статического ОЗУ, хранящей один бит информации. Ячейка содержит невидимый для пользователя D-триггер с синхронизацией по уровню, предназначенный для хранения данных. Когда $\overline{SEL}$ = 0 и $\overline{WR}$ =1, то G=0, ячейка находится в режиме чтения и Data Out = Q. Если $\overline{SEL}$ = $\overline{WR}$ =0, тогда

G =1 и данные поступают на триггер. Когда $\overline{SEL}$ или $\overline{WR}$ получают значение 1, данные сохраняются в триггере. При $\overline{SEL}$ =1 на выходе *Data Out* находится значение высокого импеданса.

Рисунок 10.2. Функциональная схема – эквивалент ячейки статического ОЗУ



Современные статические ОЗУ могут хранить до нескольких миллионов байтов данных. В качестве примера создана модель 6116 static CMOS RAM, которая может сохранять 2Кбайт данных. Но такой подход может быть применим для создания моделей памяти больших размеров. На рисунке 10.3 представлена схема памяти 6116 static RAM, хранящая 2048 8-битных слов данных. Эта память имеет 16 384 ячеек, организованных в виде матрицы 128×128. Одиннадцать линий, необходимых для адресации 2^{11} байт данных, разделены на две группы. Линии A10–A4 выбирают одну из 128 строк матрицы. Линии A3–A0 используются для адресации столбцов, поскольку память имеет 8 линий для передачи информации. Данные с выходов, перед соединением со входными/выходными (I/O) линиями, проходят через тристабильные буферы. Они позволяют пропускать сигнал только в момент чтения данных из памяти.

В таблице 10.1 истинности для ОЗУ описаны его основные операции. В режиме чтения адресные входы используются для выбора восьми ячеек памяти. Данные поступают на I/O линии по истечении времени доступа. В режиме записи данные поступают на входы триггеров, выбранных ячеек памяти, когда $\overline{WE}$ =0. Но запись в ячейки памяти не будет выполнена до тех пор, пока $\overline{WE}$ не станет 1 или будет отменен выбор микросхемы. Таблица не содержит временных параметров. Временные диаграммы и спецификации будут рассмотрены при проектировании интерфейса памяти.

Рисунок 10.3. Структурная схема памяти 6116 Static RAM

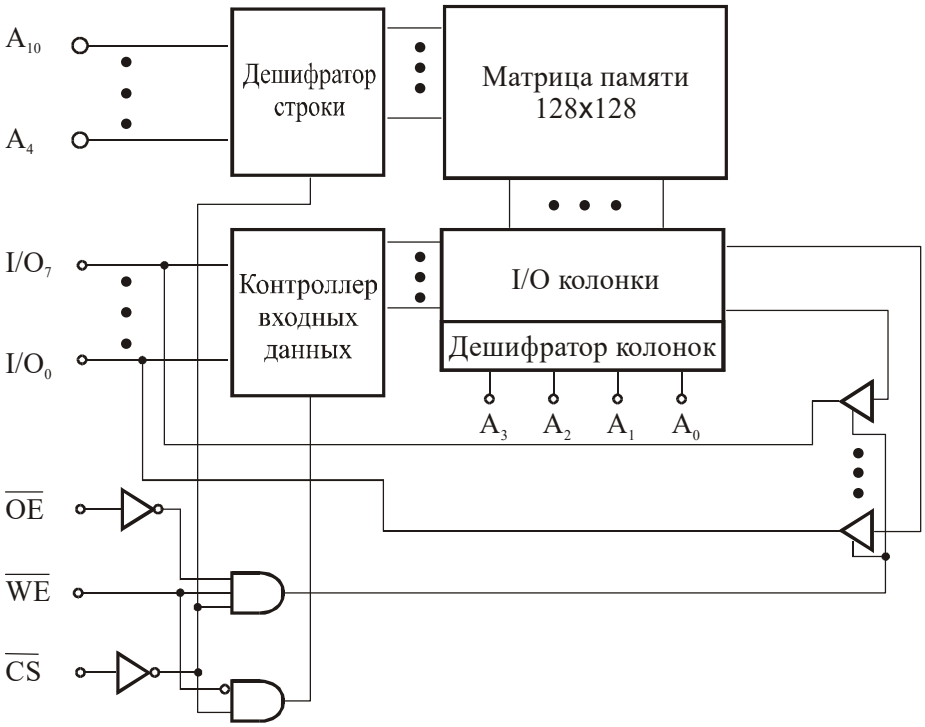


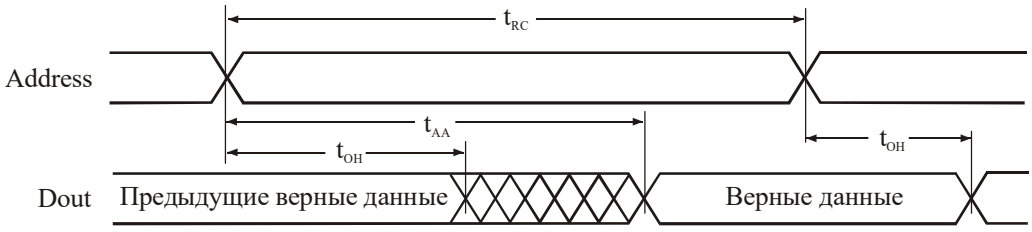
Таблица 10.1. Описание поведения статического ОЗУ

CS	OE	WE	Режим	I/O контакты
H	X	X	не выбран	high-Z
L	H	H	выходы отключены	high-Z
L	L	H	чтение	выходные данные
L	X	L	запись	входные данные

На рисунке 10.4, а представлены временные диаграммы для цикла чтения, когда $\overline{CS}$ = $\overline{OE}$=0 до изменения адреса. В этом случае, после изменения адреса в промежутке времени t_{OH}, на выходе сохраняются старые данные. Затем наступает переходный период, когда значения на выходах могут измениться. На рисунке он отмечен перекрестной штриховкой. Новые данные устанавливаются на выходах памяти после промежутка времени выборки t_{AA} (address access time). Адрес не должен изменяться в течение выполнения цикла чтения t_{RC}.

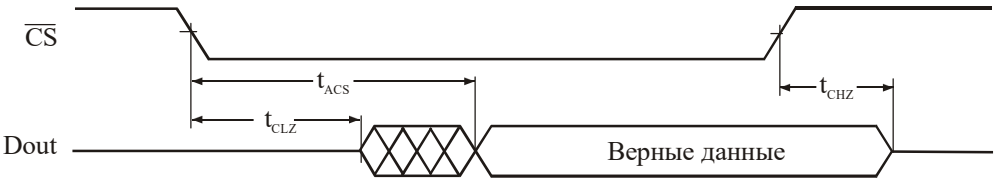
Рисунок 10.4, б содержит временные диаграммы для случая, когда $\overline{OE}$ =0 и адрес установлен до того, как значение $\overline{CS}$ перейдет в 0. Пока $\overline{CS}$ =1, Dout находится в состоянии высокого импеданса, что отмечено линией, проведенной посередине между уровнями нуля и единицы. Когда $\overline{CS}$ переключается в 0, Dout сохраняет значение высокого импеданса на протяжении времени t_{CLZ} переходного периода, в течение которого данные могут изменяться. После смены $\overline{CS}$ выходные данные становятся стабильными по истечении промежутка времени, равного t_{ACS}. Выход Dout возвращается в состояние высокого импеданса в момент времени t_{CGZ} после того, как $\overline{CS}$ станет равным 1.

Рисунок 10.4. Временные диаграммы цикла чтения



При $\overline{CS}$ =0, $\overline{OE}$=0, $\overline{WE}$ =1

а



Адрес не изменяется, $\overline{OE}$ = 0, $\overline{WE}$ =1

б

Временные параметры для циклов чтения и записи памяти CMOS static RAM приведены в таблице 10.2. Спецификация дана для памяти 6116-2 RAM, имеющей время доступа 120 ns (access time), а также для 43258A-25 RAM со временем доступа 25 ns. Прочерк обозначает несущественность данного параметра или отсутствие его описания фирмой-производителем.

Таблица 10.2. Временные параметры для двух статических ОЗУ (CMOS RAM)

Параметры	Символ	6116-2		43258A-25	
		min	max	min	max
Длина цикла чтения – Read cycle time	t_{RC}	120	–	25	–
Время доступа адреса – Address access time	t_{AA}	–	120	–	25
Время реакции на выбор микросхемы – Chip select access time	t_{ACS}	–	120	–	25
Время переключения выхода в low-Z – Chip selection to output in low-Z	t_{CLZ}	10	–	3	–
Время между поступлением сигнала OE и установкой верных значений на выходах – Output enable to output valid	t_{OE}	–	80	–	12
Время между поступлением сигнала OE и установкой выхода в low-Z – Output enable to output in low-Z	t_{OLZ}	10	–	0	–
Время между отменой выбора микросхемы и установкой выхода в high-Z – Chip deselection to output in high-Z	t_{CHZ}.	10*	40	3*	10
Время между блокировкой микросхемы и установкой на выходе значения high-Z – Chip disable to output in high-Z	t_{OHZ}	10*	40	3*	10
Время сохранения выхода после изменения адреса – Output hold from address change	t_{OH}	10	–	3	–
Длина цикла записи – Write cycle time	t_{WC}	120	–	25	–
Время между выбором микросхемы и завершением цикла записи – Chip selection to end of write	t_{CW}	70	–	15	–
Время между установкой адреса и завершением цикла записи – Address valid to end of write	t_{AW}	105	–	15	–
Время установки адреса – Address setup time	t_{AS}	0	–	0	–
Длина импульса запись – Write pulse width	t_{WP}	70	–	15	–
Время возвращения к циклу записи – Write recovery time	t_{WR}	0	–	0	–
Время между поступлением сигнала разрешения записи и установкой выхода в high-Z – Write enable to output in high-Z	t_{WHZ}	10*	35	3*	10
Время между установкой данных и завершением цикла записи – Data valid to end of write	t_{DW}	35	–	12	–
Время хранения данных после завершения цикла записи – Data hold from end of write	t_{DH}	0	–	0	–
Время разрешения активизации выхода после завершения цикла записи – Output active from end of write	t_{OW}	10	–	0	–

* Предполагаемые значения, не описанные производителем

На рисунке 10.5 представлены временные диаграммы цикла записи, когда $\overline{OE}$ =0 на протяжении всего цикла и операция записи контролируется $\overline{WE}$. Пусть $\overline{CS}$ переходит в 0 до или в тот же самый момент, когда и $\overline{WE}$ получает состояние низкого уровня. Высокий уровень $\overline{WE}$ получает до или в тот же самый момент, что и $\overline{CS}$. Перекрестная штриховка на $\overline{CS}$ обозначает интервал, в течение которого сигнал меняет значение. Адрес должен установиться (address setup time, t_{AS}) до того, как $\overline{WE}$ получит значение 0. После промежутка времени t_{WHZ} линия Data out с помощью тристабильных буферов устанавливается в значение высокого импеданса и входные данные могут подаваться на I/O линии. Данные, записываемые в память, должны быть стабильными в течение периода t_{DW} до того, как $\overline{WE}$ станет равным 1, затем не должны изменяться в течение времени сохранения t_{DH}. Адрес не может переключаться в промежутке времени t_{WR} после того, как $\overline{WE}$ получил значение 1. Когда $\overline{WE}$ переходит в состояние высокого уровня, память переключается обратно в режим чтения. После t_{OW} (min) в зоне region (a) Dout проходит переходный период и устанавливается равным данным, сохраненным в памяти. Дальнейшие изменения Dout могут произойти, если изменится адрес или $\overline{CS}$ переключится в 1. Для того чтобы избежать конфликтов шины в течение промежутка region (a), Din должен иметь значение высокого импеданса или такое же состояние как Dout.

Рисунок 10.5. Временные диаграммы цикла записи, управляемого $\overline{WE}$ ($\overline{OE}$ = 0)

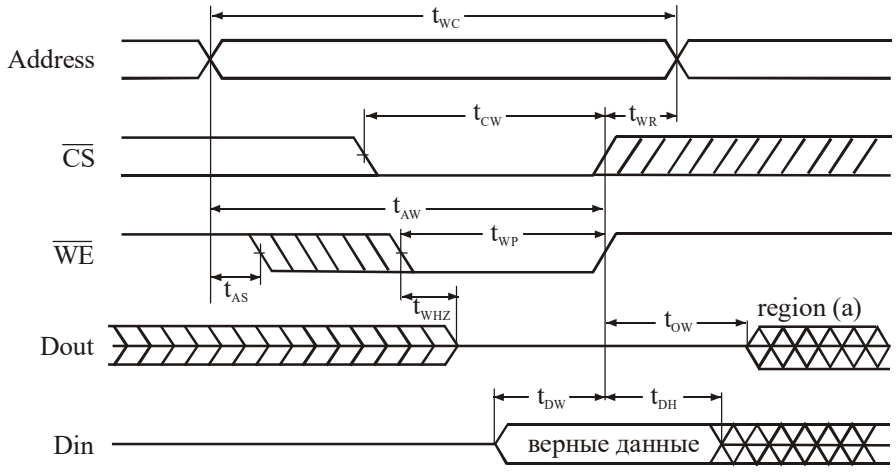
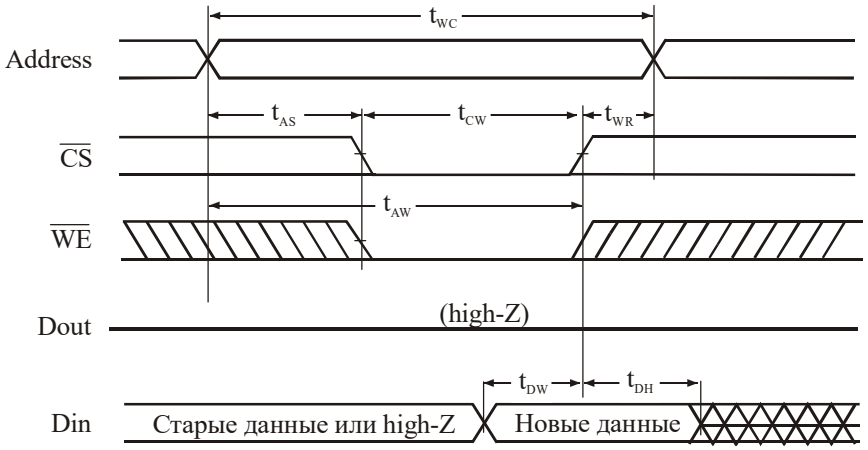


Рисунок 10.6 описывает временные диаграммы для случая, когда $\overline{OE}$=0 в течение всего цикла и запись данных в память управляется сигналом $\overline{CS}$. Пусть $\overline{WE}$ переключается в 0 до или одновременно с $\overline{CS}$, а $\overline{CS}$ возвращается в 1 до или в тот же самый момент, что и $\overline{WE}$. Адрес должен быть постоянным в течение времени установки адреса (address setup time) t_{AS}, до того как $\overline{CS}$ переключится в 0. Необходима стабильность данных, сохраняемых в памяти, на протяжении времени установки t_{DW} до переключения $\overline{CS}$ в 1, а затем еще в течение времени хранения t_{DH}. Адрес не должен изменяться в периоде t_{WR}, после того как $\overline{CS}$ перейдет в 1. Важно отметить, что данный цикл очень похож на запись, управляемую сигналом $\overline{WE}$. В обоих случаях

запись в память происходит когда $\overline{CS}$ и $\overline{WE}$ равны 0 и процесс записи завершается, когда один из этих сигналов переключается в единицу.

Рисунок 10.6. Временные диаграммы цикла записи, управляемого $\overline{CS}$ ($\overline{OE}$ = 0)



Теперь рассмотрим простую VHDL-модель памяти, не учитывающую временные параметры. Затем, чтобы получить более точную модель, будут добавлены временные характеристики и создан процесс для тестирования наиболее важных временных параметров. Пусть линия $\overline{OE}$ постоянно подключена к нулю, таким образом, ее описание может быть не указано в модели. Пусть также временные характеристики соответствуют процессу записи в память под управлением сигнала $\overline{WE}$. Для дальнейшего упрощения модели число адресных линий сокращается до 8, а размер памяти до 256 байт. Модель описывает внешнее поведение памяти и не учитывает внутреннее. В VHDL-коде идентификатор WE_b используется для представления $\overline{WE}$.

На рисунке 10.7 массив RAM-памяти представлен с помощью вектора стандартной логики RAM1. Поскольку Address набирается как битовый вектор, для адресации памяти его необходимо преобразовать в число типа integer. Процесс устанавливает входные/выходные линии в состояние высокого импеданса, если чип не выбран. Иначе, данные с входных/выходных линий сохраняются в RAM1 по переднему фронту We _ b. Если Address и We_b изменяются одновременно, будет использовано старое значение Address. Address'delayed применяется для индексации массива задержек. Оператор wait for 0 ns необходим для того, чтобы данные были сохранены в памяти до того, как они будут прочитаны обратно. Если We_b ='1', RAM находится в режиме чтения и данные из памяти поступают на входные/выхоные линии. Если We_b = '0', память в режиме записи, входные/выходные линии находятся в состоянии высокого импеданса и, таким образом, внешние данные могут быть записаны в память.

Рисунок 10.7. Упрощенная модель памяти

```
-- Упрощенная модель памяти
library IEEE;
use IEEE.std_logic_1164.all;
library BITLIB;
use BITLIB.bit_pack.all;

entity RAM6116 is
   port(Cs_b, We_b: in bit;
        Address: in bit_vector(7 downto 0);
        IO: inout std_logic_vector(7 downto 0));
```

```
  end RAM6116;
architecture simple_ram of RAM6116 is
  type RAMtype is array(0 to 255) of
                                  std_logic_vector(7 downto 0);
  signal RAM1: RAMtype:=(others=>(others=>'0'));
                       --Начальное значение всех бит '0'
begin
  process
  begin
    if Cs_b = '1' then IO <= "ZZZZZZZZ"; -- чип не выделен
    else
      if We_b'event and We_b = '1' then -- передний фронт We_b
        RAM1(vec2int(Address'delayed)) <= IO; -- запись
        wait for 0 ns;               -- ожидание обновления RAM
      end if;
      if We_b = '1' then
        IO <= RAM1(vec2int(Address));   -- чтение
      else  IO <= "ZZZZZZZZ";           -- высокий импеданс
      end if;
    end if;
    wait on We_b, Cs_b, Address;
  end process;
end simple_ram;
```

Для тестирования памяти создается модель, представленная на рисунке 10.8. Она имеет регистр адреса памяти MAR (memory address register) и регистр данных для хранения информации, прочитанной из памяти. Система считывает слово из памяти, загружает в регистр данных, увеличивает на 1 данные в регистре, сохраняет результаты обратно в память и затем увеличивает адрес памяти. Процесс продолжается, пока адрес памяти не станет равным 8. Необходимые управляющие сигналы: *ld_data* (загрузка данных с шины Data Bus), *en_data* (разрешение передачи данных на шину (Data Bus), *inc_data* (увеличение регистра данных (Data Register) и *inc_addr* (увеличение MAR). На рисунке 10.9 изображена ГСА системы. Данные из памяти загружаются во время перехода в состояние S1. Регистр данных увеличивается при переходе в состояние S2. $\overline{WE}$ является активным по низкому уровню и может быть равным нулю только в состоянии S2, во всех остальных случаях $\overline{WE}$ получает значение 1. Запись в RAM инициализируется в состоянии S2 и выполняется по переднему фронту $\overline{WE}$, который имеет место во время перехода из S2 в S3.

На рисунке 10.10 представлен VHDL-код RAM-системы. Первый процесс соответствует ГСА, второй используется для обновления регистров по переднему фронту синхросигнала. При обновлении адреса добавляется небольшая задержка, чтобы гарантировать выполнение операции записи в память до изменения адреса. Параллельные операторы используются для моделирования тристабильных буферов, разрешающих подключение выходов регистров к входным/выходным линиям.

Рисунок 10.8. Структурная схема RAM-системы

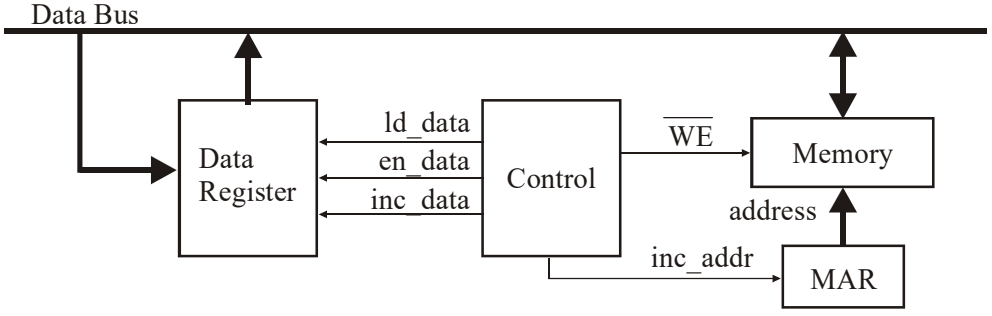


Рисунок 10.9. ГСА для RAM-системы

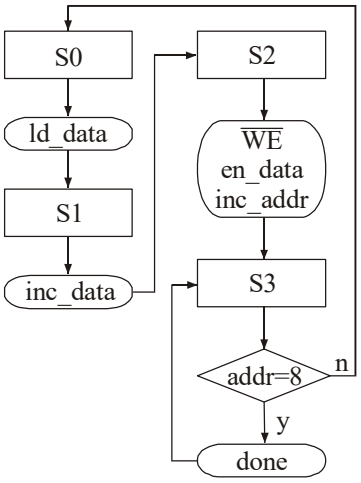


Рисунок 10.10. Устройство тестирования для упрощенной модели памяти

```
-- Устройство тестирования для упрощенной модели памяти
library ieee;
use ieee.std_logic_1164.all;
library bitlib;
use bitlib.bit_pack.all;

entity RAM6116_system is
end RAM6116_system;

architecture RAMtest of RAM6116_system is
   component RAM6116 is
      port(Cs_b, We_b: in bit;
           Address: in bit_vector(7 downto 0);
           IO: inout std_logic_vector(7 downto 0));
   end component RAM6116;
   signal state, next_state: integer range 0 to 3;
   signal inc_adrs, inc_data, ld_data, en_data, Cs_b, elk, done:
bit;
   signal We_b: bit := '1';  -- инициализация режима read
   signal Data: bit_vector(7 downto 0);    -- регистр данных
   signal Address: bit_vector(7 downto 0); -- регистр адреса
   signal IO: std_logic_vector(7 downto 0);  -- вх/вых шина
begin
   RAM1: RAM6116 port map(Cs_b, We_b, Address, IO);
   control: process(state, Address)
   begin
   -- инициализация всех управляющих сигналов (RAM всегда выбрана)
      ld_data<='0'; inc_data<='0'; inc_adrs<='0'; en_data <='0';
      done <= '0'; We_b <='1'; Cs_b <= '0';
      -- начало ГСА
      case (state) is
         when 0 => ld_data <='1'; next_state <= 1;
         when 1 => inc_data <='1'; next_state <= 2;
         when 2 => We_b <= '0'; en_data <= '1'; inc_adrs <= '1';
                   next_state <= 2;
         when 3 => if (Address = "00001000") then done <= '1';
                   else next_state <= 0;
                end if;
      end case ;
   end process control;

   -- Процесс выполняется по переднему фронту синхроимпульса
   register_update: process
```

```
   begin
      wait until clk = '1';
      state <= next_state;
      if (inc_data = '1') then data <= int2vec(vec2int(data)+1,8);
      end if;
      if (ld_data = '1') then data <= To_bitvector(IO); end if;
      if (inc_adrs = '1') then
            Address <= int2vec(vec2int(Address)+1,8) after 1 ns;
   -- задержка добавляется для гарантии выполнения записи в память
      end if;
   end process register_update;

   -- Параллельные операторы
   clk <= not clk after 100 ns;
   IO <= To_StdLogicVector(data) when en_data = '1'
      else "ZZZZZZZZ";
end RAMtest;
```

Затем происходит уточнение RAM-модели, чтобы включить в нее временную информацию, основанную на циклах чтения и записи, представленных на рисунках 10.4, 10.5 и 10.6. Пусть $\overline{OE} = 0$. VHDL-модель RAM-памяти, учитывающая временные характеристики (рисунок 10.11), использует generic-константы для уточнения задаваемых по умолчанию существенных временных параметров. Везде применяется транспортная задержка, чтобы возможные неисправности не были сглажены, как это может произойти при использовании инерционной задержки. Процесс RAM ожидает изменения CS_b, WE_b или адреса. Если передний фронт WE_b имеет место, когда CS_b ='0', или существует передний фронт CS_b, когда WE_b='0', то выполняется окончание операции записи, данные сохраняются в RAM и затем считываются обратно после промежутка времени t_{OW}. Если задний фронт WE_b случится, когда CS_b = '0', RAM переключается в режим записи и выходные линии получают значение высокого импеданса.

По переднему фронту CS_b RAM становится невыбранной и значение на выходе данных переходит в состояние Z после определенной задержки. Иначе, по заднему фронту CS_b и, если WE_b=1, память переключится в режим чтения. Выход может находиться в состоянии высокого импеданса после задержки t_{CLZ} (min), но после момента времени t_{ACS} (max) данные на шине должны быть установлены. Промежуток времени между двумя этими параметрами называется переходным периодом, считается, что в этот момент шина имеет значение 'X'. Если произошло изменение адреса и RAM находится в режиме чтения (см. рисунок 10.4, а), старые данные будут сохраняться на шине на протяжении промежутка времени t_{OH}. После переходного периода новые данные могут быть считаны через промежуток t_{AA}.

Тестирующий процесс, выполняемый одновременно с процессом, моделирующим поведение памяти, проверяет корректность временных параметров. Предопределенная переменная NOW содержит текущий момент времени. Для того чтобы исключить возможность появления сообщений об ошибках, проверка не выполняется в момент, когда NOW=0 или когда чип памяти не выбран. При изменении адреса выполняется проверка его стабильности на протяжении цикла записи (t_{WC}). Если адрес изменяется в момент выполнения теста, атрибут Address'stable(t_{WC}) всегда будет возвращать значение FALSE. Поэтому вместо Address следует использовать Address'delayed. Таким образом, Address задерживается на один период дельта и тест проверяет стабильность адреса до момента его изменения. Затем проверяются временные параметры для режима записи. Сначала проверяется стабильность адреса в промежутке времени t_{AW}. Следом за этим проверяется наличие WE_b='0' в течение t_{WP}. В заключение оцениваются временные параметры установки и хранения данных.

Рисунок 10 .11. Временная VHDL-модель 6116 Static CMOS RAM

```
-- временная модель памяти (OE_b=0)
library ieee;
use ieee.std_logic_1164.all;
library bitlib;
use bitlib.bit_pack.all;
entity static_RAM is generic (constant tAA: time := 120 ns;
                                    -- 6116 статическая RAM
   constant tACS: time:= 120 ns;
   constant tCLZ: time:= 10 ns;
   constant tCHZ: time:= 10 ns;
   constant tOH: time:= 10 ns;
   constant tWC: time:= 120 ns;
   constant tAW: time:= 105 ns;
   constant tWP: time:= 70 ns;
   constant tWHZ: time:= 35 ns;
   constant tDW: time:= 35 ns;
   constant tDH: time:= 0 ns;
   constant tOW: time:= 10 ns);

   port (CS_b, WE_b, OE_b: in bit;
         Address: in bit_vector(7 downto 0);
         Data: inout std_logic_vector(7 downto 0):= (others => 'Z'));
end Static_RAM;

architecture SRAM of Static_RAM is
type RAMtype is array (0 to 255) of bit_vector(7 downto 0);
signal RAM1: RAMtype := (others => (others => '0'));
begin
   RAM: process
   begin
      if (rising_edge(WE_b) and CS_b'delayed = '0')
         or (rising_edge(CS_b) and WE_b'delayed = '0') then
   RAM1(vec2int(Address'delayed))<=to_bitvector(Data'delayed);
                                                      --запись
   Data <= transport Data'delayed after tOW;
                                    -- чтение данных после записи
      end if;
      if falling_edge(WE_b) and CS_b = '0' then
                                       -- вход в режим записи
         Data <= transport "ZZZZZZZZ" after tWHZ;
      end if;
      if CS_b'event and OE_b = '0' then
         if CS_b = '1' then  --память не выделена
            Data <= transport "ZZZZZZZZ" after tCHZ;
         elsif WE_b = '1' then  -- чтение
            Data <= "XXXXXXXX" after tCLZ;
            Data <= transport to_stdlogicvector
                           (RAM1(vec2int(Address))) after tACS;
         end if;
   end if;
   if Address'event and CS_b ='0' and OE_b ='0' and WE_b ='1' then
                                          -- чтение
      Data <= "XXXXXXXX" after tOH;
      Data <= transport to_stdlogicvector(RAM1(vec2int(Address)))
               after tAA;
   end if;
   wait on CS_b, WE_b, Address;
end process RAM;

check: process
begin
   if CS_b'delayed = '0' and NOW /= 0 ns then
```

```
      if address'event then
         assert (address'delayed'stable(tWC)) --tRC = tWC assumed
            report "Address cycle time too short"
            severity WARNING;
      end if;
   if rising_edge(WE_b) then
         assert (address'delayed'stable(tAW))
            report "Address not valid long enough to end of write"
            severity WARNING;
         assert (WE_b'delayed'stable(tWP))
            report "Write pulse too short"
            severity WARNING;
         assert (Data'delayed'stable(tDW))
            report "Data setup time too short"
            severity WARNING;
         wait for tDH;
         assert (Data'last_event >= tDH)
            report "Data hold time too short"
            severity WARNING;
      end if;
   end if;
   wait on WE_b, address, CS_b;
   end process check;
end SRAM;
```

VHDL-код для частичного тестирования временной модели RAM представлен на рисунке 10.12. Этот код выполняет цикл чтения, за которым следуют еще два аналогичных цикла. Между ними RAM не выбрана. На рисунке 10.13 даны результаты выполнения теста. Тестируется также ситуация одновременного изменения сигналов на входах и нарушения временных параметров, однако результатов этих тестов на рисунке 10.13 нет.

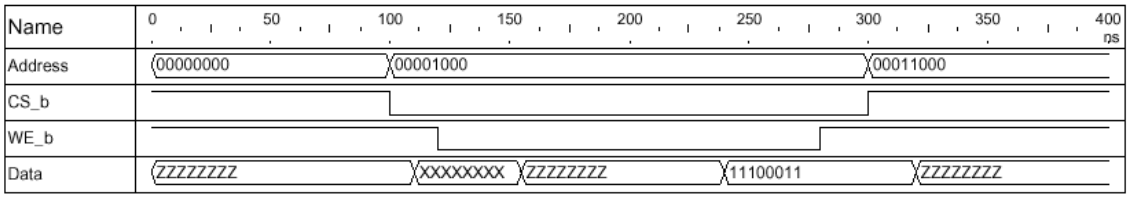
Рисунок 10.12. Testbench для тестирования временной VHDL-модели RAM

```
library IEEE;
use IEEE.std_logic_1164.all;
library BITLIB;
use BITLIB.bit_pack.all;

entity RAM_timing_tester is
end RAM_timing_tester;

architecture testi of RAM_timing_tester is
   component static_RAM is
      port (CS_b, WE_b, OE_b: in bit;
            Address: in bit_vector(7 downto 0);
            Data: inout std_logic_vector(7 downto 0));
   end component Static_RAM;

   signal Cs_b, We_b: bit:= '1'; -- active low signals
   signal Data: std_logic_vector(7 downto 0) := "ZZZZZZZZ";
   signal Address: bit_vector(7 downto 0);
begin
   SRAM1: Static_RAM port map(Cs_b, We_b, '0', Address, Data);
   process
   begin
      wait for 100 ns;

      Address <= "00001000"; -- write(2) with CS pulse
      Cs_b <= '0' ;
      We_b <= transport '0' after 20 ns;
      Data <= transport "11100011" after 140 ns;
      Cs_b <= transport '1' after 200 ns;
      We_b <= transport '1' after 180 ns;
```

```
        Data <= transport "ZZZZZZZZ" after 220 ns;
        wait for 200 ns;

        Address <= "00011000"; -- RAM deselected
        wait for 200 ns;

        Address <= "00001000"; -- Read cycles
        Cs_b <= '0';
        wait for 200 ns;
        Address <= "00010000";
        Cs_b <= '1' after 200 ns;
        wait for 200 ns;

        Address <= "00011000"; -- RAM deselected
        wait for 200 ns;
    end process;
end testi;
```

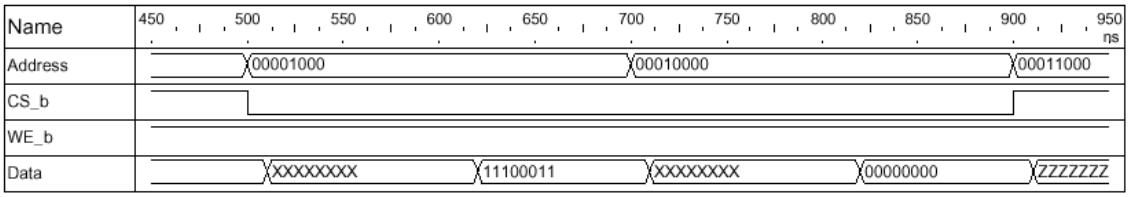
Рисунок 10.13. Результаты тестирования временной модели RAM

Цикл записи



а

Два цикла чтения



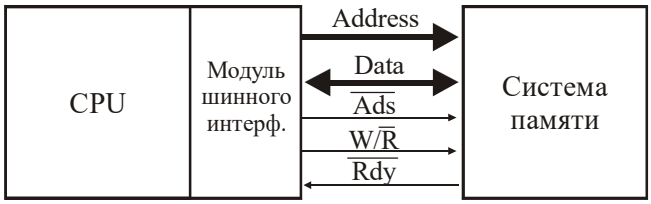
б

10.2. Упрощенная модель 486 шины

Память и входные-выходные устройства обычно соединяются с процессором через тристабильную шину. Для того чтобы гарантировать правильную передачу данных по шине, временные характеристики интерфейса шины микропроцессора и памяти должны быть тщательно рассчитаны и подобраны. В предыдущем параграфе была рассмотрена временная VHDL-модель RAM-памяти, здесь же разрабатывается временная модель шинного интерфейса микропроцессора. После этого для определения корректности временных характеристик моделируется поведение системы, состоящей из микропроцессора и памяти.

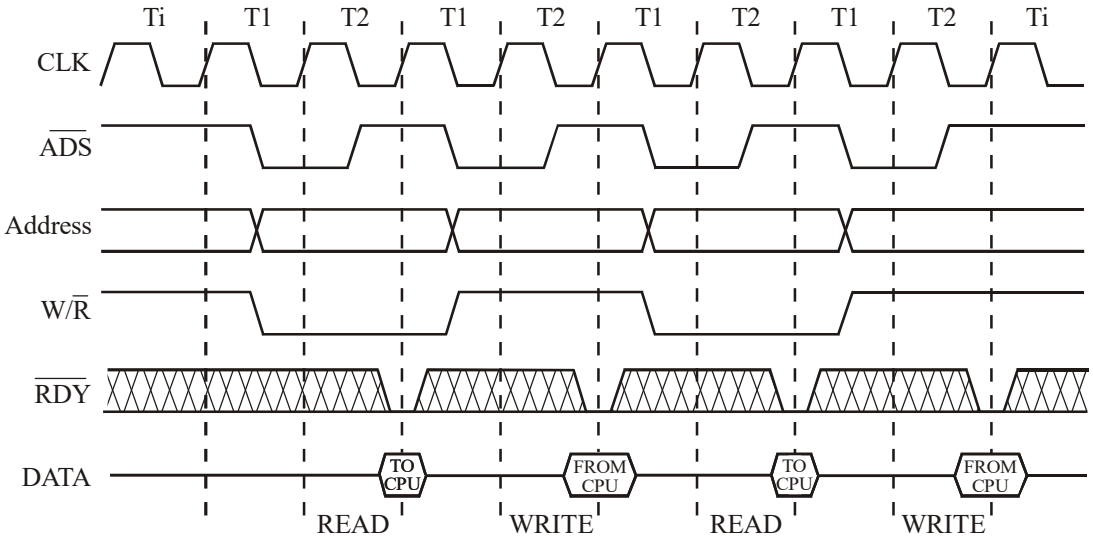
На рисунке 10.14 изображен типичный шинный интерфейс для соединения с памятью. Нормальная последовательность операций для записи в память: 1) микропроцессор передает адрес на шину и устанавливает сигнал $\overline{\text{Ads}}$ (address strobe) для индикации правильности формирования адреса; 2) процессор передает данные на шину данных и устанавливает сигнал W/$\overline{\text{R}}$ (write/$\overline{\text{read}}$) для инициализации записи данных в память. Память передает сигнал готовности $\overline{\text{Rdy}}$ (ready), означающий завершение выполнения операции записи. Для операции чтения из памяти шаг 1 остается прежним, а на втором шаге память передает данные на шину, которые считываются процессором после установки памятью сигнала $\overline{\text{Rdy}}$.

Рисунок 10.14. Шинный интерфейс микропроцессора



В качестве примера разрабатывается упрощенная модель шинного интерфейса 486 микропроцессора. Реальный шинный интерфейс очень сложен и поддерживает много различных типов циклов шины. Рисунки 10.15 и 10.16 иллюстрируют два из них. На первом одно слово данных передается между процессором и шиной за два синхротакта. Они имеют метки T1 и T2 и каждый соответствует внутреннему состоянию шинного контроллера. Также имеется состояние простоя Ti. Во время состояния Ti и между передачей информации шина данных находится в высоком импедансе, который обозначен на диаграмме линией между уровнями 0 и 1. Шина находится в состоянии простоя, пока ее интерфейс не получит запрос от микропроцессора. В T1 интерфейс передает новый адрес на шину и устанавливает $\overline{Ads}$ =0. Также для цикла чтения сигнал чтение/запись равен W/$\overline{R}$ =0 в течение тактов T1 и T2. Во время такта T2 память отвечает на новый адрес и передает данные на шину. Такие данные имеют на диаграмме метку "to CPU". Одновременно память устанавливает $\overline{Rdy}$=0, что означает наличие правильных данных на шине. По переднему фронту синхроимпульса, завершающего цикл T2, шинный интерфейс обнаруживает сигнал $\overline{Rdy}$=0 и данные с шины передаются процессору.

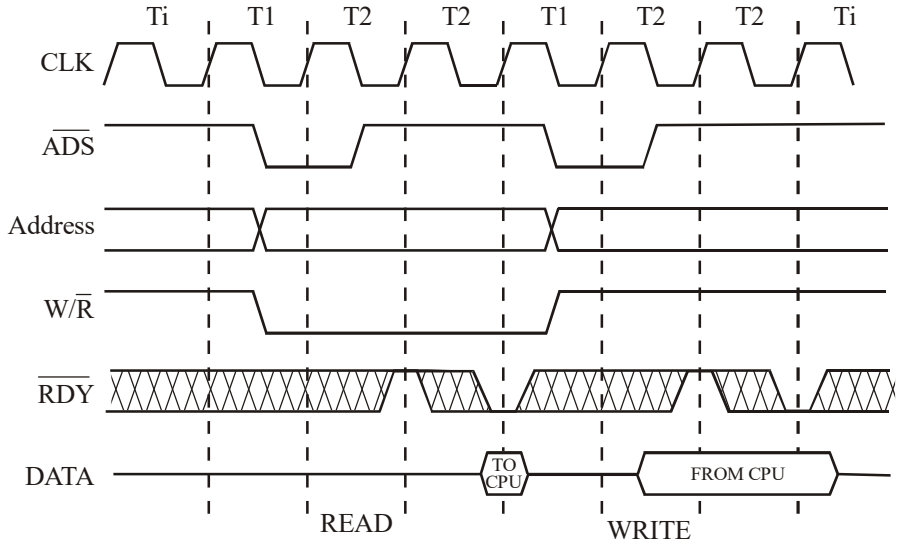
Рисунок 10.15. Циклы 2-2 шины Intel 486



Следующий цикл шины, изображенный на рисунке 10.15, является циклом записи. Как и в предыдущем цикле, новый адрес передается во время такта T1 и $\overline{Ads}$ становится 0, но W/$\overline{R}$ остается в 1. Во время такта T2 процессор передает данные на шину. Ближе к окончанию такта T2 память устанавливает сигнал $\overline{Rdy}$=0, что означает завершение цикла записи. Данные сохраняются в памяти в конце такта T2 по переднему фронту синхросигнала. После этого может быть выполнен следующий цикл чтения или записи.

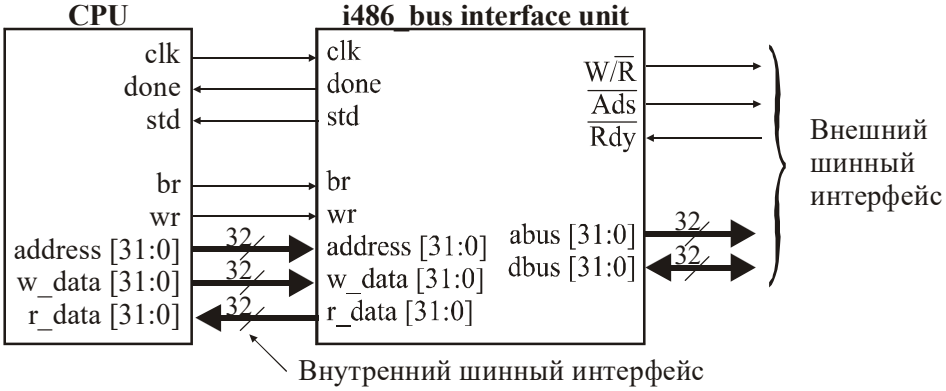
Рисунок 10.16 представляет циклы чтения и записи для случая, когда память имеет низкое быстродействие и для чтения из памяти или записи в нее одного слова данных требуется три синхротакта. Операция чтения подобна операции с рисунка 10.15, за исключением окончания цикла T2. Шинный интерфейс обнаруживает сигнал $\overline{\text{Rdy}}$=1 и добавляет второй такт T2. В конце его $\overline{\text{Rdy}}$ =0 и операция чтения завершается. Аналогично операция записи похожа на операцию записи с рисунка 10.15. В конце первого такта T2 сигнал $\overline{\text{Rdy}}$ остается равным 1, вследствие чего добавляется второй такт T2, в конце которого $\overline{\text{Rdy}}$=0 и операция записи завершается. Дополнительные состояния T2 называются состояниями ожидания, поскольку процессор ожидает память.

Рисунок 10.16. Циклы 3-3 шины Intel 486



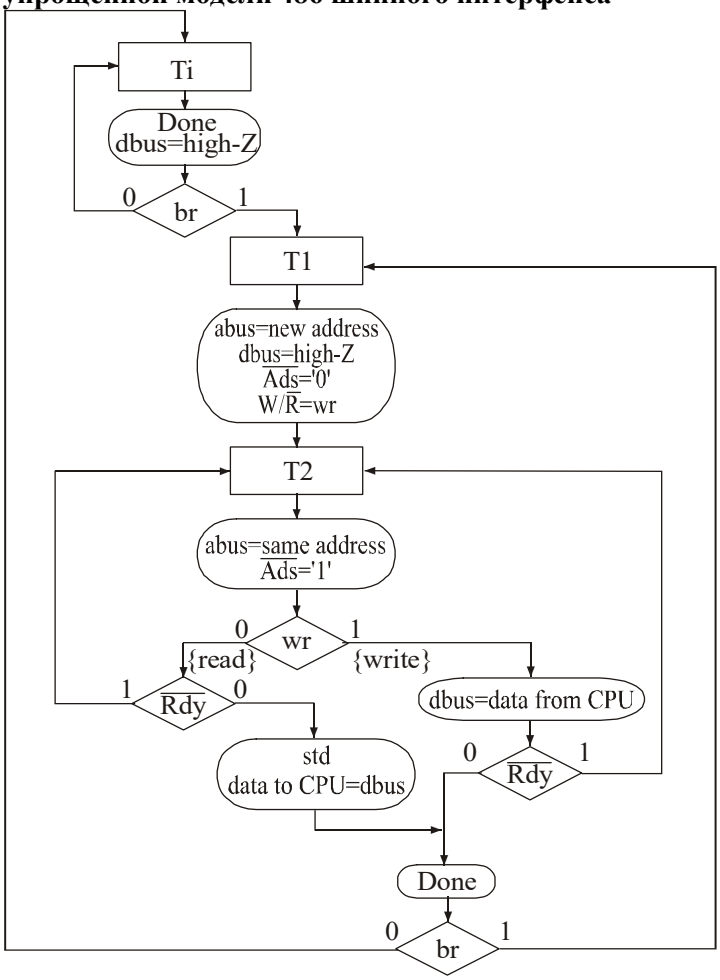
Ниже разрабатывается упрощенная модель модуля 486 шинного интерфейса. Он является составляющей частью 486 микропроцессора и обеспечивает связь между внешней 486 шиной и процессором. Модель основана на 486 формате данных и предназначена для представления внешнего поведения 486 шинного интерфейса. На структурной схеме (рисунок 10.17) представлены сигналы интерфейса, включенные в модель. Реальный шинный интерфейс намного сложнее. Его внешний интерфейс имеет сигналы: для передачи за один раз 8, 16 и 32-битных данных; для выполнения пакетных циклов памяти, повышающих скорость передачи данных; для разрешения другим устройствам контроля над внешней шиной. Разрабатываемая модель включает сигналы только для выполнения основных операций чтения и записи, проиллюстрированных рисунками 10.15 и 10.16. Создание точной модели процессора 486 CPU не является задачей данной книги. Внутренний шинный интерфейс с рисунка 10.17 включает только сигналы для передачи данных между модулем шинного интерфейса и процессором. Если процессору необходимо записать данные в память, подключенную к внешнему шинному интерфейсу, он запрашивает цикл записи, установив сигналы: запрос шины br(bus request)=1 и записи wr (write)=1. Если процессор собирается прочитать данные, он передает сигналы br = 1 и wr = 0. По завершению цикла чтения или записи шинный интерфейс возвращает процессору сигнал done = 1.

Рисунок 10.17. Упрощенный модуль 486 шинного интерфейса



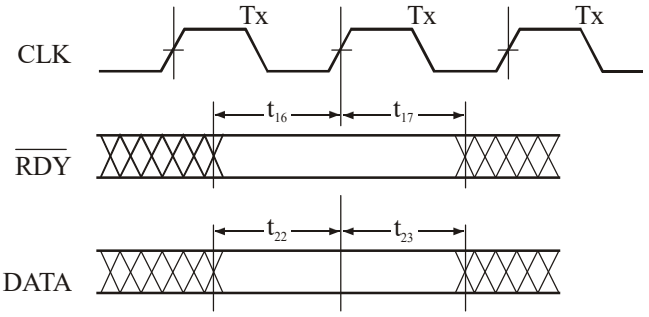
Модуль 486 шинного интерфейса включает автомат для управления операциями шины. Рисунок 10.18 представляет ГСА его упрощенной версии. В состоянии простоя Ti шина данных находится в высоком импедансе. При поступлении сигнала запроса шины br автомат переходит в состояние T1. В T1 новый адрес поступает на шину и $\overline{\text{Ads}} = 0$ означает наличие правильного адреса на шине. Сигнал чтение/запись ($W/\overline{R}$) равен 0 для цикла чтения и 1 – для записи, управляющий автомат переходит в состояние T2. В T2 $\overline{\text{Ads}}$ возвращается в 1. Для цикла чтения wr = 0 и автомат ожидает $\overline{\text{Rdy}} = 0$, что означает доступность данных из памяти. Затем устанавливается сигнал std (store data), сообщающий о наличии данных, которые следует считать процессору. В цикле записи, при wr = 1, данные из CPU размещаются на шине. Управляющий автомат ожидает $\overline{\text{Rdy}} = 0$, означающее, что данные сохранены в памяти. Для чтения и записи при поступлении сигнала $\overline{\text{Rdy}} = 0$ устанавливается сигнал done = 1.

Рисунок 10.18. ГСА для упрощенной модели 486 шинного интерфейса



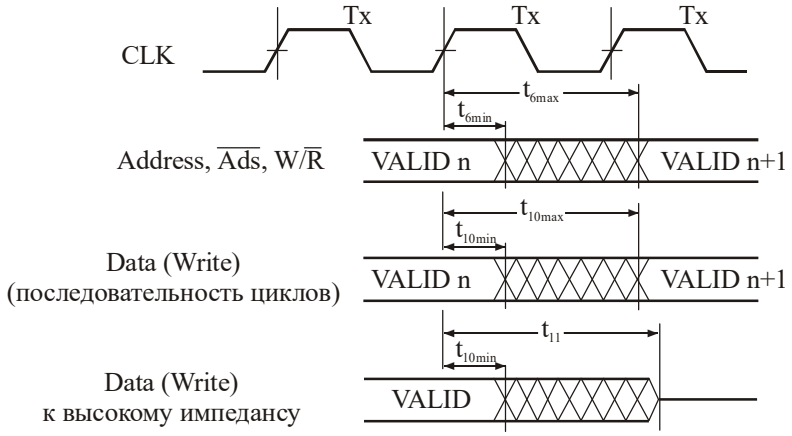
Поскольку шина 486 процессора синхронная, то все ее временные параметры измеряются от переднего фронта синхросигнала. Во время проверки шинным интерфейсом сигнала $\overline{\text{Rdy}}$ он должен быть стабильным в течение времени установки и хранения после поступленния переднего фронта синхросигнала. Эти два временных параметра обозначены на рисунке 10.19 как t_{16} и t_{17} соответственно. Во время цикла чтения, при передаче данных в процессор, должны также соблюдаться требуемые параметры установки и хранения. Данные должны быть стабильны в течение t_{22} до и t_{23} после активного фронта синхросигнала.

Рисунок 10.19. Описание времен хранения и установки



Если выходные сигналы шинного интерфейса изменяются, появляется задержка между передним фронтом и временем, когда сигналы становятся стабильными. Параметры t_{6min} и t_{6max} (рисунок 10.20) определяют промежуток времени после активного фронта синхросигнала, в течение которого адрес изменяет свое значение. На диаграмме этот период обозначен перекрестной линией. Аналогично параметры t_{10min} и t_{10max} определяют промежуток после активного фронта синхросигнала, в котором могут измениться данные. Когда шина данных переключается в состояние высокого импеданса, старая информация сохраняется в течение t_{10min} после активного фронта синхросигнала, а затем до t_{11} она должна принять значение Z. Все такты синхросигнала имеют метки Tx, но изменения на шинах адреса и данных могут происходить между определенными синхротактами. Инициализация изменения адреса происходит по переднему фронту синхросигнала между Ti и T1 или между T2 и T1 тактами, несмотря на то, что инициализация изменения записываемых данных происходит в конце такта T1. Переключение шины данных от некоторого значения к состоянию высокого импеданса начинается в конце цикла записи между T2 и Tl или Ti.

Рисунок 10.20. Временные параметры 486 шины для изменения адреса и шины



VHDL-модель 486 шинного интерфейса, основанная на ГСА (см. рисунок 10.18) и структурной схеме (см. рисунок 10.17), представлена на рисунке 10.21. Шина может работать на нескольких различных скоростных режимах, а временные параметры,

описанные с помощью generic-констант, относятся к ее версии в 50МГц. Модель автомата шины включает два процесса. Третий используется для проверки некоторых критических временных параметров. Начальным состоянием шины является Ti, когда шина данных dbus переходит в состояние высокого импеданса после задержки t_{10min}. Если шинный интерфейс переходит в состояние T1, ads_b($\overline{Ads}$)=0, то w_rb ($W/\overline{R}$) получает соответствующее значение и на шину abus передается адрес. Все изменения происходят после определенных задержек. В состоянии T2 для цикла записи wr =1 данные от процессора w_data передаются на шину dbus. Для цикла чтения wr = 0 данные с шины dbus передаются на процессор как r_data.

Рисунок 10.21. VHDL-модель модуля 486 шинного интерфейса

```
library ieee;
use ieee.std_logic_1164.all;
entity i486_bus is
   generic ( -- Параметры для i486DX 50
      constant t6_max:time:=12 ns;
      constant tl0_min:time:=3 ns;
      constant tl0_max:time:=12 ns;
      constant tll_max:time:=18 ns;
      constant tl6_min:time:=5 ns;
      constant tl7_min:time:=3 ns;
      constant t22_min:time:=5 ns;
      constant t23_min:time:=3 ns);
   port (--внешний интерфейс
      abus: out bit_vector(31 downto 0);
      dbus: inout std_logic_vector(31 downto 0):=(others => 'Z');
      w_rb, ads_b: out bit := '1';
      rdy_b, clk: in bit;
      --внутренний интерфейс
      address, w_data: in bit_vector(31 downto 0);
      r_data: out bit_vector(31 downto 0);
      wr, br: in bit;
      std, done: out bit);
   end i486_bus;
architecture simple_486_bus of i486_bus is
type state_t is (Ti, T1, T2 ) ;
signal state, next_state:state_t:=Ti;
begin
-- Следующий процесс формирует управляющие сигналы и адрес для
-- процессора во время операции чтение/запись (read/write).
-- Процесс также управляет шиной данных в зависимости от типа
-- операции. Во время выполнения операции чтение/запись
-- (read/write) сигнал done равен 0. Когда шина готова принять
-- новый запрос, сигнал done равен 1.
comb_logic: process
begin
   std <= '0';
   case (state) is
      when Ti=> done<='1';
            if (br = '1') then next_state <= T1;
            else next_state <= Ti;
            end if;
            dbus <= transport (others =>'Z') after tl0_min;
      when T1=> done <= '0';
            ads_b <= transport '0' after t6_max;
            w_rb <= transport wr after t6_max;
            abus <= transport address after t6_max;
            dbus <= transport (others =>'Z') after tl0_min;
            next_state <= T2 ;
      when T2=>
            ads_b <= transport '1' after t6_max;
            if (wr = '0') then -- чтение
               if (rdy_b ='0') then
                  r_data <= to_bitvector(dbus);
                  std <= '1';
                  done <= '1';
                  if (br = '0') then next_state <= Ti;
```

```
                  else next_state <= T1;
                end if;
                else next_state <= T2 ;
              end if;
            else -- запись
        dbus <= transport to_stdlogicvector(w_data) after tl0_max;
              if (rdy_b = '0') then done<='1';
                if (br = '0') then next_state <= Ti;
                  else next_state <= T1;
                end if;
              else next_state <= T2 ;
              end if;
          end if;
    end case;
    wait on state, rdy_b, br, dbus;
end process comb_logic;
-- Процесс обновляет текущее состояние по переднему фронту
-- синхроимпульса
seg_logic: process(clk)
begin
    if (clk = '1') then state <= next_state; end if;
end process seg_logic;
-- Процесс проверяет все задержки установки и хранения для всех
-- входящих управляющих сигналов. Все задержки установки и
-- хранения для шины данных проверяются только для режима чтения
wave_check: process (clk, dbus, rdy_b)
    variable clk_last_rise:time:= 0 ns;
begin
    if (now /= 0 ns) then
       if clk'event and clk = '1' then --проверка времен установки
          -- Следующий оператор assert допускает, что время
          -- установки для RDY равно или больше, чем для данных
          assert (rdy_b /= '0') OR (wr /= '0') OR
                    (now - dbus'last_event >= t22_min)
             report "i486 bus:Data setup too short"
             severity WARNING;
          assert (rdy_b'last_event >= tl6_min)
             report "i486 bus:RDY setup too short"
             severity WARNING;
          clk_last_rise := NOW;
       end if;
       if (dbus'event) then --Проверка времен хранения
          --Следующий оператор assert допускает, что время хранения
          -- для RDY эквивалентно или больше, чем для данных
          assert (rdy_b /= '0') OR (wr /= '0') OR
                    (now - clk_last_rise >= t23_min)
             report "i486 bus:Data hold too short"
             severity WARNING;
       end if;
       if (rdy_b'event) then
          assert (now - clk_last_rise >= tl7_min)
             report "i486 bus: RDY signal hold too short"
             severity WARNING;
       end if;
    end if;
    end process wave_check;
end simple_486_bus;
```

Проверяющий процесс тестирует время установки считываемых данных и для rdy_b. Чтобы избежать неправильного сообщения об ошибке, проверка выполняется для случая, когда чип является выбранным и now ≠ 0. Оператор assert проверяет время установки данных, если rdy_b = '0' и wr = '0', и сообщает об ошибке, если:

```
(now - dbus'event) < минимальное время установки,
```

где now – промежуток времени, в котором синхросигнал изменился; dbus'event – время изменения данных. Процесс также проверяет время хранения при считывании данных и сообщает об ошибке, если:

```
(now - clock_last_rise) < минимальное время хранения,
```

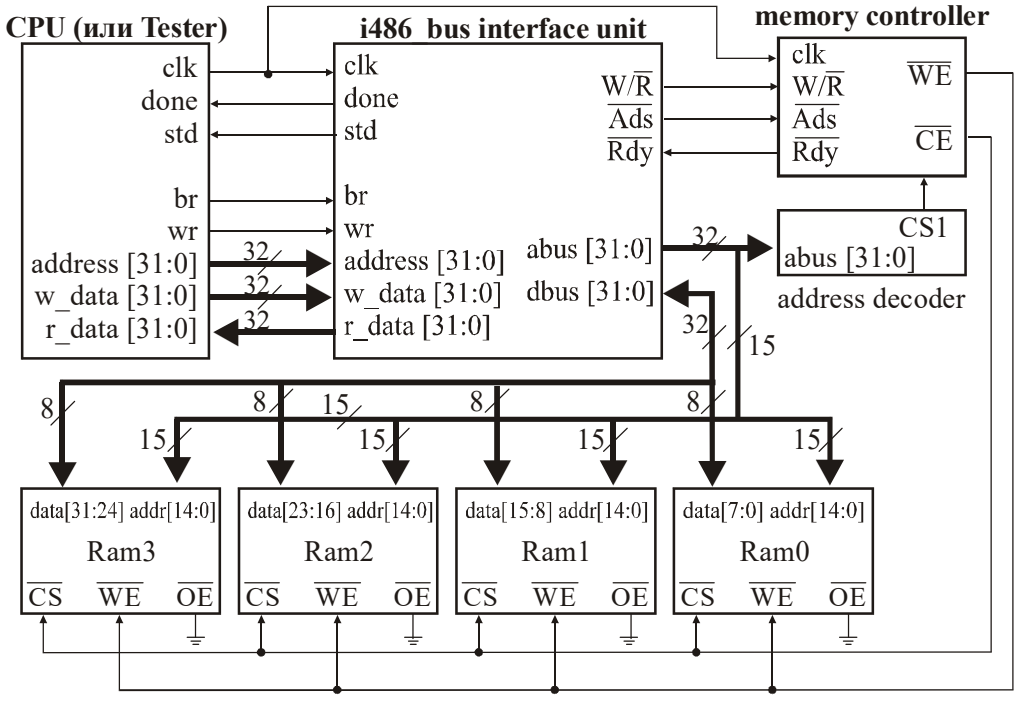
где now – время изменения данных; clock_last_rise – время последнего фронта синхросигнала clk. Процесс также проверяет время хранения сигнала при его изменении.

10.3. Подключение памяти к шине микропроцессора

Рассмотрим задачу подключения памяти к шине микропроцессора. В этом случае временные параметры памяти и микропроцессора должны подходить друг другу. Для записи в память временные параметры установки и хранения должны подходить памяти, а для чтения из памяти аналогичные временные параметры должны подходить микропроцессору. Если быстродействие памяти невысокое, может понадобиться добавление в цикл шины состояний ожидания.

Проектируется интерфейс между 486 шиной и небольшой статической памятью RAM. Затем разрабатывается VHDL testbench для тестирования временных параметров интерфейса. Используются разработанные выше модели статической памяти RAM и шинного интерфейса. На рисунке 10.22 показано соединение шинного интерфейса с памятью. Она состоит из четырех микросхем, каждая из которых содержит 2^{15} = 32,768 8-битных слов. Четыре микросхемы функционируют параллельно, составляя память разрядностью 32 бита. К первому чипу подключены (31-24) биты данных, ко второму – (23-16) и так далее. Пятнадцать младших битов адресной шины подключены параллельно ко всем четырем микросхемам. Система включает контроллер памяти, генерирующий сигналы: для памяти – $\overline{WE}$ и $\overline{CS}$, для шинного интерфейса – $\overline{\text{Rdy}}$, сигнализирующий об окончании цикла чтения или записи. Памяти присвоены адреса с 0 по 32767. В общем случае полная 486 система должна иметь большую динамическую память и карты I/O интерфейсов, подключенные к адресной шине и шине данных. Для того чтобы разрешить расширение системы, используется адресный дешифратор. Таким образом, память будет активна только в отведенном ей адресном диапазоне. Когда адрес находится в диапазоне 0-32767, дешифратор выдает сигнал CS1 = 1, активизирующий контроллер памяти.

Рисунок 10.22. Подключение памяти к 486 шинному интерфейсу

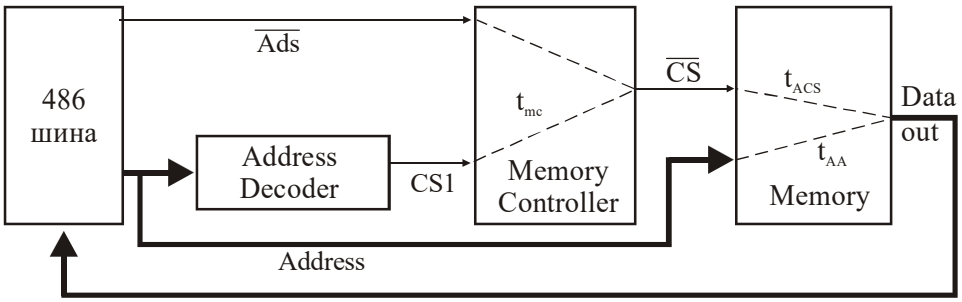


В таблице 10.2 представлены временные параметры памяти 43258A-25, быстрой статической памяти CMOS RAM с временем доступа 25 ns. Она иллюстрирует добавление в цикл шины состояния ожидания. Если использовать более быструю шину, можно исключить ожидание и применить шинный цикл 2-2. В общем случае требуется детальный временной анализ для определения точного числа состояний ожидания, если они необходимы. На рисунке 10.23 показаны пути сигнала для операции чтения из памяти. Для того чтобы сделать анализ самого наихудшего варианта, необходимо найти путь самой длинной задержки. Поскольку $t_{AA} = t_{ACS}$, общая задержка распространения сигнала $\overline{CS}$ длиннее адресного пути. Поэтому, если $\overline{\text{Ads}}$ и адрес остаются неизменными в это же время, адресный путь включает дешифратор. Используя начало состояния T1 как точку отсчета, задержка самого длинного пути есть:

– время от активного фронта синхросигнала до установки адреса = t_{6max} =12 ns;
– задержка дешифратора адреса = t_{decode} = 5 ns;
– задержка контроллера памяти = t_{mc} = 5 ns;
– время доступа к памяти = t_{ACS} = 25 ns;
– время установки данных для 486 шины = t_{22} = 5 ns;
– общая задержка = 52 ns.

Для адресного дешифратора и контроллера памяти задержка распространения предполагается равной 5 ns. Эти значения могут измениться, когда компоненты будут спроектированы. Если шина работает на частоте 50 МГц, то ее период равен 20 ns, поэтому необходимо три такта синхросигнала для выполнения операции чтения информации.

Рисунок 10.23. Путь сигнала для режима чтения

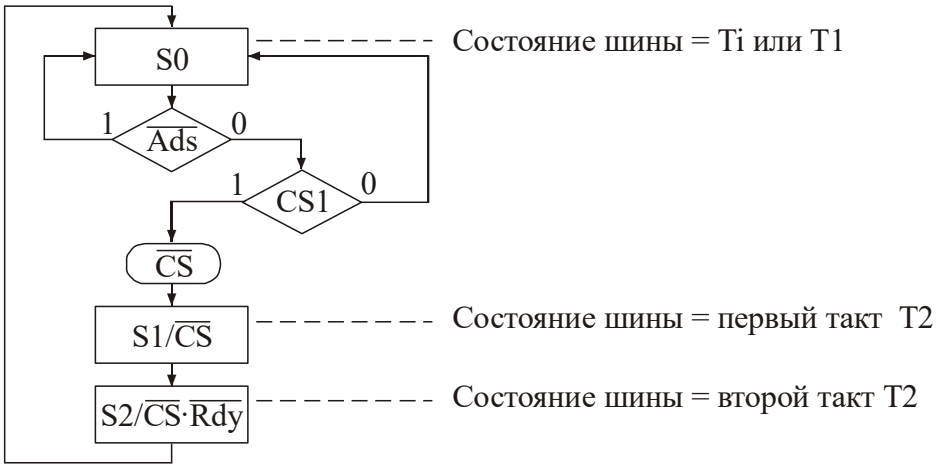


ГСА на рисунке 10.24 представляет возможный вариант контроллера памяти для циклов чтения. Контроллер находится в состоянии S0, пока $\overline{\text{Ads}}$ не сигнализирует о наличии адреса на шине для такта T1 и пока с адресного дешифратора не поступит сигнал CS1, означающий, что адрес находится в диапазоне, принадлежащем памяти. Затем сигнал $\overline{CS}$ =0 активизирует микросхемы памяти и контроллер переходит в состояние S1. В S2 контроллер устанавливает сигнал $\overline{\text{Rdy}}$, обозначая этим конец последнего такта T2. Временные требования будут выполняться, если сигнал $\overline{CS}$ не поступит до состояния S1. В таком случае задержка распространения, измеряемая от конца такта T1, будет равна 35 ns, что является суммой:

– задержки контроллера памяти = t_{mc} = 5 ns;
– времени доступа к памяти = t_{ACS} = 25 ns;
– времени установки данных для 486 шины = t_{22} = 5 ns.

Поскольку период синхросигнала равен 20 ns, правильные данные доступны через 2×20 ns - 5 ns = 35 ns, другими словами, на 5 ns раньше окончания второго такта T2.

Рисунок 10.24. ГСА контроллера памяти для цикла чтения



Затем вычисляется время установки данных для их записи в память. Используя начало такта T2 как точку отсчета, рассчитывается худшая задержка для пути данных:

– время от активного фронта синхросигнала до установки данных = t_{10max} = 12 ns;
– время между установкой данных и окончанием цикла записи в память = t_{DW} = 12 ns;
– общее время = 24 ns.

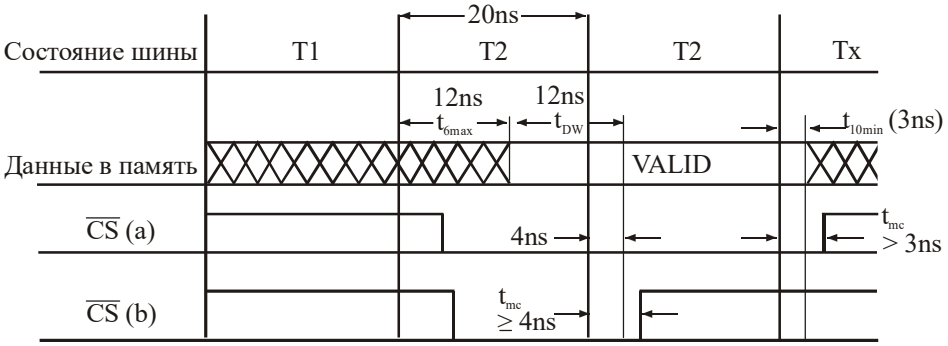
Поскольку общее время от начала T2 до завершения цикла записи должно быть как минимум равно 24 ns, одного цикла T2 длинной 20 ns недостаточно. Требуется дополнительный цикл T2. Это означает, что для выполнения записи нужно три цикла: один T1 и два T2.

Время хранения данных для цикла чтения 486 шины равняется как минимум t_{23} = 3 ns. После того, как $\overline{CS}$ переключается в 1, данные из памяти сохраняются на шине минимально в течение времени t_{CHZ} = 0 ns. Поэтому t_{23} будет достаточным, если t_{mc} равно как минимум 3 ns, следовательно, $\overline{CS}$ =1 и память становится невыделенной как минимум через 3 ns после переднего фронта синхросигнала.

Запись в память может управляться сигналами $\overline{CS}$, $\overline{WE}$, которые должны быть равны 0. По первому сигналу, переключенному в 1, выполняется операция записи. При установленном сигнале $\overline{WE}$ = (W/$\overline{R}$)' значение $\overline{CS}$ используется для управления циклом записи. Поскольку t_{CW} = 15 ns, $\overline{CS}$ должен быть равен 0 как минимум 15 ns. Эти требования выполняются, если $\overline{CS}$ переключается в 0 за один синхротакт.

Поскольку время хранения данных для памяти t_{DH}=0 ns, необходимо гарантировать, чтобы данные не изменились до того, как $\overline{CS}$ переключится в 1. Спецификация 486 шины говорит, что при записи данные на шине могут измениться как минимум через t_{10min} =3 ns после окончания последнего такта T2. Это требование может не выполниться, если $\overline{CS}$ переключится в 1 в конце второго такта T2, а задержка распространения контроллера памяти t_{mc} больше 3 ns. В данном случае, как показано на рисунке 10.25 (a), $\overline{CS}$ переключается в 1 после того, как данные на шине изменятся. Одним вариантом решения этой задачи может быть использование более быстрой логики для контроллера памяти. Другим – переключать $\overline{CS}$ в 1 в конце первого такта T2, как это показано на диаграмме b. До тех пор, пока $t_{mc} \geq 4$ ns, время установки данных t_{DW} будет удовлетворять требованиям.

Рисунок 10.25. Время выбора микросхемы для операции записи в память



На рисунке 10.26 представлена ГСА контроллера памяти, удовлетворяющего временным параметрам операций чтения и записи. Временные параметры для цикла записи основаны на диаграмме, представленной на рисунке 10.25, $\overline{CS}$ (b). Рисунок 10.27 содержит VHDL-код контроллера памяти, основанный на ГСА, и соответствует стандартному шаблону для автомата, за исключением задержек, введенных для выходных сигналов $\overline{CS}$, $\overline{WE}$ и $\overline{\text{Rdy}}$.

Рисунок 10.26. ГСА для контроллера памяти

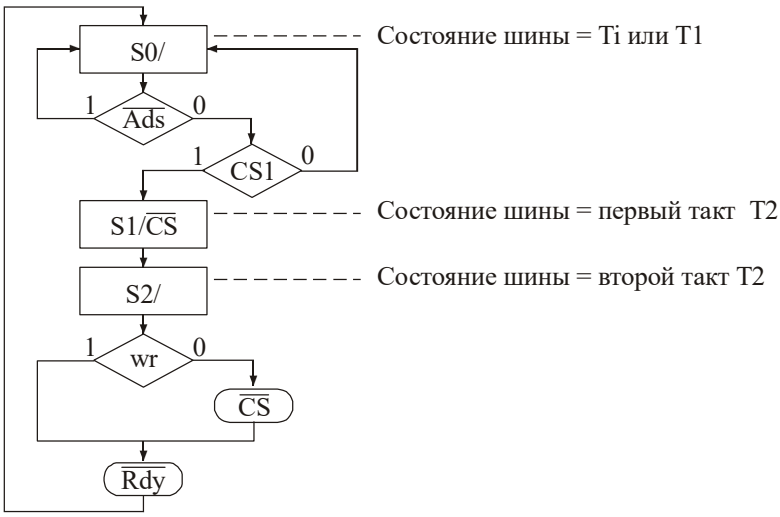


Рисунок 10.27. VHDL-код контроллера памяти

```
-- Контроллер памяти для быстрой CMOS SRAM с одним состоянием ожидания
entity memory_control is
port (  clk, w_rb, ads_b, csl: in bit;
        rdy_b, we_b, cs_b: out bit := '1');
end memory_control;

architecture behavel of memory_control is
   constant delay: time := 5 ns;
   signal state, nextstate: integer range 0 to 2;
   signal new_we_b, new_cs_b, new_rdy_b: bit := '1';
begin
   process(state,ads_b,w_rb,csl)
   begin
      new_cs_b <= '1'; new_rdy_b <= '1'; new_we_b <= '1' ;
      case state is
         when 0 => if ads_b = '0' and csl = '1' then
                                                  nextstate <= 1;
                     else nextstate <= 0;
                   end if;
         when 1 => new_cs_b <= '0' ;
```

```
                    nextstate <= 2;
          when 2 => if w_rb = '1' then new_cs_b <= '1';
                       else new_cs_b <= '0' ;
                    end if;
                    new_rdy_b <= '0';
                    nextstate <= 0;
       end case;
    end process;

    process(clk)
    begin
       if clk = '1' then state <= nextstate; end if;
    end process;

    we_b <= not w_rb after delay;
    cs_b <= new_cs_b after delay;
    rdy_b <= new_rdy_b after delay;
end behavel;
```

Для того чтобы протестировать 486 шинный интерфейс системы статической памяти, создается тестирующий модуль, замещающий процессор с рисунка 10.22. Устройство тестирования необходимо для верификации работы контроллера памяти, подключенного к шинному интерфейсу и памяти. Поскольку шинная модель и модель памяти имеют встроенную проверку временных параметров, устройство тестирования необходимо только для проверки полноты функций того, что записанные в память данные правильно считываются обратно при отсутствии шинных конфликтов. По завершении каждого шинного цикла устройство тестирования должно поддерживать значения br, wr, address и data для следующего шинного цикла. Для того чтобы легче управлять тестовыми последовательностями, они размещены в текстовый файл, а устройство тестирования считывает их. Каждая линия тестового файла содержит следующие значения:

```
br (bit)   wr (bit) address (integer) data (integer).
```

Каждое поле отделяется от другого одним или несколькими пробелами. Для address-адреса и data-данных используется формат integer, поскольку поле integer компактнее и его легче считывать, используя обычные процедуры пакета TEXTIO.

Устройство тестирования (рисунок 10.28) взаимодействует с автоматом шинного интерфейса (см. рисунок 10.18). Поскольку он обнаруживает $\overline{\text{Rdy}}$ до окончания такта шины T2, устройство тестирования должно проверить сигнал Done до переднего фронта, заканчивающего такт T2. Для облегчения этого фронт синхросигнала {testclk) проверяется внутри устройства тестирования. Синхросигнал шины Clk такой же, как и testclk, но с небольшой задержкой. Таким образом, можно проверять значение Done сразу после переднего фронта testclk, но до переднего фронта Clk. Процесс read_test_file ожидает переднего фронта testclk и затем проверяет сигнал Done. Если Done равен '1' и цикл чтения завершен, данные прочитаны из памяти и std='1'. В этом случае устройство тестирования проверяет, что данные, считанные из памяти r_datd, такие же, какие были считаны из тестового файла dataint. Затем устройство тестирования считывает в буфер строку из тестового файла и определяет значения br, wr, data и address из буфера, которые вычисляют следующий шинный цикл. Если он представляет собой цикл записи, данные из тестового файла являются выходом w_data. Значение br контролирует следующее состояние шинного автомата, которое должно быть Ti или T1.

Рисунок 10.28. VHDL- код для модуля тестирования 486 шины

```
-- Тестер для модели шины
library BITLIB;
```

```
use BITLIB.bit_pack.all;
use std.textio.all;
entity tester is
port ( address, w_data: out bit_vector(31 downto 0);
        r_data: in bit_vector(31 downto 0);
        clk, wr, br: out bit;
        std, done: in bit := '0');
end tester;

architecture testi of tester is
  constant half_period: time := 10 ns; -- 20 ns clock period
  signal testclk: bit := '1';
begin
     testclk <= not testclk after half_period;
     clk <= testclk after 1 ns; -- Delay bus clock
  read_test_file: process(testclk)
     file test_file: text open read_mode is "test2.dat";
     variable buff: line;
     variable dataint, addrint: integer;
     variable new_wr, new_br: bit;
  begin
  if testclk = '1' and done = '1' then
     if std = '1' then
        assert dataint = vec2int(r_data)
          report "Read data doesn't match data file!"
          severity error;
     end if;
     if not endfile(test_file) then
        readline(test_file, buff);
        read(buff, new_br);
        read(buff, new_wr);
        read(buff, addrint);
        read(buff, dataint);
        br <= new_br;
        wr <= new_wr;
        address <= int2vec(addrint,32);
        if new_wr = '1' and new_br = '1' then
                  w_data <= int2vec(dataint,32);
          else w_data <= (others => '0');
        end if;
     end if;
  end if;
  end process read_test_file;
end testi;
```

VHDL для полной 486 шинной системы со статической памятью представлен на рисунке 10.29. Эта модель содержит компоненты: устройство тестирования, модуль 486 шинного интерфейса, контроллер памяти и статическую память. Оператор generate используется для реализации четырех копий статической памяти. В дополнение к карте портов для памяти применяется карта generic-констант, содержащая описания временных параметров для статической памяти 43258A-25 CMOS RAM. Поскольку модель использует только 8 адресных линий, их число сокращено с 15 до 8. Дешифратор адреса реализуется с помощью одного параллельного оператора.

Рисунок 10.29. VHDL-код для полной 486 шинной системы со статической памятью

```
library IEEE;
use IEEE.std_logic_1164.all;

entity i486_bus_sys is
end i486_bus_sys;

architecture bus_sys_bhv of i486_bus_sys is
```

```
----Компоненты----
component i486_bus
  port (  -- внешний интерфейс
     abus: out bit_vector(31 downto 0);
     dbus: inout std_logic_vector(31 downto 0);
     w_rb, ads_b: out bit;
     rdy_b, clk: in bit;
          -- внутренний интерфейс
     address, w_data: in bit_vector(31 downto 0);
     r_data: out bit_vector(31 downto 0);
     wr, br: in bit;
     std, done: out bit);
end component;
component static_RAM
  generic (constant tAA, tACS, tCLZ, tCHZ, tOH, tWC, tAW, tWP,
                    tWHZ, tDW, tDH, tOW: time);
  port ( CS_b, WE_b, OE_b: in bit;
        Address: in bit_vector(7 downto 0);
        Data: inout std_logic_vector(7 downto 0));
end component;
component memory_control
  port (clk, w_rb, ads_b, csl: in bit;
        rdy_b, we_b, cs_b: out bit);
end component;
component tester
  port (address, w_data: out bit_vector(31 downto 0);
         r_data: in bit_vector(31 downto 0);
         clk, wr, br: out bit;
         std, done: in bit);
end component;
----Сигналы---
constant decode_delay: time := 5 ns;
constant addr_decode: bit_vector(31 downto 8) := (others => '0');
signal cs1: bit;
-- signals between tester and bus interface unit
signal address, w_data, r_data: bit_vector(31 downto 0);
signal clk, wr, br, std, done: bit;
-- external 486 bus signals
signal w_rb, ads_b, rdy_b: bit;
signal abus: bit_vector(31 downto 0);
signal dbus: std_logic_vector(31 downto 0);
-- signals to RAM
signal cs_b, we_b: bit;
begin
busl: i486_bus port map (abus, dbus, w_rb, ads_b, rdy_b, clk,
                    address, w_data, r_data, wr, br, std, done);
controll: memory_control port map (clk, w_rb, ads_b, cs1, rdy_b,
                                   we_b, cs_b);
RAM32: for i in 3 downto 0 generate
     ram: static_RAM
        generic map(25 ns,25 ns,3 ns,3 ns,3 ns,25 ns,15 ns,15 ns,
                  10 ns,12 ns, 0 ns,0 ns)
        port map(cs_b, we_b, '0', abus(7 downto 0),
                dbus(8*i+7 downto 8*i));
end generate RAM32;
test: tester port map(address, w_data, r_data, clk, wr, br, std, done);
-- Сигнал адресного дешифратора, посылаемого контроллеру памяти
cs1 <= '1' after decode_delay when (abus(31 downto 8) = addr_decode)
  else '0' after decode_delay;
end bus_sys_bhv;
```

В таблице 10.3 показан файл данных, который содержит различные тестовые последовательности шинного цикла: состояние idle, за которым следуют состояния read или write; две последовательности записи; две последовательности чтения;

чтение следует за записью; запись следует за чтением. Последняя линия теста содержит адрес, выходящий за адресный диапазон шины. Когда шинный цикл выполнен, контроллер памяти должен остаться неактивным, сигнал $\overline{\text{Rdy}}$ не должен генерироваться, а шинный интерфейс добавляет состояния ожидания, пока моделирование не окончится.

Таблица 10.3. Тестовые данные для 486 шинной системы

br	wr	addr	Data	Bus action
0	1	7	23	Idle
1	1	139	4863	Write
1	1	255	19283	Write
1	0	139	4863	Read
1	0	255	19283	Read
0	0	59	743	Idle
1	0	139	4863	Read
1	1	139	895	Write
1	0	139	895	Read
1	1	2483	0	Bus hang

Результаты моделирования представлены на рисунке 10.30. Поведение памяти, 486 шины и контроллера памяти такое, как и ожидалось. Поскольку r_data представляется сигналом типа integer, то r_data= 0, когда Dbus находится в состоянии высокого импеданса. Анализируя результаты моделирования, необходимо учитывать, что они настолько хороши, насколько хорошая модель была использована. Модели памяти и шины достаточны для моделирования этих компонентов, но они не являются полными. VHDL-код проверяет не все временные параметры. Во многих случаях были выбраны только максимальные или минимальные задержки для оценки наихудших ситуаций. В других условиях могут потребоваться другие крайние параметры. Для моделирования контроллера памяти используется номинальная задержка. До выполнения его проекта следует вернуться назад и определить минимальную и максимальную приемлемые задержки контроллера, гарантирующие выполнение проектом технических требований.

*Выводы***.** Рассмотрена упрощенная VHDL-модель статической памяти ОЗУ. Разработана усложненная модель, включающая временные параметры и их встроенную проверку, а также временная модель шинного интерфейса микропроцессора, содержащая встроенную проверку временных параметров. Выполнено соединение шины и статической памяти, спроектирован контроллер памяти, удовлетворяющий заданным требованиям к временным параметрам. Смоделирована полная система для проверки того, что временные параметры подходят шине и интерфейсу. В этом примере продемонстрирован принцип проектирования интерфейса для худшего случая временных параметров и показано использование VHDL для проверки корректности проекта.

10.4. Задачи

10.4.1. Ответить на следующие вопросы, касающиеся 6116-2 и 43258A-25 статической памяти CMOS RAM. Использовать временные параметры из таблицы 10.2.

1) Какая максимальная частота синхросигнала может быть использована?

2) Через какой минимальный промежуток времени после изменения адреса или шины могут быть считаны правильные данные?

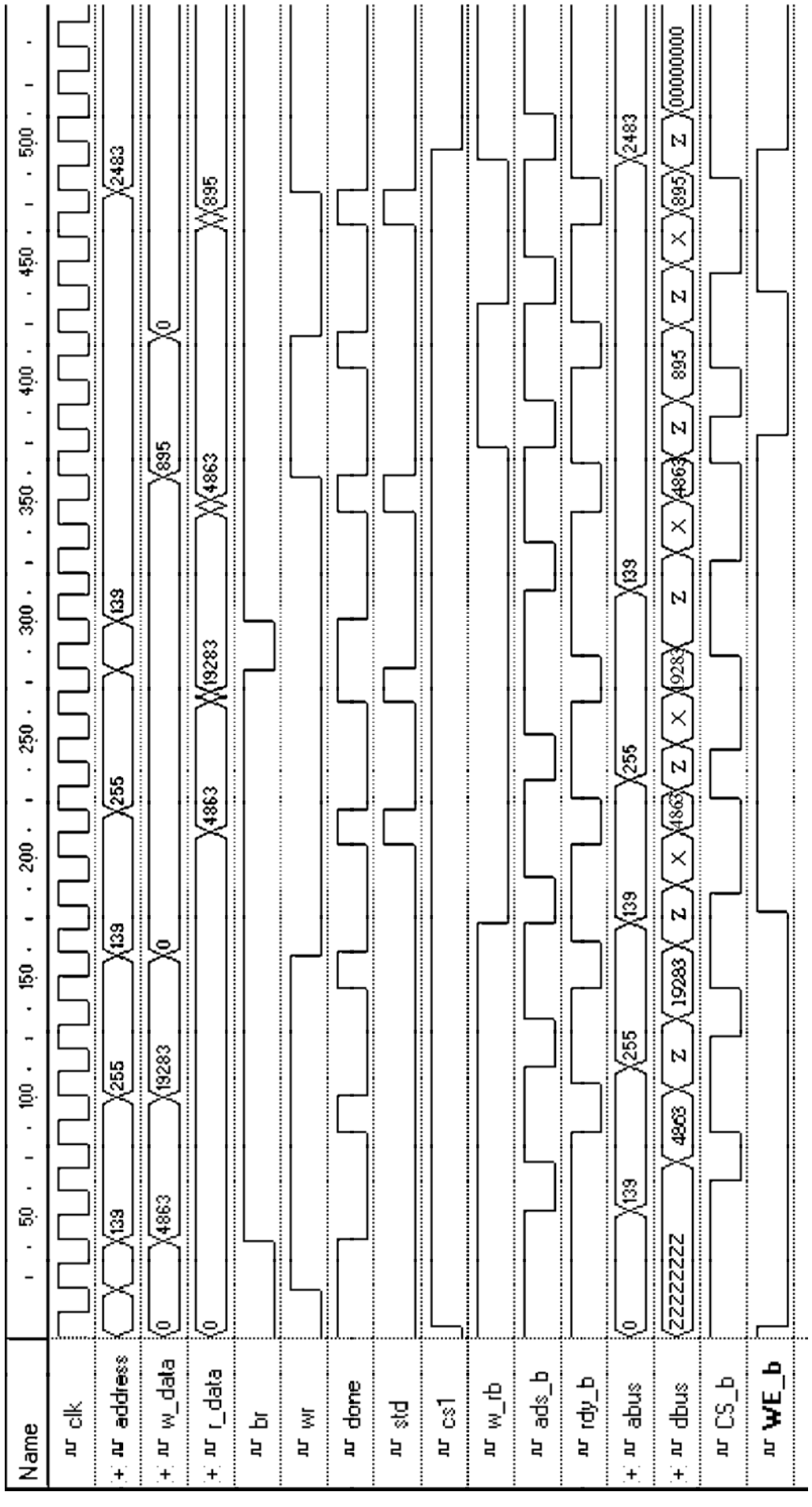


Рисунок 10.30. Результаты тестирования системы 486 шины

3) Для цикла записи, управляемого сигналом $\overline{WE}$, через какой минимальный и максимальный промежутки времени после того, как $\overline{WE}$=0, данные могут быть переданы на шину?

4) Для цикла записи, управляемого сигналом $\overline{\text{CS}}$, через какой минимальный и максимальный промежутки времени после изменения адреса данные могут быть переданы на шину?

10.4.2. Пусть в упрощенной модели памяти 6116 CMOS RAM $\overline{CS}$ =0 и $\overline{OE}$=0, таким образом, операции памяти зависят только от адреса и WE.

1) Записать упрощенную VHDL-модель памяти, игнорирующую все временные параметры. Модель не должна содержать сигналы $\overline{CS}$ или $\overline{OE}$.

2) Включить в модель следующие временные параметры: t_{AA}, t_{OH}, t_{WHZ} и t_{OW}. Для чтения Dout следует перейти в "XXXXXXXX" через промежуток времени t_{OH}, а затем передать данные на шину после t_{AA}. Для записи Dout следует иметь высокий импеданс после t_{WHZ} и "XXXXXXXX" после t_{OW}.

3) Добавить еще один процесс, который бы проверял корректность временных параметров t_{WP}, t_{DW} и t_{DH} и выдавал сообщение в случае возникновения ошибки.

10.4.3. VHDL-код, описывающий операции 6116 памяти, представлен на рисунке 10.11.

1) Добавить код, выдающий предупреждение, если время установки, время хранения данных для режима записи в память или минимальная продолжительность импульса сигнала WEb не удовлетворяют требованиям.

2) Обозначить изменения и дополнения к исходному VHDL-коду, которые необходимо выполнить для того, чтобы модель включала сигнал OE_b ($\overline{OE}$). Обратите внимание: для операции чтения, если OE_b переключился в 0 после того, как CS_b стал равным 0, должен быть введен параметр t_{OE} времени доступа. Также необходимо учитывать, что когда OE_b переключается в 1, на шине должно устанавливаться значение высокого импеданса после t_{OHZ}.

10.4.4. Какие изменения необходимо сделать в проверяющем процессе в VHDL временной модели памяти 6116 RAM (рисунок 10.11), для того чтобы выполнялась проверка времени установки адреса t_{AS} и времени восстановления записи (the write recovery time) t_{WR}?

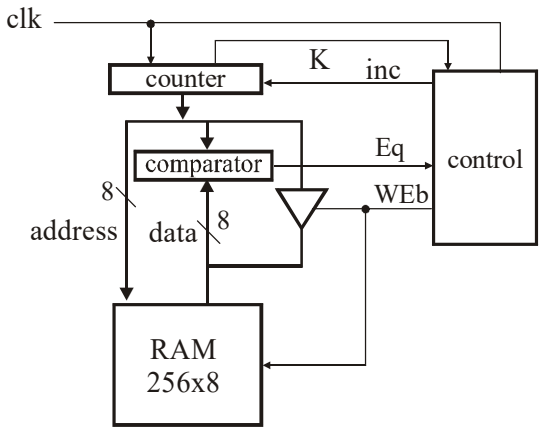
10.4.5. Рассмотреть цикл записи статической памяти CMOS RAM, управляемый сигналом $\overline{CS}$ (рисунок 10.5). Какой VHDL-код необходимо добавить в проверяющий процесс временной модели памяти (рисунок 10.11) для проверки корректности операции записи, управляемой сигналом $\overline{CS}$? Необходимо проверять временные параметры: t_{CW}, t_{DW} и t_{DH}.

10.4.6. Спроектировать систему для тестирования памяти, выполняющую проверку первых 256 байт статической памяти 6116-2 RAM. Система включает простое управляющее устройство, 8-битный счетчик, компаратор и память, как это описано и показано ниже. Счетчик подключен к адресной шине и шине данных IO таким образом, что 0 будет записан по адресу 0, 1 – по адресу 1, 2 – по адресу 2, ... ,255 – по адресу 255. Затем данные считываются обратно и сравниваются со значением адреса. Если данные не совпадают, контроллер переходит в состояние fail сразу после обнаружения ошибки, иначе он находится в состоянии pass, пока все 256 ячеек не будут проверены. Пусть OE_b=0 и CS_b = 0.

1) Нарисовать ГСА или граф состояний для управляющего устройства (5 состояний). Пусть длина периода синхроимпульса достаточна для чтения одного слова данных за один такт.

2) Определить для системы минимальный период синхроимпульса, позволяющий выполнить операцию чтения. Пусть имеются следующие задержки:

- стабильность выхода счетчика после переднего фронта синхроимпульса – 15 ns;
- стабильность управляющих сигналов, формируемых автоматом после переднего фронта синхросигнла – 20 ns;
- задержка на компараторе – 15 ns.



10.4.7. Спроектировать устройство тестирования памяти, проверяющее корректность операций для статической памяти 6116 CMOS RAM. Устройство должно сохранять шахматную последовательность (чередование 0 и 1 для четного адреса, чередование 1 и 0 для нечетного) во всех ячейках памяти, затем считывать их обратно. Тест должен повторяться с использованием обратных входных последовательностей.

1) Нарисовать структурную схему устройства тестирования памяти. Указать и описать все управляющие сигналы.

2) Нарисовать ГСА или граф состояний управляющего модуля. Пусть используется упрощенная модель памяти, не учитывающая временные параметры.

3) Написать VHDL-код устройства тестирования. Для верификации его операций использовать testbench.

10.4.8. Для упрощенной VHDL-модели 486 шины (рисунок 10.21) адрес может изменять значение с текущего VALIDn на следующее VALIDn+1 в любой момент времени между t_{6min} и t_{6max}, как это показано на рисунке 10.20 перекрестной штриховкой. Таким образом, во время этого временного интервала значение адреса может быть рассмотрено как неизвестное 'X'. В более точную модель следует включить этот период неизвестного значения адреса. Указать необходимые изменения в VHDL-коде, для того чтобы модель шины функционировала описанным образом.

1) Аналогичная ситуация происходит при установлении данных на шине. Во время цикла записи существует временной интервал между t_{10min} и t_{10max}, когда значение на шине равно 'X' перед окончательной установкой. Также при завершении состояния T2 существует период времени между t_{10min} и t_{11max}, когда данные на шине равны 'X' перед переключением ее в состояние высокого импеданса. Укажите необходимые изменения, которые требуется внести в VHDL-код для того, чтобы он моделировал эту ситуацию.

2) Период синхроимпульса должен быть $\geq t_{1min}$ и $\leq t_{1max}$. Добавить в VHDL-код процесс wave_check, который бы выдавал сообщение "clock pulse width error" , если длина синхроимпульса не соответствует этим границам.

10.4.9. Определить верхнюю и нижнюю границы для задержки контроллера памяти с рисунка 10.26. Использовать стимулятор, чтобы проверить, какие временные нарушения произойдут при выходе за эти границы.

10.4.10. Перепроектировать контроллер памяти с рисунка 10.26 для использования меньшей частоты синхросигнала, чтобы не требовалось добавление состояний ожидания и шина работала по схеме 2-2. Пусть за состоянием T1 следует состояние T2 после цикла чтения или записи. Убедитесь, что сигналы данных и управляющие удовлетворяют временным параметрам.

1) Какая максимальная частота синхроимпульса может быть использована для статической памяти 43258A-25 CMOS RAM, чтобы не требовалось введения состояний ожидания?

2) Нарисовать ГСА для нового контроллера памяти.

3) Написать VHDL-код контроллера памяти и протестировать его операции с моделью шины.

10.4.11. Перепроектировать контроллер памяти с рисунка 10.26 для использования статической памяти 43258A-15 RAM с циклами шины 2-2 без введения состояния ожидания. Память имеет следующие временные параметры: $t_{RC}=t_{AA}=t_{ACS}=t_{WC}=15$ ns, $t_{CLZ}=t_{OH}=t_{CHZ}=t_{WHZ}=3$ ns(min), $t_{CHZ}=10$ ns(max), $t_{WHZ}=8$ ns(max), $t_{CW}=t_{AW}=t_{AW}=t_{WP}=12$ ns, $t_{DW}=9$ ns.

1) Чему равна максимальная задержка, допустимая для контроллера памяти?

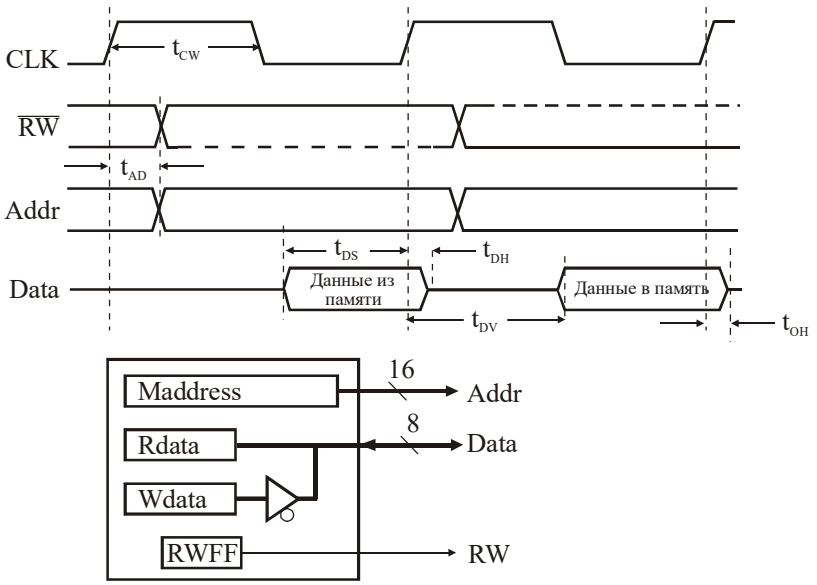
2) Нарисовать ГСА для нового контроллера памяти.

3) Написать VHDL-код для контроллера памяти и верифицировать его на работе с шиной.

10.4.12. Упрощенный микропроцессор передает за один синхротакт один бит данных, как это показано на приведенной ниже временной диаграмме. Внутренними регистрами микропроцессора являются: 16-битный адресный регистр M; 8-битный регистр данных R, в котором сохраняются данные, прочитанные из памяти; 8-битный регистр данных W, в котором сохраняются данные, записываемые в память; RWFF (read-write flip-flop) регистр, который устанавливается в 1 на протяжении цикла чтения и в 0 на протяжении цикла записи. Все регистры изменяют свои состояния по переднему фронту синхросигнала.

1) Записать VHDL-код, описывающий шинный интерфейс микропроцессора. Пусть уже существует процесс, обновляющий содержимое регистров по переднему фронту синхросигнала, поэтому создавать его нет необходимости. VHDL-модель должна генерировать в правильное время сигналы шины.

2) Написать процесс, верифицирующий временные параметры установки и хранения для цикла чтения.



ГЛАВА 11
ГРАНИЧНОЕ СКАНИРОВАНИЕ

Описываются методы, позволяющие эффективно выполнять анализ поведения цифровых систем. Ранее было показано, как для верификации и тестирования VHDL-моделей на логическом уровне используется специальное VHDL-описание – TestBench, которое дает возможность убедиться в соответствии модели техническому заданию. После реализации устройство также нуждается в верификации на физическом уровне, чтобы гарантировать правильность функционирования и корректность его временных параметров.

При промышленном выпуске устройств поведение каждого из них необходимо тестировать, для того чтобы исключить возможность существования внесенных на этапе производства ошибок. Такое тестирование может быть дорогостоящим и занимать много времени. В настоящее время оно является одним из основных компонентов себестоимости устройства, что повышает значимость разработки эффективных методов тестирования цифровых систем.

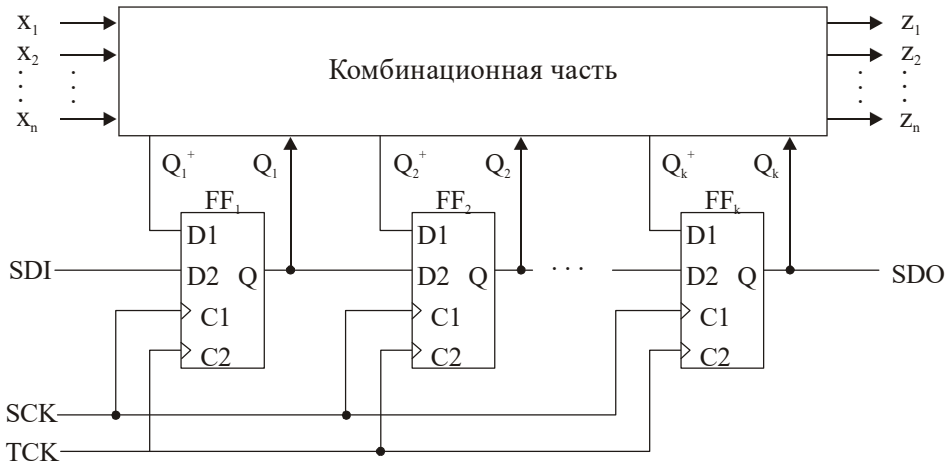
Одной из проблем, которая мешает эффективному тестированию устройств, является наблюдаемость только внешних входов и выходов при невозможности обзора состояний внутренних линий. Решением проблемы может быть использование контрольных точек, выводимых на разъем. Однако это становится невозможным из-за возрастающей плотности компонентов и уменьшения размеров цифровых систем. Введение дополнительных выходных линий увеличивает количество внешних контактов микросхемы, число которых является ограниченным. Другое решение связано с методологией сканирования, которая позволяет наблюдать внутренние состояния устройства, не требуя большого количества дополнительных входов и выходов. В дополнение к идее сканирования используется метод самотестирования цифровых устройств. Введение дополнительных компонентов в микросхему позволяет выполнять генерацию тестовых последовательностей и анализ реакций на тесте без дорогостоящих внешних приборов.

11.1. Метод сканируемого пути (Scan path testing)

Тестирование последовательностных схем значительно упрощается, если имеется возможность кроме внешних выходов наблюдать внутренние состояния триггеров. Чтобы убедиться в правильности работы устройства, необходимо верифицировать каждый комбинационный выход и состояния элементов памяти. Один из подходов ориентирован на то, чтобы выходы триггера микросхемы были соединены с внешними контактами. Но поскольку число ножек микросхемы ограничено, то такая стратегия практически нереализуема. Поэтому для того, чтобы иметь возможность наблюдать состояния триггеров, значительно не увеличивая при этом число внешних контактов, из триггеров формируется сдвиговый регистр. Таким образом, появляется возможность считывать хранящиеся в триггерах значения побитно, используя единственный дополнительный последовательный выход. Такой подход представляет собой основу метода сканируемого пути.

На рисунке 11.1 представлена схема, реализующая метод сканируемого пути, основанный на двухвходовых триггерах. Последовательностная схема делится на две части: комбинационную и память, составленную из триггеров. Каждый из них имеет два входа D и две линии синхронизации. При поступлении синхросигнала C1 (C2) значение со входа D1 (D2) сохраняется в триггере. Для формирования сдвигового регистра выход Q каждого триггера соединяется со входом D2 следующего за ним. Состояние ($Q_1^+ Q_2^+ \ldots Q_k^+$) загружается в триггеры при поступлении синхросигнала C1, а новые значения ($Q_1 Q_2 \ldots Q_k$) подаются обратно на комбинационную логику. Когда схема работает в нормальном режиме, используется системная синхронизация SCK = C1.

Рисунок 11.1. Схема сканируемого пути, построенная на двухвходовых триггерах



SDI (scan data input) - вход сканируемых данных,
SDO(scan data output) - выход сканируемых данных,
SCK(system clock) - системный синхровход,
TCK(test clock) - тестовый синхровход.

Во время тестирования триггеры устанавливаются в заданное состояние, для чего выполняется сдвиг значений через вход сканируемых данных SDI по синхроимпульсам, поступающим на тестовый синхровход TCK. На входы $X_1X_2...\ X_n$ подается тестовый набор, а с выходов $Z_1Z_2...\ Z_n$ считывается реакция схемы, при этом сигнал SCK используется для перевода схемы в следующее состояние. Затем значения, хранящиеся в триггерах, сканируются через сдвиговый регистр по синхросигналу TCK и наблюдаются на выходе SDO. Этот метод преобразует задачу тестирования последовательностной логики к тестированию комбинационной схемы. При этом для построения тестовых наборов может быть использован любой алгоритм генерации тестов для комбинационных устройств. Каждый тестовый вектор должен иметь длину (n + k) битов, поскольку схема имеет n входов X и k триггеров состояний. Первая часть теста подается непосредственно на входы X, а вторая Q – заносится через последовательный вход SDI в триггеры. В общем случае процедура тестирования включает следующие шаги:

1. Сканирование тестового вектора. Значения в разряды Q_i заносятся через вход SDI с использованием тестовой синхронизации.

2. Подача тестовой последовательности на X входы.

3. Считываются значения с выходов Z после промежутка времени, достаточного для распространения сигналов через комбинационную часть.

4. Для того чтобы сохранить значения Q_i^+ в соответствующих триггерах, подается системный синхроимпульс SCK.

5. По синхросигналу TCK значения Q_i сдвигаются на внешний выход.

6. Шаги 1-5 повторяются для каждого вектора теста.

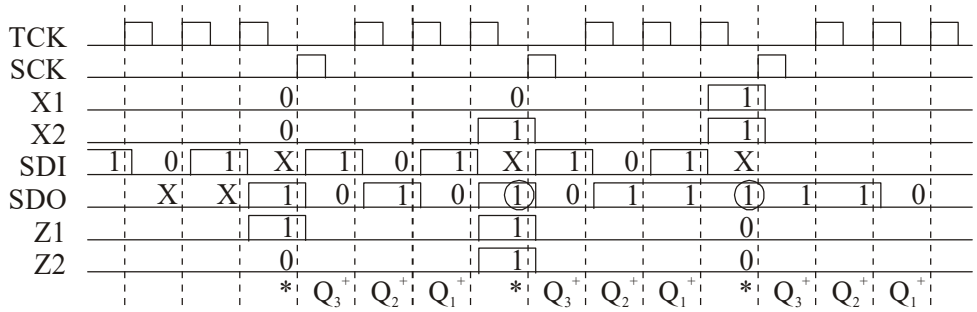
Шаги 5 и 1 могут перекрываться, поскольку имеется возможность заносить следующий тестовый вектор в момент считывания предыдущего.

Рассмотрим использование данного метода для тестирования схемы, имеющей два входа X_1X_2, три триггера $Q_1Q_2Q_3$ и два выхода Z_1Z_2. Одна строка таблицы переходов такого устройства имеет вид:

$Q_1Q_2Q_3$	$Q_1^+Q_2^+Q_3^+$				Z_1Z_2
X_1X_2=	00	01	11	10	00 01 11 10
101	010	110	011	111	10 11 00 01

На рисунке 11.2 изображены временные диаграммы для тестирования данной строки таблицы переходов. Сначала с использованием синхронизации TCK последовательность 101 сдвигается в триггеры, при этом младший бит (Q_3) заносится первым. На входы X_1X_2 подается 00, тогда выходы имеют значения $Z_1Z_2 = 10$. По синхросигналу SCK схема переходит в состояние 010. Значение из триггеров 010 сдвигается на внешний выход и одновременно в них заносится новое состояние 101, для чего используется синхроносигнал TCK. Процедура продолжается до завершения процесса тестирования.

Рисунок 11.2. Временные диаграммы для сканирующего тестирования



* Чтение сигналов на выходах, а также значения на выходах в другие моменты времени не показаны.

В общем случае, цифровые устройства реализуются на микросхемах, содержащих регистры и блоки комбинационной логики, как это показано на рисунке 11.3. Для того чтобы применить метод сканирования пути, необходимо заменить обычные триггеры на двухвходовые или другие, позволяющие сканировать триггеры, и объединить последние в цепь сканирования, как это показано на рисунке 11.4. После этого, используя тестовую синхронизацию, можно сканировать данные во всех регистрах.

Рисунок 11.3. Система регистров и комбинационных блоков без сканируемого пути

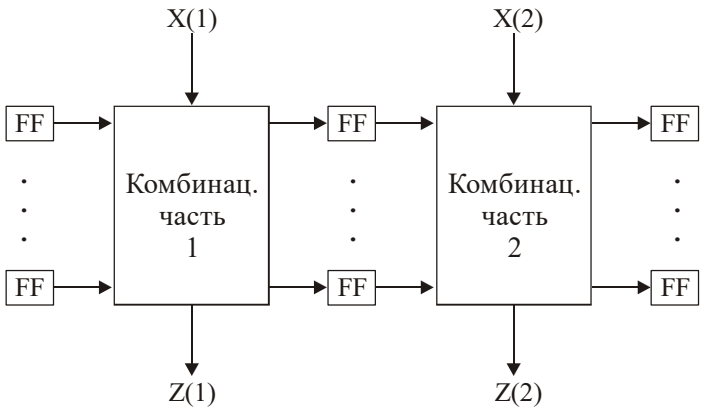
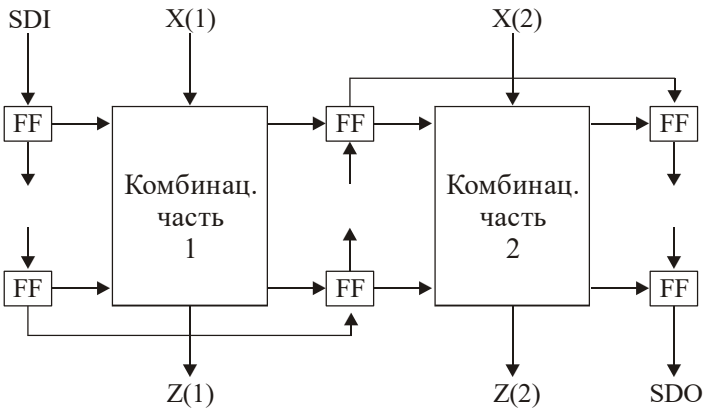
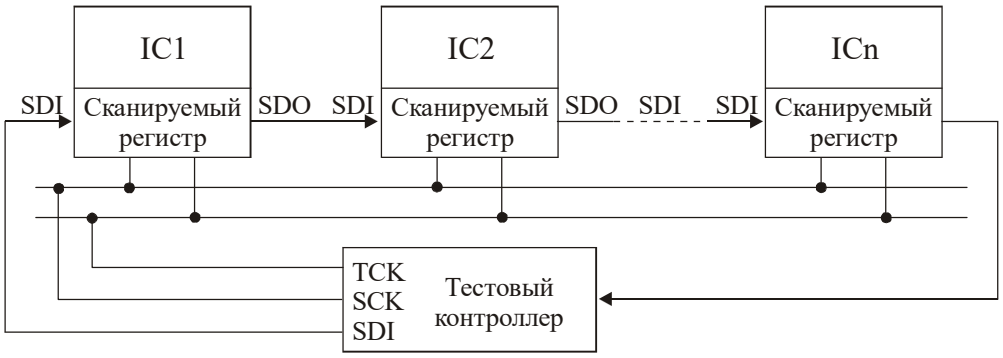


Рисунок 11.4. Система регистров и комбинационных блоков, имеющая сканируемый путь



Если на плате устанавливается несколько микросхем, то можно объединить их сканируемые регистры таким образом, чтобы все устройство было доступно для тестирования с помощью одного дополнительного последовательного порта доступа (рисунок 11.5).

Рисунок 11.5. Конфигурация сканируемого пути для нескольких микросхем



11.2. Boundary scan

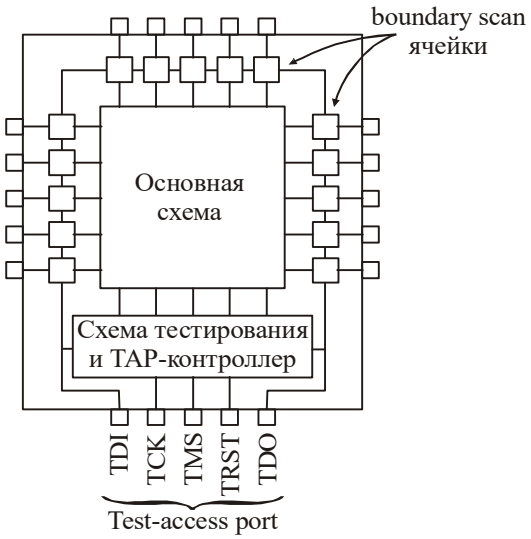
Первоначально технология граничного сканирования была ориентирована на диагностическое обслуживание печатных плат, содержащих десятки микросхем. Это связано с тем, что тенденция развития рынка микроэлектроники такова, что микросхемы становятся все более сложными, увеличивается число внешних контактов, плотность монтажа печатных плат повышается, увеличивается число слоев и уменьшается ширина линий трассировки. Тестирование таких плат, после того как на них разместили сложные микросхемы, становится очень трудной задачей. Ее решение, с помощью только внешних разъемов, может быть практически неприемлемым во времени. Когда платы персональных компьютеров имели меньшую плотность и более широкие соединительные линии трассировки, тестирование часто выполнялось с использованием зондирования. При этом применялся острый зонд, который подключался к контактным линиям схемы для подачи тестовых последовательностей и считывания реакций различных микросхем, содержащихся на плате. Однако зондирование не применяется при высокой плотности монтажа печатных плат цифровых устройств с узкими линиями трассировки и сложными микросхемами.

Метод тестирования boundary scan был введен для упрощения тестирования печатных плат, который имеет также и другое наименование – JTAG, по названию группы его создателей – Joint Test Action Group. В 80-х годах Joint Test Action Group разработала спецификацию для boundary scan тестирования. Последняя была стандартизирована в 1990 году как IEEE Std. 1149.1-1990. В 1993 году стандарт был пересмотрен, исправлен, дополнен и стал называться 1149.1a. В 1994 году в него включили описание языка Boundary-Scan Description Language (BSDL). Последняя ревизия стандарта была выполнена в 2001 году. В настоящее время большинство выпускаемых электронных устройств поддерживает boundary scan тестирование.

На рисунке 11.6 представлена микросхема с дополнительной boundary scan логикой. По одной ячейке boundary scan регистра (BSR) размещается между входными и выходными контактами и внутренним ядром микросхемы. Четыре или пять ее контактов предназначены под порты доступа при тестировании (test-access port – TAP). TAP-контроллер и дополнительная логика для тестирования также размещаются в микросхеме. TAP-контакты выполняют следующие функции: TDI – ввод тестовых последовательностей: данные вводятся последовательно, с помощью операции сдвига, в регистр BSR; TCK – синхровход для тестирования; TMS – выбор режима тестирования; TDO – вывод реакций микросхемы последователь-

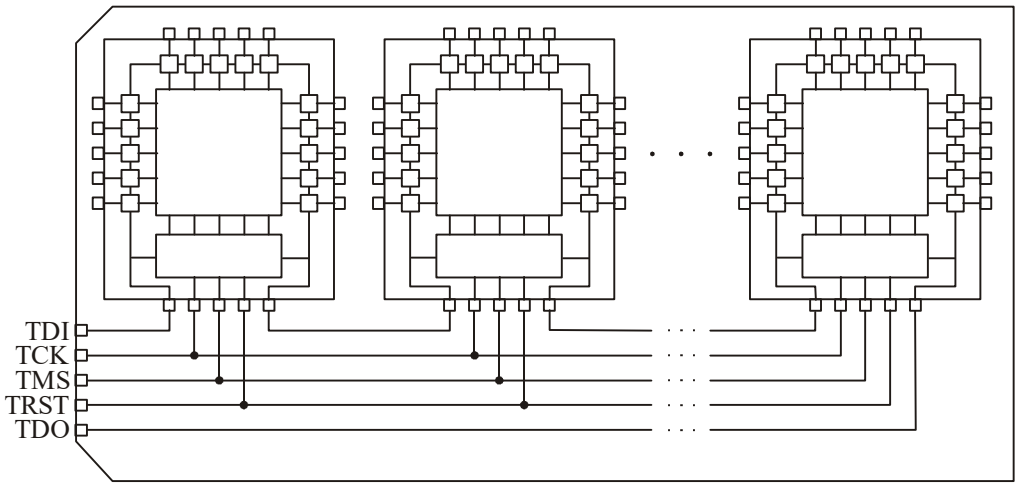
но из регистра BSR; TRST – сброс тестирования – сбрасывается TAP-контроллер и логическая схема тестирования.

Рисунок 11.6. Микросхема с boundary scan регистром



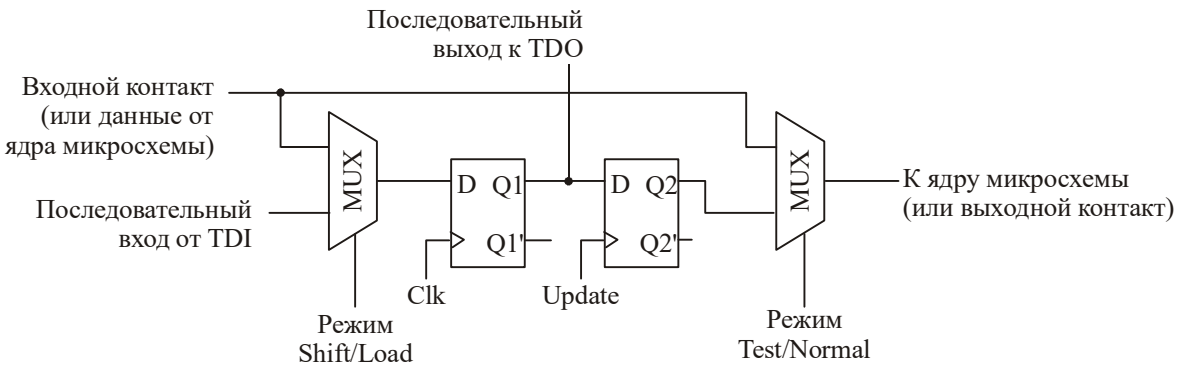
Печатная плата с несколькими boundary scan микросхемами представлена на рисунке 11.7. Они соединены вместе в единую цепочку со входом TDI и выходами TDO, TCK, TMS и TRST, которые присоединены параллельно ко всем микросхемам. Используя эти сигналы, методы тестирования и тестовые наборы, можно проверить каждую микросхему на печатной плате и межсоединения между ними.

Рисунок 11.7. Печатная плата с микросхемами, поддерживающими стандарт boundary scan



На рисунке 11.8 изображена схема типичной boundary scan ячейки. В нормальном режиме данные со входов микросхемы передаются в ее внутреннюю часть, которые далее поступают на ее выходы.

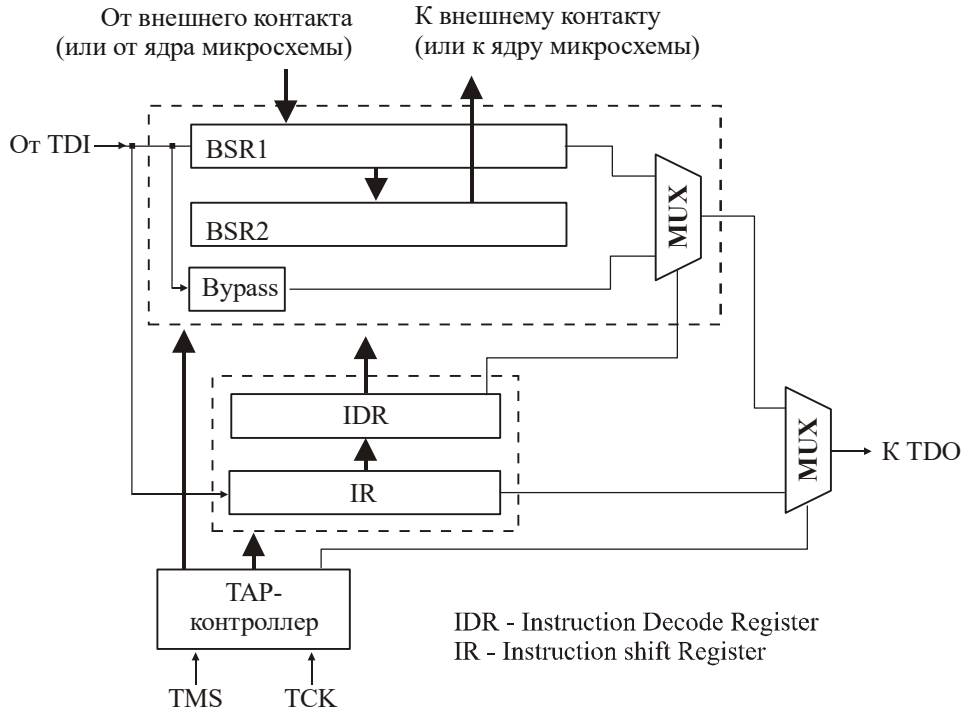
Рисунок 11.8. Ячейка boundary scan



В режиме сдвига последовательные данные из предыдущей ячейки поступают в триггер Q1, в то время как информация, хранящаяся в Q1, передается в следующую boundary scan ячейку. После того, как значение в Q2 будет обновлено состоянием Q1, оно может передаваться в микросхему или на ее выход.

На рисунке 11.9 показана основная boundary scan архитектура, которая реализуется в каждой boundary scan микросхеме. Boundary scan регистр (BSR) разделен на две части. Первый – BSR1 состоит из триггеров Q1 boundary scan ячейки. Второй – BSR2 представлен Q2 триггерами, данные в которые могут быть параллельно загружены из BSR1 при поступлении сигнала Update. Данные с последовательного входа подаются в BSR1, через bypass регистр или в командный регистр (instruction register). Boundary scan устройство может также включать необязательный 32-битный регистр идентификационного кода. TAP-контроллер каждой микросхемы представляет собой автомат (рисунок 11.10), где TMS – его вход. Последовательность из нулей и единиц, поступающая на TMS, определяет, будут ли TDI данные переданы в командный регистр или в boundary scan ячейки. TAP-контроллер и командный регистр оперируют boundary scan ячейками.

Рисунок 11.9. Основная boundary scan архитектура



Последняя версия спецификации boundary scan – IEEE 1149.1-2001 – определяет четыре обязательных и шесть дополнительных (необязательных) команд. К обязательным относятся:

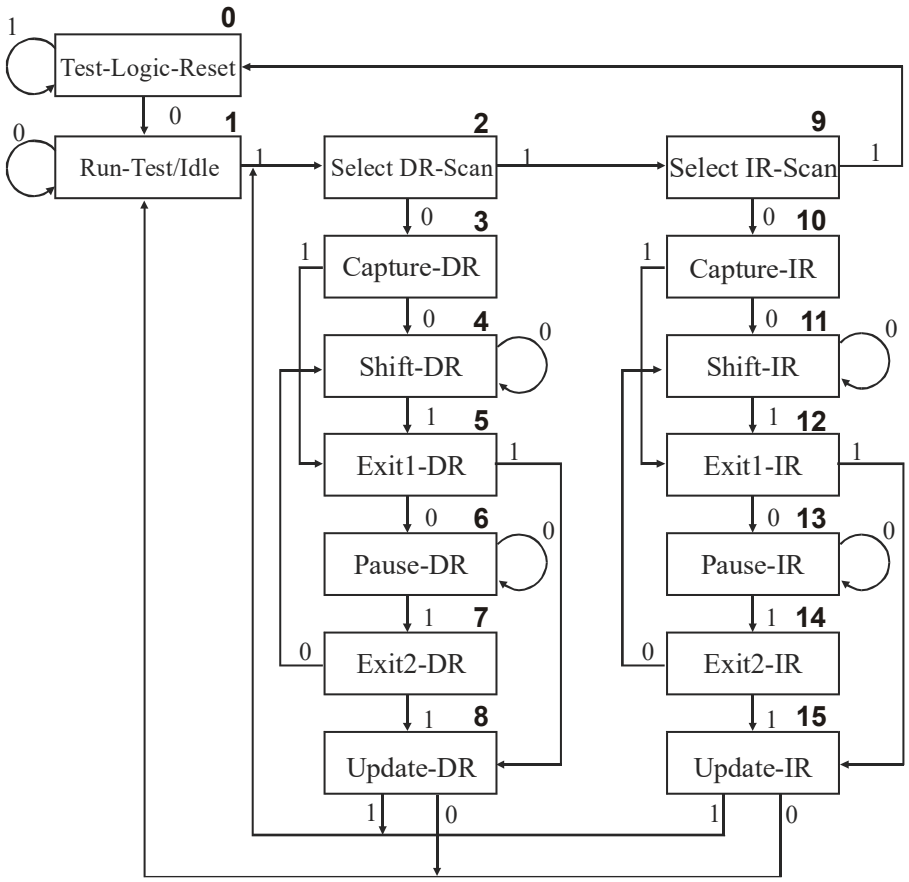
– EXTEST. Команда позволяет выполнять тестирование межсоединений на уровне платы, а также допускает тестирование групп компонентов, не поддерживающих boundary scan тестирование. Тестовые последовательности помещаются в BSR-регистр, затем передаются на выходы микросхемы. Данные со входов собираются в BSR регистр.

– BYPASS. Эта команда позволяет последовательности данных, поступающие на TDI, пропустить через 1-битовый bypass регистр микросхемы в обход BSR. Таким образом , одна или несколько микросхем на плате могут быть пропущены, в то время как другие будут тестироваться.

– SAMPLE. Команда используется для сканирования boundary scan регистра, не мешая нормальному функционированию внутренней логики. Данные поступают из ядра микросхемы на ее внешние контакты и наоборот без влияния на них boundary scan логики.

– PRELOAD. Подобна команде SAMPLE, за исключением того, что она позволяет вводить тестовые последовательности через BSR.

Рисунок 11.10. Граф переходов для TAP-контроллера



Дополнительные команды:

– INTEST. Данная команда позволяет проверять ядро микросхемы, подавая тесты через boundary scan регистр. Тестовая последовательность размещается в BSR и занимает место данных, идущих со входов микросхемы, а реакции на тест загружаются в BSR.

– INCODE. Позволяет читать данные из необязательного 32-битного идентификационного регистра, если он реализован, содержащего информацию производителя об устройстве.

– USERCODE. Вариант команды INCODE. Для устройств, имеющих двойное авторство, например CPLD и FPGA, команда позволяет загружать альтернативные данные в 32-битный идентификационный регистр.

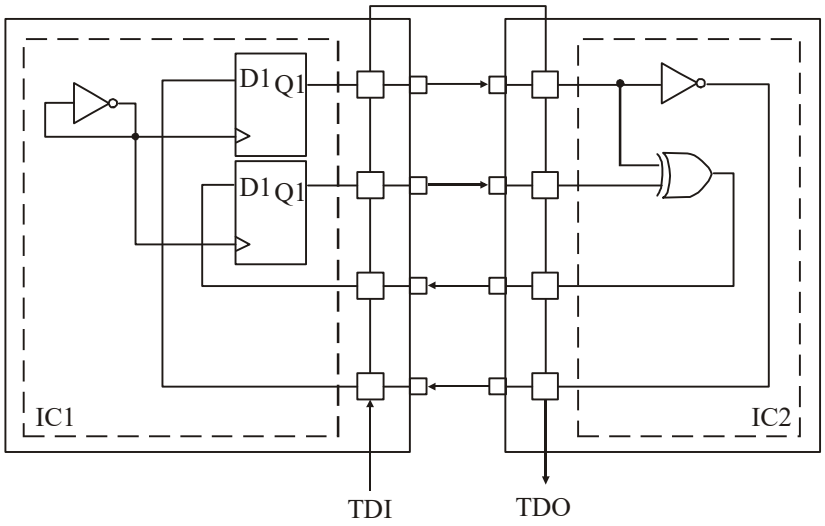
– RUNBIST, необязательный. Команда запускает самотестирование (BIST), встроенное в микросхему. Далее объясняется, как BIST может быть использовано для генерации тестовых последовательностей и проверки результатов тестирования. Могут иметь место другие, необязательные, определенные пользователем, команды.

– CLAMP. Применяется вместе с командой PRELOAD, позволяющей загружать предопределенные данные в ячейки boundary scan. Команда CLAMP передает эти значения на выходы устройства, после чего выбирает регистр Bypass. Может быть использована для удержания определенных значений на выходах устройства, например, для того, чтобы избежать конфликта шины.

– HIGHZ. Подобна команде CLAMP, но применяется для установки выходов в состояние высокого импеданса.

Следующий простой пример иллюстрирует, как с помощью команд SAMPLE/PRELOAD и EXTEST может быть протестировано соединение двух микросхем. Тестирование определяет разрывы и замыкания в линиях трассировки. Обе микросхемы, как показано на рисунке 11.11, имеют два входа и два выхода. Тестовые последовательности поступают в регистр BSR через вход TDI. Затем данные через входы микросхемы параллельно загружаются в BSR и считываются через выход TDO. Допустим, что командный регистр каждой микросхемы имеет длину три бита и командам соответствуют коды: EXTEST – 000 и SAMPLE/PRELOAD – 001. Ядро первой микросхемы IC1 представляет собой инвертор, подключенный как осциллятор к двум триггерам, второй IC2 – инвертор и XOR-вентиль. Вместе две микросхемы IC образуют двухбитный счетчик.

Рисунок 11.11. Межсоединение, тестируемое методом boundary scan



Тестирование соединения между двумя микросхемами выполняется следующим образом:

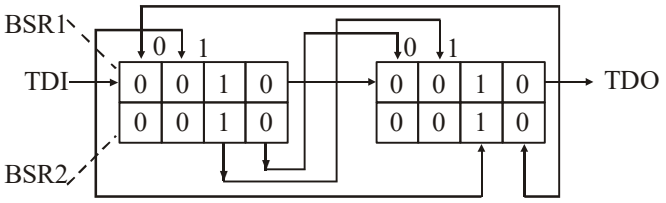
1. На вход *TMS* последовательно подаются значения (11111) для сброса TAP-автомата в состояние Test-Logic-Reset. TAP-контроллер спроектирован таким образом, что последовательность из пяти единиц сбрасывает его, независимо от предыдущего состояния. Альтернативно, если это возможно, может быть получено значение TRST.
2. Установив команду SAMPLE/PRELOAD, сканируются обе микросхемы IC с помощью последовательностей для TMS и TDI, приведенных ниже:

```
State:  0  1  2  9  10 11 11 11 11 11 11 12 15 2
TMS:    0  1  1  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  -  -  -  1  0  0  1  0  0  —  -
```

   Номера состояний относятся к рисунку 11.10.
   TMS последовательность 01100 устанавливает TAP-контроллер в Shift-IR состояние. В этом состоянии копия SAMPLE/PRELOAD команды (код 001) помещается в командный регистр обеих микросхем. В состоянии Update-IR команды загружаются в регистр дешифрации команд. Затем TAP возвращается обратно в Select DR-scan состояние.
3. Предварительная загрузка тестового набора в микросхему осуществляется с помощью следующих последовательностей для TMS и TDI:

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2
TMS:    0  0  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  0  1  0  0  0  1  0  0  -  -
```

В состоянии Shift-DR данные заносятся в BSR1, затем переносятся в регистр BSR2 в состоянии Update-DR. Результат имеет следующий вид:



4. Ввод в обе микросхемы команды EXTEST с помощью следующей входной последовательности:

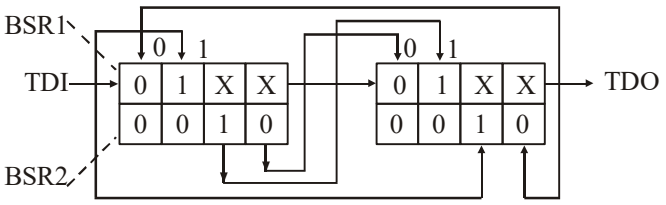
```
State:  2  9  10 11 11 11 11 11 11 12 15 2
TMS:    1  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  -  0  0  0  0  0  0  -  -
```

Команда EXTEST (000) заносится в регистр (instruction register) в состоянии Shift-IR и загружается в регистр дешифрации команды (instruction decode register) в состоянии Update-IR. В этот момент предварительно загруженные тестовые даные поступают на выходные контакты и передаются на входы соседней микросхемы через проводники печатной платы.

5. Захват результатов теста со входов микросхемы. Следующая последовательность используется для вывода данных через выход TDO и ввода второй тестовой последовательности:

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2
TMS:    0  0  0  0  0  0  0  0  0  1  1  1
TDI:    -  -  1  0  0  0  1  0  0  0  -  -
TDO:    -  -  x  x  l  O  x  x  1  O  -  -
```

Данные со входных контактов загружаются в регистр BSR1 в состоянии Capture-DR. В этот момент, если не было обнаружено никаких ошибок, регистры BSR должны содержать следующую информацию:



где X обозначает несущественную информацию для теста.

В состоянии Shift-DR результаты теста выталкиваются из регистра BSR1, так как в него заносятся новые данные. В состоянии Update-IR новые данные заносятся в регистр BSR2.

6. Считывание результатов тестирования. Для выдачи данных из микросхемы через выход TDO и занесения нулей используется такая последовательность:

```
State:  2  3  4  4  4  4  4  4  4  4  5  8  2  9  0
TMS:    0  0  0  0  0  0  0  0  0  1  1  1  1  1
TDI:    -  -  0  0  0  0  0  0  0  0  -  -  -  -
TDO:    -  -  x  x  O  l  x  x  O  l  -  -  -  -
```

Данные с входных контактов загружаются в BSR1 в состоянии Capture-DR. Затем они выталкиваются наружу в состонии Shift-DR и в BSR2 загружаются нули в состоянии Update-IR. Контроллер возвращается в состояние Test-Logic-Reset и возобновляется нормальное функционирование микросхем. Тест межсоединений выполнился успешно, если полученные последовательности соответствуют описанным выше.

На рисунке 11.12 приведен VHDL-код базовой boundary scan архитектуры с рисунка 11.9. В нем с помощью 3-битового регистра реализованы только три обязательных команды: EXTEST, SAMPLE/PRELOAD и BYPASS. Эти команды имеют коды 000, 001 и 111 соответственно. Число ячеек в регистре BSR описывает generic-константа. Параметр CellType является бит-вектором, указывающим для каждой ячейки, является ли она входом или выходом. Оператор case реализует автомат TAP-контроллера. Код команды заносится и перегружается в регистр IDR в состояниях Capture-IR, Shift-IR и Update-IR. Команды выполняются в состояниях Capture-DR, Shift-DR и Update-DR. Действия, выполняемые в этих состояниях, определяются загруженными командами. Обновление регистров и изменение состояний выполняется по переднему фронту синхросигнала TCK. VHDL-код реализует большинство функций, указанных в стандарте IEEE boundary scan, но он не соответствует полностью стандарту.

Рисунок 11.12. VHDL-код для основной архитектуры boundary scan

```
-- VHDL-код для boundary scan архитектуры с рисунка 11.9
entity BS_arch is
   generic (NCELLS: natural range 2 to 120 := 2);
         -- number of boundary scan cells
   port (TCK, TMS, TDI: in bit;
         TDO: out bit;
         BSRin: in bit_vector(1 to NCELLS);
         BSRout: inout bit_vector(1 to NCELLS);
         cellType: in bit_vector(1 to NCELLS));
              -- '0' for input cell, '1' for output cell
end BS_arch;
architecture behavior of BS_arch is
   signal IR,IDR: bit_vector(1 to 3);        -- регистры команд
   signal BSR1,BSR2: bit_vector(1 to NCELLS);-- BSRi
   signal BYPASS: bit;                       -- регистр bypass
   type TAPstate is (TestLogicReset, RunTest_Idle, SelectDRScan,
      CaptureDR, ShiftDR, Exit1DR, PauseDR, Exit2DR, UpdateDR,
      SelectIRScan, CaptureIR, ShiftIR, Exit1IR, PauseIR,Exit2IR,
       UpdateIR) ;
   signal St: TAPstate;           --  состояние TAP-контроллера
begin
   process (TCK)
   begin
      if (TCK= '1') then
         -- TAP Controller State Machine
         case St is
         when TestLogicReset =>
            if TMS='0' then St<=RunTest_Idle;
                       else St<=TestLogicReset;  end if;
      when RunTest_Idle =>
         if TMS='0' then St<=RunTest_Idle;
                       else St<=SelectDRScan;    end if;
      when SelectDRScan =>
         if TMS='0' then St<=CaptureDR;
                       else St<=SelectIRScan;    end if;
      when CaptureDR =>
            if IDR = "111" then BYPASS <= '0';
               -- EXTEST (входные ячейки получают данные со входов)
         elsif IDR = "000" then
         BSR1 <= (not CellType and BSRin) or (CellType and BSR1);
         elsif IDR = "001" then     -- SAMPLE/PRELOAD
            -- данные со входов поступают во все ячейки
                 BSR1<= BSRin; end if;
            if TMS='0' then St<=ShiftDR;
                       else St<=Exit1DR; end if;
         when ShiftDR =>
```

```
                -- передача даных через bypass регистр
            if IDR = "111" then BYPASS <= TDI;
                        else  BSR1 <= TDI & BSR1(1 to NCELLS-1);
            end if;
            -- сдвиг данных в BSR
            if TMS='0' then St<=ShiftDR; else St<=Exit1DR; end if;
          when Exit1DR =>
            if TMS='0' then St<=PauseDR; else St<=UpdateDR; end if;
          when PauseDR =>
            if TMS='0' then St<=PauseDR; else St<=Exit2DR; end if;
          when Exit2DR =>
            if TMS='0' then St<=ShiftDR; else St<=UpdateDR; end if;
          when UpdateDR =>
            if IDR = "000" then
       --EXTEST (обновление выходного регистра для выходных ячеек)
            BSR2 <= (CellType and BSR1) or (not CellType and BSR2);
            elsif IDR = "001" then       -- SAMPLE/PRELOAD
            -- обновление выходного регистра для всех ячеек
               BSR2 <= BSR1; end if;
            if TMS='0' then St<=RunTest_Idle;
                       else St<=SelectDRScan;  end if;
          when SelectIRScan =>
            if TMS='0' then St<=CaptureIR;
                       else St<=TestLogicReset; end if;
          when CaptureIR =>

  --загрузка в младшие биты регистра IR значения 01 определяется стандартом
            IR <= "001";
            if TMS='0' then St<=ShiftIR; else St<=Exit1IR; end if;
          when ShiftiR =>
            IR <= TDI & IR(1 to 2);   --shift in instruction code
            if TMS='0' then St<=ShiftIR; else St<=Exit1IR; end if;
          when Exit1IR =>
            if TMS='0' then St<=PauseIR; else St<=UpdateIR; end if;
          when PauseIR =>
            if TMS='0' then St<=PauseIR; else St<=Exit2IR; end if;
          when Exit2IR =>
          if TMS='0' then St<=ShiftIR; else St<=UpdateIR;   end if;
          when UpdateIR =>
          IDR <= IR;      --  обновление регистра дешифрации команд
            if TMS='0' then St<=RunTest_Idle;
                       else St<=SelectDRScan; end if;
          end case;
       end if;
    end process;
    TDO <= BYPASS when St = ShiftDR and IDR = "111"   -- BYPASS
          else BSR1(NCELLS) when St=ShiftDR -- EXTEST or SAMPLE/
PRELOAD
          else IR(3) when St=ShiftIR;
    BSRout <= BSRin when (St = TestLogicReset or not (IDR = "000"))
          else BSR2;                          -- define cell outputs
end behavior;
```

На рисунке 11.13 приведен VHDL-код, реализующий тестирование межсоединений для рисунка 11.11. Константы TMSpattern и TDIpattern содержат входные комбинации, описанные в шагах 2–6. Для микросхем IC1 и IC2 реализуются копии базовой архитектуры boundary scan. Затем описываются внешние соединения и внутренняя логика каждой микросхемы. Внуренняя частота выбрана произвольным образом, чтобы она отличалась от частоты сигнала тестирования. В процессе тестирования выполняется внутренняя логика, затем скан-тест и снова внутренняя логика. В результате тестирования определяется правильность работы логики микросхем и межсоединений между ними.

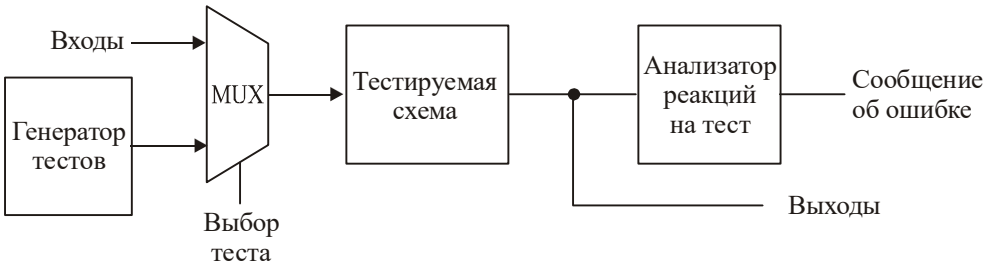
Рисунок 11.13. VHDL-код для примера тестирования межсоединений

```
entity system is
end system;

architecture IC_test of system is
  component BS_arch is
  generic (NCELLS:natural range 2 to 120 := 4);
    port (TCK, TMS, TDI: in bit;
          TDO: out bit;
          BSRin: in bit_vector(1 to NCELLS);
          BSRout: inout bit_vector(1 to NCELLS);
          CellType: in bit_vector(1 to NCELLS));
            --'0' for input cell, '1' for output cell
  end component;
signal TCK,TMS,TDI,TDO,TD01: bit;
signal Q0, Q1, CLK1: bit;
signal BSR1in, BSR1out, BSR2in, BSR2out: bit_vector(1 to 4);
signal count: integer := 0;
constant TMSpattern: bit_vector(0 to 62) :=
"01100000001110000000001111000000011100000000011100000000111111";
constant TDIpattern: bit_vector(0 to 62) :=
"00000100100000001000100000000000000000100010000000000000000000";
begin
  BS1: BS_arch port map(TCK, TMS, TDI, TD01, BSR1in, BSR1out,
"0011");
  BS2: BS_arch port map(TCK, TMS, TD01, TDO, BSR2in, BSR2out,
"0011");
  -- each BSR has two input cells and two output cells
    BSR1in(1) <= BSR2out(4);        --внутреннее соединение
микросхемыIC1 external connections
  BSR1in(2) <= BSR2out(3);
  BSR1in(3) <= Q1;                  --внутренняя логика микро-
схемы IC1
  BSR1in(4) <= Q0;
  CLK1 <= not CLK1 after 7 ns;      -- внутренняя синхронизация
  process(CLK1)
  begin
    if (CLK1='1') then              -- D-триггер
      Q0 <= BSR1out(1);
      Q1 <= BSR1out(2);
    end if;
  end process;
  BSR2in(1) <= BSR1out(4);-- внутренние соединения микросхемы IC2
  BSR2in(2) <= BSR1out(3);
  BSR2in(3) <= BSR2out(1) xor BSR2out(2); --IC2 внутренняя логика
  BSR2in(4) <= not BSR2out(1);
  TCK <= not TCK after 5 ns;   -- синхронизация тестирования
  process
  begin
    TMS <= '1';
    wait for 70 ns;             -- выполнение внутренней логики
    wait until TCK='1';              -- выполнение сканирования
      TMS <= TMSpattern(i);
      TDI <= TDIpattern(i);
      wait for 0 ns;
      count <= i+1;
      wait until TCK='1';
    end loop;
    wait for 70 ns;             --  выполнение внутренней логики
    wait;                       -- стоп
  end process ;
end IC_test;
```

11.3. Самотестирование

В связи с возрастающей сложностью цифровых систем их тестирование становится все более трудоемкой и дорогостоящей задачей. Одним из способов удовлетворительного и практически приемлемого ее решения является включение диагностических функций в саму микросхему для целей самотестирования – Built-in Self-Test, или BIST. Рисунок 11.14 иллюстрирует общую методику использования BIST. Если выбран режим тестирования и установлен сигнал выбора теста, входные последовательности с генератора тестов поступают на схему. Реакции схемы обрабатываются в анализаторе реакций на тест, который выдает сигнал об ошибке при ее обнаружении.

Рисунок 11.14. Общая BIST-схема



Технология BIST часто используется для тестирования памяти. Регулярная структура микросхем памяти значительно облегчает генерацию тестовых последовательностей. На рисунке 11.15 представлена самотестируемая структурная схема памяти RAM.

Рисунок 11.15. Самотестируемая схема RAM

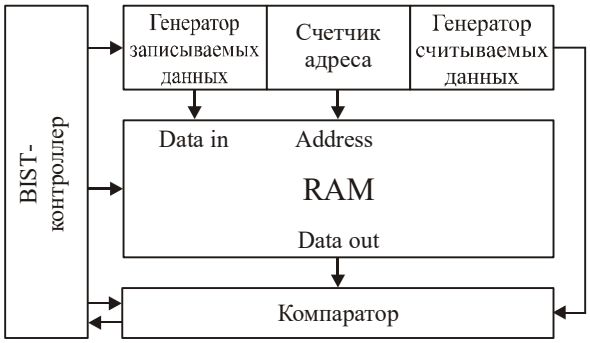
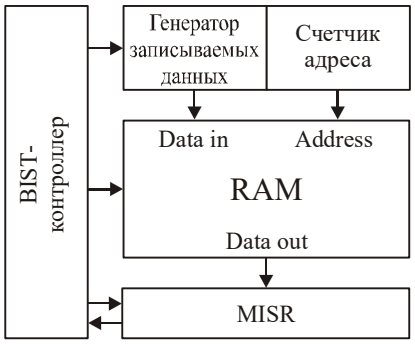


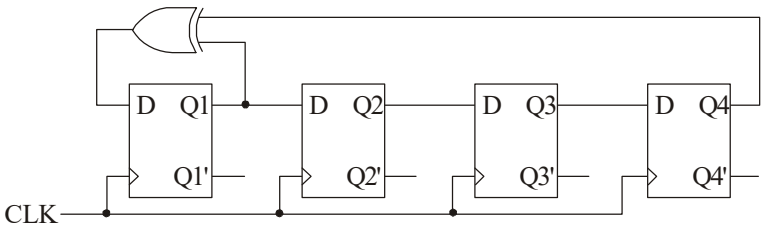
Схема тестирования может быть значительно упрощена благодаря использованию сигнатурного регистра, который преобразует выходные данные в короткую строку-код, называемый сигнатурой. Затем она сравнивается с эталонной сигнатурой, сформированной при исправном функционировании схемы. Многовходовой сигнатурный регистр MISR (multiple-input signature register) комбинирует и сжимает несколько выходных потоков в одну сигнатуру. На рисунке 11.16 представлена упрощенная самотестируемая схема RAM. Регистр MISR выполняет функции генератора считываемых данных (read-data generator) и компаратора. MISR может формировать проверяемую сумму путем сложения битов данных, хранящихся в RAM. Для тестирования ROM схема, представленная на рисунке 11.16, становится еще более простой, поскольку нет необходимости включать в нее генератор записываемых данных.

Рисунок 11.16. Самотестируемая схема RAM с сигнатурным регистром



Линейный сдвиговый регистр с обратными связями LFSR (linear feedback shift register) часто используется для генерации тестовых последовательностей. На рисунке 11.17 представлен пример такого LFSR. Значения с выходов первого и четвертого триггера через элемент "исключающее ИЛИ" подаются обратно на D-вход первого триггера. В общем случае в регистре LFSR значения с выходов двух или большего числа триггеров через вентиль XOR поступают на вход первого триггера. Название "линейный" (linear) связано с тем, что используемый элемент "исключающее ИЛИ" реализует функцию сложения по модулю 2, которая является линейной.

Рисунок 11.17. Четырехбитовый линейный сдвиговый регистр с обратными связями



При соответствующем выборе выходов, подключаемых к элементу "исключающее ИЛИ", можно генерировать 2^n-1 различных последовательностей на n-битовом сдвиговом регистре. Возможна генерация любых наборов, за исключением вектора, состоящего из одних нулей. Схема LFSR с рисунка 11.17 может генерировать такие последовательности:

```
1000, 1100, 1110, 1111, 0111, 1011, 0101, 1010, 1101, 0110,
0011, 1001, 0100, 0010, 0001, 1000,...
```

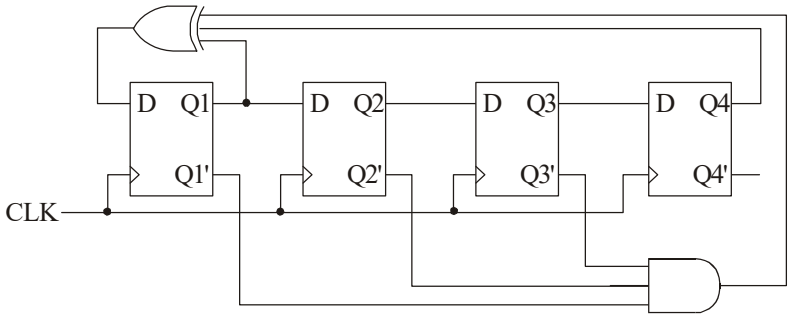
Эти последовательности идут не по порядку, а имеют некоторый элемент случайности. LFSR называется псевдослучайным генератором последовательностей или PRPG (pseudo-random pattern generator). Он очень полезен в BIST, поскольку позволяет генерировать большое число тестовых наборов при использовании небольшого количества логики. В таблице 11.1 представлены различные комбинации обратных связей для генерации 2^n-1 битов последовательностей для регистров LFSR длиной от 4 до 32 разрядов.

Если необходима последовательность, состоящая из нулей, в схему n-битового LFSR может быть добавлен элемент И с n-1 входами. Пример такой схемы для n = 4 приведен на рисунке 11.18. Если состояние регистра равно 0001, то следующим кодом будет 0000, затем – 1000. Остальная последовательность кодовых комбинаций останется прежней.

Таблица 11.1. Варианты LFSR

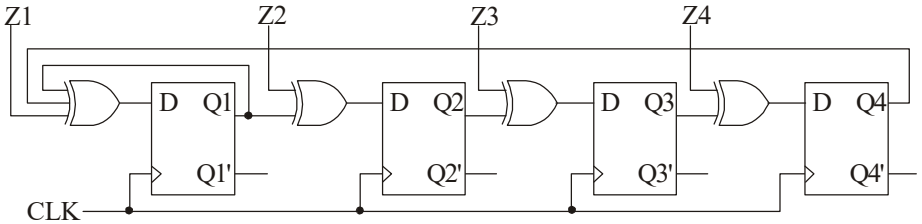
n	Обратные связи
4,6,7	$Q_1 \oplus Q_n$
5	$Q_1 \oplus Q_2 \oplus Q_5$
8	$Q_2 \oplus Q_3 \oplus Q_4 \oplus Q_8$
12	$Q_1 \oplus Q_4 \oplus Q_6 \oplus Q_{12}$
14,16	$Q_3 \oplus Q_4 \oplus Q_5 \oplus Q_6$
24	$Q_1 \oplus Q_2 \oplus Q_7 \oplus Q_{24}$
32	$Q_1 \oplus Q_2 \oplus Q_{22} \oplus Q_{32}$

Рисунок 11.18. Регистр LFSR, модифицированный для генерации состояния 0000



Регистр MISR может быть получен из LFSR путем включения в схему дополнительных элементов "исключающее ИЛИ" (рисунок 11.19). Тестовые данные $Z_1Z_2Z_3Z_4$ по синхросигналу заносятся в регистр через элементы "исключающее ИЛИ". Полученная сигнатура сравнивается с эталонной. Такой тип сигнатурного анализатора обнаруживает большинство ошибок устройства, но не все. Регистр, состоящий из n триггеров, преобразует любой входной поток сигналов в одну из возможных 2^n сигнатур. Одна из них будет соответствовать исправному поведению, остальные – неисправным состояниям. Вероятность того, что из неправильного набора будет получена сигнатура, соответствующая эталонной, равна $1/2^n$.

Рисунок 11.19. Многовходовой сигнатурный регистр



Пусть для MISR, который представлен на рисунке 11.19, исправной входной последовательностью является: 1010, 0001, 1110, 1111, 0100, 1011, 1001, 1000, 0101, 0110, 0011, 1101, 0111, 0010, 1100, которой соответствует сигнатура 1010. Любая другая последовательность, отличающаяся только одним битом от приведенной выше, будет иметь отличную от предыдущей сигнатуру. Например, если набор 0001 изменяется на 1010, то генерируется сигнатура 1000. Таким образом, можно обнаружить большинство двойных ошибок. Если же набор 0001 изменится на 1001, а 0010 на 0110, то результирующей сигнатурой будет 1010, что соответствует эталонной. Другими словами, такая ошибка не может быть определена.

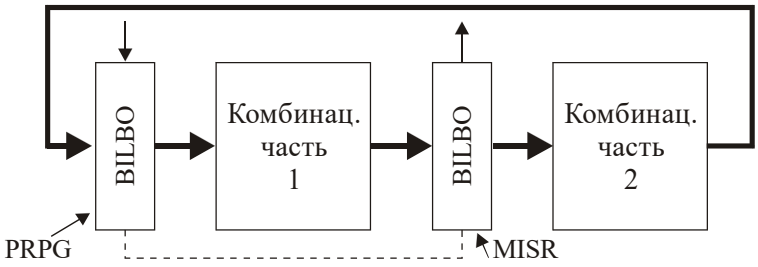
Для того чтобы сканируемую схему, изображенную на рисунке 11.4, адаптировать для BIST, сканируемый регистр модифицируется таким образом, чтобы он мог быть использован как регистр состояний, генератор тестовых последовательностей, сигнатурный анализатор и сдвиговый регистр. Если он работает в режиме сдвигового регистра, данные сдвигаются в регистр или выходят из него. Затем

часть регистра используется как генератор теста PRPG, а другая часть – как сигнатурный регистр MISR для тестирования одного из комбинационных блоков. После этого роли меняются для тестирования другого фрагмента комбинационной части. По завершении тестирования в режиме сдвигового регистра триггеры устанавливаются в состояние нормального функционирования.

Схема одного из таких устройств получила название BILBO (built-in logic block observation). На рисунке 11.20 представлено расположение регистров BILBO для тестирования схемы с двумя комбинационными фрагментами. Комбинационный блок 1 тестируется, когда первый элемент BILBO работает в режиме PRPG, а второй – в режиме MISR. Для тестирования комбинационного блока 2 роли регистров меняются. В режиме нормальной работы элементы BILBO используются как регистры, связывающие комбинационную логику. Данные сканирования заносятся в регистры или считываются из них, если оба работают как сдвиговые.

Рисунок 11.20. Самотестируемое устройство с использованием регистров BILBO:

а – тестирование комбинационной схемы 1;



б – тестирование комбинационной схемы 2

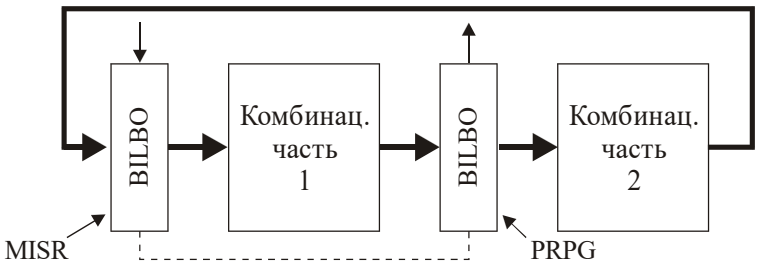


Рисунок 11.21 содержит схему четырехбитного регистра BILBO. Значения, подаваемые на входы B_1 и B_2, определяют режимы работы регистра; Z_i и Z_o – последовательный вход и выход; Z_1-Z_4 входы, на которые поступают сигналы с комбинационной части. Поведение регистра BILBO представлено уравнениями:

$$D_1 = Z_1B_1 \oplus (S_iB_2' + FB \cdot B_2)(B_1' + B_2);$$
$$D_i = Z_iB_1 \oplus Q_{i-1}(B_1' + B_2) \quad (i > 1).$$

Когда B_1=B_2=0, уравнения имеют следующий вид:

$$D_1 = S_i, \quad D_i = Q_{i-1} \quad (i > 1),$$

что соответствует режиму работы в качестве сдвигового регистра. Если B_1=0, а B_2=1, то уравнения принимают форму:

$$D_1 = FB, \quad D_i = Q_{i-1} \quad (i > 1)$$

и описывают режим работы PRPG. Если B_1=1, а B_2=0, то уравнения сокращаются до:

$$D_1 = Z_1, \quad D_i = Z_i.$$

Это нормальный режим работы устройства. Значениям $B_1=B_2=1$ соответствуют уравнения:

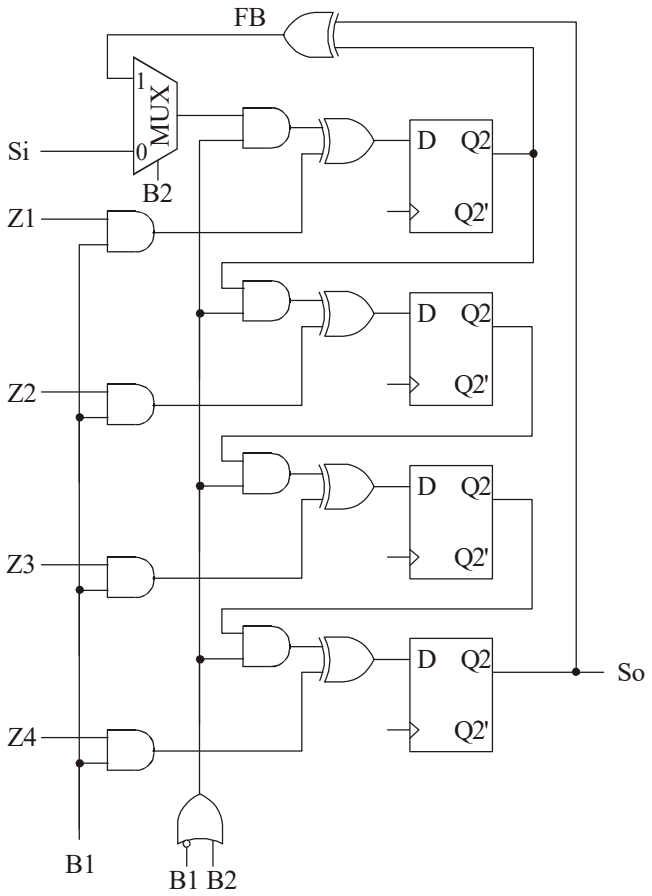
$$D_1 = Z_1 \oplus FB\,, \quad D_i = Z_i \oplus Q_{i-1}\,.$$

Они описывают режим MISR. Если обобщить сказанное выше, то регистр BILBO имеет следующие режимы работы:

B_1B_2	Режим
00	сдвиговый регистр
01	PRPG
10	нормальный
11	MISR

,

Рисунок 11.21. Четырехбитный регистр BILBO



На рисунке 11.22 представлен VHDL-код для n-битного регистра BILBO. Его разрядность описывается generic-константой NBITS, имеющей диапазон от 4 до 8. Модель функционально эквивалентна схеме с рисунка 11.21, за исключением того, что первая дополнена входом разрешения синхронизации (CE). Обратные связи для LFSR зависят от числа битов в регистре.

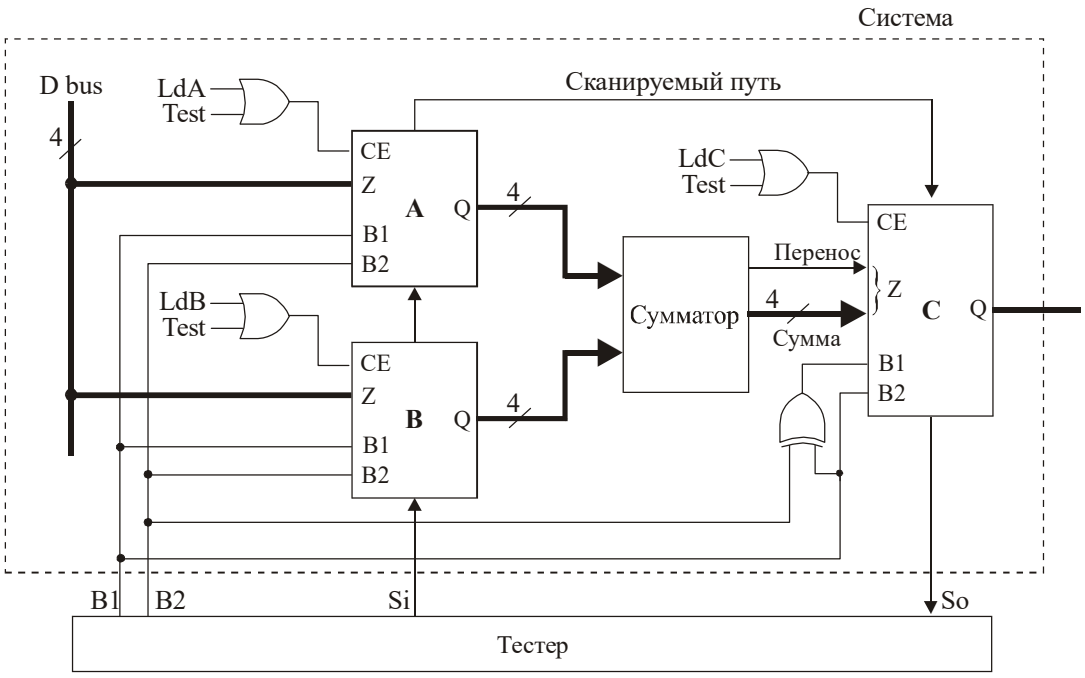
Система, изображенная на рисунке 11.23, иллюстрирует использование регистра BILBO. Регистры A и B загружаются с шины Dbus по сигналам LDA и LDB соответственно. При выполнении операции сложения сумма и перенос сохраняются в регистре C. Когда B1B2 =10, регистры работают в нормальном режиме (Test = 0) и управляются сигналами LDA, LDB и LDC. Для тестирования сумматора сначала на входы B1и B2 подаются значения 00, что переводит регистры в сдвиговый режим. Выполняется ввод начальных значений в регистры A, B и C. Затем B1 и B2 = 10, регистры A и B работают в режиме PRPG, а C – в режиме MISR. Через

15 синхротактов тестирование завершается, B1 и B2 устанавливаются в 00 и выполняется сканирование сигнатуры.

Рисунок 11.22. VHDL-код регистра BILBO

```
entity BILBO is  -- Регистр BILBO
   generic (NBITS: natural range 4 to 8:= 4);
   port (Clk, CE, Bl, B2, Si:in bit;
         So: out bit;
         Z: in bit_vector(l to NBITS);
         Q: inout bit_vector(l to NBITS));
end BILBO;

architecture behavior of BILBO is
   signal FB: bit;
begin
   FB <= Q(2) xor Q(3) xor Q(4) xor Q(NBITS) when (NBITS=8) else
         Q(2) xor Q(NBITS) when (NBITS=5) else
         Q(1) xor Q(NBITS);
   process(Clk)
      variable mode: bit_vector(l downto 0);
   begin
      if (Clk = '1' and CE = '1') then
         mode:= Bl & B2 ;
         case mode is
            when "00" =>    -- Сдвиговый регистр
               Q <= Si & Q(1 to NBITS-1);
            when "01" => --Генерация псевдослучайной последовательности
               Q <= FB & Q(1 to NBITS-1);
            when "10" => --Нормальный режим функционирования
               Q <= Z;
            when "11" => --Формирования сигнатуры
               Q <= Z(1 to NBITS) xor (FB & Q(1 to NBITS-1));
         end case ;
      end if;
   end process ;
   So <= Q(NBITS);
end;
```

Рисунок 11.23. Структура, содержащая регистр BILBO и тестер



VHDL-код для системы представлен на рисунке 11.24. В ней используется три регистра BILBO и 4-битный сумматор, модель которого была описана на рисунке 9.12. TestBench (рисунок 11.25) устанавливает регистры BILBO в начальные состояния, затем выполняется тестирование. При этом регистры A и B используются в режиме PRPG, а C – как MISR. Результирующая сигнатура сдвигается наружу и сравнивается с эталонной.

Рисунок 11.24. VHDL-код структуры, содержащей регистр BILBO и тестер

```
entity BILBO_System is
port (Clk, LdA, LdB, LdC, Bl, B2, Si: in bit;
     So: out bit;
     DBus: in bit_vector(3 downto 0);
     Output: inout bit_vector(4 downto 0));
end BILBO_System;

architecture BSysl of BILBO_System is
  component Adder 4 is
     port (A, B: in bit_vector(3 downto 0);
           Ci: in bit;
           S: out bit_vector(3 downto 0);
           Co:out bit);
  end component;
  component BILBO is
     generic (NBITS: natural range 4 to 8:= 4);
     port (Clk, CE, Bl, B2, Si: in bit;
           So: out bit;
           Z: in bit_vector(l to NBITS);
           Q: inout bit_vector(l to NBITS));
  end components;
  signal Aout, Bout: bit_vector(3 downto 0);
  signal Cin: bit_vector(4 downto O);
  alias Carry: bit is Cin(4);
  alias Sum: bit_vector is Cin(3 downto O);
  signal ACE, BCE, CCE, CBl, Test, SI, S2: bit;
begin
  Test <= not Bl or B2 ;
  ACE <= Test or LdA;
  BCE <= Test or LdB;
  CCE <= Test or LdC;
  CBl <= Bl xor B2 ;
  RegA: BILBO generic map (4)
       port map(Clk, ACE, Bl, B2, SI, S2, DBus, Aout);
  RegB: BILBO generic map (4)
       port map(Clk, BCE, Bl, B2, Si, SI, DBus, Bout);
  RegC: BILBO generic map (5)
       port map(Clk, CCE, CBl, B2, S2, So, Cin, Output);
  Adder: Adder4 port map(Aout, Bout, '0', Sum, Carry);
end BSysl;
```

Рисунок 11.25. TestBench для системы BILBO

```
-- TestBench для структуры, содержащей регистр BILBO
entity BILBO_test is
end BILBO_test;

architecture Btest of BILBO_test is

component BILBO_System is
  port (Clk, LdA, LdB, LdC, Bl, B2, Si: in bit;
        So: out bit;
        DBus: in bit_vector(3 downto 0);
        Output: inout bit_vector(4 downto 0));
end component;

signal Clk: bit:= '0';
```

```
signal LdA, LdB, LdC, Bl, B2, Si, So: bit := '0' ;
signal DBus: bit_vector(3 downto 0);
signal Output: bit_vector(4 downto 0);
signal Sig: bit_vector(4 downto 0);
constant test_vector: bit_vector(12 downto 0) := "1000110000000";
constant test_result: bit_vector(4 downto 0) := "01011";

begin
   clk <= not clk after 10 ns;
   Sys: BILBO_System
      port map(Clk, Lda, LdB, LdC, Bl, B2, Si, So, DBus, Output);
   process
   begin
      B1 <= '0'; B2 <= '0';      -- Сдвиг тестовых векторов
      for i in test_vector'right to test_vector'left loop
         Si <= test_vector(i);
         wait until (clk = '1');
      end loop;
      B1 <= '0'; B2 <= '1'; -- Использование режимов PRPG и MISR
      for i in 1 to 15 loop
         wait until (clk = '1');
      end loop;
      Bl <= '0' ; B2 <= '0' ;    -- Сканирование сигнатуры
      for i in 0 to 5 loop
         Sig <= So & Sig(4 downto 1);
            wait until (clk = '1');
      end loop;
      if (Sig = test_result) then     --Сравнение сигнатур
         report "System passed test.";
      else
         report "System did not pass test!" ;
      end if;
      wait;
   end process;
end Btest;
```

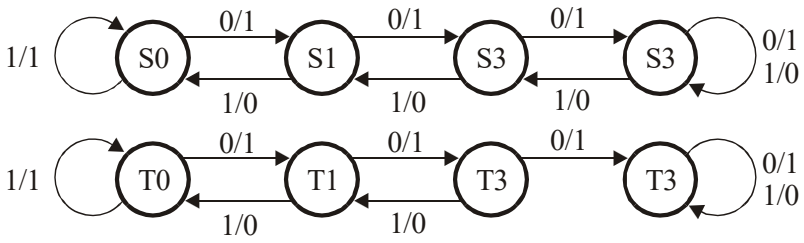
Выводы. Рассмотрены вопросы использования технологии сканирования пути и самотестирования, которые становятся все более популярными в связи с увеличением сложности и размеров интегральных микросхем и печатных плат. Максимальный эффект от применения идеологии тестопригодного проектирования достигается при рассмотрении вопросов тестируемости на ранних стадиях создания цифровых систем. Это позволит провести тестирование готовых цифровых устройств более эффективно и с меньшими материальными затратами.

11.4. Задачи

11.4.1. Изменить схему с рисунка 2.21, заменив триггеры на двухвходовые и представив ее в форме, показанной на рисунке 11.1. Определить тестовую последовательность для верификации первых двух строк таблицы переходов устройства (таблица 4.1). Для тестовой последовтельности нарисовать временные диаграммы, подобные представленным на рисунке 11.2.

11.4.2. Написать VHDL-код для двухвходового триггера и для устройства, разработанного в задаче 11.4.1. Написать TestBench для выполнения тестов, записанных для задачи 11.4.1. Сравнить результаты моделирования с временными диаграммами, созданными в предыдущей задаче.

11.4.3. Ниже приведены два графа состояний для двух автоматов. Первый соответствует исправному поведению автомата, а второй – неисправному. Пусть оба автомата установлены в свои начальные состояния S0 и T0. Определить минимальную входную последовательность, позволяющую различить эти два автомата.



11.4.4. Промоделировать тест для boundary scan (рисунок 11.13). Убедиться, что результаты соответствуют ожидаемым. Изменить код для случая, когда младший вход микросхемы IC1 подключен к земле (=0), выполнить моделирование и оценить результат.

11.4.5. Написать VHDL-код для boundary scan ячейки (рисунок 11.8). Изменить VHDL-код с рисунка 11.12, используя модель boundary scan ячейки в качестве компонента для замены поведенческого кода регистра BSR. Для реализации NCELL копий компонента использовать оператор generate. Протестировать полученный код, используя тест с рисунка 11.13.

11.4.6. Написать структурную схему для LFSR, n = 5. Затем добавить в схему И вентиль для генерации состояния 00000. Вычислить реальную последовательность состояний.

11.4.7. Написать VHDL-код для 8-битного регистра MISR, подобного представленному на рисунке 11.19.

Спроектировать самотестируемую схему для 6116 статического ОЗУ, аналогичную изображенной на рисунке 11.16. Генератор записываемых данных должен заносить тесты в такой последовательности: 00000000,10000000,11000000, ...,11111111, 01111111, 00111111,...,00000000.

Записать VHDL-код для тестирования проекта. Промоделировать систему как минимум для одного случая исправного поведения, при наличии одной ошибки, двух и трех.

ГЛАВА 12
ПРОЕКТИРОВАНИЕ УСТРОЙСТВА UART

Ранее приводились примеры простых устройств, иллюстрирующие применение VHDL в процессе проектирования. Далее рассмотрим реальный проект, который иллюстрирует, как VHDL вместе со средствами синтеза может быть применен для разработки сложных цифровых систем.

Проектируется и реализуется на FPGA асинхронный универсальный приемник-передатчик UART (Universal Asynchronous Receiver-Transmitter). Для достижения поставленной цели выполняется разработка функциональной иерархической модели устройства; создание тестов для верификации моделей устройства на различных уровнях проектирования; аппаратное моделирование проекта; синтез и реализация. В результате формируется файл для программирования микросхемы.

12.1. Анализ технического задания

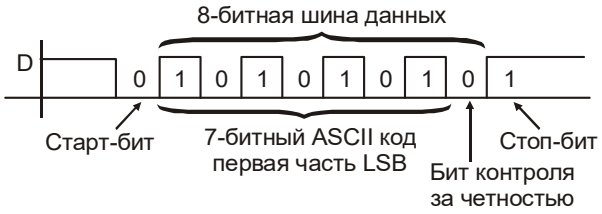
Разработать асинхронный универсальный приемник-передатчик UART – коммуникационный интерфейс для передачи и приема последовательных данных.

Большинство компьютеров и микроконтроллеров имеют один или несколько последовательных портов данных, используемых для обмена информацией с устройствами через последовательные входы/выходы, например для связи с клавиатурой или последовательным принтером. С помощью модема, присоединенного к последовательному порту, информация может быть передана или получена от удаленных объектов через телефонные линии (рисунок 12.1). Здесь представлены: последовательный коммуникационный интерфейс, который передает и получает данные, иначе UART, *RxD* – передаваемый последовательный сигнал данных, TxD – поступающий сигнал.

Рисунок 12.1. Передача последовательных данных

На рисунке 12.2 представлен формат данных, передаваемых последовательно. Поскольку здесь нет линии синхронизации, данные (D) передаются асинхронно, по одному биту. Когда информация не передается, на D сохраняется логическая единица. Чтобы обозначить начало передачи, D переходит в '0' на один такт. Это и будет старт-бит. Затем передаются 8 битов данных. Младший значащий бит передается первым. Например, для передачи текста может быть применен ASCII-код. Восьмой бит используется как бит контроля четности. После передачи данных D должен перейти в состояние '1' как минимум на один такт. Этот бит рассматривается как стоп-бит. Затем в любой момент может начаться передача другого символа.

Рисунок 12.2. Последовательный формат данных



Во время работы в режиме передачи информации UART преобразует восемь битов параллельных данных в последовательный поток, состоящий из старт-бита

(логический 0), 8 битов данных (младший бит – первый) и одного или более стоп-битов (логическая 1). Во время приема данных UART обнаруживает первый бит, получает 9 битов данных и преобразует их в параллельную форму после поступления стоп-бита. Поскольку синхроимпульс не передается, UART синхронизирует поступающий поток данных с помощью внутреннего синхросигнала.

На рисунке 12.3 представлена упрощенная схема UART, входящего в состав MC6805, MC6811 и других микроконтроллеров. Интерфейс UART соединен с 8-битной шиной данных. Используются следующие шесть восьмибитных регистров:

RSR – сдвиговый регистр для поступающих данных;

RDR – регистр получаемых данных;

TOR – регистр передаваемых данных;

TSR – сдвиговый регистр для передаваемых данных;

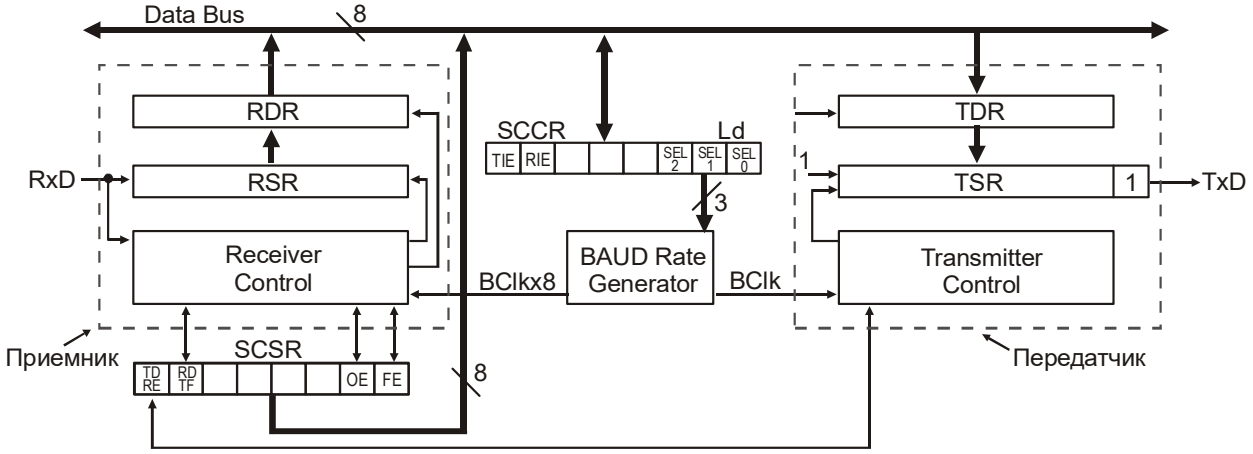
SCCR – последовательный коммуникационный управляющий регистр;

SCSR – последовательный коммуникационный регистр состояния устройства.

UART соединен с шиной данных и адресной шиной микроконтроллера таким образом, что CPU может осуществлять считывание и запись информации в регистры. Регистры RDR, TDR, SCCR и SCSR распределены в памяти, так что каждый из них является адресуемым пространством памяти. RDR, SCSR и SCCR могут передавать значения на шину данных через тристабильные буферы, TDR и SCCR могут получать значения с шины данных.

Помимо регистров, UART содержит три существенных компонента: делитель частоты BAUD (BAUD rate generator), контроллер приема (receiver control) и контроллер передачи (transmitter control). BAUD делит системную частоту, генерируя синхроимпульс (BClk) с периодом, равным времени передачи одного бита. Кроме того, здесь есть BClkX8, который имеет частоту, в 8 раз превышающую частоту BClk.

Рисунок 12.3. Блок-схема UART



Флаг *TDRE* (transmit data register empty – регистр передаваемых данных пуст) в *SCSR* устанавливается, если регистр *TDR* пуст. Когда микроконтроллер готов передавать данные, происходит следующее:

1. Микроконтроллер ждет значения *TDRE* = '1', затем загружает байт данных в *TDR* и очищает *TDRE.*
2. UART передает данные из TDR в TSR и устанавливает TDRE.
3. UART посылает старт-бит ('0'), затем выполняет сдвиг TSR вправо для передачи восьми битов данных, за которыми следует стоп-бит ('1').

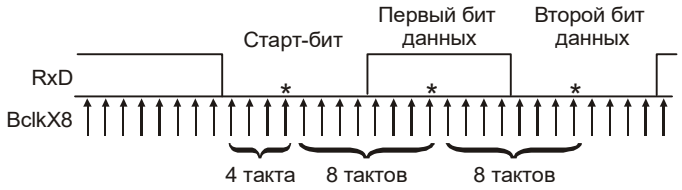
Операцию приема данных UART выполняет следующим образом:

1. Когда UART обнаруживает старт-бит, он последовательно считывает оставшиеся биты и сдвигает их в RSR.

2. После того как все биты данных и стоп сигнал будут получены, RSR загружается в RDR и выставляется флаг RDRF (Receive Data Register Full) в регистре SCSR.
3. Микроконтроллер проверяет флаг RDRF, и если он установлен, считывает данные из RDR и сбрасывает флаг RDRF.

Поток данных, поступающих с RxD, не синхронизируется внутренним синхросигналом (Bclk), поскольку если сигнал считывается по переднему фронту Bclk и при этом RxD изменяется в момент времени, близкий к активному фронту синхроимпульса, могут появляться ошибки при приеме информации. Например, чтение данных может закончиться в неправильный промежуток времени. Чтобы избежать этого, сигнал *RxD* считывается 8 раз в течение длительности каждого передаваемого бита. (Некоторые системы повторяют процедуру чтения 16 раз на один бит). Считывание выполняется по переднему фронту сигнала *BclkX8.* Стрелки на рисунке 12.4 обозначают передний фронт сигнала *BclkX8.* Для максимального качества считывания каждый бит идеально считывать, начиная со средины периода передачи бита. Когда *RxD* установится в 0, устройство ожидает 4 такта *BclkX8*. Это произойдет приблизительно в средине старт-бита. Затем, после 8 следующих периодов, будет средина первого бита данных. Чтение продолжается один раз через каждые 8 тактов *BclkX8,* пока не будет получен стоп-сигнал. На рисунке 12.4 чтение данных происходит в точках, отмеченных (*).

Рисунок 12.4. Пример RxD с BclkX8 синхросигналом



BAUD – это программируемый генератор синхроимпульсов. Три бита в *SCCR* используются для выбора одного из 8 коэффициентов BAUD. Пусть системный синхросигнал равен 8 МГц и следует получить частоту синхросигнала 300, 600, 1200, 2400,4800, 9600, 19200 и 38400 Гц. Максимальная частота *BclkX8* будет определяться как 38400 x 8 = 307200 Гц. Чтобы получить такую частоту, необходимо разделить 8 МГц на 26,04. Поскольку деление возможно только на целое число, следует допустить небольшую погрешность в коэффициенте BAUD или подкорректировать системную частоту на 7,9877 МГц для компенсации.

12.2. Разработка функциональной модели

Необходимо разработать функциональные модели блоков Transmitter, Receiver и делителя частоты. Затем создать общую модель всего устройства, включающую упомянутые компоненты.

На рисунке 12.5 представлена ГСА трансмиттера . Соответствующая последовательностная схема (SM) синхронизируется синхросигналом микроконтроллера(CLK). В состоянии IDLE SM ожидает загрузки регистра TDR и сброса бита TDRE. В состоянии SYNCH SM ожидает переднего фронта сигнала синхронизации битов (Bclk↑). Затем обнуляется младший бит регистра TSR для передачи сигнала '0' в течение одного такта. В состоянии TDATA по каждому переднему фронту сигнала Bclk↑ выполняется сдвиг регистра TSR для передачи следующего бита данных, затем счетчик битов (Bct) увеличивается на единицу. Когда Bct = 9, это означает, что 8 битов данных и стоп-сигнал переданы. Bct очищается и SM возвращается в состояние IDLE.

VHDL-код для трансмиттера UART (рисунок 12.6) получен по ГСА автомата (см. рисунок 12.5). Трансмиттер содержит TDR и TSR регистры и контроллер передачи. Он связан с TDRE и шиной данных (DBUS). Первый процесс представляет собой комбинационную схему, которая генерирует сигналы следующего состояния и управляющие сигналы. Второй процесс обновляет содержание регистров по переднему фронту

синхроимпульса. Сигнал Bclk_rising равен '1' в течение одного такта, следующего за передним фронтом сигнала Bclk. Для генерации Bclk_rising Bclk сохраняется в триггере с именем Bclk_Dlayed. Bclk_rising равен '1', если текущее значение Bclk равно '1', а предыдущее (сохраненное в Bclk_Dlayed) равно '0'. Таким образом, Bclk_rising <= Bclk **and not** Bclk_Dlayed.

Рисунок 12.5. ГСА для трансмиттера

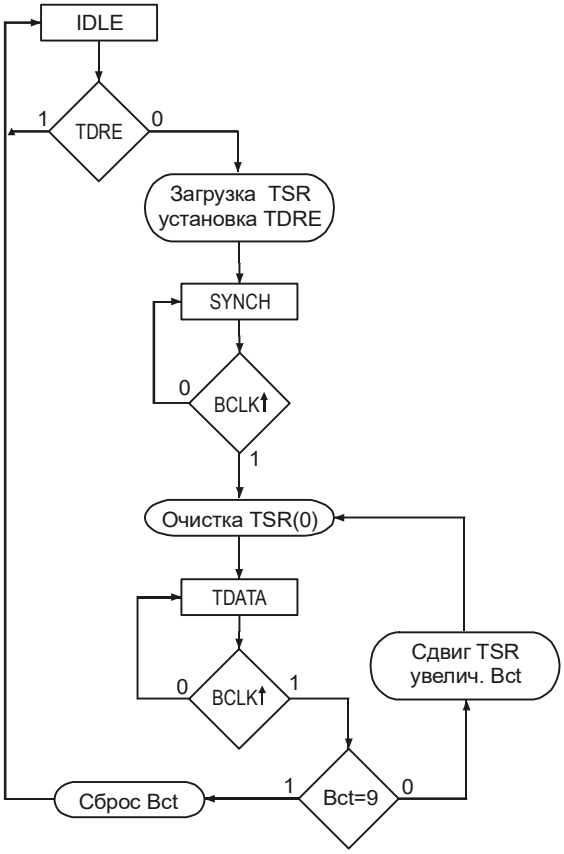


Рисунок 12.6. VHDL-код для UART-трансмиттера

```
library ieee;
use ieee.std_logic_1164.all;

entity UART_Transmitter is
port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
     DBUS:in std_logic_vector(7 downto 0);
     setTDRE, TxD: out std_logic) ;
end UART_Transmitter;

architecture xmit of UART_Transmitter is
type stateType is (IDLE, SYNCH, TDATA);
signal state, nextstate : stateType;
signal TSR: std_logic_vector (8 downto 0);-- Transmit Shift Register
signal TDR: std_logic_vector(7 downto 0); --Transmit Data Register
signal Bct: integer range 0 to 9; --counts number of bits sent
signal inc, clr, loadTSR, shftTSR, start: std_logic;
signal Bclk_rising, Bclk_Dlayed: std_logic;

begin
TxD <= TSR(0) ;
setTDRE <= loadTSR;
   -- определение переднего фронта синхросигнала
Bclk_rising <= Bclk and (not Bclk_Dlayed);
```

```
Xmit_Control: process(state, TDRE, Bct, Bclk_rising)
begin
     -- сброс управляющих сигналов
   inc <='0'; clr <='0'; loadTSR <='0'; shftTSR <='0'; start <='0';
   case state is
      when IDLE => if (TDRE ='0' ) then
         loadTSR <='1'; nextstate <= SYNCH;
         else nextstate <= IDLE; end if;
      when SYNCH =>--synchronize with the bit clock
         if (Bclk_rising ='1') then
         start <='1'; nextstate <= TDATA;
         else nextstate <= SYNCH; end if;
      when TDATA =>
         if (Bclk_rising ='0') then nextstate <= TDATA;
         elsif (Bct /= 9) then
            shftTSR <='1'; inc <='1'; nextstate <= TDATA;
         else clr <='1'; nextstate <= IDLE; end if;
   end case;
end process;

Xmit_update: process (sysclk, rst_b)
begin
   if (rst_b ='0') then
      TSR <= "111111111"; state <= IDLE; Bct <= 0; Bclk_Dlayed <='0';
   elsif (sysclk'event and sysclk ='1') then
      state <= nextstate;
      if (clr ='1') then Bct <= 0;
         elsif (inc ='1') then Bct <= Bct + 1; end if;
      if (loadTDR ='1') then TDR <= DBUS; end if;
      if (loadTSR ='1') then TSR <= TDR &'1'; end if;
      if (start ='1') then TSRout <='0'; end if;
         -- выталкивание одного бита
      if (shftTSR ='1') then TSR <='1' & TSR(8 downto 1); end if;
      Bclk_Dlayed <= Bclk; --задержка сигнала Bclk на 1 период sysclk
   end if;
end process;
end xmit;
```

Рисунок 12.7 содержит ГСА автомата SM для UART Receiver. Используются два счетчика: Ct1 считает число BclkX8 синхроимпульсов;. Ct2 – число битов, полученных после старт-бита. В состоянии IDLE SM ожидает старт-бит, затем выполняется присвоение RxD ='0' и осуществляется переход RxD в состояние Start Detected. SM ожидает передний фронт сигнала BclkX8 (BclkX8↑) и затем выполняется считывание RxD снова. Поскольку старт-сигнал должен быть равен '0', то в течение восьми тактов BclkX8 будет считан '0', при этом Ct1 еще равен 0. Таким образом, Ct1 увеличивается на 1 и SM ожидает BclkX8↑. Если RxD = '1', это свидетельствует об ошибке, а SM сбрасывает Ct1 и возвращается в состояние IDLE. Когда RxD равно '0' в четвертый момент времени, Ct1 = 3, то Ct1 очищается и автомат переходит в состояние Receive Data. В этом состоянии SM увеличивает Ctl по каждому переднему фронту сигнала BclkX8. После восьмого синхроимпульса Ct1 = 7 проверяется Ct2. Если он не равен 8, текущее значение RxD сдвигается в регистр RSR, Ct2 увеличивается на 1, и Ct1 очищается. Если Ct2 = 8, все 8 битов считаны и автомат находится в середине стоп-бита. Если RDRF = 1, это означает, что микроконтроллер еще не считал полученный байт данных. В таком случае возникает ошибка увеличения темпа работы. Тогда будет установлен флаг OE в регистре состояния и новые данные будут игнорироваться. Если RxD = '0', это означает, что стоп-бит не был определен правильно и в регистре состояний установится флаг (FE) ошибки кадрирования. Если ошибок не произошло, данные из RSR загружаются в RDR. Во всех случаях устанавливается флаг RDRF, обозначающий окончание операции считывания и сброс всех счетчиков.

VHDL-код UART-приемника (рисунок 12.8) соответствует ГСА автомата с рисунка 12.7. Приемник содержит RDR и RSR регистры и контроллер приема. Последний связан с SCSR. RDR может передавать данные на шину. Первый процесс представляет комбинационную схему, которая генерирует сигналы следующего состояния и управляющие сигналы. Второй регистр обновляет регистры по переднему фронту синхроимпульса. Сигнал BclkX8_rising равен '1' в течение одного промежутка времени, следующего после переднего фронта BclkX8. BclkX8_rising генерируется таким же способом, как и Bclk_rising.

Рисунок 12.7. ГСА автомата UART-приемника

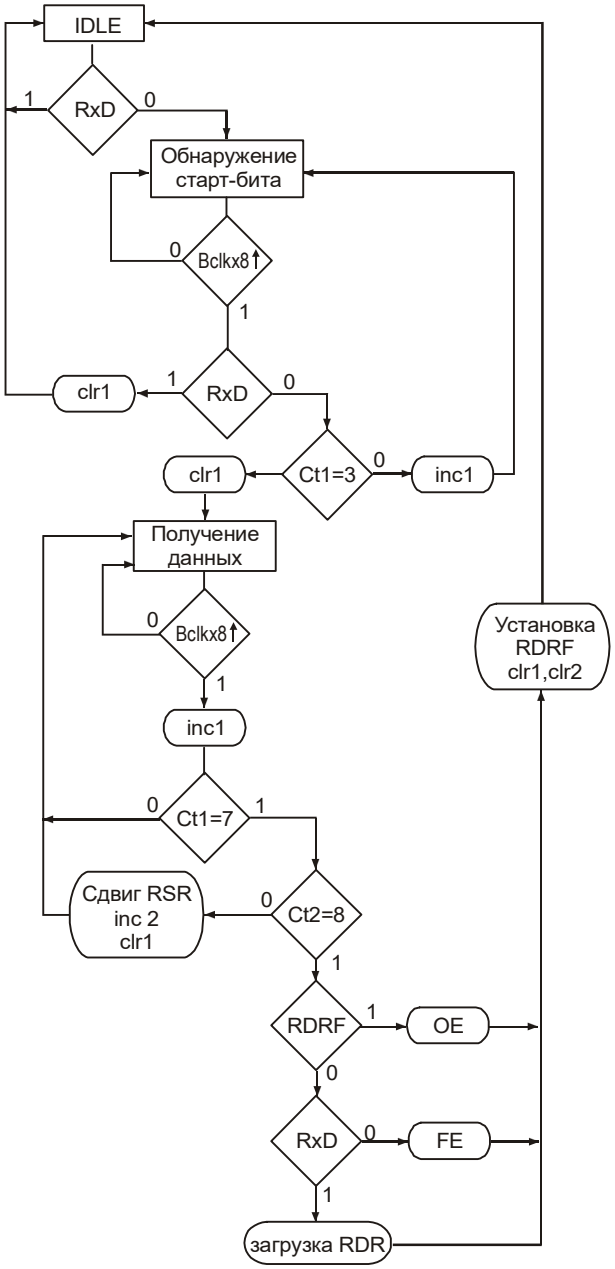


Рисунок 12.8. VHDL-код для UART-приемника

```
library ieee;
use ieee.std_logic_1164.all;

entity UART_Receiver is
port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
     RDR: out std_logic_vector(7 downto 0);
```

```
     setRDRF, setOE, setFE: out std_logic);
end UART_Receiver;

architecture rcvr of UART_Receiver is

type stateType is (IDLE, START_DETECTED, RECV_DATA);
signal state, nextstate: stateType;

signal RSR: std_logic_vector (7 downto 0);-- Receive Shift Register
signal ctl : integer range 0 to 7;
               -- обозначает момент считавания данных со входа RxD
signal ct2 : integer range 0 to 8;-- счетчик числа полученных битов
signal inc1, inc2, clr1, clr2, shftRSR, loadRDR : std_logic;
signal BclkX8_Dlayed, BclkX8_rising : std_logic;

begin
-- определение переднего фронта сигнала BclkX8
BclkX8_rising <= BclkX8 and (not BclkX8_Dlayed);

Rcvr_Control: process(state, RxD, RDRF, ctl, ct2, BclkX8_rising)
begin
     -- сброс управляющих сигналов
   inc1 <='0'; inc2 <='0'; clr1 <='0'; clr2 <='0'; shftRSR <='0';
   loadRDR <='0'; setRDRF <='0'; setOE <='0'; setFE <='0';

   case state is
      when IDLE => if (RxD ='0' ) then nextstate <= START_DETECTED;
         else nextstate <= IDLE; end if;
      when START_DETECTED =>
         if (BclkX8_rising ='0') then nextstate <= START_DETECTED;
         elsif (RxD ='1') then clr1 <='1'; nextstate <= IDLE;
         elsif (ctl = 3) then clr1 <='1'; nextstate <= RECV_DATA;
         else inc1 <='1'; nextstate <= START_DETECTED; end if;
      when RECV_DATA =>
         if (BclkX8_rising ='0') then nextstate <= RECV_DATA;
         else inc1 <='1';
            if (ctl /= 7) then nextstate <= RECV_DATA;
               -- wait for 8 clock cycles
            elsif (ct2 /= 8) then
            shftRSR <='1'; inc2 <='1'; clr1 <='1';
                                          -- чтение следующих данных
               nextstate <= RECV_DATA;
            else
               nextstate <= IDLE;
               setRDRF <='1'; clr1 <='1'; clr2 <='1';
               if (RDRF ='1') then setOE <='1';--ошибка переполнения
               elsif (RxD ='0') then setFE <='1';--ошибка кадрирования
               else loadRDR <='1'; end if;
         end if;
      end if;
   end case;
end process;

Rcvr_update: process (sysclk, rst_b)
begin
   if (rst_b ='0') then state <= IDLE; BclkX8_Dlayed <= '0';
      ctl <= 0; ct2 <= 0;
   elsif (sysclk'event and sysclk ='1') then
      state <= nextstate;
      if (clr1 ='1') then ctl <= 0;
           elsif (inc1 ='1') then
           ctl <= ctl + 1; end if;
      if (clr2 ='1') then ct2 <= 0;
           elsif (inc2 ='1') then
```

```
            ct2 <= ct2 + 1; end if;
          if (shftRSR ='1') then RSR <= RxD & RSR(7 downto 1); end if;
            -- обновление сдвигового регистра
          if (loadRDR ='1') then RDR <= RSR; end if;
          BclkX8_Dlayed <= BclkX8;  -- задержка BclkXS на 1 период sysclk
      end if;
    end process;
    end rcvr;
```

Следующим проектируется программируемый генератор частоты. Три бита в SCCR используются для выбора одного из 8 коэффициентов BAUD. Рисунок 12.9 отображает блок-схему делителя частоты BAUD. Системный синхроимпульс (8 МГц) вначале делится на 13 с помощью счетчика. Выход со счетчика идет на 8-битный двоичный счетчик. Выходы с этого счетчика соответствуют частоте, деленной на 2, 4,..., 256. Один из следующих выходов выбирается мультиплексором. Управляющими сигналами для мультиплексора являются три младших бита регистра SCCR. Выход мультиплексора MUX соответствует сигналу BclkX8, который дальше делится на 8, образуя сигнал Bclk. Предполагая, что системная частота равна 8 МГц, генерируемую частоту можно представить таблицей 12.1.

Рисунок 12.9. Делитель частоты BAUD

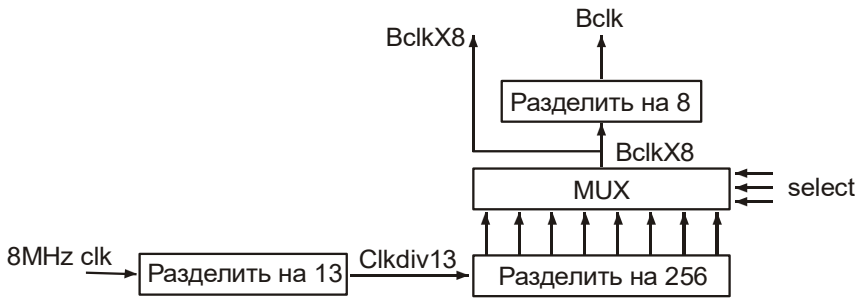


Таблица 12.1. Зависимость частоты сигнала Bclk от коэффициента деления

Выбираемые биты	Частота Bclk
000	38462
001	19231
010	9615
011	4808
100	2404
101	1202
110	601
111	300.5

Рисунок 12.10 содержит VHDL-код для генератора BAUD. Первый процесс по переднему фронту синхроимпульса увеличивает на единицу счетчик divide-by-13. Второй процесс увеличивает содержимое счетчика divide-by-256 по переднему фронту сигнала *Clkdivl3*. Параллельный оператор генерирует выход мультиплексора *BclkX8.* Третий процесс увеличивает счетчик divide-by-8 по переднему фронту *BclkX8*, образуя сигнал *Bclk*.

Рисунок 12.10. VHDL-код генератора частоты BAUD

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.std_logic_unsigned.all -- исп. оператора '+' и функ. CONVER_INT

entity clk_divider is port(Sysclk, rst_b: in std_logic;
   Sel: in std_logic_vector(2 downto 0);
   BclkX8: buffer std_logic;
   Bclk: out std_logic);
end clk_divider;

architecture baudgen of clk_divider is
signal Ctr1: std_logic_vector (3 downto 0):= "0000";
                                        -- счетчик деления на 13
signal ctr2: std_logic_vector (7 downto 0):= "00000000";
                                        -- счетчик деления на 256
signal ctr3: std_logic_vector (2 downto 0):= "000";
                                        -- счетчик деления на 8
signal Clkdivl3, Bclkx8_in: std_logic;

begin
process (Sysclk, rst_b)   -- деление синхросигнала на 13
begin
   if rst_b='0' then  Ctr1<= "0000";
   elsif (Sysclk'event and Sysclk = '1') then
      if (Ctr1 = "1100") then Ctr1 <= "0000";
      else Ctr1 <= Ctr1 + 1; end if;
   end if;
end process;
Clkdivl3 <= Ctr1(3);

process (Clkdivl3, rst_b)
begin
   if rst_b='0' then
      ctr2<= "00000000";
   elsif (rising_edge(Clkdivl3)) then
      ctr2 <= ctr2 + 1;
   end if;
end process;

BclkX8_in <= ctr2(bit_vec2int(sel)); -- выбор коэффициента деления
Bclkx8<=Bclkx8_in;
process (BclkX8_in, rst_b)
begin
   if rst_b='0' then
        ctr3<= "000";
   elsif (rising_edge(BclkX8_in)) then
      ctr3 <= ctr3 + 1;
   end if;
end process;
Bclk <= ctr3(2);
end baudgen;
```

Для выполнения UART-проекта необходимо объединить спроектированные компоненты, соединить их с управляемыми регистрами и регистром состояния и добавить схему генерации прерываний с интерфейсом шины. Рисунок 12.11 содержит код верхнего уровня для всего проекта. SCI_IRQ – сигнал прерывания, который запрашивает CPU, когда UART-приемник или передатчик нуждается в управлении. Когда RIE (receive interrupt enable) установлен в SCCR, SCI_IRQ генерируется всякий раз, когда RDRE или OE равны '1'. Если TIE (transmit interrupt enable) установлен в SCCR, то SCIJRQ генерируется всякий раз, когда TDRE='1'.

UART связан с адресами микроконтроллера и шиной данных таким образом, что CPU может читать и писать в его регистры, когда UART выбирается сигналом

SCIsel='1'. В таблице 12.2 указано, каким образом происходит выбор регистра в зависимости от значений последних двух битов адреса и сигнала R_W. Когда UART не выбран для чтения, на шину данных подается значение высокого импеданса 'Z'.

Таблица 12.2. Выбор регистров в соответствии со значением сигналов ADDR2 и R_W

ADDR2	R_W	Action
00	1	DBUS←RDR
00	0	TDR←DBUS
01	1	DBUS←SCSR
01	0	DBUS←hi-Z
1-	1	DBUS←SCCR
1-	9	SCCR←DBUS

Рисунок 12.11. VHDL-код верхнего уровня для UART

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity UART is
  port(SCI_sel, R_W, clk, RxD, rst_b: in std_logic;
       ADDR: in std_logic_vector(1 downto 0);
       DBUS: inout std_logic_vector(7 downto 0);
       SCI_IRQ, TxD: out std_logic);
end UART;

architecture uart1 of UART is
component UART_Receiver
  port(RxD, BclkX8, sysclk, rst_b, RDRF: in std_logic;
       RDR: out std_logic_vector(7 downto 0);
       setRDRF, setOE, setFE: out std_logic);
end component;

component UART_Transmitter
  port(Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
       DBUS: in std_logic_vector(7 downto 0);
       setTDRE, TxD: out std_logic) ;
end component;

component clk_divider
  port(Sysclk, rst_b: in std_logic;
       Sel: in std_logic_vector(2 downto 0);
       BclkXS, Bclk: out std_logic);
end component ;

signal RDR : std_logic_vector(7 downto 0);-- Receive Data Registers
signal SCSR : std_logic_vector(7 downto 0);     -- Status Register
signal SCCR : std_logic_vector(7 downto 0);     -- Control Register
signal TDRE, RDRF, OE, FE, TIE, RIE : std_logic;
signal BaudSel : std_logic_vector(2 downto 0);
signal setTDRE, setRDRF, setOE, setFE, loadTDR, loadSCCR: std_logic;
signal drRDRF, Bclk, BclkX8, SCI_Read, SCI_Write : std_logic;
begin
RCVR: UART_Receiver
port map(RxD, BclkX8, clk, rst_b, RDRF, RDR, setRDRF,setOE, setFE);
XMIT: UART_Transmitter
  port map(Bclk, clk, rst_b, TDRE, loadTDR, DBUS, SetTDRE, TxD);
CLKDIV: clk_divider port map(clk, rst_b, BaudSel, BclkXS, Bclk);

-- Процесс обновляет управляющие регистры и регистры статуса
process (clk, rst_b)
begin
  if (rst_b = '0') then
    TDRE <= '1'; RDRF <= '0'; OE<= '0'; FE <= '0';
    TIE <= '0'; RIE <= '0';
```

```
   elsif (rising_edge(clk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
      RDRF <= (setRDRF and not RDRF) or (not drRDRF and RDRF);
      OE <= (setOE and not OE) or (not drRDRF and OE);
      FE <= (setFE and not FE) or (not drRDRF and FE);
   if (loadSCCR = '1') then TIE <= DBUS(7); RIE <= DBUS(6);
      BaudSel <= DBUS(2 downto 0); end if;
   end if ;
end process;

-- логика, генерирующая прерывания IRQ
   SCI_IRQ <= '1' when  ((RIE = '1' and (RDRF = '1' or OE = '1'))
                  or (TIE - '1' and TDRE = '1'))
               else '0';
-- интерфейс шины
SCSR <= TDRE & RDRF & "0000" & OE & FE;
SCCR <= TIE & RIE & "000" & BaudSel;
SCI_Read <= '1' when (SCI_sel = '1' and R_W = '0') else '0';
SCI_Write <= '1' when (SCI_sel = '1' and R_W = '1') else '0';
drRDRF <= '1' when (SCI_Read = '1' and ADDR = "00") else '0';
loadTDR <= '1' when (SCI_Write = '1' and ADDR = "00") else '0';
loadSCCR <= '1' when (SCI_Write = '1' and ADDR = "10") else '0';
DBUS <= "ZZZZZZZZ" when (SCI_Read = '0')
        -- состояний тристабильной шины при отсутствии режима чтения
   else RDR when (ADDR = "00")
        -- передача соответствующего регистра на шину
   else SCSR when (ADDR = "01")
   else SCCR; -- dbus = sccr, если addr равен "10" или "11"
end uart1;
```

12.3. Разработка TestBench для тестирования UART

12.3.1. Разработка TestBench для модуля Transmitter и его верификация

Этот компонент выполняет преобразование параллельных данных в последовательные и их передачу. Блок имеет входы: Bclk – синхросигнал, генерируемый делителем частоты BAUD; sysclk – системный синхросигнал; rst_b – сброс, TDRE – флаг Transmit Data Register Empty (регистр передаваемых данных пуст); loadTDR – сигнал, разрешающий загрузку регистра TDR; DBUS – шина. Формирует выходы: setTDRE – сигнал, указывающий на необходимость сброса флага TDRE; TxD – последовательный выход.

Сигнал rst_b, активный по низкому уровню, выполняет загрузку регистра TSR единицами и сброс автомата в начальное состояние. Поэтому в тесте его следует установить в 0, а через два синхротакта – в 1. Значение входного сигнала TDRE зависит от значений сигналов rst_b и setTDRE и равняется единице, если хотя бы один из них равен '1'. Сигнал loadTDR должен быть равен '1', разрешая считывание передаваемого вектора с шины.

Модуль Transmitter содержит два синхровхода sysclk и Bclk. При этом значение частоты сигнала Bclk зависит от sysclk. Модуль Reciever имеет еще вход синхронизации BclkX8. Sysclk – это внешний вход синхронизации устройства UART. Два других сигнала генерируются модулем Clk_divider, который делит системную частоту на определенный коэффициент. Выберем системную частоту равной 50 МГц. Тогда ее период равен 20 ns. Реальная частота Bclk меньше Sysclk, как минимум в 26 раз, но для удобства тестирования модуля Transmitter целесообразно лишь незначительно ее понизить. Например, использовать Bclk-сигнал с периодом 80 ns (12,5МГц).

На основе этих рассуждений создан TestBench для тестирования трансмиттера (рисунок 12.12). Модуль Transmitter используется как компонент в TestBench. Входные передаваемые последовательности содержатся в константе in_data. Сигнал TDRE

формируется в архитектуре проекта на основе значений сигнала setTDRE, получаемых при моделировании. Этот же TestBench и файл с результатами моделирования можно использовать для тестирования разработанного устройства после синтеза. Для подключения к TestBench VHDL-модели, сформированной в результате синтеза, предназначена конфигурация:

```
configuration TESTBENCH_FOR_UART_Transmitter of uart_transmitter_tb
is
   for TB_ARCHITECTURE
      for UUT : UART_Transmitter
         use entity work.UART_Transmitter(xmit);
      end for;
   end for;
end TESTBENCH_FOR_UART_Transmitter;
```

Она используется для связи между описанным в архитектуре компонентом и фактическим VHDL-кодом. Для автоматизации выполнения процесса тестирования предназначен следующий макрофайл:

```
acom "$DSN\src\Transmitter.vhd"
acom "$DSN\src\TestBench\UART_Transmitter_TB.vhd"
asim TESTBENCH_FOR_UART_Transmitter
wave -noreg Bclk
wave -noreg sysclk
wave -noreg rst_b
wave -noreg TDRE
wave -noreg loadTDR
wave -noreg DBUS
wave -noreg setTDRE
wave -noreg TxD
run 10 000 ns
```

Рисунок 12.12. TestBench и командный файл для тестирования трансмиттера

```
library ieee;
use ieee.std_logic_1164.all;

entity uart_transmitter_tb is
end uart_transmitter_tb;

architecture TB_ARCHITECTURE of uart_transmitter_tb is
   -- декларация компонента для тестируемого модуля
   component UART_Transmitter
   port( Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
      DBUS:in std_logic_vector(7 downto 0);

      setTDRE, TxD: out std_logic) ;
   end component;

   type arr is array (1 to 5) of std_logic_vector(7 downto 0);

   -- массив с тестовыми последовательностями
   constant in_data: arr:= ( "10101010", "01010101","11001100",
               "11111111", "00000000");

   -- Сигналы, подаваемые на тестируемый модуль
   signal Bclk : std_logic:='0';
   signal sysclk :std_logic:='0';
   signal rst_b : std_logic:='0';
   signal TDRE : std_logic;
   signal loadTDR : std_logic:='1';
   signal DBUS : std_logic_vector(7 downto 0);
```

```
   -- Наблюдаемые сигналы
   signal setTDRE : std_logic;
   signal TxD : std_logic;
   -- вспомогательные сигналы
    signal i: integer:=1;
begin
   -- реализация тестируемого модуля
   UUT : UART_Transmitter
      port map   (Bclk => Bclk,  sysclk => sysclk,  rst_b => rst_b,
                  TDRE => TDRE,loadTDR => loadTDR, DBUS => DBUS,
                  setTDRE => setTDRE, TxD => TxD );
sysclk<= not sysclk  after 10 ns; --системный синхросигнал
bclk<= not bclk after 40 ns; -- синхронизация передачи
rst_b<='1'after 40 ns;

process(sysclk, rst_b)
begin
   if (rst_b = '0') then   TDRE <= '1' ;
   elsif (rising_edge(sysclk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
   end if;
end process;
process (TDRE)
   begin
      if (TDRE'event)and (TDRE = '1') then
         if i<=5 then DBus<=in_data(i);
            i<=i+1;
         else loadTDR<='0';
         end if;
         end if;
    end process;
end TB_ARCHITECTURE;
```

12.3.2. Разработка TestBench для модуля Receiver и его верификация

При тестировании модуля Receiver необходимо формировать входной сигнал для принимаемых данных RxD. Сигнал BclkX8 имеет частоту в 8 раз выше, чем Bclk. Иными словами, его период в 8 раз короче периода сигнала Bclk. Определим частоты синхросигналов и промоделируем еще раз Transmitter, применяя измененное значение сигнала Bclk. Полученная временная диаграмма для сигнала RxD будет использоваться в TestBench для Receiver. Выберем период BclkX8 равным 40 ns (25 МГц), тогда Bclk = 40*8=320 ns (3,125 МГц).

Модуль Receiver служит для преобразования последовательного кода в параллельный. Данный компонент имеет входы: RxD – последовательный вход данных; BclkX8 – синхросигнал с частотой в 8 раз выше сигнала Bclk; sysclk – системный синхросигнал; rst_b – сигнал сброса, активный по низкому уровню; RDRF – флаг Reciever Data Register Full (регистр получаемых данных полон). Выходы: RDR – параллельный код полученных данных; setRDRF – сигнал, по которому устанавливается флаг RDRF; setOE – флаг, свидетельствующий об ошибке “регистр для приема данных полон”; setFE – флаг, свидетельствующий о неправильной длине поступаемых данных.

TestBench должен выполнять тестирование нормальной работы модуля и моделировать экстремальные ситуации, в которых генерируются сигналы ошибок.

Для первого теста сигнал RDRF будет всегда равен 0, другими словами, RDR всегда пуст и готов к приему входных данных. Синхровходы имеют частоты: sysclk – 50 МГц (20 ns), BclkX8 – 25 МГц (40 ns). Для генерации сигнала RxD используется протестированный модуль Transmitter. Частота, с которой он передает последовательные данные: Bclk = 25 МГц/8 = 3,125 МГц (320 ns). Рисунок 12.13 содержит TestBench

для тестирования модуля Receiver. Этот же TestBench может быть использован для тестирования модуля Receiver после синтеза.

Модуль Receiver будет выдавать сообщения об ошибке в первом случае, когда RDRF='1'. При этом сигнал setOE будет установлен в 1, что свидетельствует о возникновении переполнения. Другими словами, регистр RDR содержит не переданную информацию, когда в него уже нужно записывать новую. Второй ошибкой, о которой свидетельствует сигнал setFE='1', является некорректность принимаемых данных. Для тестирования этих ситуаций создан еще один файл TestBench. Входная последовательность RxD сформирована автоматически программой Active-HDL из временной диаграммы сигнала TxD, полученного после моделирования трансмиттера со входными частотами Clk=50 МГц и Bclk = 3,125 МГц. Экспорт временной диаграммы в VHDL выполняется командой Waveform>Export Waveforms. Из полученного процесса выбраны операторы, формирующие сигнал до 7000 ns. Это отвечает передаваемым данным: "10101010" и "01010101". В момент приема первого вектора будет установлен сигнал RDRF='1', чтобы промоделировать ситуацию переполнения. Диаграмма второго изменена так, чтобы она соответствовала передаче дополнительного нуля в конце вектора. Таким образом, создается ситуация, когда на вход модуля Receiver подаются неправильные входные данные. Полученный VHDL-код представлен на рисунке 12.14.

Рисунок 12.13. TestBench и командный файл для тестирования модуля Receiver

```
library ieee;
use ieee.std_logic_1164.all;

entity uart_receiver_tb is
end uart_receiver_tb;

architecture TB_ARCHITECTURE of uart_receiver_tb is
component UART_Transmitter
  port( Bclk, sysclk, rst_b, TDRE, loadTDR: in std_logic;
        DBUS:in std_logic_vector(7 downto 0);
        setTDRE, TxD: out std_logic) ;
  end component;
  -- декларация компонента для тестируемого модуля
component UART_Receiver
  port( RxD, BclkX8, sysclk, rst_b, RDRF : in std_logic;
       RDR : out std_logic_vector(7 downto 0);
       setRDRF : out std_logic;
       setOE, setFE : out std_logic );
end component;

type arr is array (1 to 5) of std_logic_vector(7 downto 0);
constant in_data: arr:= ("10101010","01010101","11001100",
                         "11111111", "00000000");
  signal Bclk,sysclk,BclkX8: std_logic:='0';
  signal rst_b: std_logic:='0';
  signal TDRE: std_logic;
  signal loadTDR: std_logic:='1';
  signal DBUS: std_logic_vector(7 downto 0);

  signal RxD : std_logic;
  signal RDRF : std_logic:='0';
-- Наблюдаемые сигналы
  signal setTDRE : std_logic;
  signal TxD : std_logic;
  signal RDR : std_logic_vector(7 downto 0);
  signal setRDRF : std_logic;
  signal setOE : std_logic;
  signal setFE : std_logic;
```

```
   signal i: integer:=1; -- номер теста

begin
   UUT1 : UART_Transmitter
      port map
         (Bclk => Bclk,  sysclk => sysclk, rst_b => rst_b,
         TDRE => TDRE, loadTDR => loadTDR, DBUS => DBUS,
         setTDRE => setTDRE,  TxD => TxD );
   UUT2 : UART_Receiver
      port map
         (RxD => RxD, BclkX8 => BclkX8, sysclk => sysclk,
         rst_b => rst_b, RDRF => RDRF, RDR => RDR,
         setRDRF => setRDRF,  setOE => setOE, setFE => setFE );
sysclk<= not sysclk  after 10 ns;   -- период=20 ns
bclkx8<= not bclkx8 after 20 ns;    -- период=40.1/4 of   bclk
bclk<= not bclk after 160 ns;   -- период=320, в 8 раз больше
                                -- чем период сигнала bclkx8
rst_b<='1'after 160 ns;   -- два синхротакта
--Формирование управляющих сигналов для модуля Transmitter
process(sysclk, rst_b)
begin
   if (rst_b = '0') then
      TDRE <= '1';
   elsif (rising_edge(sysclk)) then
      TDRE <= (setTDRE and not TDRE) or (not loadTDR and TDRE);
   end if;
end process;
   process (TDRE)
   begin
      if (TDRE'event)and (TDRE = '1') then
         if i<=5 then DBus<=in_data(i);
            i<=i+1;
         else loadTDR<='0';
         end if;
         end if;
    end process;

 RDRF<='0';     -- регист RDR всегда пуст
 RxD<=TxD; -- данные, сформированные transmitter, поступают на receiver
end TB_ARCHITECTURE;

--конфигурация
configuration TESTBENCH_FOR_UART_Receiver of uart_receiver_tb is
   for TB_ARCHITECTURE
      for UUT1 : UART_Transmitter
      use entity work.UART_Transmitter(xmit);
   --    use entity work.UART_Transmitter(beh);
      end for;
      for UUT2 : UART_Receiver
         use entity work.UART_Receiver(rcvr);
      end for;
   end for;
end TESTBENCH_FOR_UART_Receiver;

Командный файл
acom "$DSN\src\Receiver.vhd"
acom "$DSN\src\Transmitter.vhd"
acom "$DSN\src\TestBench\UART_Receiver_TB.vhd"
asim TESTBENCH_FOR_UART_Receiver
wave -noreg rst_b
wave -noreg sysclk
wave -noreg Bclk
wave -noreg BclkX8
wave -noreg DBUS
wave -noreg setTDRE
```

```
wave -noreg TxD
wave -noreg RxD
wave -noreg RDR
wave -noreg RDRF
wave -noreg setRDRF
wave -noreg setOE
wave -noreg setFE
run 25000 ns
```

Рисунок 12.14. TestBench для тестирования экстремальных ситуаций функционирования модуля Receiver

```
library ieee;
use ieee.std_logic_1164.all;

entity uart_receiver_tb_error is
end uart_receiver_tb_error;

architecture TB_ARCHITECTURE of uart_receiver_tb_error is
component UART_Receiver
  port( RxD, BclkX8, sysclk, rst_b, RDRF : in std_logic;
       RDR : out std_logic_vector(7 downto 0);
       setRDRF : out std_logic;
       setOE, setFE : out std_logic );
end component;

  signal sysclk,BclkX8: std_logic:='0';
  signal rst_b : std_logic:='0';
  signal RxD : std_logic;
  signal RDRF : std_logic:='0';
  signal TxD : std_logic;
  signal RDR : std_logic_vector(7 downto 0);
  signal setRDRF : std_logic;
  signal setOE : std_logic;
  signal setFE : std_logic;

begin
  UUT2 : UART_Receiver
     port map
        (RxD => RxD, BclkX8 => BclkX8, sysclk => sysclk,
        rst_b => rst_b, RDRF => RDRF, RDR => RDR,
        setRDRF => setRDRF,  setOE => setOE, setFE => setFE );
sysclk<= not sysclk  after 10 ns;   -- период=20 ns
bclkx8<= not bclkx8 after 20 ns;    -- период=40.1/4 of   bclk
rst_b<='1'after 160 ns;           -- два периода sysclk
RDRF<='1', '0' after 3500 ns;     -- регистр RDR пуст все время
RxD<=TxD;       -- данные с трансмиттера передаются на ресивер
 STIMULUS: process
begin
  TxD <= '1';    wait for 170 ns; --0 ps
  TxD <= '0';    wait for 640 ns; --170 ns
  TxD <= '1';    wait for 320 ns; --810 ns
  TxD <= '0';    wait for 320 ns; --1130 ns
  TxD <= '1';    wait for 320 ns; --1450 ns
  TxD <= '0';    wait for 320 ns; --1770 ns
  TxD <= '1';    wait for 320 ns; --2090 ns
  TxD <= '0';    wait for 320 ns; --2410 ns
  TxD <= '1';    wait for 960 ns; --2730 ns
  TxD <= '0';    wait for 320 ns; --3690 ns
  TxD <= '1';    wait for 320 ns; --4010 ns
  TxD <= '0';    wait for 320 ns; --4330 ns
  TxD <= '1';    wait for 320 ns; --4650 ns
  TxD <= '0';    wait for 320 ns; --4970 ns
```

```
   TxD <= '1';    wait for 320 ns; --5290 ns
   TxD <= '0';    wait for 320 ns; --5610 ns
   TxD <= '1';    wait for 320 ns; --5930 ns
   TxD <= '0';  wait for 640 ns; --6570 ns  -- увеличена задержка,
                -- моделирующая десятый знак кода, равный 0

   TxD <= '1';    wait; --6570 ns
end process;
end TB_ARCHITECTURE;

--Конфигурация
configuration TESTBENCH_FOR_UART_Receiver of uart_receiver_tb_error
is
   for TB_ARCHITECTURE
      for UUT2 : UART_Receiver
         use entity work.UART_Receiver(rcvr);
      end for;
   end for;
end TESTBENCH_FOR_UART_Receiver;
```

12.3.3. Тестирование делителя частоты

Делитель частоты используется для формирования сигналов Bclk и BclkX8, синхронизирующих передачу и прием данных устройством UART. Модуль имеет входы: Sysclk – системная частота; Sel – выбор коэффициента деления частоты. Формируемые выходы: BclkX8 и Bclk. Сигнал BclkX8 имеет в 8 раз большую частоту, чем Bclk.

Для тестирования модуля также создается TestBench. Для сигнала Sysclk устанавливается частота 8 МГц. Вектор Sel последовательно получает значения от "000" до "111" с задержками, равными двум периодам сигнала Bclk для данного коэффициента деления. Периоды полученных сигналов должны соответствовать данным, приведенным в таблице 12.3, сформированной из таблицы 12.1. TestBench приведен на рисунке 12.15. Полученная временная диаграмма сохранена и используется как эталонная для последующей верификации проекта. Ввиду того, что тестирование модуля проводилось в большом промежутке времени и полученные данные при небольшой информационной нагрузке занимают слишком большой объем на распечатке, анализ результатов удобнее провести на основе процесса, сформированного из временной диаграммы сигнала Bclk. Он содержит всю информацию о временных характеристиках и приведен на рисунке 12.16. Из него видно, что генерируемый модулем сигнал Bclk имеет описанные в таблице 12.3 временные параметры.

Таблица 12.3. Зависимость частоты сигнала Bclk от коэффициента деления

Выбираемые биты	Частота Bclk	Период Bclk
000	38462	25999,688 ns
001	19231	51999,376 ns
010	9615	104004,16 ns
011	4808	207986.689 ns
100	2404	415973.378 ns
101	1202	831946.755 ns
110	601	1663893.511 ns
111	300.5	3327787.022 ns

Рисунок 12.15. TestBench для тестирования делителя частоты

```
library ieee;
use ieee.NUMERIC_STD.all;
```

```
use ieee.std_logic_1164.all;
entity clk_divider_tb is
end clk_divider_tb;
architecture TB_ARCHITECTURE of clk_divider_tb is
component clk_divider
  port( Sysclk: in std_logic;
        rst_b: in std_logic;
        Sel: in std_logic_vector(2 downto 0);
        BclkX8: inout std_logic;
        Bclk: out std_logic);
end component;
  signal Sysclk: std_logic:='0';
  signal rst_b: std_logic;
  signal Sel: std_logic_vector(2 downto 0);
  signal BclkX8: std_logic;
  signal Bclk: std_logic;
begin
  UUT : clk_divider
     port map(Sysclk => Sysclk, rst_b => rst_b, Sel => Sel,
              BclkX8 => BclkX8, Bclk => Bclk);

Sysclk<= not Sysclk after 62.5 ns; -- период 125 ns, 8 МГц
Process
begin
  sel<="000";  wait for 51999.376 ns; --two period
  sel<="001";  wait for 103998.752 ns; --two period
  sel<="010";  wait for 208008.32 ns; --two period
  sel<="011";  wait for 416016.64 ns; --two period
  sel<="100";  wait for 832033.28 ns; --two period
  sel<="101";  wait for 1664066.56 ns; --two period
  sel<="110";  wait for 3328133.12 ns; --two period
  sel<="111";  wait for 6656266.24 ns; --two period
  -- полное время моделирования =  13207841,504 ns
end process;
end TB_ARCHITECTURE;
-- Конфигурация
configuration TESTBENCH_FOR_clk_divider of clk_divider_tb is
  for TB_ARCHITECTURE
     for UUT : clk_divider
        use entity work.clk_divider(baudgen);
     end for;
  end for;
end TESTBENCH_FOR_clk_divider;
```

Рисунок 12.16. Процесс, сформированный из временной диграммы сигнала Bclk

```
--************************************************************
--* This file is automatically generated Test Bench     *
--* template and can be used in Test Bench generator.   *
--* ACTIVE-VHDL Testbench Generator ver. 1.16.          *
--* Copyright (C) ALDEC Inc.                            *
--************************************************************
--Bellow are listed ports used in STIMULUS process,
--Begin Comment
--Bclk : sig std_logic
--13.207842 ms : END_SIMULATION_TIME
--End Comment
STIMULUS: process
begin  -- of stimulus process
--wait for <time to next event>; -- <current time>
  Bclk <= '0';    wait for 10687500 ps; --0 ps
  Bclk <= '1';    wait for 13 us; --10687500 ps
  Bclk <= '0';    wait for 13 us; --23687500 ps
  Bclk <= '1';    wait for 13 us; --36687500 ps
```

```
    Bclk <= '0';    wait for 24375 ns; --49687500 ps
    Bclk <= '1';    wait for 26 us; --74062500 ps
    Bclk <= '0';    wait for 26 us; --100062500 ps
    Bclk <= '1';    wait for 26 us; --126062500 ps
    Bclk <= '0';    wait for 48750 ns; --152062500 ps
    Bclk <= '1';    wait for 52 us; --200812500 ps
    Bclk <= '0';    wait for 52 us; --252812500 ps
    Bclk <= '1';    wait for 52 us; --304812500 ps
    Bclk <= '0';    wait for 97500 ns; --356812500 ps
    Bclk <= '1';    wait for 104 us; --454312500 ps
    Bclk <= '0';    wait for 104 us; --558312500 ps
    Bclk <= '1';    wait for 104 us; --662312500 ps
    Bclk <= '0';    wait for 195 us; --766312500 ps
    Bclk <= '1';    wait for 208 us; --961312500 ps
    Bclk <= '0';    wait for 208 us; --1169312500 ps
    Bclk <= '1';    wait for 208 us; --1377312500 ps
    Bclk <= '0';    wait for 338 us; --1585312500 ps
    Bclk <= '1';    wait for 416 us; --1923312500 ps
    Bclk <= '0';    wait for 416 us; --2339312.5 ns
    Bclk <= '1';    wait for 416 us; --2755312.5 ns
    Bclk <= '0';    wait for 676 us; --3171312.5 ns
    Bclk <= '1';    wait for 832 us; --3847312.5 ns
    Bclk <= '0';    wait for 832 us; --4679312.5 ns
    Bclk <= '1';    wait for 832 us; --5511312.5 ns
    Bclk <= '0';    wait for 1352 us; --6343312.5 ns
    Bclk <= '1';    wait for 1664 us; --7695312.5 ns
    Bclk <= '0';    wait for 1664 us; --9359312.5 ns
    Bclk <= '1';    wait for 1664 us; --11023312.5 ns
    Bclk <= '0';    wait for 520529004 ps; --12687312.5 ns
    wait;
  end process;
```

12.3.4. Тестирование полной модели проекта UART

Структурная модель устройства UART состоит из рассмотренных и протестированных модулей, именуемых: Trancmitter, Receiver, делитель частоты BAUD, а также содержит код, формирующий управляющие сигналы. Устройство подключено к шине. Прием или передача данных в UART происходит через шину. Помимо шины DBus, имеются следующие входные линии: SCI_sel – выбор микропроцессором устройства UART; R_W – выбор режима чтение/запись; ADDR – адресация регистров UART; Clk – синхронизация; rst_b – сброс; RxD – последовательные принимаемые данные. Формируемые выходы: TxD – последовательные передаваемые данные; SCI_IRQ – сигнал прерывания.

Устройство начинает работу после того, как CPU выбирает его и устанавливает сигнал SCI_sel =1. Тогда CPU может прямо читать информацию из UART-регистров и заносить в них данные. В таблице 3.2 описаны режимы передачи данных в зависимости от значений сигналов R_W и ADDR. Когда UART не выбран для чтения, на шину данных подается значение высокого импеданса 'Z'.

Чтобы убедиться в правильности функционирования спроектированного устройства, необходимо промоделировать следующие ситуации:

1. Устройство не активно. Сигнал SCI_sel =0. На шине DBus должно быть значение высокого импеданса.
2. Сброс устройства в начальное состояние. SCI_sel =1. Rst_b=0.
3. ADDR=01, R_W=1. Выполняется чтение регистра SCSR. Он должен содержать нули, которые были записаны в него операцией сброса.
4. Моделирование режима работы, когда ADDR=01, R_W=0. Шина DBus получает значение 'Z'. Другие сигналы: SCI_sel =1, Rst_b=1.
5. Чтение информации из SCCR и запись в него. Значения сигналов: ADDR=1-, R_W=1

и ADDR=1-, R_W=0, соответственно. Другие сигналы: SCI_sel =1, Rst_b=1. Для выполнения режима записи на шину подаются значения:1111111, 0000000, 01010101 и 10101010.

6. Режим передачи данных. ADDR=00, R_W=0. На шину DBus подаются тестовые последовательности. Данные поступают на выход TxD. Другие сигналы: SCI_sel =1, Rst_b=1.
7. Режим приема данных. ADDR=00, R_W=1. Последовательный код поступает на вход RxD. Полученные данные записываются в регистр RDR, а затем передаются на шину. Другие сигналы: SCI_sel =1, Rst_b=1.

Также в момент приема и передачи данных можно выполнить чтение информации из регистра SCSR.

Для моделирования первых четырех ситуаций построен первый TestBench, который представлен на рисунке 12.17. Результаты выполнения данного теста отражены в рисунке 12.18. Из него видно, что до 40 ns, пока не подан сигнал выбора устройства (пока SCI_sel =0), на шине находится значение "ZZZZZZZ". После поступления сигнала сброса rst_b=0 внутренние регистры состояния SCCR и SCSR сбрасываются в начальное состояние. В момент времени, равный 80 ns, управляющие режимом работы входы получают значения ADDR=01, R_W=1. Данные из регистра SCSR поступают на шину. Затем R_W переключается в 0 и шина снова переходит в состояние высокого импеданса.

Для моделирования 5, 6 и 7 режимов работы построены отдельные тесты (рисунки 12.19, 12.20 и 12.23 соответственно). Сигнал для RxD проверки режима приема данных (рисунок 12.23) формировался на основе временной диаграммы сигнала TxD, задаваемого на этапе выполнения теста 6. Автоматическое формирование VHDL-процесса, генерирующего заданную форму сигнала TxD, выполнено с помощью функций программы Aldec-HDL. Затем в текстовом редакторе имя TxD было заменено на RxD и полученный VHDL-код вставлен в TestBench.

На рисунке 12.21 приведен результат моделирования пятого режима работы. В регистр последовательно заносятся данные FF, 00, 55 и AA. Они передаются обратно на шину после переключения ее в режим высокого импеданса. Эти данные уже имеют значения C7, 00, 45 и 82. Изменение информации связано с тем, что разряды 3-5 регистра SCCR не используются и всегда равны 0.

Рисунок 12.22 соответствует моделированию передачи данных "10101010", "01010101", "11001100", "11111111", "00000000". Для хранения тестовых последовательностей создается константа test типа массив бит-векторов.

Результаты моделирования приема данных представлены рисунком 12.24.

Рисунок 12.17. TestBench для тестирования модуля UART

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
entity uart_tb is
end uart_tb;
architecture TB_ARCHITECTURE of uart_tb is
   component UART
      port( SCI_sel, R_W, clk, RxD, rst_b: in std_logic;
            ADDR: in std_logic_vector(1 downto 0);
            DBUS: inout std_logic_vector(7 downto 0);
            SCI_IRQ, TxD: out std_logic );
   end component;
   signal SCI_sel, clk, RxD, rst_b : std_logic;
```

```
   signal ADDR : std_logic_vector(1 downto 0);
   signal DBUS : std_logic_vector(7 downto 0);
   -- наблюдаемые сигналы
   signal SCI_IRQ, TxD : std_logic;
   -- Сигнал, используемый для завершении генерации синхросигнала.
   signal END_SIM: BOOLEAN:=FALSE;
begin
   UUT : UART
      port map(SCI_sel => SCI_sel, R_W => R_W, clk => clk,
         RxD => RxD, rst_b => rst_b,  ADDR => ADDR,   DBUS => DBUS,
         SCI_IRQ => SCI_IRQ,  TxD => TxD);
STIMULUS: process
begin  -- процесс, генерирующий входные последовательности

   SCI_sel <= '0'; wait for 40 ns; --0 ps
   rst_b <= '0'; wait for 40 ns; --40 ns
   ADDR <= "01"; R_W <= '1'; wait for 40 ns; --80 ns
   SCI_sel <= '1'; wait for 40 ns; --100 ns
   R_W <= '0';  wait for 40 ns; --140 ns
   END_SIM <= TRUE;
   wait;
end process;
CLOCK_clk : process
begin
   --генерируется синхросигнал по формуле: 0 0, 1 10000 -r 20000
   if END_SIM = FALSE then clk <= '0';  wait for 10 ns; --0 ps
      else  wait;
   end if;
   if END_SIM = FALSE then clk <= '1';  wait for 10 ns; --10 ns
      else  wait;
   end if;
end process;
end TB_ARCHITECTURE;
-- Конфигурация
configuration TESTBENCH_FOR_UART of uart_tb is
   for TB_ARCHITECTURE
      for UUT : UART
         use entity work.UART(uart1);
      end for;
   end for;
end TESTBENCH_FOR_UART;
```

Рисунок 12.18. Результаты тестирования первых четырех режимов работы

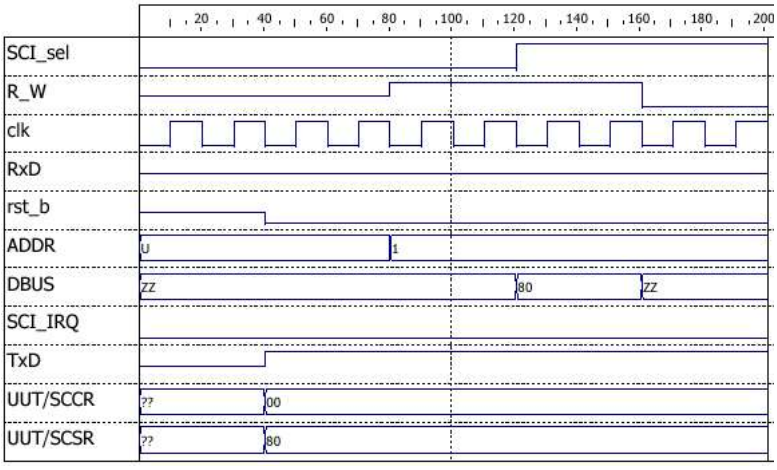


Рисунок 12.19. TestBench для тестирования модуля UART в режиме обмена информацией между шиной DBus и регистром SCCR

```
entity uart_tb5 is
end uart_tb5;

architecture TB_ARCHITECTURE of uart_tb5 is
   component UART
      port( SCI_sel, R_W, clk, RxD, rst_b: in std_logic;
            ADDR: in std_logic_vector(1 downto 0);
            DBUS: inout std_logic_vector(7 downto 0);
         SCI_IRQ, TxD : out std_logic );
   end component;

   signal SCI_sel : std_logic;
   signal R_W : std_logic;
   signal clk : std_logic:='0';
   signal RxD : std_logic;
   signal rst_b : std_logic;
   signal ADDR : std_logic_vector(1 downto 0);
   signal DBUS : std_logic_vector(7 downto 0);
   -- Наблюдаемые сигналы
   signal SCI_IRQ : std_logic;
   signal TxD : std_logic;

   -- Сигнал, используемый для завершения генерации синхросигнала.
   signal END_SIM: BOOLEAN:=FALSE;

   Type arr is array (1 to 4)of std_logic_vector(7 downto 0);
   constant test: arr:=("11111111", "00000000", "01010101",
                        "10101010");
    signal i: natural:=1;
begin
   UUT : UART
        port map   (SCI_sel => SCI_sel, R_W => R_W, clk => clk,
                    RxD => RxD, rst_b => rst_b, ADDR => ADDR,
                    DBUS => DBUS, SCI_IRQ => SCI_IRQ, TxD => TxD);
clk<=not clk after 10 ns;-- 50 МГц
SCI_Sel<='1';
rst_b<= '0', '1' after 5 ns;
Process
begin
   wait on clk until clk='1';
      wait on clk until clk='1';
   ADDR<="1-"; R_W<='0'; DBus <= test(i);
      if i < 4 then i<=i+1; else i<=1; end if;
   wait until clk'event and clk='1';
      wait until clk'event and clk='1';
   ADDR<="01"; R_W<='0'; DBus <="ZZZZZZZZ";
   wait until clk'event and clk='1';
      wait until clk'event and clk='1';
   ADDR<="1-"; R_W<='1';
end process;
end TB_ARCHITECTURE;

-- Конфигурация
configuration TESTBENCH_FOR_UART of uart_tb is
   for TB_ARCHITECTURE
      for UUT : UART
         use entity work.UART(uart1);
      end for;
   end for;
end TESTBENCH_FOR_UART;
```

Рисунок 12.20. Моделирование режима передачи данных устройством UART

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;

entity uart_tb6 is
end uart_tb6;

architecture TB_ARCHITECTURE of uart_tb6 is
  component UART
  port( SCI_sel, R_W, clk, RxD, rst_b: in std_logic;
     ADDR: in std_logic_vector(1 downto 0);
     DBUS: inout std_logic_vector(7 downto 0);
     SCI_IRQ, TxD: out std_logic );
  end component;

  signal SCI_sel: std_logic;
  signal R_W: std_logic;
  signal clk: std_logic:='0';
  signal RxD: std_logic;
  signal rst_b: std_logic;
  signal ADDR: std_logic_vector(1 downto 0);
  signal DBUS: std_logic_vector(7 downto 0);
  -- Наблюдаемые сигналы
  signal SCI_IRQ : std_logic;
  signal TxD : std_logic;
  -- Сигнал, используемый для завершения генерации синхросигнала.
  signal END_SIM: BOOLEAN:=FALSE;
  -- Тесты
  type arr is array (1 to 5) of std_logic_vector(7 downto 0);
  constant test: arr:=("10101010", "01010101", "11001100",
                       "11111111", "00000000");
  signal i: natural:=1; -- номер теста
begin
  UUT : UART
       port map   (SCI_sel => SCI_sel, R_W => R_W, clk => clk,
                  RxD => RxD,  rst_b => rst_b, ADDR => ADDR,
                  DBUS => DBUS, SCI_IRQ => SCI_IRQ, TxD => TxD);
clk<=not clk after 10 ns; -- 50 МГц
SCI_Sel<='1';
rst_b<= '0', '1' after 5 ns;
ADDR<="00"; R_W<='0';
Process
  begin
     DBus <= test(1); wait for 43300 ns;
     DBus <= test(2); wait for 43320 ns;
     DBus <= test(3); wait for 43320 ns;
     DBus <= test(4); wait for 43320 ns;
     DBus <= test(5); wait for 43320 ns;
     if i < 5 then i<=i+1; else i<=1; end if;
end process;
end TB_ARCHITECTURE;

--Конфигурация
configuration TESTBENCH_FOR_UART of uart_tb is
  for TB_ARCHITECTURE
     for UUT : UART
     use entity work.UART(uart1);
--   use entity work.UART(beh); -- для тестирования после синтеза
     end for;
  end for;
end TESTBENCH_FOR_UART;
```

Рисунок 12.21. Временные диаграммы тестирования модуля UART в режиме обмена информацией между шиной DBus регистром SCCR

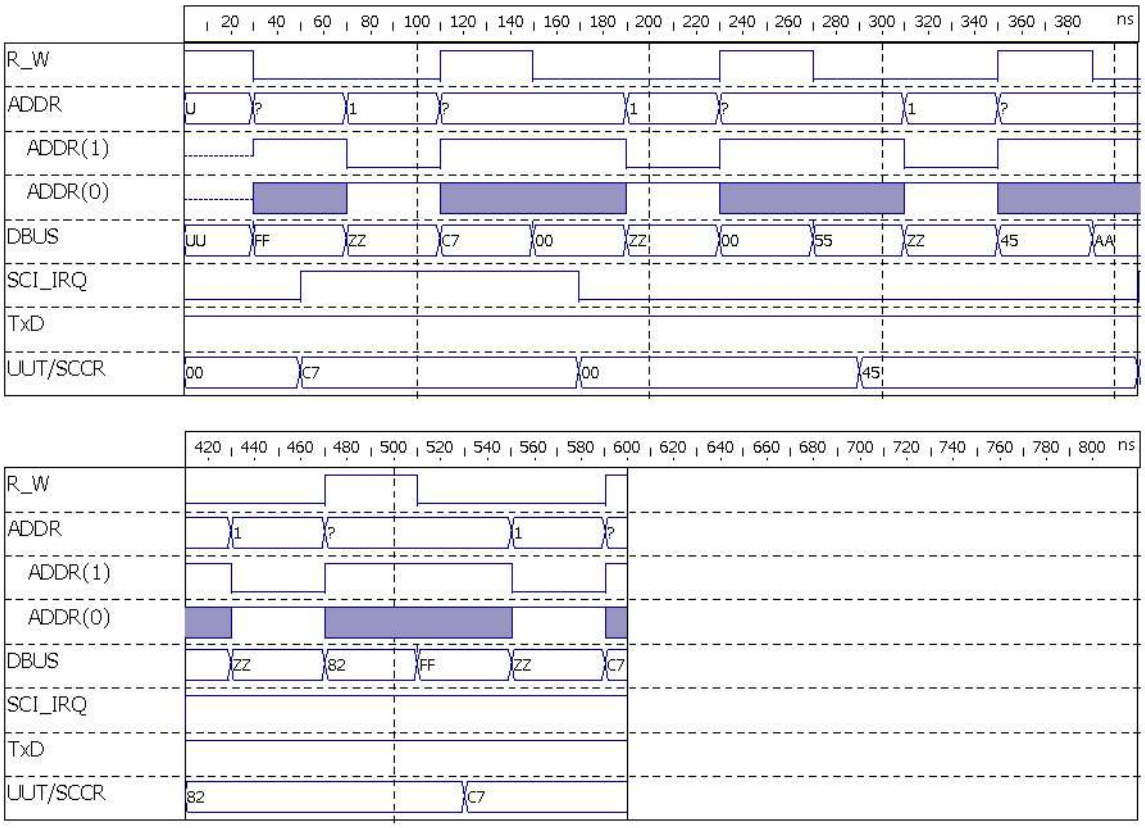


Рисунок 12.22. Временные диаграммы моделирования режима передачи данных

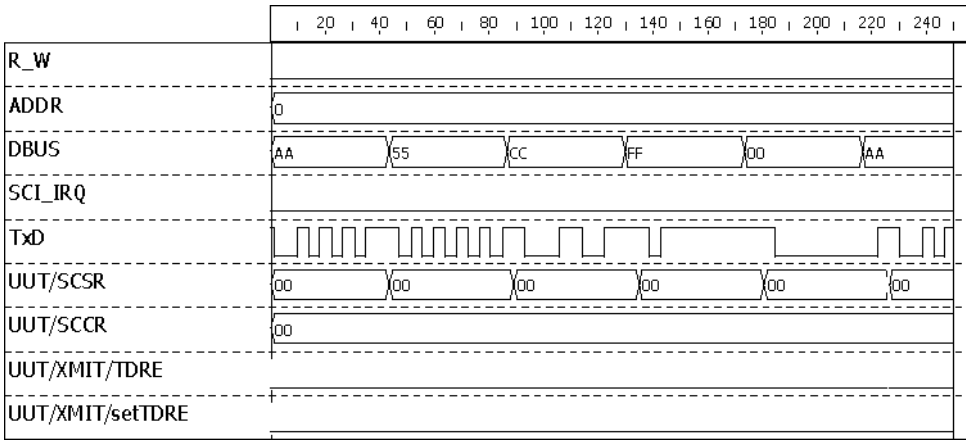


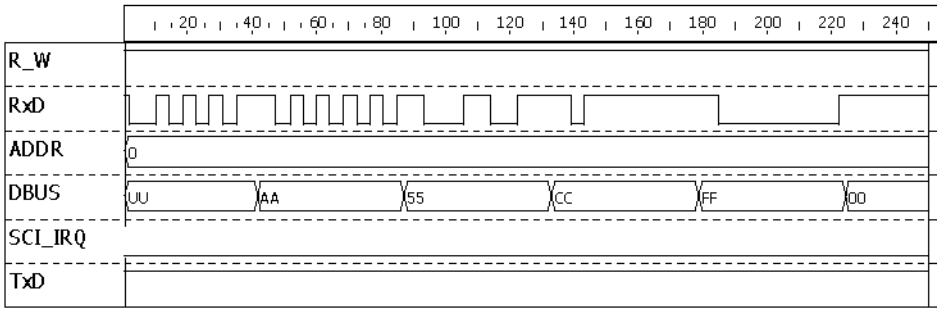
Рисунок 12.23. Моделирование режима приема данных устройством UART

```
library ieee;
use ieee.std_logic_arith.all;
use ieee.std_logic_1164.all;
entity uart_tb7 is
end uart_tb7;
architecture TB_ARCHITECTURE of uart_tb7 is
  component UART
    port( SCI_sel, R_W, clk, RxD, rst_b: in std_logic;
          ADDR: in std_logic_vector(1 downto 0);
          DBUS: inout std_logic_vector(7 downto 0);
          SCI_IRQ, TxD: out std_logic );
```

```
  end component;

  signal SCI_sel : std_logic;
  signal R_W : std_logic;
  signal clk : std_logic:='0';
  signal RxD : std_logic;
  signal rst_b : std_logic;
  signal ADDR : std_logic_vector(1 downto 0);
  signal DBUS : std_logic_vector(7 downto 0);
  -- Наблюдаемые сигналы
  signal SCI_IRQ : std_logic;
  signal TxD : std_logic;
  -- Сигнал, используемый для завершения генерации синхросигнала.
  signal END_SIM: BOOLEAN:=FALSE;

begin
  UUT : UART
    port map (SCI_sel => SCI_sel,   R_W => R_W, clk => clk,
              RxD => RxD, rst_b => rst_b, ADDR => ADDR,
              DBUS => DBUS, SCI_IRQ => SCI_IRQ, TxD => TxD);

clk<=not clk after 10 ns;-- 50 МГц
SCI_Sel<='1';
rst_b<= '0', '1' after 5 ns;
ADDR<="00"; R_W<='1';

STIMULUS: process
begin
  RxD <= '1';    wait for 1730 ns; --0 ps
  RxD <= '0';    wait for 8320 ns; --1730 ns
  RxD <= '1';    wait for 4160 ns; --10050 ns
  RxD <= '0';    wait for 4160 ns; --14210 ns
  RxD <= '1';    wait for 4160 ns; --18370 ns
  RxD <= '0';    wait for 4160 ns; --22530 ns
  RxD <= '1';    wait for 4160 ns; --26690 ns
  RxD <= '0';    wait for 4160 ns; --30850 ns
  RxD <= '1';    wait for 12480 ns; --35010 ns
  RxD <= '0';    wait for 4160 ns; --47490 ns
  RxD <= '1';    wait for 4160 ns; --51650 ns
  RxD <= '0';    wait for 4160 ns; --55810 ns
  RxD <= '1';    wait for 4160 ns; --59970 ns
  RxD <= '0';    wait for 4160 ns; --64130 ns
  RxD <= '1';    wait for 4160 ns; --68290 ns
  RxD <= '0';    wait for 4160 ns; --72450 ns
  RxD <= '1';    wait for 4160 ns; --76610 ns
  RxD <= '0';    wait for 4160 ns; --80770 ns
  RxD <= '1';    wait for 8320 ns; --84930 ns
  RxD <= '0';    wait for 12480 ns; --93250 ns
  RxD <= '1';    wait for 8320 ns; --105730 ns
  RxD <= '0';    wait for 8320 ns; --114050 ns
  RxD <= '1';    wait for 16640 ns; --122370 ns
  RxD <= '0';    wait for 4160 ns; --139010 ns
  RxD <= '1';    wait for 41600 ns; --143170 ns
  RxD <= '0';    wait for 37440 ns; --184770 ns
  RxD <= '1';    wait for 8320 ns; --222210 ns
  wait;
end process;
end TB_ARCHITECTURE;

-- Конфигурация
configuration TESTBENCH_FOR_UART of uart_tb is
  for TB_ARCHITECTURE
    for UUT : UART
    -- use entity work.UART(uart1);
```

```
            use entity work.UART(beh);
        end for;
    end for;
end TESTBENCH_FOR_UART;
```

Рисунок 12.24. Временные диаграммы моделирования режима приема данных



12.4. Синтез схемы

Современные инструментальные средства проектирования позволяют автоматически синтезировать из VHDL-кода описания моделей RTL и вентильного уровня. Примером таких программ являются FPGA Express фирмы Synopsys и Synplify фирмы Synplicity. В данной работе использовалась программа Synplify фирмы Synplicity. Эта программа сначала преобразует VHDL-описание в модель уровня регистровых передач, состоящую из типовых элементов, таких как логические элементы, триггеры, регистры, счетчики, мультиплексоры, сумматоры, автоматы. Затем полученная схема преобразуется в модель вентильного уровня и отображается в элементах, составляющих целевое устройство.

В результате синтеза генерируется VHDL- и EDIF-код. VHDL-модель используется для тестирования проекта после синтеза, EDIF – необходима для последующей реализации схемы.

Для разрабатываемого здесь устройства в качестве целевой реализации выбрана микросхема FPGA фирмы Xilinx серии Virtex 800.

12.4.1. Синтез и тестирование отдельных модулей

Проектирование отдельных модулей, именуемых: Transmitter, Receveir и делитель частоты BAUD, выполняется для проверки синтезируемости их VHDL-кодов. На этом этапе также устраняются ошибки, которые могут возникнуть в результате синтеза, оцениваются временные характеристики полученных схем.

В результате синтеза модуля Transmitter установлено, что данное устройство может работать на частоте до 104,3 МГц. Схема RTL-уровня приведена на рисунке 12.25.

Модуль Receveir может работать на частоте до sysclk = 66,3 МГц. Схема блока уровня регистровых передач изображена на рисунке 12.26.

Синтезированная схема RTL-уровня делителя частоты BAUD приведена на рисунке 12.27. Данное устройство может работать на системной частоте до 238,8 МГц На этом этапе проводится тестирование моделей модулей Transmitter, Receveir и делителя частоты BAUD, полученных в результате синтеза. Для тестирования используются те же самые файлы TestBench и командные файлы, что применялись для функционального моделирования.

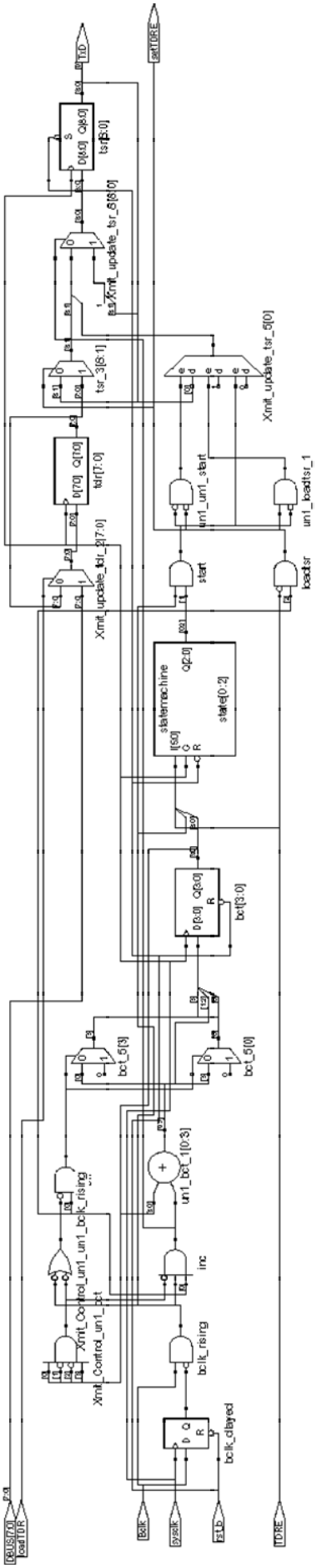


Рисунок 12.25 RTL-схема модуля Transmitter

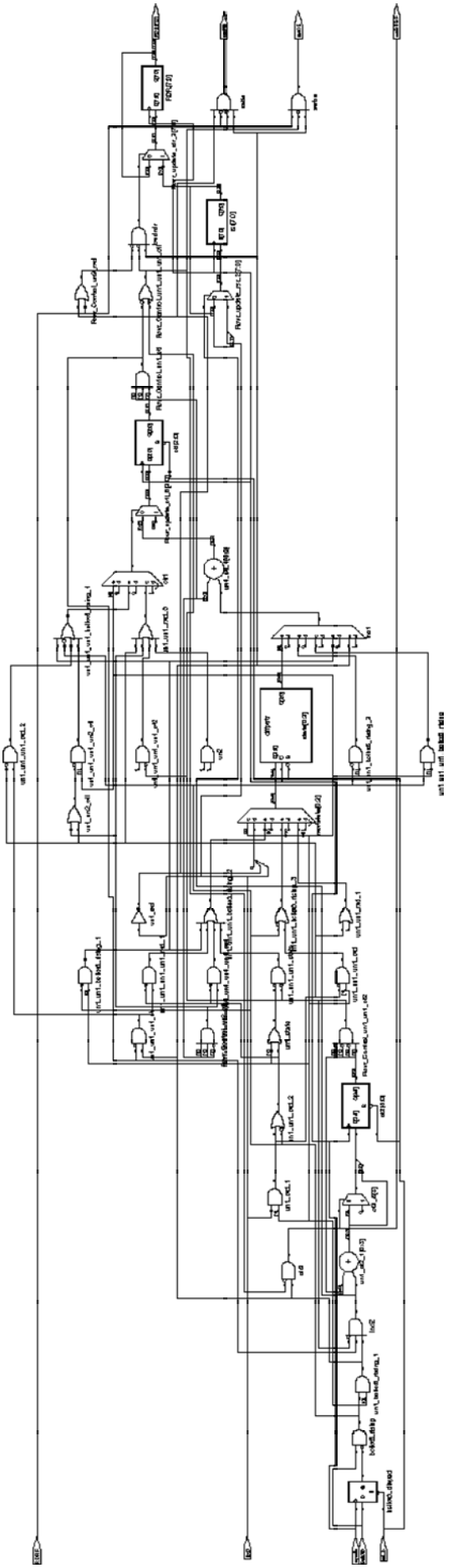


Рисунок 12.26 RTL-схема модуля Receiver

Для того чтобы обрабатывался код, полученный в результате синтеза, в конфигурации для Transmitter строка

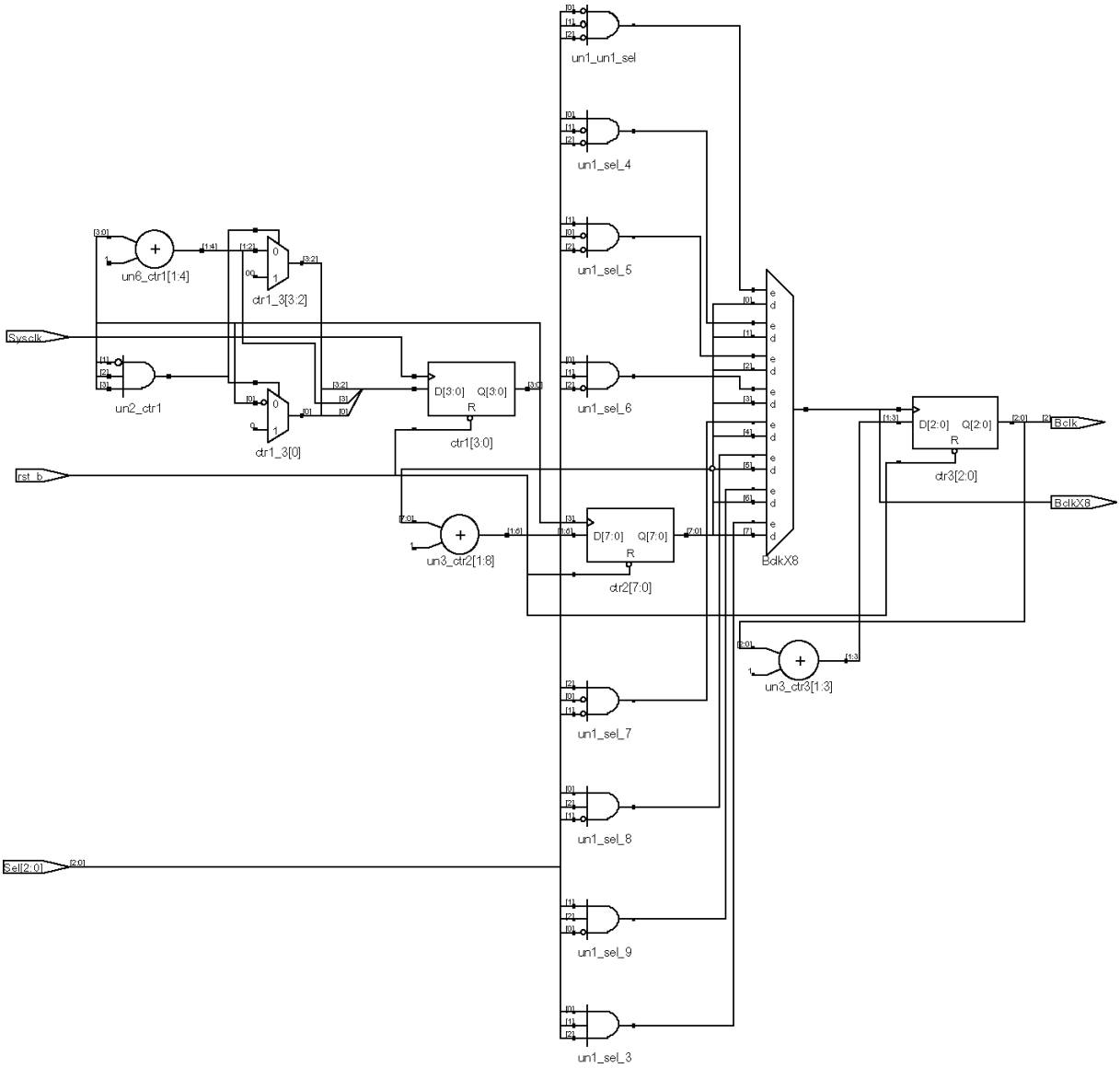
```
use entity work.UART_Transmitter(xmit);
```

заменяется на

```
use entity work.UART_Transmitter(beh);.
```

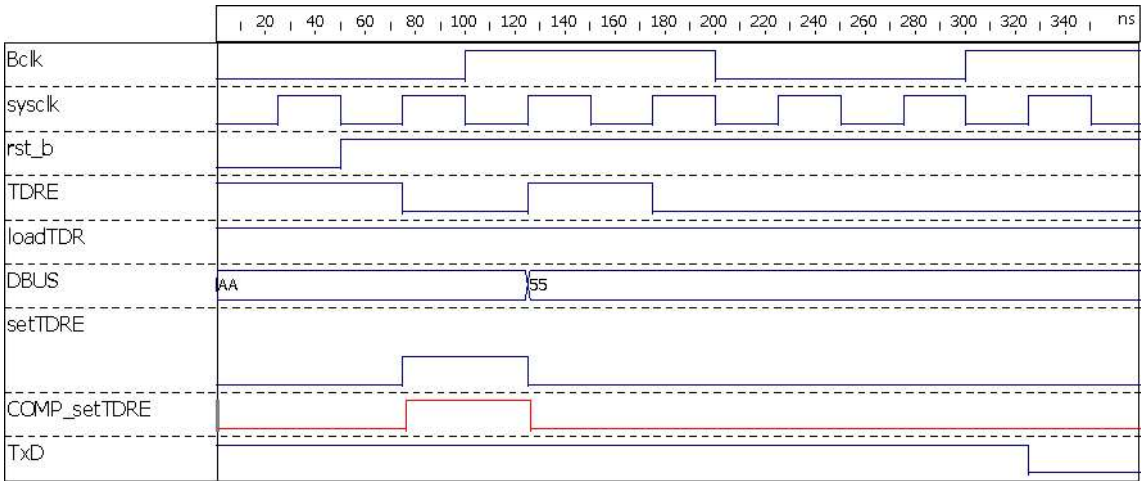
Аналогичным образом редактируется конфигурация для TestBench всех остальных модулей.

Рисунок 12.27. RTL-схема делителя частоты BAUD



Фрагмент сравнения двух временных диаграмм, полученных в результате моделирования функциональной модели и схемы после синтеза для модуля Transmitter, приведен на рисунке 12.28. Отмечен небольшой временной сдвиг сигнала setTDRE, который не влияет на общую работу модуля. Диаграмма сигнала COMP_setTDRE соответствует полученной после анализа функциональной модели.

Рисунок 12.28. Фрагмент сравнения временных диаграмм для модуля Transmitter



При функциональном анализе модуля Receiver на сигналах были отмечены статические риски сбоев сигналов setRDRF, setOE и setFE. При наблюдении поведения модели, полученной после синтеза, они исчезли. На рисунках 12.29 и 12.30 приведены фрагменты сравнения двух диаграмм. Временные эпюры сигналов с именем COMP_ соответствуют эталонному значению, сформированному при функциональном моделировании проекта.

Временные диаграммы сигналов Bclk и Bclkx8, полученные в результате тестирования созданной после синтеза VHDL-модели, имеют небольшой временной сдвиг по сравнению с данными функционального моделирования. Тем не менее, генерируемые сигналы имеют указанную в требованиях частоту. Для удобства анализа из временных диаграмм сформирован VHDL-процесс.

Рисунок 12.29. Фрагменты сравнения временных диаграмм для исправного поведения

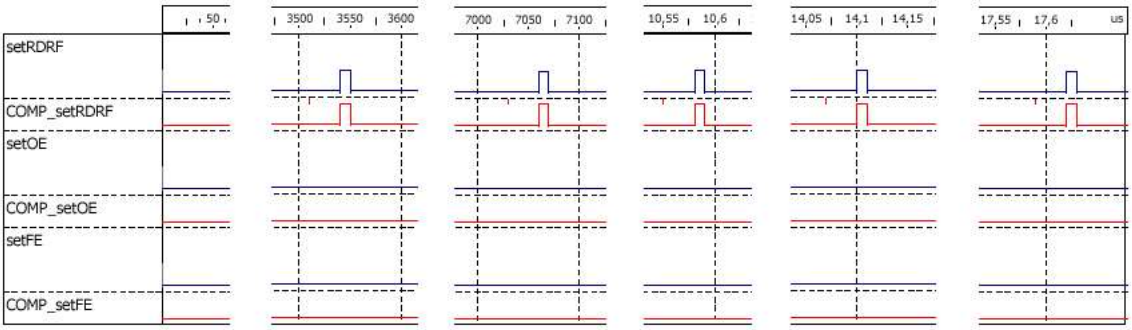
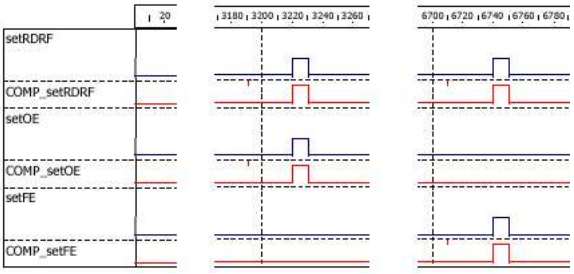


Рисунок 12.30. Фрагменты сравнения временных диаграмм моделирования экстремальных ситуаций



12.4.2. Синтез устройства UART

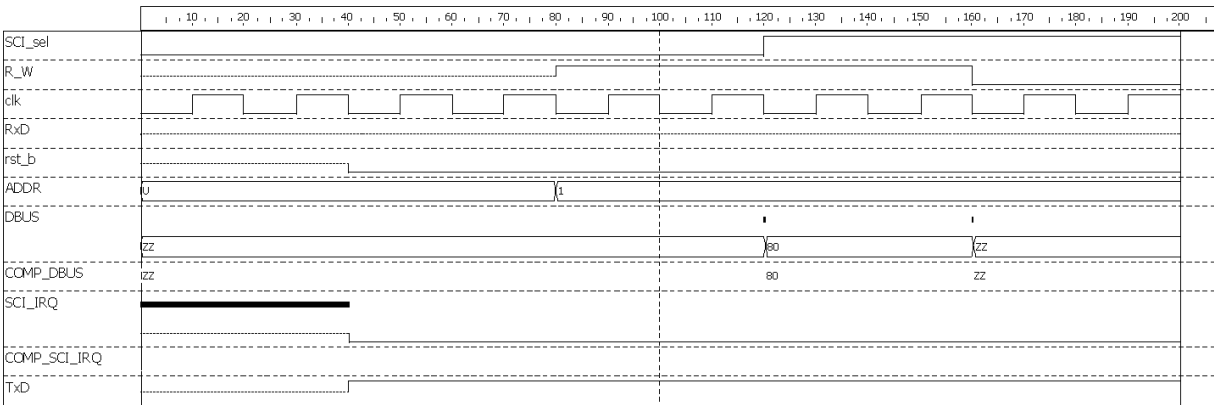
Выполнить синтез схемы всего устройства и протестировать его. Для моделирования после синтеза используется VHDL-описание разрабатываемого проекта. Чтобы выполнить последующую реализацию, создается файл EDIF. Устройство может работать на частоте до 54,2 МГц . Его RTL-схема приведена на рисунке 12.31. Схема иерархическая, поэтому компоненты Transmitter, Receveir и делитель частоты BAUD представлены в ней отдельными элементами: UART_ Transmitter, UART_Receiver и clk_ divider, схемы которых получены на предыдущем этапе проектирования.

Выполнить тестирование синтезированной модели устройства, чтобы убедиться в правильности ее работы. Проверка выполняется с помощью тестов, разработанных для функциональной модели. Принимая полученные ранее результаты моделирования за эталонные, нужно сделать анализ текущих временных диаграмм. Для первого теста (см. рисунок 12.17) отмечены временные сдвиги в диаграмме сигнала DBus, не влияющие на работу устройства. Сравнение двух временных диаграмм приведено на рисунке 12.32. Для второго теста, проверяющего режимы загрузки данных в регистр SCCR и считывание из него информации, отмечено возникновение задержки переключения шины на 0,2 ns. При переключении шины в момент перехода устройства UART из режима передачи данных на их прием она в течение 0,2 ns находится в неопределенном состоянии. Этот промежуток времени значительно меньше длины периода синхросигнала, поэтому в момент поступления следующего активного фронта синхросигнала шина будет иметь правильные данные. Эти события отражены на рисунке 12.33. При этом переключение шины происходит следующим образом:

```
   ps    delta   clk   R_W   ADDR    DBUS
150000     0      1     1     1-   11000111
150000     1      1     0     1-   XX000XXX
150200     0      1     0     1-   00000000

270000     0      1     1     1-   00000000
270000     1      1     0     1-   0X0X0X0X
270200     0      1     0     1-   01010101

390000     0      1     1     1-   01000101
390000     1      1     0     1-   XXX0XXXX
390200     0      1     0     1-   10101010

510000     0      1     1     1-   10000010
510000     1      1     0     1-   1XXXXX1X
510200     0      1     0     1-   11111111.
```

При тестировании режима передачи данных устройством временные диаграммы, которые отображены на рисунке 12.34, идентичны эталонным, за исключением периода 200 ps после начала моделирования. В течение 100 ps сигналы имеют значение U, а сигнал SCI_IQR переключится из U в 0 в момент 200 ps.

Рисунок 12.31. Сравнение результатов моделирования с использованием первого теста



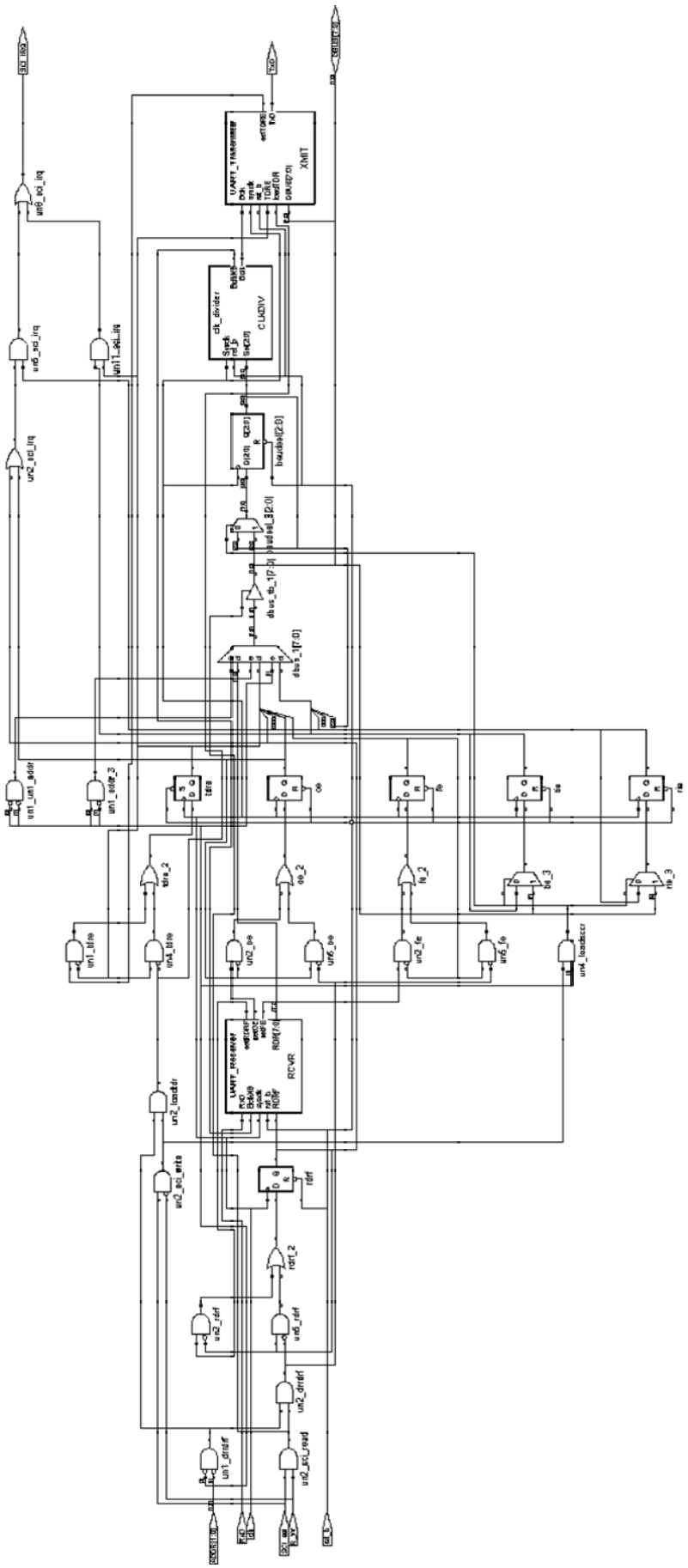


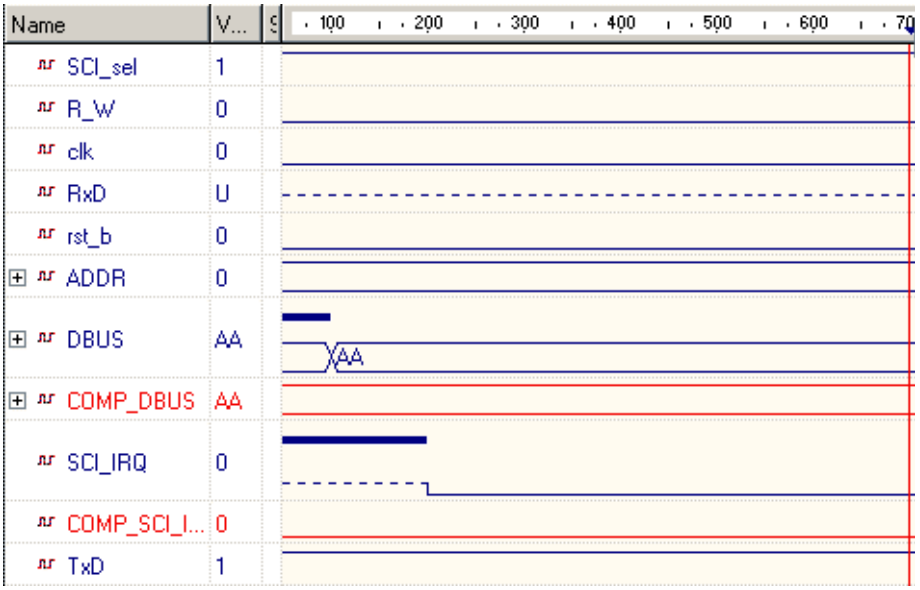
Рисунок 12.32. RTL-схема устройства UART

Временные диаграммы, полученные при тестировании режима приема последовательных данных, полностью совпадают с эталонными, сформированными в результате верификации функциональной модели.

Как видно из результатов тестирования, сгенерированная во время синтеза модель устройства полностью соответствует техническим требованиям, а отмеченные небольшие изменения не отражаются на качестве ее функционирования и принципиально не изменяют ее поведение.

Рисунок 12.33. Сравнение двух временных диаграмм для теста, выполняющего загрузку и считывание регистра SCCR (рисунок 12.18)

Рисунок 12.34. Фрагмент сравнения временных диаграмм, полученных в результате моделирования функциональной модели и синтезированной схемы



12.5. Тестирование проекта с помощью аппаратного моделирования

Для тестирования разработанной модели устройства UART можно использовать средства аппаратного моделирования HES – Hardware Embedded Simulation. Это технология, позволяющая повысить скорость тестирования сверхсложных FPGA и

ASIC ориентированных проектов, облегчая их верификацию. Вначале модули проекта верифицируются на верхнем уровне, затем выполняется их синтез, реализация и загрузка в FPGA, расположенную на плате акселератора, после чего выполняется аппаратно-программное совместное моделирование. HES-платы подключаются к персональному компьютеру через PCI-шину.

Весь проект моделируется в HES-среде, которая состоит из HES-программного симулятора и PCI-плат. Эта среда обеспечивает правильное взаимодействие модулей, помещенных в HES, и модулей, тестируемых с помощью программы.

HES-акселератор работает под управлением операционных систем UNIX, Linux и Windows NT и полностью совместим с программами моделирования Cadence, Model Technology, Aldec's Riviera™ и Active-HDL™. HES-технология позволяет обрабатывать смешанные цифровые проекты, содержащие блоки, написанные на VHDL, Verilog, EDIF и Си.

Она также используется вместо моделирования после синтеза. Все аспекты функционального анализа проекта могут быть верифицированы в среде HES. Она позволяет повысить скорость моделирования в 10–1000 раз по сравнению с самым быстрым программным моделированием.

С помощью HES-технологии проект, или его часть, можно поместить в микросхему FPGA, расположенную на плате акселератора. Для этого необходимо синтезировать модули, предназначенные для аппаратного моделирования. Для конфигурации HES-окружения используется HES Wizard.

Синтез схемы системы при работе с HES также выполняется с помощью программы Synplify фирмы Synplicity, реализация – программой, входящей в пакет Alliance фирмы Xilinx.

На первом шаге один компонент устройства размещается в HES. Выполняется совместное аппаратно-программное моделирование устройства. Модель верхнего уровня UART, а также входящие в нее компоненты Transmitter и генератор частоты BAUD моделировались с помощью программных средств, а комнонент Receiver – с помощью HES. Полученные результаты совпадают с эталонными, за исключением последнего теста. На рисунке 12.35 временные диаграммы, сформированные в результате моделирования, сравниваются с полученными данными функционального моделирования. На эталонных диаграммах до получения значений сигналы DBus и RDR имели состояние 'U', тогда как при моделировании с использованием HES эти сигналы равны '0'. Это связано с тем, что для данного теста сигналы DBus и RDR, имеющие тип std_logic, получают значения от моделируемого с помощью аппаратуры компонента Receveir. Тип std_logic имеет девять значений. Однако HES-моделирование основано на реальном устройстве, работающем в компьютере. Реальные устройства могут оперировать только двумя состояниями: 0 и 1. Поэтому U для сигналов DBus и RDR изменяется на 0. В таблице 12.4 представлены преобразования значений типа std_logic в состояния, используемые при аппаратном моделировании.

На втором шаге в HES размещались модули Transmitter, Receiver и генератор частоты BAUD. Программным образом моделировался модуль верхнего уровня – UART. Различие во временных диаграммах наблюдалось только для первого и последнего теста. Временные диаграммы приведены на рисунках 12.36 и 12.37. Изменение значений сигналов также связано с тем, что аппаратное моделирование оперирует только двумя значениями: 0 и 1, что не отражается на правильности функционирования устройства.

Рисунок 12.35. Сравнение результатов моделирования (один из них на основе HES)

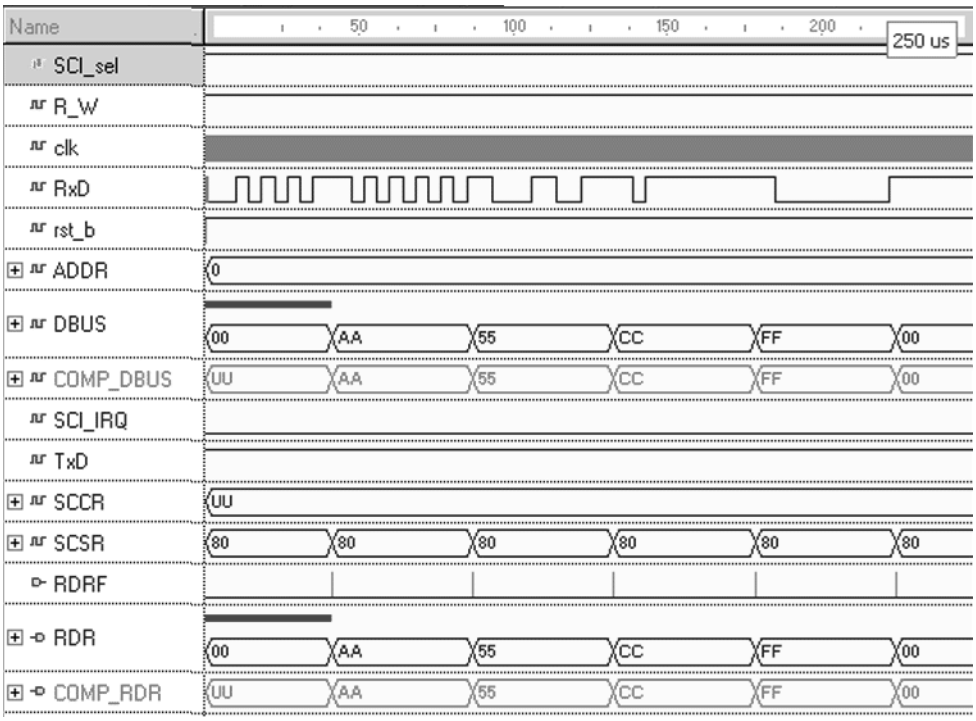


Таблица 12.4. Преобразование значений std_logic для аппаратного моделирования

Программное моделирование	Аппаратное моделирование
'U'	'1' или '0'
'X'	'1' или '0'
'0'	'0'
'1'	'1'
'Z'	'1' или '0'
'W'	'1' или '0'
'L'	'0'
'H'	'1'
'-'	'1' или '0'

Рисунок 12.36. Сравнение временных диаграмм для теста, моделирующего с 1-го по 4-й режим работы устройства UART

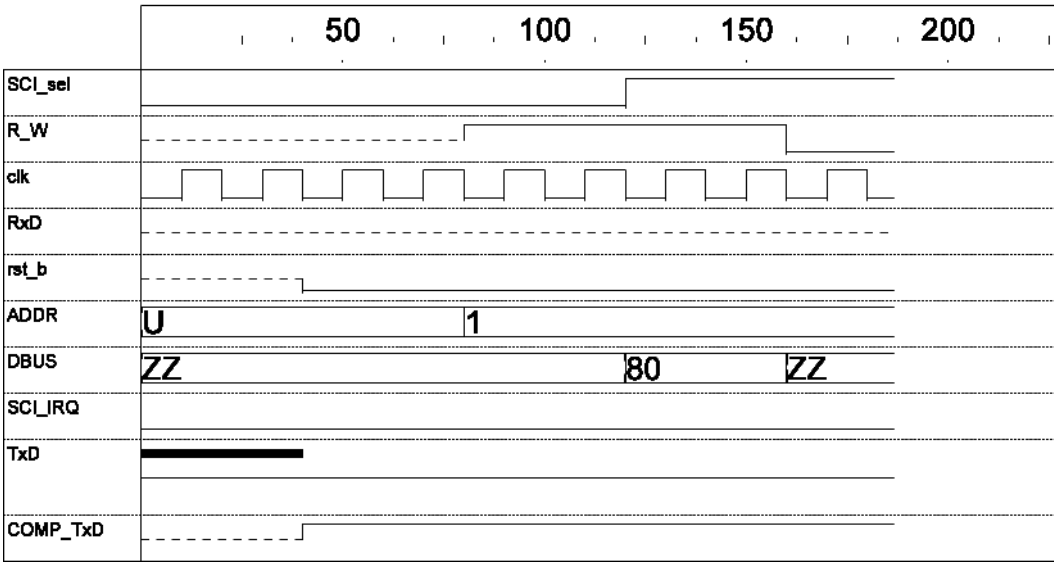
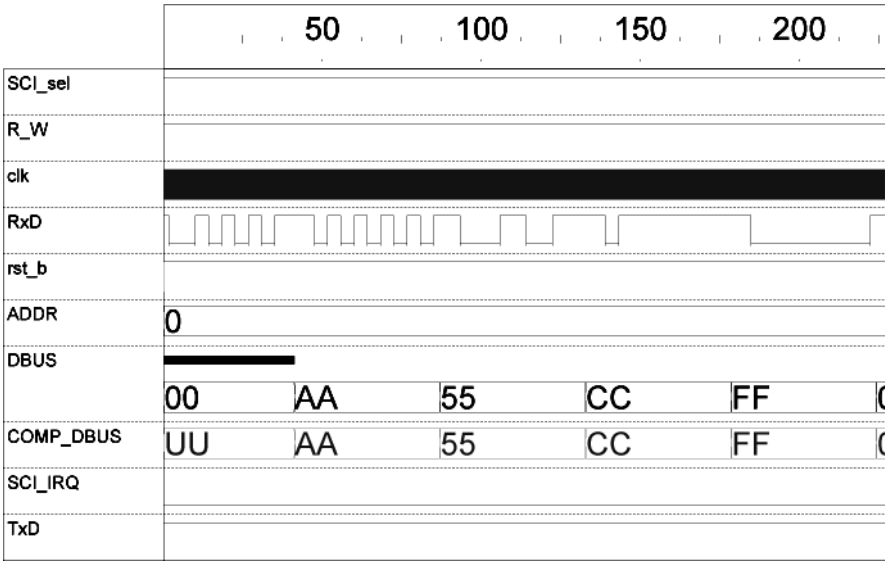


Рисунок 12.37. Сравнение временных диграмм для теста, моделирующего 7-й режим работы: прием данных устройства UART



На последнем шаге разработанное устройство UART полностью тестировалось с помощью аппаратного моделирования. Поскольку в проекте использовался двунаправленный порт DBus (рисунок 12.38), до синтеза в HES он разбивался на два: входной DBus_in и выходной DBus_out (рисунок 12.39). Необходимо указать управляющий сигнал для шины и значение, по которым она становится активной (выходом). В данном случае – это сигнал R_W. DBus становится выходом, когда R_W='1'. Новый модуль UART_unitdir синтезируется и реализуется. До окончания процесса совместного моделирования вход DBus_in и выход DBus_out следует преобразовать обратно в двунаправленный порт. Рисунок 12.40 иллюстрирует эту операцию.

Рисунок 12.38. Проект, содержащий двунаправленный порт

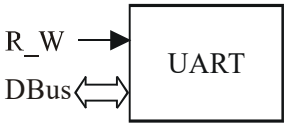


Рисунок 12.39. Проект с разделенным двунаправленным портом

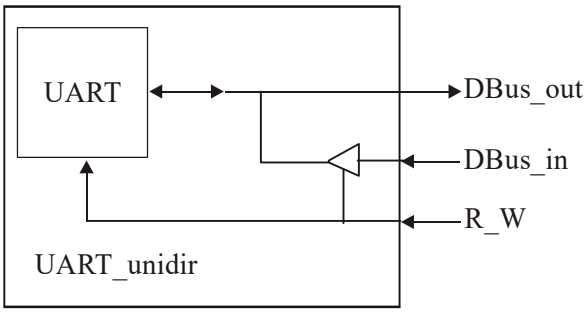
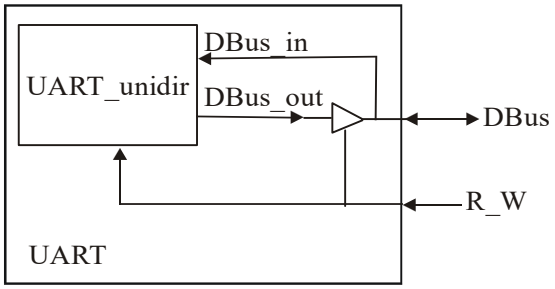


Рисунок 12.40. Слияние двунаправленных портов



Реализуемый UART_unidir модуль внутри UART и порты DBus_in и DBus_out объединяются в двунаправленный порт DBus модуля Module. Весь процесс разделения/слияния двунаправленных портов автоматически выполняется в HES Wizard.

Сравнение временных диаграмм, полученных в результате HES моделирования, с эталонными (рисунки 12.41 и 12.42) показало различие только для первого и последнего тестов. Несовпадение значений связано с невозможностью для аппаратуры представить модельное значение 'Z' или 'U'.

Рисунок 12.41. Сравнение временных диаграмм для теста, моделирующего с 1-го по 4-й режим работы устройства UART

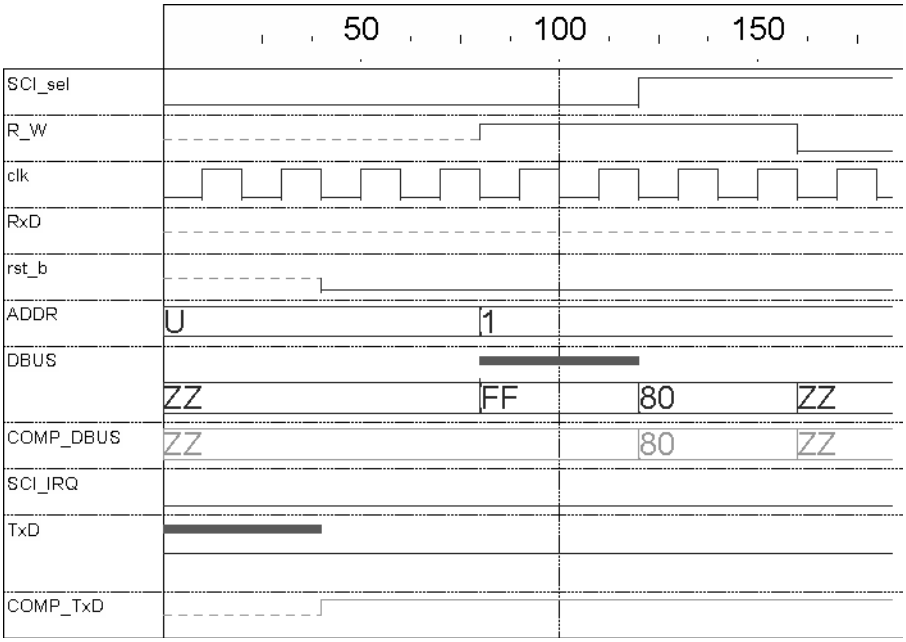
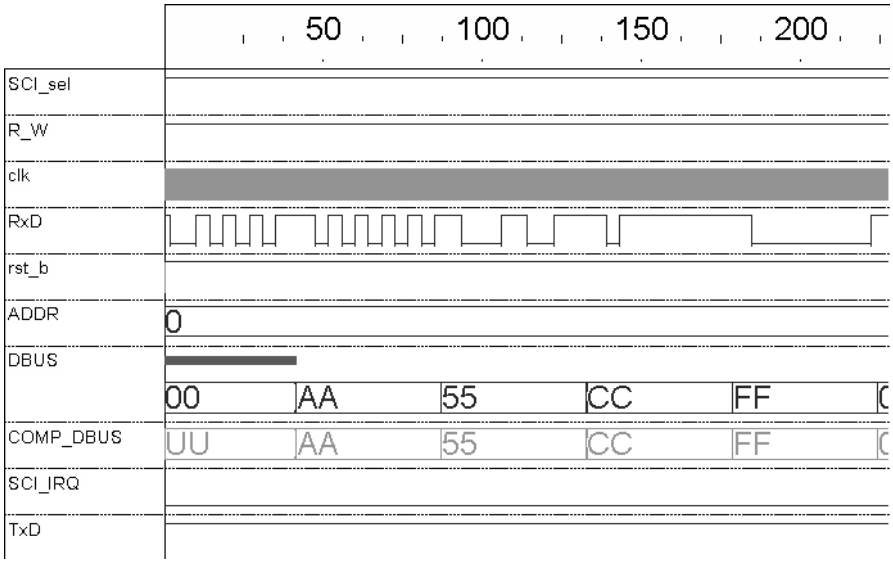


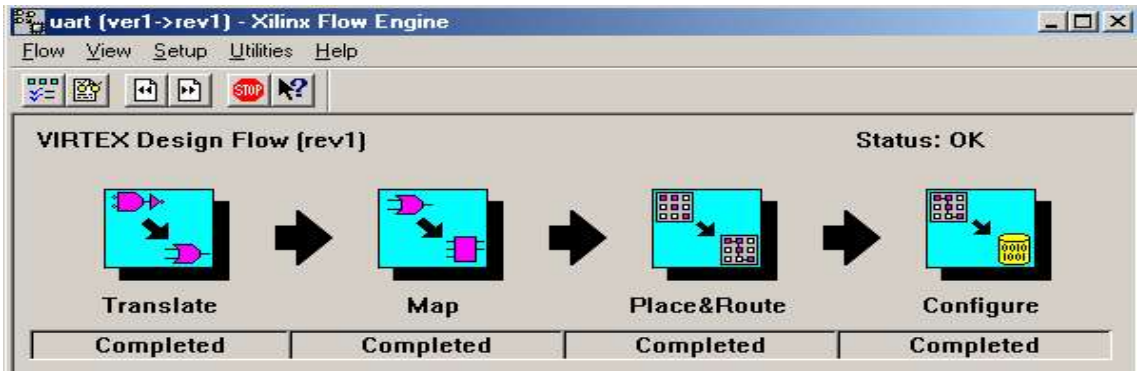
Рисунок 12.42. Сравнение временных диаграмм для теста, моделирующего 7-й режим работы: прием данных устройством UART



12.6. Реализация проекта

Реализация проекта необходима для преобразования EDIF-файла, полученного в результате синтеза, в набор битов для программирования целевой FPGA. На этом шаге используется программа Xilinx Flow Engine (рисунок 12.43), входящая в состав Xilinx Design Manager.

Рисунок 12.43. Этапы процесса реализации



Процесс реализации для FPGA включает четыре шага. Первый Translaton – преобразование схемы в вентильную модель. Второй Map – разбиение схемы устройства по блокам FPGA. Третий Place&Route – создание внутренних связей между блоками микросхемы. Четвертый Configure – создание битового файла для программирования микросхемы. На рисунке 12.44 приведен 1 сектор блока CLB R51C1 с отмеченной конфигурацией.

Устройство занимает 72 сектора из 9408 доступных в нем. Использует 76 триггеров, расположенных в секторах. Общее число необходимых LUT равно 97, из них 93 применяются как LUT и 4 – для маршрутизации. Занято 16 входных-выходных блоков IOB. Размер проекта в эквивалентных вентилях – 1268.

Минимальный период синхроимпульса может составлять 16,024 ns, что соответствует 62,406 МГц – максимальной частоте, на которой может работать спроектированное устройство.

Выводы. В процессе разработки, проведенной в рамках выполнения целевого проекта, получены следующие основные результаты:

1. Разработана функциональная VHDL-модель UART (Universal Asynchronous Receiver-Transmitter) – коммуникационного интерфейса, выполняющего прием и передачу последовательных данных. Модель имеет блоки: Transmitter, Receiver, делитель частоты и интерфейс, генерирующий управляющие сигналы.

2. Для тестирования отдельных модулей и всего проекта разработаны тестовые программы на языке VHDL, называемые TestBench. Они использовались для верификации проекта на всех этапах его разработки. Результаты тестирования функциональных моделей приняты за эталонные. Для удобства моделирования разработаны командные файлы, автоматизирующие работу по вводу команд.

3. Выполнен синтез схемы проекта с помощью программы Synplify фирмы Synplicity. Целевое устройство выбрано с учетом тестирования проекта в HES, плата которого содержит микросхему FPGA фирмы Xilinx серии Virtex. В результате синтеза сгенерирован EDIF-файл и VHDL-файл для тестирования модели после синтеза. Отличие полученных временных диаграмм от эталонных заключается в небольшом временном сдвиге диаграмм отдельных сигналов, не влияющем на реализуемую функцию. Это связано с наличием в полученной после синтеза модели устройства временных параметров – задержек.

4. Работа отдельных блоков и всего устройства протестирована с использованием аппаратного моделирования HES. Полученные диаграммы соответствуют эталонным, за исключением моментов, когда выходные сигналы должны иметь значения X, U или Z. При аппаратном моделировании эти значения заменяются на 0 или 1, поскольку реальное устройство может оперировать только этими значениями.

5. Выполнен процесс реализации устройства программой Xilinx Flow Engine, входящей в состав Xilinx Design Manager. Сформирован битовый файл, используемый для программирования микросхемы FPGA. Полученная схема использует 97 LUT, 76 триггеров из CLB, 16 IOB. Размер устройства равен 1268 эквивалентным вентилям. Максимальная допустимая частота синхросигнала 62,406 МГц.

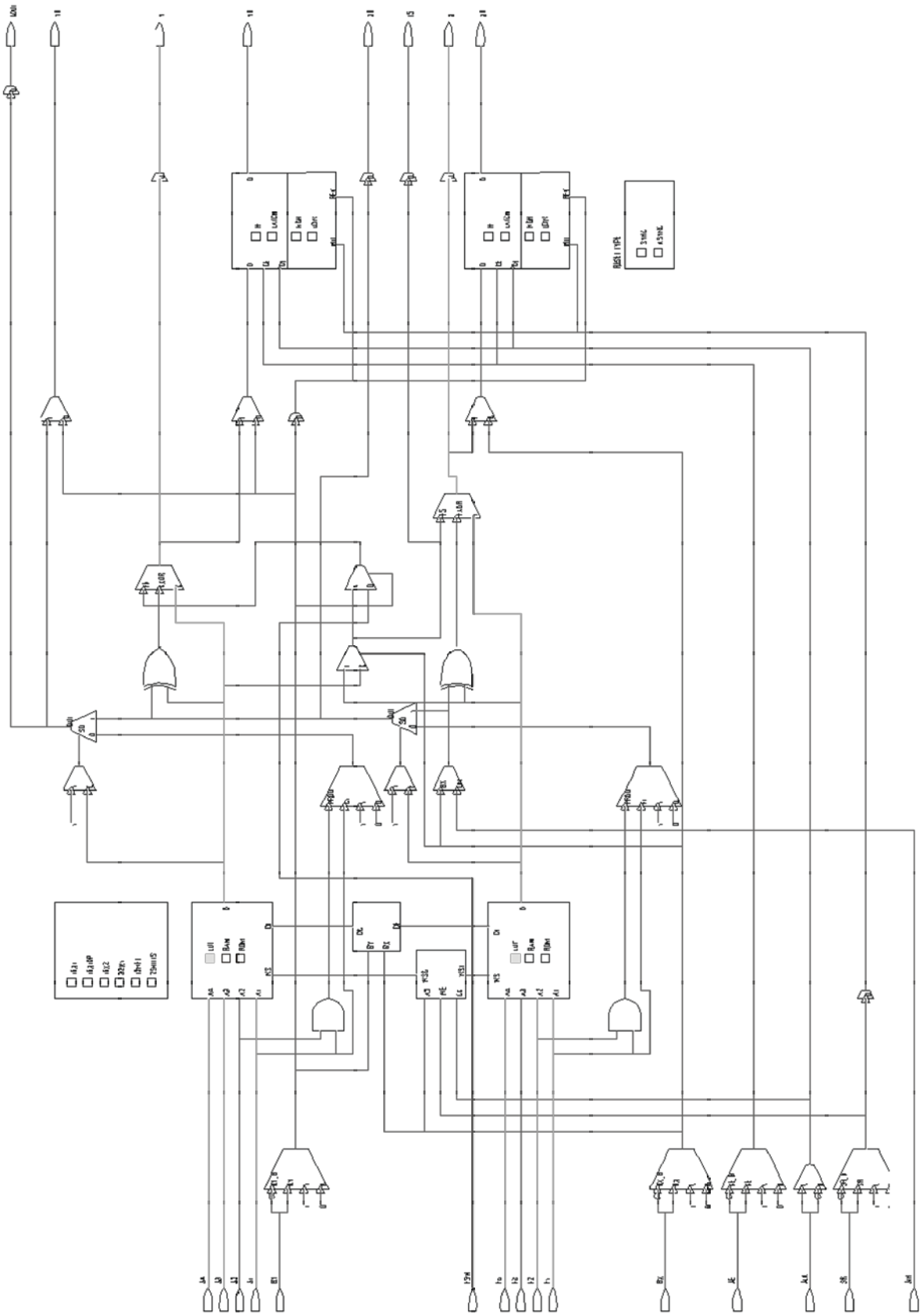


Рисунок 12.44. Сектор блока CLB R51C1 с отмеченной конфигурацией

ГЛАВА 13
МОДЕЛИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ

Представлены методы моделирования исправного поведения и неисправностей цифровых систем, являющиеся основой для верификации, синтеза тестов, контроля и поиска дефектов на стадиях проектирования и эксплуатации изделий вычислительной техники. Основной акцент сделан на обработку сверхсложных проектов, реализуемых на основе кристаллов программируемой логики, включающих миллионы вентилей.

13.1. Технологии проектирования цифровых систем

Основу совершенствования цифровых систем составляют достижения микроэлектроники. Эффективность ее развития такова, что каждую неделю тактовая частота универсального процессора увеличивается на 5 МГц, в то время как в восьмидесятые годы XX века ее повышение с 5 до 50 КГц происходило за три года. Относительно динамики стоимостных параметров следует заметить, что двадцать лет назад средняя цена одного транзистора (вентиля) составляла 5 долларов, а сегодня на рынке микроэлектроники за один доллар можно купить 5 миллионов транзисторов [Proceeding of the 9th International Coference “Mixed Design if Integrated Circuits and Systems.– Wroclaw, Poland.– 20-24 June 2002.– 722p.]. Наблюдается также существенное снижение потребляемой мощности кристаллов микросхем и применение субмикронных технологий при их производстве.

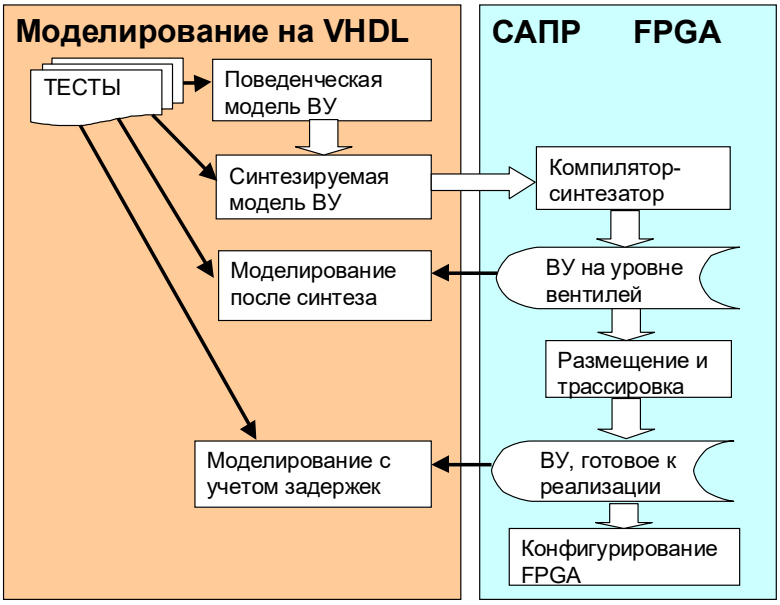
Кроме того, наблюдаемый прогресс связан с расширением многообразия предлагаемых схемотехнических решений на рынке электронных технологий, которое можно представить тремя направлениями: микропроцессоры (CPU), заказные БИС (ASIC – Application Specific Integrated Circuits, VLSI – Very Large Scale Integration) и программируемая логика (FPGA, CPLD). Первые два, наряду с высоким быстродействием, имеют недостатки, связанные со значительными временными (не менее 18 недель) и финансовыми затратами проектирования вычислительных устройств. Что касается третьего типа, то современный уровень развития субмикронных технологий микроэлектроники позволяет создавать на одном кристалле программируемой логической интегральной схемы (ПЛИС) сверхсложные цифровые системы, содержащие 5-8 миллионов эквивалентных вентилей. Это дает возможность проектировать сложные специализированные вычислительные устройства в течение 2-4 недель с помощью средств автоматизированного проектирования известных фирм, таких как: Aldec, Cadence, Altera, Xilinx, Synopsys, Mentor Graphics.

В настоящее время мировыми лидерами в области производства программируемой логики являются фирмы: Xilinx (FPGA – Virtex, Spartan), Altera (CPLD, FLEX), Lattice/Vantis (SPLD, PAL, GAL), CPLD), Actel (FPGA), Cypress, Atmel, Quicklogic. Максимальные параметры программируемой логики: тактовая частота – 500 МГц, число эквивалентных вентилей – 9 млн., объем памяти – 400 Кбит, поддерживает Boundary Scan. Объем продаж программируемой логики на рынке микроэлектроники составляет более 60 миллиардов долларов в год. Лидером является фирма Xilinx – 28 миллиардов долларов.

Что касается средств автоматизированного проектирования (Electronic Design Automation), здесь лидируют фирмы, работающие в трех направлениях: 1) ввод и моделирование цифровых проектов (Design Entry&Simulation): Aldec (20000 instalations), Mentor Graphics (12000 instalations); синтез и имплементация – (Synthesis): Synopsys (74% продаж на рынке САПР), Mentor Graphics (15%), Synplicity (11%); тестирование и верификация (Digital System Testing): Logic Vision, Mentor Graphics, Synopsys, Syntest, Simucad, Cadence.

Один из возможных вариантов структуры процесса создания цифровой системы в виде IP-core с использованием пакета Active-HDL фирмы Aldec (США) имеет вид, изображенный на рисунке 13.1.

Рисунок 13.1. Стадии проектирования IP-core в системе HDL-Active



Предложенная технология является модификацией общего подхода к разработке цифровых систем, который показан на рисунке 13.2.

Рисунок 13.2. Обобщенная структура стадий проектирования

Что касается мотивации появления IP-core, как новой технологии в создании и использовании интеллектуальных средств, следует отметить,. что почти все фирмы занимаются разработкой и практическим применением аппаратурной реализации эффективных методов и алгоритмов приема, передачи и преобразования информации. Это обусловлено прежде всего:

1. Соизмеримостью временных и материальных затрат программной и аппаратурной реализации проблемных алгоритмов, благодаря наличию средств автоматического ввода проекта, его верификации, синтеза, трассировки, размещения и имплементации в кристалле микросхемы.

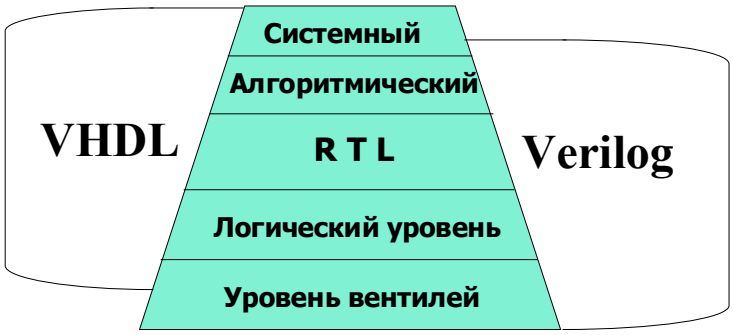
2. Более высоким быстродействием аппаратурной реализации метода или алгоритма, распараллеливанием процессов обработки больших объемов информации путем использования многоразрядных регистров (100-1000 и более битов), а также достаточно низкой стоимостью разработки, соизмеримой с программным продуктом.

3. Низкой рыночной стоимостью реализованного в микросхеме IP-core благодаря большим тиражам выпуска программируемых логических схем и новым энергосберегающим технологиям их изготовления.

4. Узкой специализацией вычислительного устройства – IP-core (стандартизированный повторно используемый верифицированный проект), позволяющей многократно уменьшить эксплуатационные и капитальные затраты его промышленного прменения, по сравнению с программной реализацией алгоритма, работающей под управлением дорогостоящей универсальной вычислительной машины.

Существенным моментом при самом быстром и современном проектировании по технологии System on Chip (SOC) – (частный случай есть изделие IP-core) на программируемой логике является иерархическое представление проекта во взаимосвязи с двумя наиболее популярными языками описания аппратуры. Типичной классификацией является деление уровней, соответствующее технологиям проектирования (рисунок 13.3):

Рисунок 13.3. Уровни иерархии моделей при описании их аппаратурными языками



1. Системное (алгоритмическое) проектирование. Характеризуется высоким уровнем абстракции описания моделей на языках: VHDL, Verilog, System C. Скорость выполнения проектов – более 1 миллиона вентилей в месяц на одного проектировщика, сложность проектов – 10^5– 10^6 вентилей. Объем выполняемых на рынке проектов – 60%. Используются технологии совместного аппаратно-программного проектирования – Hardware-Software Cooperation Design и стратегия повторного применения готовых проектов – IP-core reuse.

2. Схемотехническое проектирование. Описание моделей на RTL уровне с помощью языков: VHDL (более универсальный и академический, популярен в Европе, в среде университетских ученых), Verilog (более простой язык, промышленно-ориентированный, популярен в Японии, США; в списке пользователей – фирмы SUN, Apple, Motorola), System C (в стадии разработки компиляторов и тестирования), EDIF (стандарт внутреннего описания цифровых проектов и представления схем после синтеза). Скорость выполнения проектов – около ста тысяч вентилей в месяц на одного проектировщика, сложность проектов – 10^4 вентилей. Объем выполняемых на рынке проектов – 40%.

3. Логическое проектирование цифровых систем. Описание моделей на вентильном уровне в виде булевых уравнений с помощью следующих языков: VHDL, Verilog, System C, EDIF. Используется самостоятельно для создания несложных проектов (около 10^4 вентилей), требующих сверхвысокого быстродействия и минимальных аппаратурных затрат. В настоящее время данный уровень рассматривается как автоматическая процедура синтеза цифровой системы.

При использовании действующих языков (VHDL, Verilog, EDIF) для описания исходной модели после синтеза получается цифровая структура (например, 40 тыс. вентилей), имеющая следующие наиболее существенные относительные характеристики: объем файла (1/0,7/2,8), объем ОЗУ (1/2,8/1), время моделирования (1/2,7/1,2) соответственно.

Проблема тестопригодного проектирования решается в рамках системы диагностического обслуживания (рисунок 13.4), разработанной на кафедре АПВТ ХНУРЭ, входящей в состав САПР. Основные задачи, решаемые в целях создания диагностического обеспечения: 1) тестопригодное проектирование с использованием технологий Boundary Scan; 2) построение моделей, минимально достаточных и технологичных для решения задач тестирования; 3) синтез тестов проверки исправности или неисправностей наперед заданного класса; 4) моделирование неисправностей в целях оценки качества построенных тестов; 5) диагностирование цифрового устройства на стадиях его проектирования и эксплуатации.

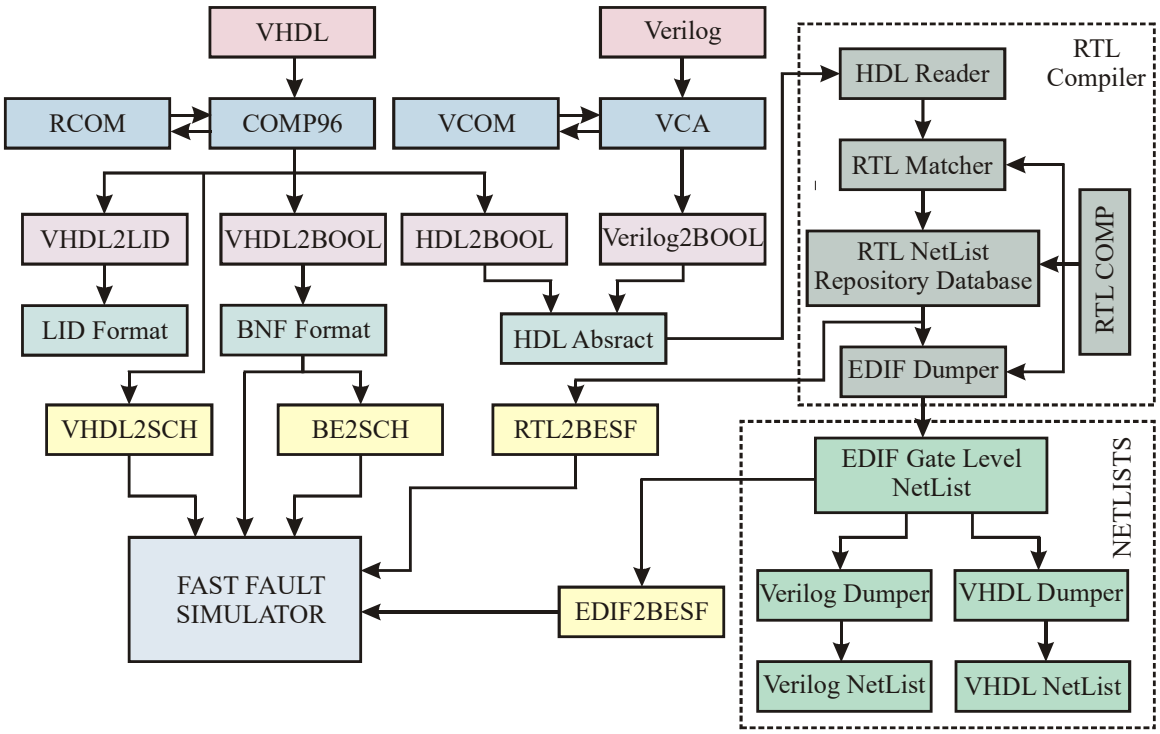
Рисунок 13.4. Диагностическое обслуживание проектируемых цифровых систем

Наиболее сложной проблемой является синтез тестов диагностирования или проверки неисправностей, которая решается на основе использования совокупности алгоритмических генераторов входных последовательностей, реализуемых как программно, так и аппаратурно, в виде встроенных или внешних, по отношению к объекту диагностирования, средств. Генераторы представляют собой реализацию псевдослучайных и быстродействующих алгоритмов, а также детерминированных точных методов, ориентированных на проверку отдельных дефектов. В целях синтеза тестов и их анализа для сверхбольших проектов используется их структурный анализ и декомпозиция на подсхемы, что дает возможность обрабатывать объекты до 1 млн. вентилей.

Весьма существенной для создания сложных проектов представляется возможность конвертирования исходных данных в различные языки описания аппаратуры. Это связано, прежде всего, с тем, что IEEE-стандарты для языков: VHDL, Verilog, EDIF, System C могут принимать участие в описании проекта одновременно, что характерно при создании сложных проектов, в которых участвует несколько фирм или коллективов разработчиков. В связи с таким интегрированным подходом становится необходимой разработка и использование некоторой внутренней структуры данных,

универсальной по отношению ко всем языкам описания аппаратуры. При этом следует иметь парсеры (трансляторы) для синтаксического анализа и преобразования конструкций одного языка в форматы другого посредством использования универсального промежуточного стандарта HDL-abstract, как показано на рисунке 13.5. Однако сложность решения проблемы по преобразованию конструкций одного языка в другой сравнима с разработкой компилятора. Если же рассматривать оптимальное решение такой задачи, то оно может быть выполнено только в рамках синтезируемого подмножества языков описания аппаратуры с последующим рассмотрением более эффективных решений в рамках постоянной доработки программных средств.

Рисунок 13.5. Поддержка IEEE-стандартов описания аппаратуры при проектировании



С позиции методологии изучения интерес представляет общая картина развития моделей, методов и алгоритмов для моделирования цифровых систем (рисунок 13.6).

Далее предлагается краткий аналитический обзор сущности моделей, методов и алгоритмов моделирования исправного поведения и неисправностей (см. рисунок 13.6) с иллюстрацией на отдельных примерах.

13.2. Синхронные методы моделирования

Предназначены для синхронного анализа переходных процессов в цифровых устройствах вентильного и функционального уровней описания, которое характеризуется, во-первых, интервалом времени между сменой очередного входного набора, определяемым как $\Delta t = t_{i+1} - t_i$, во-вторых, тем, что для задержек схемы t и элемента t_i должно выполняться условие $\Delta t >> t >> t_i$. Использование трехзначного алфавита {0,1,X}, где X – переходный процесс, не идентифицируемый нулем или единицей, ориентировано на выявление состязаний в последовательностных схемах. При анализе КП (последовательностных) схем значения выходов определяются состоянием переменных в предыдущий момент времени: $(Z^t,Y^t)=g(X^{(t-1,t)},Z^{t-1},Y^{t-1})$. Такой вычислительный процесс получил название простых итераций. Альтернативный вариант – итерации Зейделя – предполагает проведение вычислений в одном временном такте, но с предварительным псевдоразрывом глобальных обратных связей, ранжированием линий и элементов: $(Z^t,Y^t)=g(X^t,Z'^t,Y'^t)$, где Z'^t,Y'^t – псевдопеременные разорванных обратных связей.

До начала моделирования всем линиям объекта присваивается значение X. Моделирование носит итеративный характер обработки примитивов схемы. Если при выполнении алгоритма моделирования в схеме возникает генераторный режим, то при достижении предельного, наперед заданного числа итераций (для реальных схем около 20), всем изменяющимся линиям присваивается значение X, после чего сходимость алгоритма гарантируется. Синхронное моделирование предполагает, что время существования переходного процесса намного больше номинальной задержки схемы. Для комбинационных схем моделирование в троичном алфавите не различает статические и динамические риски сбоев. Данный недостаток устраняется введением многозначного алфавита описания переходных процессов.

Для последовательностных схем увеличение значности алфавита переходных процессов (K>3) не повышает точность моделирования, но ухудшает сходимость алгоритма, увеличивая число итераций. Дальнейшее повышение адекватности связано с введением параметра времени при использовании номинальных задержек элементов и разброса интервала переключения входных сигналов. Тем не менее, для анализа комбинационных схем существует большое количество работ, связанных с введением многозначных алфавитов.

Рисунок 13.6. Модели, методы и алгоритмы при моделировании цифровых систем

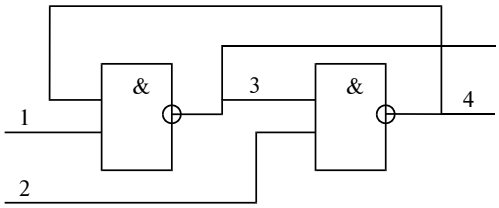
Метод Эйхельбергера. Предназначен для синхронного анализа переходных процессов в цифровых устройствах вентильного уровня описания. Использует трехзначный алфавит {0,1,X}, где X – переходный процесс, не идентифицируемый нулем или единицей. (У автора символ X определяется значением дроби 1/2 – не нуль и не единица.) Алгоритм анализа содержит итеративные процедуры A и B. Первая служит для моделирования собственно переходного входного набора A/B, который определяется по правилам следующей операции:

*	0	1	X
0	0	X	X
1	X	1	X
X	X	X	X

.

Вторая заключается в моделировании статического двоичного вектора B и предназначена для идентификации возможных состязаний. Обе процедуры являются итеративными, с гарантированной сходимостью вычислительного процесса негенераторных схем. Для режима A характерно увеличение символов X на линиях от одной итерации к другой, для B – уменьшение. Вычисления прекращаются при получении двух одинаковых векторов моделирования в двух соседних итерациях. При анализе КП (последовательностных) схем значения выходов определяются методом простых итераций. На примере схемы триггера, изображенной на рисунке 13.7, рассмотрим особенности анализа переходных процессов эквипотенциальных линий.

Рисунок 13.7. Пример триггерной схемы



Последовательное выполнение процедур A, B со всеми итерациями для трех входных наборов B=(01,00,11) дает результат:

A1	XXXX	A = XX
	XXXX	A * B = XX
B1	011X	B = 01
	0110	
	0110	A = 01
A2	0X1X	A * B = 0X
	0X1X	B = 00
B2	0011	
	0011	A = 00
A3	XXXX	A * B = XX
	XXXX	B = 11
B3	11XX	
	11XX	

Значения выходов 3 и 4 примитивов определяются состояниями входов в момент t и невходных линий в предыдущем такте t-1. При выполнении B3 получены состояния выходов, равные XX. Это свидетельствует о наличии опасных состязаний, приводящих схему в непредсказуемое двоичное состояние. Поэтому для реального объекта следует исключить последовательность функциональных или тестовых наборов 00-11. Моделирование предыдущей пары векторов 01-00 по результату выполнения процедуры B свидетельствует об отсутствии состязаний. Если при выполнении A или B в схеме возникает генераторный режим, то при достижении предельного, наперед заданного

числа итераций (для реальных схем около 20) всем изменяющимся линиям присваивается значение X, после чего сходимость алгоритма гарантируется. Рассмотренный метод представляет собой синхронное моделирование, когда время существования переходного процесса намного больше номинальной задержки схемы. Многообразие других синхронных алгоритмов определяется значностью алфавитов описания переходных процессов, основные из которых представлены на рисунке 13.8.

Наиболее распространенными следует считать: A^3 – троичный алфавит Эйхельбергера; A^9 – девятизначную символику Фантози.

Сложность описания примитивов в многозначном алфавите определяет только вентильный уровень обрабатываемых объектов, представленных в базисе двухвходовых элементов И, ИЛИ, одновходового НЕ.

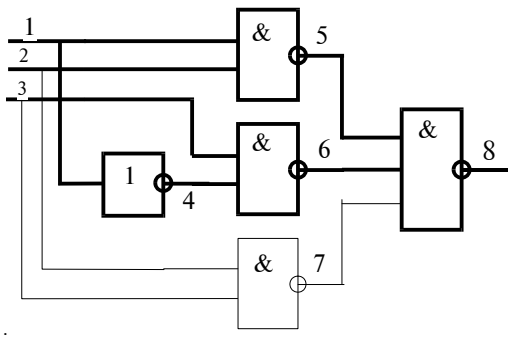
Для анализа комбинационных схем существенной представляется классификация рисков сбоев на функциональные и структурные. Первые присущи любой реализации и не могут быть устранимы. Со вторыми можно бороться введением аппаратурной избыточности. Примером может служить функция, минимизация которой определяет переключательную схему: Y=X1X2 V ^X1X3.

Рисунок 13.8. Алфавит переходных процессов

Символ	Интерпретация		
	Логическая	Графическая	Физическая
0	0		Постоянное значение 0
1	1		Постоянное значение 1
X A3	X		Неопределенное значение X
E	01		Гладкий переход из 0 в 1
H A5	10		Гладкий переход из 1 в 0
P	0X0		Статический риск сбоя в 0
V	1X1		Статический риск сбоя в 1
F	0X1		Динамический риск сбоя из 0 в 1
L A9	1X0		Динамический риск сбоя из 1 в 0
O	X0		Переход из неопределенности в 0
I	X1		Переход из неопределенности в 1
A	0X		Переход из 0 в неопределенность
B A13	1X		Переход из 1 в неопределенность

Для схемы, изображенной на рисунке 13.9, которая соответствует функции Y, существуют структурные состязания на наборах 011-111, что устраняется введением избыточного элемента с выходом 7.

Рисунок 13.9. Пример избыточной схемы



Наличие непроверяемой одиночной константной неисправности 1 на линии 7 возобновляет состязания в схеме на наборах 011-111, что ставит под сомнение правомерность устранения нежелательных переходных процессов введением избыточности, поскольку она приводит к непроверяемым дефектам. Примером функциональных состязаний могут служить для данной схемы переходы 001-111, 101-010, которые на схеме задают переходные процессы

1	2	3	4	5	6	7	8
E	E	1	H	H	E	H	V
H	E	H	E	V	V	V	P

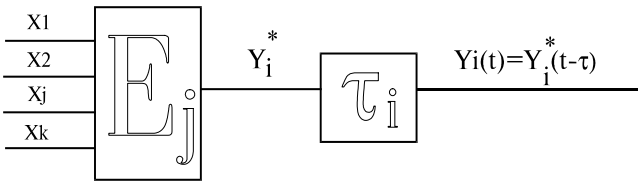
Учитывая, что динамические риски сбоев являются следствием статических, устранение последних служит условием отсутствия каких-либо состязаний. Наиболее действенный способ борьбы с ними – введение промежуточных наборов. В пределе функциональные (тестовые) входные наборы должны иметь кодовое расстояние, равное 1, благодаря переупорядочению теста или использованию промежуточных векторов. Однако и в этом случае для схем со сходящимися разветвлениями могут иметь место состязания. Дальнейшее повышение адекватности связано с введением параметра времени при использовании номинальных задержек элементов и разброса параметра времени переключения входных сигналов в асинхронных методах моделирования, которые модифицируются в дельта-троичные процедуры и в алгоритмы с нарастанием неопределенностей.

13.3. Асинхронные методы моделирования

Многозначное синхронное моделирование позволяет обнаруживать все реальные состязания, но иногда указывает на них в тех случаях, когда рисков сбоев и гонок на самом деле нет. Такой запас прочности приводит к дополнительным издержкам при логической верификации цифровых систем, связанным с оценкой качества теста и устранением несуществующих, но нежелательных модельных переходных процессов. Способом решения данной проблемы и дальнейшего повышения адекватности моделирования являются асинхронные методы анализа. Их разнообразие определяется значностью алфавита моделирования и степенью адекватности моделей по реальным временным параметрам.

Метод асинхронного двоичного моделирования оперирует описанием примитива, представленного на рисунке 13.10.

Рисунок 13.10. Асинхронная модель примитивного элемента



Элемент E_j реализует логическую функцию, t_i определяет задержку появления сигнала на выходе Y_i. Для упрощения процедуры анализа схемы с асинхронными моделями примитивов выбирается единый масштаб времени Δt, как наибольший общий делитель номинальных задержек элементов t_i, благодаря чему значения модельных задержек r_i являются целыми числами. При этом реальная задержка ПЭ $t_i = r_i \Delta t$, где r_i – целое положительное число; Δt – единица масштаба.

Если три элемента имеют задержки (60, 90, 150 ns), то масштаб, определяемый как наибольший общий делитель, имеет значение 30 ns. Тогда модельные задержки находятся по выражению $r_i = t_i / \Delta t$ и равны соответственно 2, 3, 5. Смена значений сигналов на линиях схемы осуществляется только в моменты 0, Δt, $2\Delta t$,... реального

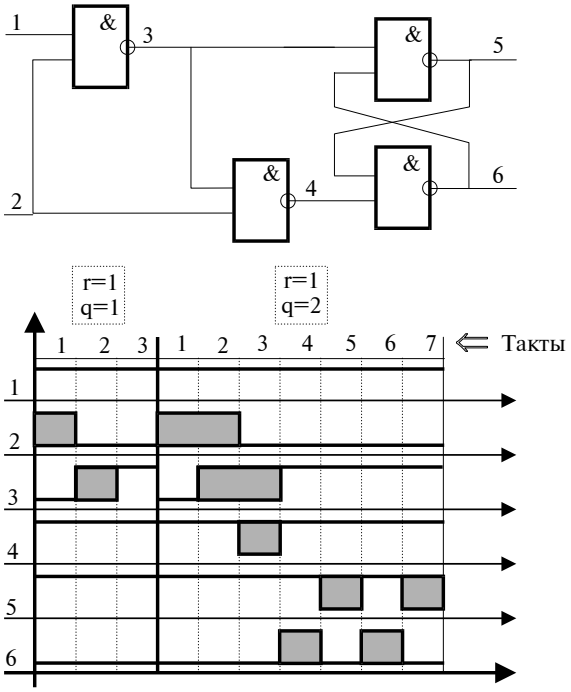
времени. Значение сигнала на выходе Y_i в такте j определяется состоянием входов примитива в такте $j - r_i$.

Двоичное асинхронное моделирование никогда не указывает на ложные состязания, но иногда пропускает реальные. Такой недостаток практически перечеркивает целесообразность применения данного метода. Тем не менее, оно используется при логическом анализе, необходимом для синтеза и верификации тестов.

Δ-Троичное моделирование, являясь "золотой" серединой, устраняет недостатки двоичного асинхронного и троичного синхронного методов. При этом в дополнение к номинальной задержке $t_i=r_i \Delta t$ примитива добавляется разброс момента переключения входных сигналов $q \Delta t$, где q – целое неотрицательное число. При $q = 0$ метод сводится к двоичному асинхронному в троичном алфавите. Если $q >> T_{max}$, где T_{max} – максимальная задержка всей схемы, получаем метод синхронного троичного моделирования. Варьируя параметром q, можно уменьшать или увеличивать адекватность анализа цифровых схем за счет соответствующего уменьшения или увеличения времени обработки проектируемого объекта.

На примере вентильной структуры D-триггера, заданной рисунком 13.11, рассмотрим Δ-троичное моделирование устройства на показанных здесь же временных диаграммах переходных процессов.

Рисунок 13.11. Δ-Троичное моделирование D-триггера



Каждому элементу схемы приписана модельная задержка $r_i=1$. При изменении на входах двоичных наборов 11-10 троичное синхронное моделирование дает по выходам 5 и 6 состояние неопределенности XX. Символу X соответствует закрашенная клеточка на диаграмме. Если анализ схемы выполнять D-троично, то при q=1 состязания отсутствуют, а выходы 5 и 6 сохраняют предыдущее значение. При $q \geq 2$ метод фиксирует состязания на выходах триггера. Для машинной реализации более технологичным является процесс анализа, представленный в табличном эквиваленте временных диаграмм.

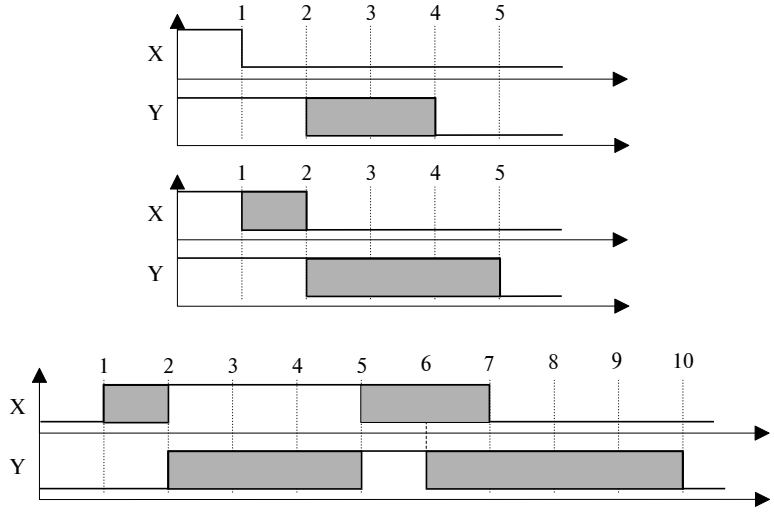
Сходимость в Δ-троичном моделировании не гарантирована, поэтому при достижении максимального числа итераций (тактов) в процессе анализа последовательностных схем на изменяющихся линиях фиксируется значение X. Выбор конкретного значения q не имеет достаточно формализованного теоретического обоснования,

поэтому адекватность результата есть функция от интуиции и квалификации пользователя.

Асинхронное троичное моделирование с нарастающей неопределенностью оперирует описанием элемента E_j , модельная задержка которого находится в интервале $0 \le r_{min} \le r_i \le r_{max}$. Значение сигнала на выходе ПЭ в момент j определяется состоянием его входов в интервале $<j-r_{max}, j-r_{min}>$. И наоборот, если произошло изменение сигнала на входе в момент j, то в интервале $<j+r_{min}, j+r_{max}>$ существует неопределенность. Варианты моделирования функции повторителя приведены на рисунке 13.12 для $r_{min}=1$ и $r_{max}=3$.

Итак, обоснование введения Δ -троичного моделирования связано с неопределенностью момента переключения сигнала на входе ПЭ, что является следствием энтропии задержек соединительных линий при детерминированных t_i каждого элемента. Моделирование с нарастающей неопределенностью убирает также детерминизм в модельной задержке элемента, заключая ее в некоторый интервал. Соединение обоих методов в единый алгоритм дает возможность изменением параметров q и интервала $<0 \le r_{min} \le r_i \le r_{max}>$ получать любой из ранее рассмотренных методов анализа переходных процессов и статического моделирования. Но универсальность такого подхода связана со сложностью его практической реализации и слабым быстродействием. Поэтому в пользу проектирования такой системы анализа должны быть представлены весомые аргументы.

Рисунок 13.12. Моделирование с нарастающей неопределенностью



13.4. Методы моделирования неисправностей

Практически любой из методов моделирования неисправностей по реализации намного сложнее алгоритмов анализа исправного поведения, которые являются частными случаями первых. Сущность моделирования неисправностей состоит в определении влияния одного или нескольких дефектов на состояния линий объекта при подаче тестовых последовательностей.

Если значения наблюдаемых выходных линий искажаются при наличии неисправностей, последние являются проверяемыми на заданном тесте, который называется проверяющим. Под дефектом будем понимать каждое отдельное несоответствие объекта нормативно-технической и/или конструкторской документации. Отказ, приводящий объект в неработоспособное состояние, называется неисправностью.

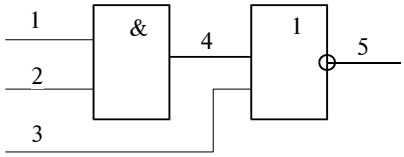
Неисправность проверяется на тесте (векторе), если она изменяет состояние хотя бы одного наблюдаемого выхода.

На сегодня известны десятки типов дефектов, сгруппированных в классы. Здесь и далее рассматриваются одиночные константные неисправности, если не оговорено иное. Данному классу моделей дефектов соответствуют в реальности: обрыв в цепи эмиттера, базы, коллектора; короткие замыкания эмиттер-база, база-коллектор; обрывы в цепях резисторов. Другими логическими неисправностями могут быть: короткое замыкание контактов, их перепутывание, изменения функций, I_{DDQ} дефекты, сочетания упомянутых случаев.

Большинство практически реализуемых методов используют для анализа неисправностей синхронные модели исправного поведения. На этой основе далее предлагаются алгоритмы одиночного, параллельного, дедуктивного, кубического и совместного моделирования.

Одиночное моделирование неисправностей. В основу положена идея внесения одной ОКН эквипотенциальной линии в схему в целях анализа ее проявления на внешних выходах объекта при подаче тестовых последовательностей. Если внесенный дефект изменяет состояние исправного поведения хотя бы одного наблюдаемого выхода при подаче теста (одна или более входных последовательностей), то он является проверяемым. Для схемы, имеющей K линий, число неисправностей равно произведению 2*K констант нуля или единицы. Качество теста оценивается формулой Q=P/(2K), где P – число проверяемых дефектов. Выполним анализ проверяющих свойств тестовых наборов 001,100 для схемы, изображенной на рисунке 13.13.

Рисунок 13.13. Фрагмент схемы для моделирования дефектов



Поочередно вносим неисправности є0, є1 на все линии схемы, учитывая, что векторы исправного поведения равны 00100, 10001. На первом наборе дефекты $\{3^0,5^1\}$ искажают состояние выхода 5, на втором проверяются неисправности $\{2^1,3^1,4^1,5^0\}$. Объединение проверяемых подмножеств дает множество $\{2^1,3^0,3^1,4^1,5^0,5^1\}$, что формирует качество теста, равное Q=(6/10)*100=60%. При машинной реализации данный метод не требует больших затрат времени и ориентирован на обработку любых схем нерегистрового уровня.

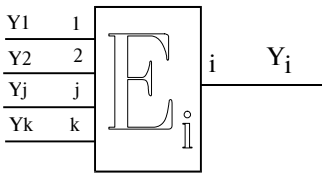
Параллельное моделирование неисправностей. Используются машинные команды параллельной обработки слов (регистров): логическое сложение, умножение, инверсия, исключающее ИЛИ. Моделирование компилятивное, поскольку поведение примитивов описывается процедурами на уровне алгоритмических языков или ассемблеров. Одновременно выполняется анализ P неисправностей на входном наборе, P – разрядность машинного слова, доступного для параллельной обработки устройства. Иллюстрацией метода может служить результат моделирования ОКН схемы (см. рисунок 13.13) на входном наборе 100:

L	И	1^0	2^0	3^0	4^0	5^0	1^1	2^1	3^1	4^1	5^1
1	1	0	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	1	0
5	1	1	1	1	1	0	1	0	0	0	1
D	=	–	–	–	–	+	–	+	+	+	–

Строки таблицы определяют состояния линий: И – столбец в случае отсутствия неисправностей; остальные – при наличии дефектов, идентифицируемых заголовком. Заштрихованные клетки указывают на состояние неисправной линии. Каждый столбец содержит одну такую координату, которая соответствует номеру дефектной переменной. Перед началом моделирования следует выполнить маскирование диагоналей нулевых и единичных неисправностей. Остальные входные координаты таблицы определяются состоянием линий столбца И. Процедура анализа сводится к доопределению пустых клеток, относящихся к невходным линиям (строки 4 и 5). В компьютере это осуществляется путем выполнения параллельной операции И над строками 1 и 2, что формирует вектор 4. Затем выполняется команда ИЛИ над строками 4 и 5, после чего применяется операция НЕ, что определяет состояние линии 5. При этом заштрихованные координаты невходных строк 4 и 5 остаются без изменения. Анализ дефектов, проверяемых набором 10001, сводится к сравнению состояний выходных координат каждого столбца неисправности с вектором И. При фиксации несовпадения неисправность, соответствующая столбцу, является проверяемой, что в последней строке D таблицы отмечается знаком “+”, в противном случае – знаком “-”. По результату параллельного моделирования получилось качество вектора 10001, относительно ОКН, равное Q=(4/10)*100=40%. Недостатком метода является ориентация на конкретную ЭВМ, сложность проектирования моделей и проблемных программ. Быстродействие метода в P раз выше одиночного моделирования неисправностей. Идея параллельного вычисления может быть использована для анализа исправного поведения. При этом за один машинный такт обрабатывается P двоичных входных наборов.

Дедуктивное моделирование неисправностей позволяет на одном входном наборе одновременно обрабатывать все ОКН схемы, выделяя при этом подмножество проверяемых. Анализ примитивного элемента сводится к выполнению операций над списками неисправностей с помощью аналитических выражений, уникальных для каждого примитива. Сложность получения аналитических преобразователей затрудняет широкое использование метода. Модель примитивного элемента для дедуктивного алгоритма представлена на рисунке 13.14.

Рисунок 13.14. Фрагмент схемы для моделирования дефектов



Элемент E_i имеет K входов и один выход, которым соответствуют значения двоичных сигналов Y_1, Y_2, ..., Y_j, ..., Y_k, Y_i. Для каждого входа j известен входной список неисправностей для фиксированного значения Y_j. Выходной список Z_i при заданном Y_i зависит от функции элемента, значений сигналов на его входах, входных списков неисправностей. Для ПЭ И (И-НЕ) формула получения выходного списка имеет вид:

$$Z_i = \begin{cases} (\cup Z_i) \cup (\equiv \overline{Y_i}) \leftarrow J_0 = \varnothing, \forall j \in J; \\ (\cap Z_i) \setminus (\cup Z_k) \cup (\equiv \overline{Y_i}) \leftarrow J_0 \neq \varnothing, \forall j \in J_0, \forall k \in J_1, \end{cases}$$

где $J_0(J_1)$ – подмножество входов, имеющих значение 0(1), $J=\{J_0,J_1\}$; $(\equiv \overline{Y}_i)$ – неисправность выхода i, определяемая инверсией его исправного состояния.

Для элемента ИЛИ (ИЛИ-НЕ) формула имеет следующий вид:

$$Z_i = \begin{cases} (\cup Z_i) \cup (\equiv \overline{Y_i}) \leftarrow J_1 = \varnothing, \forall j \in J; \\ (\cap Z_i) \setminus (\cup Z_k) \cup (\equiv \overline{Y_i}) \leftarrow J_1 \neq \varnothing, \forall j \in J_1, \forall k \in J_0. \end{cases}$$

Для линии разветвления i с исходящими ветвями ($i_1, i_2, ..., i_j,...,i_p$) справедлива формула

$$Z_{ij} = Z_i \cup (\equiv \overline{Y_j}).$$

Она имеет смысл, если выполняется моделирование неисправностей внешних контактов элементов, но не линий. Генерация списков неисправностей для внешних входов выполняется в соответствии с выражением $Z_j = (\equiv \overline{Y_j})$.

Для реализации дедуктивного алгоритма необходимо получить аналитические формулы для каждого типа ПЭ или преобразовывать схему к вентильному уровню в базисе И-ИЛИ-НЕ, для которого имеются выражения вычисления списков.

Демонстрацию алгоритма выполним на ранее приведенном примере (см. рисунок 13.13) при подаче двоичной последовательности 10001. Порядок обработки элементов не отличается от исправного моделирования. Списки неисправностей внешних входов с учетом приведенных формул имеют вид $Z1=\{1^0\}$; $Z2=\{2^1\}$; $Z3=\{3^1\}$. Далее формируем $Z4=(Z2\backslash Z1) \cup \{4^1\}=\{2^1,4^1\}$. После этого вычисляем результат: $Z5=\{Z3,Z4\} \cup \{5^0\}=\{3^1,2^1,4^1,5^0\}$.

Рассмотренный метод требует больших затрат памяти для хранения списков, а его эффективность зависит от программной реализации операций над множествами. В упомянутом изложении он ориентирован на вентильный уровень описания модели объекта как и метод параллельного моделирования. Однако аналитические выражения вычисления списков транспортируемых неисправностей можно заменить на явные условия модификации выходов при наличии ОКН. Например, для ПЭ 2И при входном слове 11 проверяемые дефекты определяются кубами (1^0: 010, 2^0: 100, 3^0: 000). Учитывая, что таким образом можно записать как константные, так и функциональные неисправности для ПЭ произвольного базиса, дедуктивный алгоритм является универсальным. Его работа при анализе последовательностных схем может иметь значительное число итераций, вследствие изменения входного слова или наличия неисправностей, вызывающих генераторный режим в ЦУ. Это приводит к модификации списков проверяемых дефектов на каждой итерации, а значит к повторению времяемкой процедуры пересечения и объединения подмножеств проверяемых неисправностей.

В совместном (конкурентном) моделировании, как и в дедуктивном алгоритме, за один проход по схеме определяются все неисправности, проверяемые на тест-векторе. Но в отличие от последнего, где дефекты моделируются дедуктивно (неявно), конкурентный алгоритм анализирует явно исправную работу и те неисправности, которые модифицируют состояния входов или выходов. (Параллельный алгоритм также выполняет явное, но одновременное моделирование дефектов и исправного поведения с помощью логических операций над машинными словами.) Класс обрабатываемых дефектов при совместном моделировании включает ОКН, кратные константные (ККН) и функциональные неисправности. Для примитива модель неисправности задается кубом, который модифицирует состояние выхода или входа относительно вектора исправного поведения для рассматриваемого ПЭ. Неисправностный список элемента для ОКН есть функция от входного слова.

Например, для полусумматора с выходами суммы и переноса при входном векторе 00, для которого вектор моделирования равен 0000, такой список имеет вид $A^1=1010$, $B^1=0110$, $S^1=0010$, $P^1=0001$. Внесение неисправности по входу ПЭ требует анализа КП в целях определения состояния его выходов. Для задания дефекта на выходной линии следует записать его тип в соответствующую координату вектора исправного моделирования. При обработке схемы для каждого ПЭ относительно вектора исправного поведения определяется подмножество дефектов, транспортируемых через анализируемый элемент. При этом важно, чтобы общий неисправностный список примитива был упорядочен в соответствии с возрастанием номеров линий

схемы (возможен и другой порядок), что исключает переборный характер процедуры анализа условий транспортирования дефекта через ПЭ. Построение входных условий проверки неисправности основано на маскировании состояний входов ПЭ. Если, например, для четырехвходового конъюнктора на входном наборе 0110 имеются внешние дефекты (7^1,7^1) на первом и четвертом входах, то куб неисправности 7^1 равен 1111, который после моделирования КП доопределяется значением выхода и принимает вид 11111. Видимость кратного дефекта на входах элемента есть следствие наличия в схеме сходящегося разветвления. Чтобы выявить двоичную или большую кратность, следует выполнять сравнение неисправностей линий, которые являются соседними элементами в неисправностном списке ПЭ. Такая процедура всегда приводит к положительному результату при наличии кратности, поскольку список упорядочен.

Алгоритм анализа ПЭ при совместном моделировании неисправностей эквипотенциальных линий включает пять пунктов.

1. Исправное моделирование входного набора.

2. Определение кубов входных неисправностей и их моделирование (если ОКН входной линии проверяется, то на выходе рассматриваемого ПЭ обнаруживаются все внешние ОКН, протранспортированные к данному входу).

3. Определение кубов кратных внешних дефектов и их моделирование (если внешние неисправности одиночные, то условия их транспортирования задаются в п.2).

4. Сравнение состояний выходов ПЭ при отсутствии и наличии дефектов в целях определения выходного списка проверяемых ОКН.

5. Если выходы ПЭ есть внешние линии схемы, то следует в множество обнаруженных неисправностей включить инверсные, по отношению к исправному поведению, значения дефектов выходных линий. При состоянии исправного поведения, равного символу X, его инверсия равна пустому множеству.

13.5. Дедуктивно-параллельное моделирование неисправностей цифровых систем

Достижения микроэлектроники позволяют создавать интегральные микросхемы, содержащие миллионы эквивалентных вентилей на кристалле. Это дает возможность проектировать сложные специализированные вычислительные устройства с помощью средств автоматизированного проектирования известных фирм (Aldec, Cadence, Altera, Xilinx, Synopsys). Однако автоматизация процесса верификации таких структурно- и функционально-сложных цифровых систем требует создания новых методов и средств синтеза тестов и моделирования неисправностей, способных за приемлемое время построить входные последовательности проверки дефектов необходимой полноты. Быстродействие реализации особенно псевдослучайных алгоритмов генерации тестов на 80-90% зависит от времени анализа неисправностей. Поэтому разработка нового быстродействующего метода моделирования одиночных константных дефектов для синтеза тестов верификации проектируемых цифровых систем на основе программируемой логики является актуальной проблемой.

Объект тестирования представлен в форме булевых уравнений, записанных на языке VHDL, реализующих сложную цифровую систему, имплементируемую в кристаллы программируемой логики.

Ниже рассмотрены вопросы, связанные с теоретическим обоснованием объединения методов кубического, дедуктивного и параллельного моделирования в целях повышения быстродействия анализа дефектов и определения качества теста. Рассмотрены особенности и достоинства алгоритмической и аппаратурной реализации предлагаемого метода, вычислительная сложность и примеры моделирования тестовых схем.

13.5.1. Математическая модель анализа неисправностей

Основное уравнение тестирования цифровой системы $F = (F_1, F_2, ..., F_i, ...F_n)$, включающей n линий и/или функциональных элементов, представлено в виде:

$$L \oplus F = T, \tag{13.1}$$

где тест $T = (T_1, T_2, ..., T_t, ..., T_k)$ для реконфигурируемых на нем моделей $L = (L_1, L_2, ..., L_t, ..., L_k)$ анализа неисправностей определяется линейным взаимодействием упомянутых компонентов по правилу (13.1). Данное уравнение может быть трансформировано к виду

$$L \oplus T = F. \tag{13.2}$$

В этом случае формализуется процесс идентификации функции при решении задач диагностирования.

Последний вариант линейной перестановки, задаваемый уравнением

$$T \oplus F = L, \tag{13.3}$$

определяет правила формирования множества дедуктивных функций параллельного моделирования неисправностей (ДФПМН или ДФ) на тесте Т для модели исправного поведения F. Естественно, что каждой ДФПМН можно поставить в соответствие схему, тогда аббревиатура будет иметь вид ДСПМН или ДС.

Компонент функционального описания цифровой системы $F_i \in F$ представляет собой булеву функцию для вычисления состояния i-й линии:

$$F_i = f_i(X_{i1}, X_{i2}, ..., X_{ij}, ...X_{in_i}). \tag{13.4}$$

В качестве результата исправного моделирования функции f_i выступает значение координаты $T_{ti} \in T_t$, причем $T_{ti} = F_i$ на тесте t. При этом имеется в виду, что тест есть матрица исправного поведения цифровой системы

$$T = [T_{ti}] = (T_{t1}, T_{t2}, ..., T_{ti}, ..., T_{tn}). \tag{13.5}$$

С учетом разбиения теста на составляющие векторы уравнение (13.3) получения ДФПМН для $T_t \in T$ принимает следующий вид:

$$L_t = T_t \oplus F.$$

При условии, что функциональное описание цифровой системы представлено компонентами, формирующими состояния всех линий схемы, в качестве формулы преобразования исправной модели в дедуктивную функцию выступает выражение

$$L_{ti} = f_i[(X_{i1} \oplus T_{t1}), (X_{i2} \oplus T_{t2}), ..., (X_{ij} \oplus T_{tj}), ..., (X_{in_i} \oplus T_{tn_i})] \oplus T_{ti}, \tag{13.6}$$

которое по существу аналогично классической формуле дедуктивного анализа цифровых схем.

Пример 13.1 Для элемента 2И определить ДФПМН на тест-векторе Т = (111).

Решение. В соответствии с (13.6) выполняются эквивалентные преобразования с использованием тождеств алгебры логики ($\neg$ – знак инверсии):

$$L(T = 11, Y = 1, X_1 \wedge X_2] = [(X_1 \oplus T_1) \wedge (X_2 \oplus T_2)] \oplus T_3 =$$
$$= [(X_1 \oplus 1) \wedge (X_2 \oplus 1)] \oplus 1 = (\overline{X}_1 \wedge \overline{X}_2) \oplus 1 = \neg(\overline{X}_1 \wedge \overline{X}_2) = (X_1 \vee X_2).$$

Аналогично получается ДФПМН для функции 2ИЛИ и тест-вектора Т = (111):

$$L(T = 11, Y = 1, X_1 \vee X_2) = [(X_1 \oplus T_1) \vee (X_2 \oplus T_2)] \oplus T_3 =$$
$$= [(X_1 \oplus 1) \vee (X_2 \oplus 1)] \oplus 1 = (\overline{X}_1 \vee \overline{X}_2) \oplus 1 = \neg(\overline{X}_1 \vee \overline{X}_2) = (X_1 \wedge X_2).$$

Путем построения ДФПМН на всех входных наборах для двухвходового элемента И можно получить универсальную дедуктивную функцию, инвариантную инверсии:

$$L[T=(00,01,10,11),(X_1 \wedge X_2)] = L\{(\bar{x}_1\bar{x}_2 \vee \bar{x}_1 x_2 \vee x_1\bar{x}_2 \vee x_1 x_2)[(X_1 \oplus T_1 \wedge X_2 \oplus T_2) \oplus T_3)]\} =$$
$$= (\bar{x}_1\bar{x}_2)[(X_1 \oplus 0) \wedge (X_2 \oplus 0)] \oplus 0 \vee (\bar{x}_1 x_2)[(X_1 \oplus 0) \wedge (X_2 \oplus 1)] \oplus 0 \vee$$
$$\vee (x_1\bar{x}_2)[(X_1 \oplus 1) \wedge (X_2 \oplus 0)] \oplus 0 \vee (x_1 x_2)[(X_1 \oplus 1) \wedge (X_2 \oplus 1)] \oplus 1 =$$
$$= (\bar{x}_1\bar{x}_2)(X_1 \wedge X_2) \vee (\bar{x}_1 x_2)(X_1 \wedge \bar{X}_2) \vee (x_1\bar{x}_2)(\bar{X}_1 \wedge X_2) \vee (x_1 x_2)(X_1 \vee X_2).$$

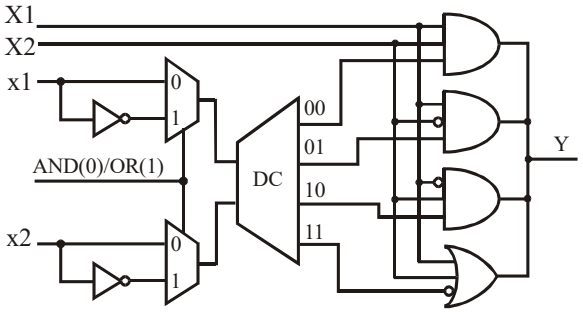
Здесь строчными буквами представлены термы, обозначающие входные тест-наборы, прописными – регистровые переменные, кодирующие векторы проверяемых неисправностей каждой существенной переменной или линии цифрового устройства.

Аналогичное преобразование для функции ИЛИ дает следующую ДФПМН:

$$L[T=(00,01,10,11),(X_1 \vee X_2)] = L\{(\bar{x}_1\bar{x}_2 \vee \bar{x}_1 x_2 \vee x_1\bar{x}_2 \vee x_1 x_2)[(X_1 \oplus T_1 \vee X_2 \oplus T_2) \oplus T_3)]\} =$$
$$= (\bar{x}_1\bar{x}_2)[(X_1 \oplus 0) \vee (X_2 \oplus 0)] \oplus 0 \vee (\bar{x}_1 x_2)[(X_1 \oplus 0) \vee (X_2 \oplus 1)] \oplus 1 \vee$$
$$\vee (x_1\bar{x}_2)[(X_1 \oplus 1) \vee (X_2 \oplus 0)] \oplus 1 \vee (x_1 x_2)[(X_1 \oplus 1) \vee (X_2 \oplus 1)] \oplus 1 =$$
$$= (\bar{x}_1\bar{x}_2)(X_1 \vee X_2) \vee (\bar{x}_1 x_2)(\bar{X}_1 \wedge X_2) \vee (x_1\bar{x}_2)(X_1 \wedge \bar{X}_2) \vee (x_1 x_2)(X_1 \wedge X_2).$$

Как следствие объединения полученных ДФПМН логических элементов И, ИЛИ дедуктивная схема их параллельного моделирования представлена на рисунке 13.15.

Рисунок 13.15. Схема параллельного моделирования неисправностей для AND, OR



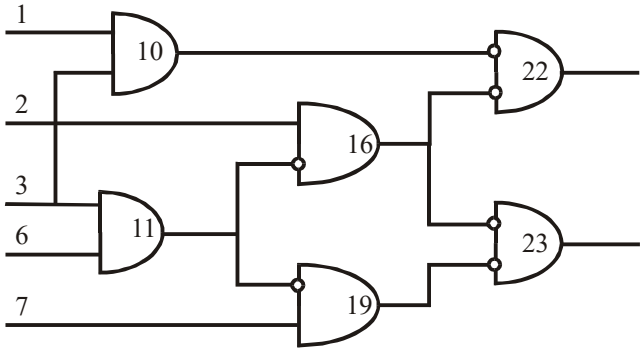
Здесь имеются регистровые переменные X1 и X2, которые представляют собой векторы проверяемых дефектов для каждой линии цифрового устройства, объединенные в матрицу $M=[M_{ij}]$ размерностью n^2. Предварительно данная матрица инициализируется нулями с единичными диагональными элементами, иначе является единичной матрицей:

$$[M_{ij}]\big|_{(i,j=\overline{1,n})} = \begin{cases} 0 \Leftarrow (i \neq j); \\ 1 \Leftarrow (i = j). \end{cases} \qquad (13.7)$$

Затем ее строки обрабатываются с помощью векторных операций AND,OR,NOT, в соответствии с моделью анализа неисправностей, представленной в виде ДСПМН.

Пример 13.2. Дана вентильная структура устройства, изображенная на рисунке 13.16.

Рисунок 13.16. Структура цифрового устройства

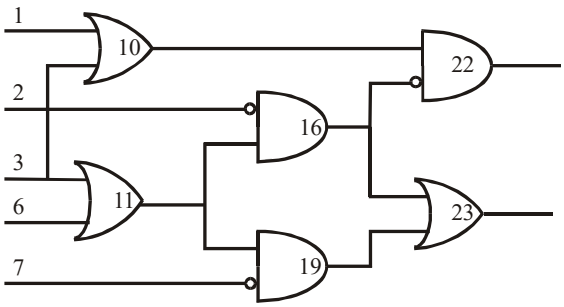


Матрица $M = [M_{ij}]$, соответствующая приведенной выше схеме, имеет вид

M^0	1	2	3	6	7	10	11	16	19	22	23
1	1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	1	0	0	0	0
16	0	0	0	0	0	0	0	1	0	0	0
19	0	0	0	0	0	0	0	0	1	0	0
22	0	0	0	0	0	0	0	0	0	1	0
23	0	0	0	0	0	0	0	0	0	0	1

На входном наборе T=11111110001 схема исправного поведения трансформируется в ДФПМН, изображенную на рисунке 13.17.

Рисунок 13.17. Дедуктивная модель цифрового устройства



Результаты последовательной обработки логических элементов (см. рисунок 13.17) модифицируют исходную матрицу M^0 с помощью формулы

$$M_i = M_i \vee L_{ti},$$

где L_{ti} – дедуктивный элемент, формирующий состояние строки M_i на тесте T_t, приводя ее к следующему виду:

M^1	1	2	3	6	7	10	11	16	19	22	23
1	1	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	0	0	0	0
10	1	0	1	0	0	1	0	0	0	0	0
11	0	0	1	1	0	0	1	0	0	0	0
16	0	0	1	1	0	0	1	1	0	0	0
19	0	0	1	1	0	0	1	0	1	0	0
22	1	0	0	0	0	1	0	0	0	1	0
23	0	0	1	1	0	0	1	1	1	0	1

Упорядоченное множество строк матрицы делится на три подмножества (X, Z, Y) – входных, внутренних и выходных линий цифровой системы.

Строки, относящиеся к наблюдаемым выходам, формируют совместно с вектором исправного поведения T=11111110001 два вектора проверяемых дефектов (S^0,S^1), первый из которых идентифицирует единицами проверяемые константы нуля, второй – единицы:

N	1	2	3	6	7	10	11	16	19	22	23
22	1	0	0	0	0	1	0	0	0	1	0
23	0	0	1	1	0	0	1	1	1	0	1
S	1	0	1	1	0	1	1	1	1	1	1
T	1	1	1	1	1	0	0	1	1	1	0
S^0	1	0	1	1	0	0	0	1	1	1	0
S^1	0	0	0	0	0	1	1	0	0	0	1
D^0	1	0	1	1	0	0	0	1	1	1	0
D^1	0	0	0	0	0	1	1	0	0	0	1

Строка S определяется дизъюнкцией всех $M_i^r (i \in Y)$, относящихся к наблюдаемым выходным линиям схемы:

$$S = \bigvee_{\forall i \in Y} M_i^r . \tag{13.8}$$

В данном случае $S = M_{23}^1 \vee M_{24}^1$. Функции для определения строк S^0, S^1 имеют вид:

$$S^0 = S \wedge T, (S_j^0 = S_j \wedge T_j);$$
$$S^1 = S \wedge \overline{T}, (S_j^1 = S_j \wedge \overline{T}_j). \tag{13.9}$$

Строки D^0 и D^1 представляют проверенные на тестовых наборах неисправности, которые заполняются по мере обработки входных последовательностей, используя выражения

$$D^0 = D^0 \vee S^0, \quad D^1 = D^1 \vee S^1. \tag{13.10}$$

Естественно, что на первом тест-векторе наблюдается эквивалентность $D^0 = S^0, \quad D^1 = S^1$. Тест T проверяет 100% всех одиночных константных неисправностей, если все координаты векторов D^0 и D^1 равны 1. В общем случае качество теста определяется выражением

$$Q(T) = \frac{1}{2n}[\sum_{j=1}^{n} (D_j^0 + D_j^1)]. \tag{13.11}$$

Аналогично вычисляется процент покрытия неисправностей тест-вектором

$$Q(T_t) = \frac{1}{2n}[\sum_{j=1}^{n} (S_j^0 + S_j^1)]. \tag{13.12}$$

Для повышения быстродействия метода моделирования цифровая система представляется двумя моделями $W = \{F, L_0\}$. Первая является компилятивной, включающей реализацию булевых уравнений в коде языка C++, и предназначена для исправного моделирования . Вторая – модифицированная интерпретация первой в табличном исполнении, что необходимо для быстрой модификации

$$L_0 \to L_1 \to ... \to L_t \to ... \to L_k$$

в целях выполнения дедуктивно-параллельного алгоритма анализа неисправностей цифровой системы.

Интерпретативная модель получается из компилятивной путем устранения всех инверсий из термов функционального описания F: $L_0 = \{F \setminus \neg\}$. Такая модификация осуществляется на основании следующего правила: все знаки инверсии над термами должны быть удалены. Обоснование данной модификации представлено в следующем утверждении.

Утверждение 13.1. Если цифровая система представлена в базисе элементарных функций И, ИЛИ, НЕ, то для любого двоичного тест-вектора исходная ДФПМН $L_0 \in W$ не имеет инверсий на выходах логических элементов.

Доказательство. Описание цифровой системы, включающее операции инверсии, может быть упрощено на основании следующих тождеств:

$$(\overline{a \oplus 0}) \oplus 1 = a; \quad (\overline{a \oplus 1}) \oplus 0 = a.$$

С другой стороны, всякая инверсия на выходе функционального элемента может рассматриваться как отдельный инвертор. При этом применение выражения (13.6) к данной функции от одной переменной доказывает несущественность операции инверсии при построении модифицированной интерпретативной модели цифровой системы, в которой исключены все инверсии из термов.

Утверждение 13.2. Тестовая модификация интерпретативной модели цифровой системы может иметь инверсии только на входных переменных элементарных функций.

Доказательство. Основано на рассмотрении всех возможных вариантов двоичных условий на входах логических элементов И, ИЛИ:

$$a \wedge b = [(a \oplus 0) \wedge (b \oplus 0)] \oplus 0 = a \wedge b; \qquad a \vee b = [(a \oplus 0) \vee (b \oplus 0)] \oplus 0 = a \vee b;$$
$$a \wedge b = [(a \oplus 0) \wedge (b \oplus 1)] \oplus 0 = a \wedge \bar{b}; \qquad a \vee b = [(a \oplus 0) \vee (b \oplus 1)] \oplus 1 = \bar{a} \wedge b;$$
$$a \wedge b = [(a \oplus 1) \wedge (b \oplus 0)] \oplus 0 = \bar{a} \wedge b; \qquad a \vee b = [(a \oplus 1) \vee (b \oplus 0)] \oplus 1 = a \wedge \bar{b};$$
$$a \wedge b = [(a \oplus 1) \wedge (b \oplus 1)] \oplus 1 = a \vee b; \qquad a \vee b = [(a \oplus 1) \vee (b \oplus 1)] \oplus 1 = a \wedge b.$$

Другие элементы, имеющие инверсии, приводятся к упомянутым выше в соответствии с утверждением 13.1.

Следствия: 1. Инвертор в исправной схеме не влияет на транспортирование дефектов. 2. Дедуктивный терм не может быть составлен только из инверсных переменных. 3. Дизъюнктивный дедуктивный терм не имеет инверсных переменных.

13.5.2. Дедуктивно-параллельный алгоритм моделирования неисправностей

1. Формирование компилятивной и исходной интерпретативной моделей цифровой системы $W = \{F, L_0\}$. Определение начального значения тест-вектора t=0.

Инициализация векторов проверенных на тесте дефектов

$$\forall_{j=1}^{n} (D_j^0 = 0;\ D_j^1 = 0).$$

2. Определение номера очередного входного набора t=t+1 для $T_t \in T$. Если входных наборов нет (t > k) – конец моделирования.

3. Исправное моделирование всех примитивов (невходных линий) $F_i (i = \overline{1,n})$ цифровой схемы на входном наборе $T_t^X \in T_t$ с использованием компилятивной модели $F \in W$ в целях доопределения невходных координат вектора $T_t^{\overline{X}} \in T_t$:

$$T_t^{\overline{X}} = f(T_t^X, F). \qquad (13.13)$$

Идентичность векторов исправных состояний линий в двух соседних итерациях $T_t^r = T_t^{r-1}$ является условием перехода к следующему пункту.

Для моделирования последовательностных схем и организации событийности [8] используется анализ пары соседних векторов (T_{t-1}, T_t).

4. Инициализация матрицы проверяемых на тест-векторе дефектов $M = [M_{ij}]$ в соответствии с выражением (13.7).

Инициализация векторов проверяемых на тест-векторе дефектов $\forall_{j=1}^{n} (S_j^0 = 0;\ S_j^1 = 0)$.

Реконфигурация примитивов $L_i (i = \overline{1,n})$ интерпретативной модели $L_i \in W$ на основе применения формулы (13.6) для текущего вектора исправного состояния

$$T_t = (T_{t1}, T_{t2}, ..., T_{tj}, ..., T_{tn})$$

в целях получения модификации $L_{ti} = T_t \oplus F_i$.

5. Формирование невходных строк матрицы проверяемых неисправностей путем их параллельного моделирования с помощью примитивов $L_{ti} \in L_t$.

6. Формирование объединенного вектора проверяемых неисправностей S путем применения формулы (13.8). При выполнении условия $\overset{n}{\underset{j=1}{\forall}}(S_j = S_j^0 \vee S_j^1)$ осуществляется вычисление качества тестового набора по (13.12) и переход к следующему пункту, иначе – формирование пары $\{S^0, S^1\}$ по (13.9) или, если наблюдается исчезновение проверяемых неисправностей в векторе S по отношению к паре $\{S^0, S^1\}$: $\overset{n}{\underset{j=1}{\exists}}[(S_j = 0) \& (S_j^0 \vee S_j^1 = 1)]$, выполняется исключение таких дефектов из процесса моделирования по правилу

$$(S_j^0 = S_j^1 = 0) \Leftarrow \overset{n}{\underset{j=1}{\forall}}[(S_j = 0) \& (S_j^0 \vee S_j^1 = 1)].$$

Выполнение перехода к п. 5.

7. Формирование векторов проверенных неисправностей в соответствии с выражением (13.10) и вычисление качества теста по формуле (13.11). Переход к п. 2.

Предложенный алгоритм ориентирован как на табличное описание примитивов произвольной сложности RTL уровня, так и на вентильное представление цифровых систем. Быстродействие алгоритма зависит от представления моделей цифровых устройств, которые могут быть реализованы в компилятивном и интерпретативном исполнении.

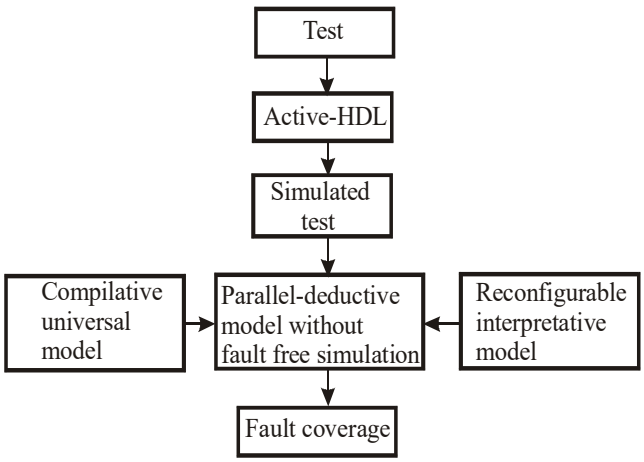
Преимущества дедуктивно-параллельного метода моделирования неисправностей:

1). Исключение операций над списками неисправностей, характерных для дедуктивного метода, имеющих вычислительную сложность n^2. Переход к регистровым параллельным логическим операциям (по 64 или 128 битов) над строками матрицы (таблицы) проверяемых дефектов.

2). Уменьшение времени обработки упомянутой выше таблицы путем использования матрицы достижимостей, позволяющей не обрабатывать координаты, функционально не связанные с моделируемой линией.

3). Возможность реализации параллельных операций на процессоре PRUS, акселераторе HEFS, использование компилятора Active-HDL (рисунок 13.18) для получения тест-векторов исправного поведения, доопределенных по невходным координатам.

Рисунок 13.18. Использование системы Active-HDL для моделирования векторов



4). Событийность обработки матрицы проверяемых дефектов по принципу изменения хотя бы в одном разряде модельного регистра.

5). Реализация событийного моделирования исправного поведения на входном наборе при использовании компилятивной модели цифровой системы.

6). Интерактивная событийная модификация дедуктивной интерпретативной модели параллельного моделирования неисправностей на каждом входном наборе без инверторов на выходах логических элементов.

7). Возможность использования универсального элемента моделирования неисправностей для синтеза схемы анализа дефектов цифровой системы или для построения компилятивной модели параллельно-дедуктивного моделирования неисправностей.

8). Возможность уменьшения размерности матрицы проверяемых дефектов путем сокращения множества неисправностей на основе их эквивалентирования – неразличимости относительно наблюдаемых выходов.

13.5.3. Сравнительный анализ с дедуктивным и параллельным методами

Учитывая, что разработанный метод моделирования неисправностей основывается на использовании преимуществ дедуктивного и параллельного алгоритмов, естественным представляется выполнить сравнительный анализ всех трех реализаций.

Параллельный алгоритм имеет вычислительную сложность C_p, определяемую функциональной зависимостью от числа неэквивалентных неисправностей (b), длины компьютерного слова (W), количества эквивалентных вентилей (G):

$$C_p = (b^2 / W) \times G^3 .$$

Дедуктивный алгоритм имеет некоторые отличия в формуле оценки быстродействия (система CHIEFS):

$$C_d = b^2 \times Q \times G^2 \big|_{Q=G} = b^2 G^3,$$

где Q – среднее число активизированных неисправностями вентилей.

Предлагаемый дедуктивно-параллельный метод имеет быстродействие, определяемое выражением

$$C_{dp} = G^2 + (b^2 / W) \times G^2.$$

Первое слагаемое задает время исправного моделирования, второе – анализа неисправностей цифрового устройства, линии которого не ранжированы. Для комбинационной схемы с ранжированными линиями и элементами быстродействие метода будет иметь следующую оценку:

$$C_{dp}^r = G + (b^2 / W) \times G.$$

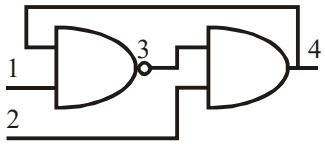
Быстродействие параллельно-дедуктивного метода выше параллельного и дедуктивного ($C_{dp}^r << \{C_p, C_d\}$), благодаря разделению фаз исправного и неисправного моделирований.

13.5.4. Особенности моделирования неисправностей в последовательностных схемах

Особенности анализа последовательностных схем связаны с возникновением генераторного режима – отсутствие условий проверки отдельных неисправностей на функциональных входных наборах.

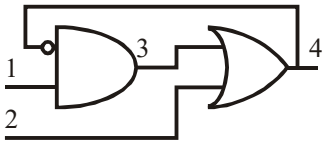
Пример 13.3 Выполнить моделирование дефектов для триггерной схемы (рисунок 13.19) на входном наборе 01.

Рисунок 13.19. Схема с обратными связями



Результаты исправного моделирования схемы определяются вектором 0111. Дедуктивно-параллельная схема для данного тестового набора имеет вид, представленный на рисунке 13.20.

Рисунок 13.20. Дедуктивная схема моделирования



Матрицы проверяемых неисправностей на трех итерациях иллюстрируют генераторный режим – появление и исчезновение дефекта 1^1 среди множества проверяемых:

M^1	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	1	0	1	0
4	1	1	1	1

M^2	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	1	1	1

M^3	1	2	3	4
1	1	0	0	0
2	0	1	0	0
3	1	0	1	0
4	1	1	1	1

.

В этом случае необходимо корректировать списки проверяемых дефектов цифрового устройства. В данном примере уже на второй итерации (матрица M^2) следует исключить из рассмотрения неисправность 1^1. Тогда выполнение итерации I_3, представленное матрицей

M^3	1	2	3	4
1	0	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	1	1	1

,

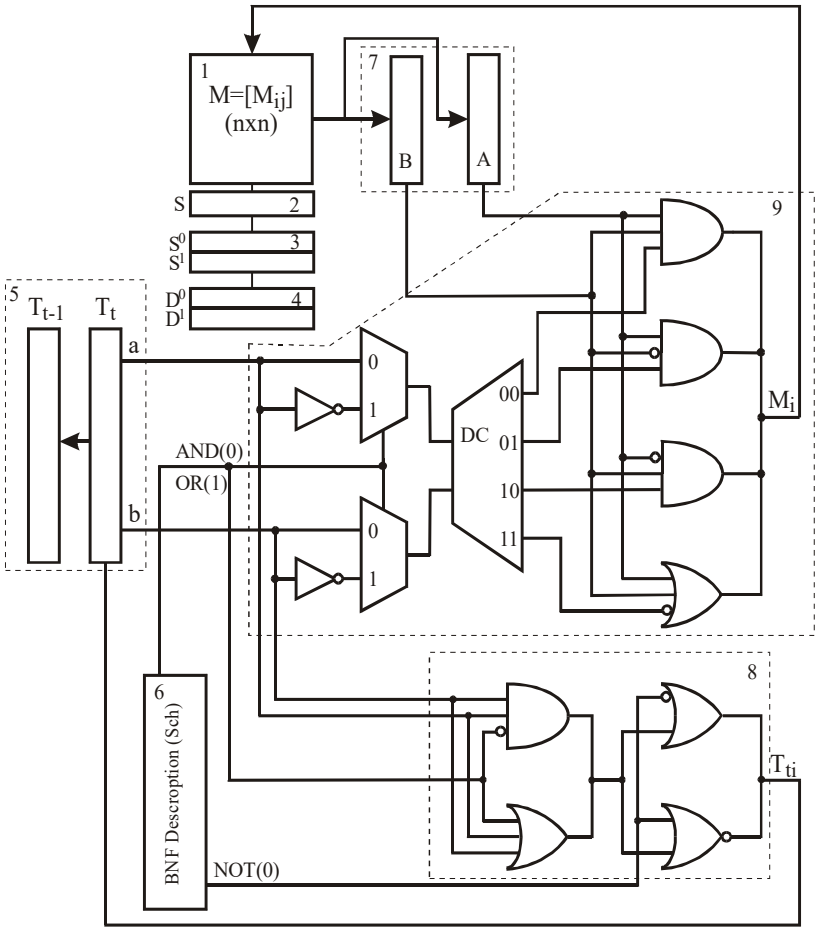
дает окончательный список проверяемых неисправностей на тест-векторе 0111, равный $L_4 = \{2^0, 3^0, 4^0\}$.

Здесь физическая причина исключения дефекта 1^1 заключается в инициировании им режима генерации изменяющихся сигналов при моделировании линий 3 и 4: 11-00-11-...-00. В этом случае состояния невходных линий цифрового устройства принудительно определяются символами неопределенности X, что является условием отсутствия проверки дефекта 1^1.

13.5.5. Аппаратурная реализация дедуктивно-параллельного моделирования дефектов

Учитывая, что параллельные процессы являются доминирующими и наиболее времяемкими в предлагаемом методе моделирования, представляется целесообразной его аппаратурная реализация (рисунок 13.21).

Рисунок 13.21. Устройство дедуктивно-параллельного моделирования неисправностей



Основная идея при этом заключается в разделении устройства моделирования на следующие компоненты:

1. Блок памяти для хранения матрицы $M = [M_{ij}]$ проверяемых дефектов размерностью n^2.

2. Регистр S[1...n] для хранения признаков проверки (1), непроверки (0) неисправностей линий цифрового устройства, инверсных по отношению к состояниям линий T_{ti} тест-вектора T_t, формируемый по правилу, определенному в (13.9).

3. Регистровая пара (S^0[1...n], S^1[1...n]) для хранения признаков проверки (1), непроверки (0) одиночных константных неисправностей $(\equiv 0, \equiv 1)$ соответственно на тест-векторе T_t.

4. Регистровая пара (D^0[1...n], D^1[1...n]) для хранения признаков проверки (1), непроверки (0) одиночных константных неисправностей $(\equiv 0, \equiv 1)$ соответственно на тесте T.

5. Регистровая пара (T_{t-1}[1...n], T_t[1...n]) для хранения двоичных состояний линий цифрового устройства при подаче на него двух соседних тест-векторов ($T_{t-1}T_t$).

6. Блок памяти для хранения схемного описания (BNF description), включающего: номер и тип элемента, число входов, номера (идентификаторы) входных переменных и выхода.

7. Буферные регистры (A, B) для хранения операндов перед выполнением регистровых операций AND, OR, NOT над строками матрицы проверяемых дефектов $M = [M_{ij}]$.

8. Модуль вычисления исправных состояний невходных переменных $T_{ti}\,(i = \overline{1,n})$ цифрового устройства на тест-векторе T_t по значениям булевых переменных (a,b) бинарных логических операций.

9. Модуль векторных логических операций над регистровыми переменными (A,B), формирующий матрицу $M = [M_{ij}]$ на тест-векторе T_t.

Алгоритм работы устройства моделирования состоит из реализации процедур исправного моделирования и анализа дефектов. Первоначально заносится информация – схемное описание в блок 6, формируется единичная матрица в блоке 1, обнуляются регистры блоков 2-5. Информация с блока 6 – тип элемента (И, ИЛИ) поступает на блок 9 в целях выбора операций для формирования векторов проверяемых дефектов совместно с тестовыми сигналами, поступающими на входы a,b блоков 8, 9 с блока 5, которые формируют сигналы на выходе блока 8, поступающие далее в блок 5, доопределяя невходные координаты. После выбранного дешифратором одного из четырех элементов на его входы подается содержимое регистровых переменных A и B, поступающее за два временных такта в блок 7. Результат векторной логической операции с выхода M_i в третьем такте поступает в блок памяти 1, где формируются векторы проверяемых дефектов для всех линий схемы. После обработки всех элементов, находящихся в блоке 6, выполняется формирование ячеек блока 2 и повторение процедуры в целях установления факта сходимости моделирования исправного поведения и неисправностей. После этого формируется содержимое ячеек блока 3 и 4. По окончании моделирования всех наборов теста в блоке 4 формируется вектор проверенных дефектов, на основе анализа которого определяется качество теста в процентах.

Метод дедуктивно-параллельного моделирования неисправностей на реконфигурируемых моделях цифровых устройств, представляющий сочетание достоинств дедуктивного и параллельного алгоритмов, ориентирован на обработку цифровых устройств вентильного и регистрового уровней описания. Его основу составляет объединение достоинств дедуктивного и параллельного моделирования в целях повышения быстродействия анализа дефектов и определения качества тестов. Технические характеристики программы моделирования неисправностей, реализованной на языке Visual C++ для IBM PC (Pentium II, 500Мгц): среднее быстродействие – 1000 векторов в секунду (в/с) для схем, содержащих 1 000 линий (2 000 вентилей); 100 в/с при 3 000 линий (6 000 вентилей); 30 в/с при 5 000 линий (10 000 вентилей). Статистика обработки тест-примеров цифровых схем дедуктивным и дедуктивно-параллельным методами при моделировании 1000 входных последовательностей представлена в таблице (рисунок 13.22). Графики сравнительного анализа реализации двух методов моделирования неисправностей цифровых устройств изображены на рисунках 13.22-13-25.

Рисунок 13.22. Табличный анализ быстродействия метода моделирования

Схема	n	Время		dt	Nt
		ДМ	ДПМ		
c17	17	1,372	0,010	1,362	137,200
c432	432	14,030	0,160	13,870	87,688
c499	499	18,206	0,240	17,966	75,858
c880	880	32,056	0,450	31,606	71,236
c1355	1355	46,037	0,751	45,286	61,301
c1908	1908	82,139	1,842	80,297	44,592
c2670	2670	134,566	6,409	128,157	20,996
c3540	3540	176,367	8,592	167,775	20,527
c5315	5315	285,356	18,787	266,569	15,189
c6288	6288	319,456	17,185	302,271	18,589
c7552	7552	502,082	44,004	458,078	11,410

Обозначения (таблица): n – количество вентилей; ДМ – дедуктивный метод; ДПМ – дедуктивно-параллельный метод; $dt = t_{дм} - t_{дпм}$ – разность времен обработки двумя методами; $Nt = t_{дм} / t_{дпм}$ – отношение времен обработки двумя методами.

Рисунок 13.23. Сравнительный анализ времени моделирования двух методов

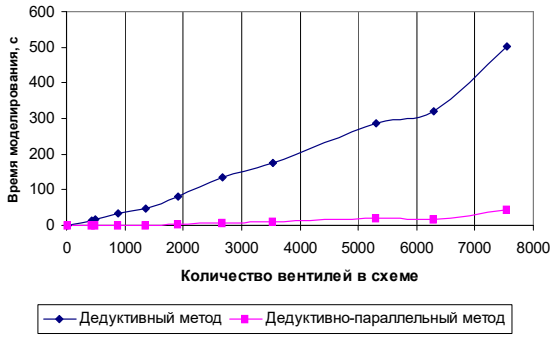


Рисунок 13.24. Абсолютный и относительный $Nt = t_{дм}/t_{дпм}$ временной выигрыш

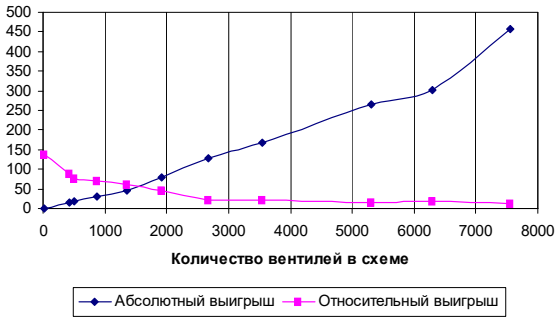
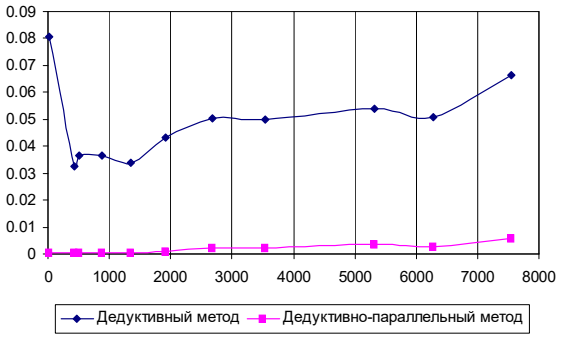


Рисунок 13.25. Время обработки одного вентиля $(t_{дм}/n),(t_{дпм}/n)$



13.6. BDP-метод моделирования неисправностей для синтеза тестов цифровых проектов

Рассматривается быстродействующий дедуктивно-параллельный метод обратного моделирования неисправностей, использующий процедуру суперпозиции решений, ориентированный на обработку сверхбольших проектов вентильного и регистрового уровней описания. Описаны структуры данных и программно-ориентированные алгоритмы для реализации метода в составе автоматической системы генерации тестов.

Актуальность описанного метода определяется необходимостью значительного улучшения средств моделирования и генерации тестов для структурно, и функционально, сложных цифровых систем, имплементированных в кристаллы программируемой логики. Существующие автоматические системы тестирования известных фирм: Cadence, Mentor Graphics, Synopsys, Logic Vision [www.cadence.com, www.logicvision.com, www.simucad.com, www.syntest.com, www.synopsys.com, www.mentorgraphics.com] ориентированы на обработку кристаллов размерностью около 100 тыс. вентилей за приемлемое время, составляющее несколько часов. Но уже сейчас ясно, что данные

средства синтеза тестов и моделирования неисправностей могут быть неприемлемыми относительно времени обработки чипов, насчитывающих несколько миллионов вентилей. Нужны принципиально новые подходы, позволяющие на порядок повысить быстродействие анализа цифровой системы на стадии ее проектирования в целях построения тестов верификации. Один из них, решающий проблему создания быстродействующего метода моделирования одиночных константных дефектов для оценки качества тестов проверки неисправностей проектируемых цифровых систем на основе программируемой логики, предлагается в данном разделе.

Объект тестирования – цифровая система, представленная в форме булевых уравнений, записанных на VHDL, имплементируемая в кристаллы программируемой логики.

Цель метода – моделирование одиночных константных неисправностей для оценки качества синтезируемых тестов цифровых систем, имплементируемых в ПЛИС, содержащих миллионы вентилей. Практически приемлемым следует считать время моделирования, линейно зависящее от числа линий, если количество сходящихся разветвлений составляет не более десяти процентов от общего числа.

Задачи, подлежащие решению:

1. Создание обобщенной модели процесса дедуктивно параллельного анализа цифровой схемы на основе процедуры обратной суперпозиции.

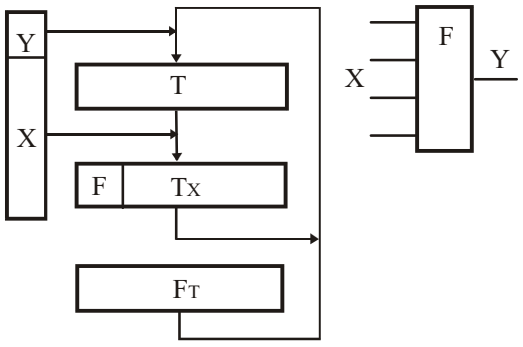
2. Разработка алгоритмов структурно-функционального анализа цифровых систем в целях определения множества сходящихся разветвлений и реконфигурации структуры для реализации процедуры суперпозиции.

3. Создание внутренней интерпретативно-компилятивной модели цифрового устройства для эффективного исправного анализа логических элементов и их неисправностей одиночного константного типа.

4. Алгоритмическая реализация метода моделирования на основе реконфигурирования модели устройства в процессе моделирования неисправностей в целях существенного уменьшения времени оценки качества тестов.

Основу обратного дедуктивно-параллельного (Backtraced Deductive-Parallel) (ОДП) метода моделирования неисправностей составляют: кубическое моделирование дефектов, дедуктивная модель транспортирования неисправностей (13.6), дедуктивно-параллельный алгоритм моделирования неисправностей, рассмотренный ранее, и алгоритм обратного прослеживания примитивов при обработке цифрового устройства. Следует заметить, что быстродействие алгоритма исправного поведения практически инвариантно к компилятивным и интерпретативным моделям цифровых устройств, однако чисто интерпретативная реализация является более технологичной с позиции программирования. Поэтому интерес представляет реализация структур данных внутреннего машинного представления обрабатываемого цифрового устройства в виде интерпретативной модели. Структурная модель логического анализа примитивного элемента цифрового устройства представлена на рисунке 13.26.

Рисунок 13.26. Структурная модель анализа примитивного элемента схемы

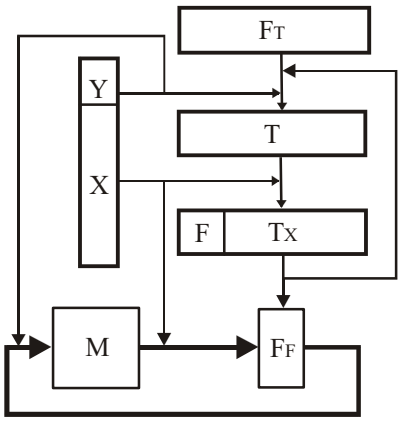


Здесь речь идет о процедуре определения состояния координаты тест-вектора T(Y), соответствующей выходу Y логического элемента F, по его входным значениям, представленным вектором $X = (X_1, X_2, ..., X_k)$, при условии, что F_T есть таблица истинности многообразия булевых функций, определенная на векторе двоичных переменных $(F * T_X)$:

$$T(Y) = F_T(F * T_X) = F_T(F * T(X_1) * T(X_2) * ... * T(X_k)).$$

Иначе, чтобы определить состояние координаты T(Y), необходимо сформировать двоичный вектор состояния входных переменных T_X на основе вектора номеров линий X (см. рисунок 13.26). Затем следует выполнить конкатенацию полученного вектора с двоичным кодом типа примитива (функции) F в целях получения строки $(F * T_X)$ обобщенной таблицы истинности F_F, где в столбце Y, соответствующем значению функции, находится искомое состояние координаты T(Y). Модель дедуктивно-параллельного анализа неисправностей, кроме структуры, участвующей в исправном моделировании, содержит два дополнительных модуля (M, F_F), как показано на рисунке 13.27.

Рисунок 13.27. Модель дедуктивно-параллельного анализа



Аналитическое выражение для вычисления векторов проверяемых неисправностей, объединенных в матрицу M, с помощью дедуктивной функции F_F, являющейся модификацией F по выражению (13.6), имеет вид:

$$M(Y) = F_F(F * T(X), M(X_1) o M(X_2) o ... o M(X_k)) = F_{(F*T(X))}(M(X_1) o M(X_2) o ... o M(X_k)).$$

Здесь операция, обозначенная символом $o = \{\wedge, \vee\}$, может быть представлена дизъюнкцией или конъюнкцией;

$F_{(F*T(X))}$ – дедуктивный элемент, определяемый двоичным словом-адресом $(F * T(X))$.

Для определения состояния вектор-строки M(Y) необходимо установить адрес (тип) дедуктивной компилятивно реализуемой функции, используя полученную для исправного моделирования конкатенацию двоичных последовательностей $(F * T(X))$. Входные переменные для элемента $F_{(F*T(X))}$ являются регистровыми, теоретическая размерность которых равна числу линий в цифровом устройстве. Далее осуществляется последовательное выполнение (k-1) регистровых операций над входными векторами $M(X_i) \in M$. Результат в виде последовательности M(Y) заносится в матрицу M. Векторная переменная X_i может иметь знак инверсии. Тогда перед

выполнением операции $o=\{\wedge,\vee\}$ осуществляется инверсия содержимого регистровой переменной $M(\overline{X_i}) = \overline{M}(X_i)$.

В качестве примера рассмотрим обобщенную таблицу истинности для определения состояния исправного поведения и выбора двоичного адреса дедуктивного функционального элемента анализа неисправностей:

S	F	X_1X_2	Y	F_F
$\wedge$	00	00	0	00
	00	01	0	01
	00	10	0	10
	00	11	1	11
$\vee$	01	00	0	11
	01	01	1	10
	01	10	1	01
	01	11	1	00
$\overline{\wedge}$	10	00	1	00
	10	01	1	01
	10	10	1	10
	10	11	0	11
$\overline{\vee}$	11	00	1	11
	11	01	0	10
	11	10	0	01
	11	11	0	00

Здесь столбец F – код функции исправного поведения, (X_1,X_2) – двоичные входные наборы таблицы истинности каждой из четырех функций, Y – состояние исправного поведения выхода функций, F_F – код адреса компилятивной модели дедуктивного элемента, которая представлена четырьмя примитивами:

$$F_F = \begin{cases} 00 \to X_1 \wedge X_2; \\ 01 \to X_1 \wedge \overline{X_2}; \\ 10 \to \overline{X_1} \wedge X_2; \\ 11 \to X_1 \vee X_2. \end{cases}$$

Вычислительная сложность обработки цифровой схемы, состоящей из n двухвходовых вентилей, определяется выражением

$$Q = [(2K + A) + A + (2n\tau)/W] = [2(K + A) + (2n\tau)/W] \times n,$$

где K – время, затрачиваемое на конкатенацию битов для получения адреса состояния выхода примитива; A – время выборки содержимого ячейки (бита) по его адресу; τ – время выполнения регистровой операции (and, or, not); W – разрядность регистра.

Если учесть, что первое слагаемое 2(K+A) является несущественным по сравнению со вторым, то вычислительная сложность будет представлена формулой

$$Q = (2n^2\tau)/W \cdot$$

Таким образом, затраты времени обработки цифрового устройства пропорциональны квадрату числа вентилей.

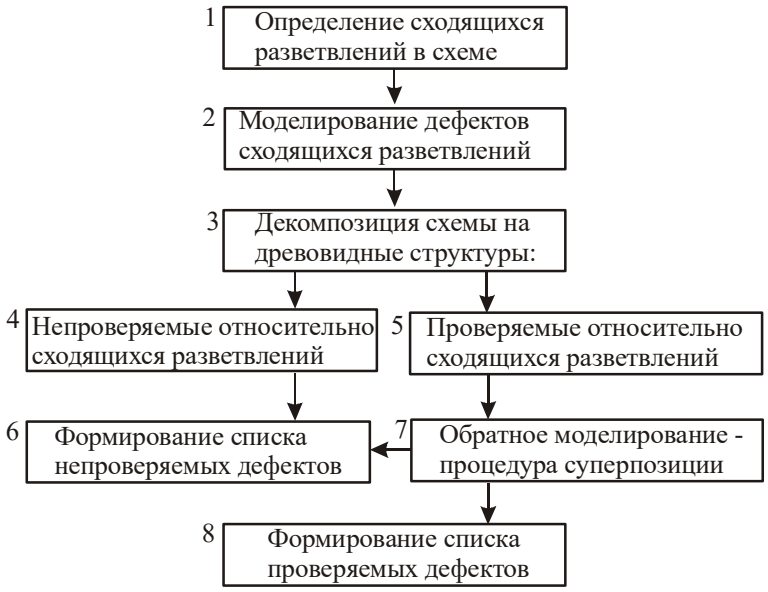
13.6.1. Алгоритмическая реализация ОДП-метода моделирования

Предложенная интерпретативно-компилятивная модель дедуктивно-параллельного анализа неисправностей и исправного поведения является базовой для ОДП-метода и гарантирует нахождение решения в виде множества всех одиночных дефектов,

проверяемых на тест-векторе за n^2 итераций. Для упрощения вычислительной сложности нахождения решения предлагается стратегия моделирования цифровых устройств, представленная на рис.13.28.

Основная идея повышения быстродействия моделирования неисправностей связана с преобразованием сходящихся разветвлений в псевдовыходы в целях последующего применения процедуры суперпозиции для древовидных структур и их необработки в случае фиксации непроверяемости линий сходящихся разветвлений.

Рисунок 13.28. Стратегия ОДП-метода моделирования



Стратегия ОДП-метода моделирования неисправностей цифрового устройства с предварительным структурным анализом (см. рисунок 13.28) включают следующие шаги:

1. Идентификация линий сходящихся разветвлений, инвариантных по отношению к тест-векторам. Вычислительная сложность данной процедуры $Q_r = n^2$, но она выполняется на стадии предварительного анализа и практически не влияет на быстродействие моделирования тест-векторов.

2. Моделирование неисправностей линий сходящихся разветвлений на тест-векторе. Модификация схемной структуры путем преобразования сходящихся разветвлений в псевдовыходы цифрового устройства.

3-4. Вычисление линий подграфов схемы, моделирование неисправностей которых на тест-векторе не должно проводиться вследствие существования формального доказательства их непроверяемости.

5. Определение фрагментов графа цифрового устройства, корректных для выполнения суперпозиции решений на тест-векторе.

6-8. Выполнение процедуры суперпозиции векторов проверяемых неисправностей примитивов на скорректированной модели цифрового устройства.

Интерес представляет алгоритм определения сходящихся разветвлений на основе анализа графовой структуры цифрового устройства. Сложность выполнения алгоритма равна квадрату числа линий в схеме. Для комбинационных схем основные шаги представлены пунктами:

1). Пусть все линии цифрового устройства можно разбить на подмножества:

$$V = (V^Y, V^S, V^R),$$

где Y – идентификатор линии, относящейся к внешним выходам; S – обозначение линии, имеющей одного преемника, соединенной с одним элементом; R – идентификатор линии разветвления, имеющей более одного элемента-преемника.

Определение множества преемников для каждой линии схемы путем формирования вектора числа преемников:

$$V = (V_1, V_2, ..., V_i, ..., V_n), \quad V_i = \begin{cases} 0 \leftarrow V_i \in V^Y; \\ 1 \leftarrow V_i \in V^S; \\ \geq 2 \leftarrow V_i \in V^R. \end{cases}$$

2). Выбор очередной линии $V_i \in V^R$ для определения ее принадлежности к множеству сходящихся разветвлений $V_i^R \in V^{RC}$. Данная процедура выполняется путем логического моделирования графовой структуры от линии $V_i^R \in V$ на множестве всех ее преемников до внешних выходов схемы. Первоначально все линии обнуляются:

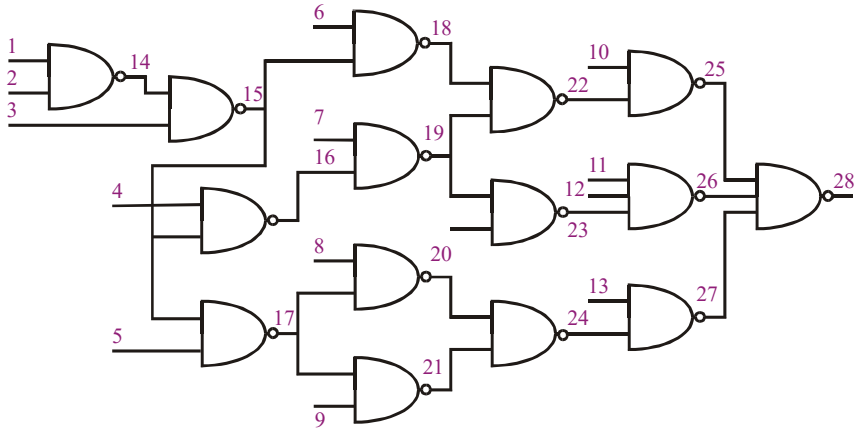
$$\forall_{i=1}^{n} V_i = 0.$$

3). Присвоение рассматриваемой линии разветвления значения 1: $V_i^R \in 1$. После этого реализуется последовательность операций $V_i = V_i + 1$ для всех линий, являющихся преемниками V_i^R. Если на некотором шаге будет зафиксирован результат $V_i = V_i + 1 = 2$, являющийся критерием сходимости для разветвления V_i^R, то оно заносится в список $V_i^R \in V^{RC}$ и осуществляется переход к пункту 2.

Повторение пунктов 2 и 3 выполняется для всех линий разветвления.

В качестве примера структурного анализа выступает схема, представленная на рисунке 13.29, где линии 15, 17, 19 есть сходящиеся разветвления.

Рисунок 13.29. Схема с разветвлениями



В результате применения алгоритма их поиска схема реконфигурируется в четыре древовидные структуры, определяемые подграфами с корневыми вершинами, которые являются выходами или псевдовыходами устройства (рисунок 13.30).

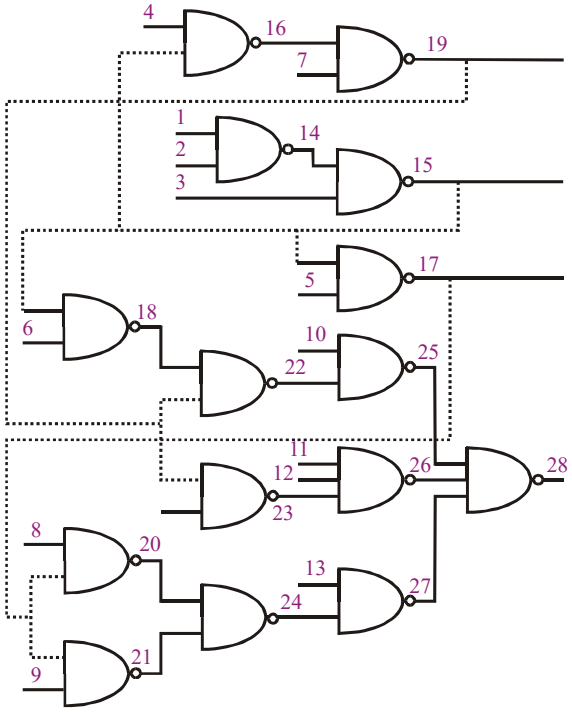
Моделирование неисправностей такой схемы с помощью обратной суперпозиции требует уже линейных затрат памяти и времени в функции от числа эквипотенциальных линий и квадратичных затрат для обработки сходящихся разветвлений:

$$Q = (r^2 / W) + n_r + n_p + (n - r - r^0),$$

где (r^2 / W) – время моделирования неисправностей r сходящихся разветвлений, число которых определяется как $r = 0{,}2 \times n$; $n_r = n$ – время реконфигурирования примитивов схемы на входном наборе; $n_p = n$ – время поиска подграфов линий, соответствующих непроверяемым сходящимся разветвлениям; $(n - r - r^0) = n - 0{,}2 \times n - 0{,}4 \times n = 0{,}4 \times n$ – время выполнения процедуры суперпозиции на множестве линий схемы без сходящихся разветвлений и предшественников для непроверяемых сходящихся разветвлений. Учитывая фактические значения указанных параметров в функции от числа линий схемы, можно получить следующую оценку быстродействия ОДП-метода:

$$Q = [(0{,}2 \times n)^2 / W] + n + n + (n - 0{,}2 \times n - 0{,}4 \times n) = [(0{,}2 \times n)^2 / W] + 2{,}4 \times n).$$

Рисунок 13.30. Древовидные фрагменты схемы



Таким образом, выигрыш в быстродействии предложенного метода тем больше, чем меньше сходящихся разветвлений в схеме цифрового устройства.

С учетом предварительного вычисления сходящихся разветвлений алгоритмическая реализация ОДП-метода представлена следующими пунктами:

1. Фаза прямого исправного моделирования цифровой схемы. Предназначена для определения реакции всех невходных линий схемы на тест-вектор $T_t \in T = [T_{ti}]$.

Множество всех линий дифференцируется на входные, внутренние и выходные: (X, Y, Z). Это означает, что тест-строка (-столбец) из матрицы исправного поведения T представляется как $T_t = (T_t^X, T_t^Z, T_t^Y)$. То же самое относится и к вектору моделирования неисправностей $S = (S^X, S^Z, S^Y)$, который каждый раз строится для новой строки T_t.

2. Инициализация вектора моделирования неисправностей цифровой схемы $S = (S_i^X = 0, S_i^Z = 0, S_i^Y = 1)$. Единичное значение координаты вектора $S_i = 1$ является

индикатором проверки одиночной неисправности, инверсной двоичному исправному состоянию линии T_{ti}.

3. Дизъюнкция вектора проверяемых входных дефектов i-го примитива $S^i = (S^i_1, S^i_2, ..., S^i_j, ..., S^i_{n_i})$ с вектором S моделирования неисправностей схемы при условии, что на линии, соответствующей выходу i-го элемента, имеется единичное значение $S_i = 1$:

$$S(I^i_j) = S(I^i_j) \bigvee_{j=1}^{n_i} S^i_j \Leftarrow S_i = 1,$$

где $I^i = (I^i_1, I^i_2, ..., I^i_j, ..., I^i_{n_i})$ – вектор номеров входных линий i-го примитива. Последний анализируется на основе применения дедуктивно-параллельного алгоритма к матрице проверяемых дефектов, но не схемы, а рассматривемого элемента. Такой анализ можно выполнять и дедуктивно, используя собственные входные списки проверяемых дефектов примитива. Алгоритм анализа неисправностей сходящихся разветвлений:

1. Определение двоичного вектора линий сходящихся разветвлений

$$R = (R_1, R_2, ..., R_i, ..., R_n),$$

$$R_i = \begin{cases} 1 \leftarrow R_i \in R^{RC}; \\ 0 \leftarrow R_i \notin R^{RC}, \end{cases}$$

относительно которого выполняется процедура дедуктивно-параллельного анализа.

2. Генерирование исходных списков неисправностей линий сходящихся разветвлений схемы

$$\{S_j\} = \begin{cases} j^{\overline{T}tj} \leftarrow R_j = 1; \\ \varnothing \leftarrow R_j = 0, \end{cases}$$

где $\{S_j\} \subseteq S; j = \overline{1,n}$; S – вектор списков дефектов.

3. Моделирование неисправностей линий сходящихся разветвлений $L_R \subseteq L = (L_R, L_{\overline{R}})$ дедуктивно-параллельным или дедуктивным методом на реконфигурируемой модели устройства, соответствующей тест-вектору T_t. Использование упомянутого метода обусловлено незначительным ($\approx 20\%$) числом сходящихся разветвлений, поскольку $L_R / L_{\overline{R}} << 1$.

4. Исключение из процесса моделирования неисправностей древовидных подграфов с корневой вершиной, являющейся сходящимся разветвлением, неисправность которого не проверяется:

$$L_U = L_U \setminus L^0_R \cup f^{-1}(L^0_R).$$

5. Моделирование неисправностей линий, дополняющих сходящиеся разветвления и подграфы с непроверяемыми корневыми вершинами до полного множества $L_U = L_U \setminus [L^1_R \cup L^0_R \cup f^{-1}(L^0_R)]$ на основе суперпозиции. Анализ проверки дефектов выполняется только относительно выходных наблюдаемых линий схемы, дополненных разветвлениями, неисправности которых обнаруживаются на тест-векторе T_t:

$$L_Y = L_Y \cup L^1_R \subseteq L_R = \{L^0_R, L^1_R\}; |L_R| = r,$$

где L_Y, L_R^0, L_R^1 – выходные линии схемы, разветвления с непроверяемыми и обнаруживаемыми неисправностями соответственно.

Последовательно рассматриваются все линии, которые являются предшественниками для полученного множества $L_Y = L_Y \cup L_R^1$. Относительно каждой линии из списка L_Y выполняется процедура суперпозиции для входного тестового набора.

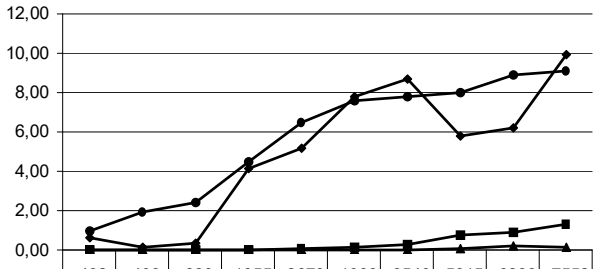
ОДП-метод моделирования неисправностей ориентирован на обработку сверхсложных цифровых систем на основе ПЛИС, содержащих миллионы вентилей. Тестовые эксперименты программной реализации метода на нескольких десятках цифровых комбинационных и последовательностных схемах дают хорошие результаты по быстродействию с традиционными алгоритмами параллельного и дедуктивного анализа. Отдельные примеры сравнения быстродействия разработанного метода и существующих базовых представлены на рисунке 13.31. Ускорение моделирования составляет не менее десяти раз. Таким образом, основным преимуществом метода является усовершенствование дедуктивно-параллельного метода моделирования неисправностей цифровых систем является его усовершенствование, заключающееся в:

1) создании обобщенной модели процесса дедуктивно-параллельного анализа цифровой схемы на основе процедуры обратной суперпозиции, имеющей линейную вычислительную сложность от числа линий схемы;

2) разработке алгоритмов структурно-функционального анализа цифровых систем в целях определения множества сходящихся разветвлений и реконфигурации структуры для реализации процедуры суперпозиции;

3) создании внутренней интерпретативно-компилятивной модели цифрового устройства для эффективного исправного анализа логических элементов и их неисправностей одиночного константного типа.

Рисунок 13.31. Анализ быстродействия: Nem – система Nemesis; TT– Turbo Tester; ДПМ – дедуктивно-параллельный метод; МОМ – ОДП-метод



	c432	c499	c880	c1355	c2670	c1908	c3540	c5315	c6288	c7552
Nem	0,65	0,10	0,33	4,11	5,16	7,81	8,68	5,80	6,24	9,93
МОМ	0,003	0,003	0,003	0,006	0,009	0,017	0,028	0,088	0,208	0,167
ДПМ	0,005	0,014	0,011	0,025	0,053	0,169	0,272	0,772	0,905	1,314
TT	1,00	1,90	2,40	4,50	6,50	7,60	7,80	8,00	8,90	9,10

Выводы. Описанные методы: дедуктивно-параллельный и ОДП являются совершенной реализацией, ориентированной на обработку цифровых проектов, имплементируемых в кристаллы, размерностью в несколько миллионов вентилей. При этом рассматривается модель дефектов одиночного константного типа. В случае необходимости описанные методы можно использовать и для анализа кратных константных дефектов, а также функциональных. В этом случае необходимо ввести процедуру идентификации и генерирования списков неисправностей. Дальнейшее совершенствование методов моделирования связано с использованием аппаратурных ускорителей, подобных тем, которые были разработаны фирмой Aldec (HES – Hardware Embedded Simulation) для повышения быстродействия методов исправного моделирования.

ГЛАВА 14
АППАРАТУРНОЕ МОДЕЛИРОВАНИЕ (HES™)

Hardware Embedded Simulation – встроенное аппаратурное моделирование – технология, позволяющая значительно, в сотни раз, сократить время проектирования и тестирования FPGA и ASIC проектов, которые все более усложняются, облегчая их верификацию. Первоначально модули проекта верифицируются на верхнем уровне описания, затем выполняется их синтез, реализация и загрузка в FPGA, расположенную на плате акселератора, после чего происходит аппаратно-программное моделирование. HES-платы подключаются к персональному компьютеру через шину PCI. HES-технология поддерживает до 4 плат акселераторов на одном компьютере. Максимальный размер проекта, который может быть размещен в 4 платах, равен 90 Мб вентилей.

Весь проект моделируется в HES-среде, которая состоит из HES программного симулятора и PCI плат. Эта среда обеспечивает правильное взаимодействие модулей, помещенных в HES, и модулей, тестируемых с помощью программы.

HES-акселератор (Aldec Inc., USA, www.aldec.com) работает под операционными системами: UNIX, Linux и Windows NT и полностью совместим с Cadence, Model Technology, Aldec's Riviera™ и Active-HDL™ программами моделирования. HES-технология позволяет обрабатывать смешанные проекты, содержащие блоки, написанные на VHDL, Verilog, EDIF и Си.

В настоящее время при разработке проектов создаваемые модули добавляются к уже созданной части проекта.Каждый модуль может верифицироваться отдельно в программной среде, а затем подключаться к остальным элементам системы. После этого система тестируется целиком. И хотя проектировщика интересует работа последнего подключенного модуля и он уверен, что остальные модули работают правильно, программа моделирования обрабатывает все блоки системы. Такой процесс моделирования требует много времени и становится очень неудобным.

HES-технология позволяет также моделировать весь проект целиком. Однако та часть проекта, которая уже верифицирована на RTL-уровне, помещается в FPGA. Программный симулятор моделирует только новый модуль. Поскольку быстродействие аппаратного анализа практически не зависит от размера проекта, то скорость моделирования значительно повышается. Также HES-технология может использоваться вместо моделирования после этапа синтеза. Все аспекты функционального анализа проекта могут быть верифицированы в среде HES. Такая технология позволяет повысить скорость моделирования в 10 –1000 раз по сравнению с самым быстрым программным моделированием.

С помощью HES-технологии проект, или только его часть, может быть помещен в микросхему FPGA, расположенную на плате акселератора. Для этого необходимо только синтезировать модули, предназначенные для аппаратного моделирования. Для конфигурации HES-окружения можно использовать HES Wizard. Однако применяя HES, нельзя спроектировать систему, включающую различные типы микросхем, как это можно сделать в программной среде моделирования.

HES-технологию можно использовать для верификации внешних систем. Внешняя система или ее фрагмент в виде процессора или таймера может быть подключена через дочернюю плату (Daughter Board) к HES. Затем выполняется совместное моделирование с другой частью системы или с другими проектами, тестируемыми с помощью программного симулятора.Также можно размещать новые модули в плату HES, продолжая моделировать внешнее устройство.

14.1. Инсталляция

Для использования HES-технологии необходимо иметь:

1) персональный компьютер, поддерживающий PCI шину;
2) средства синтеза Synopsys или Synplicity;
3) программу имплементации для микросхем Xilinx серии Virtex;
4) программу моделирования;
5) HES Service Pack;
6) HES плату – Virtex 800 или 2000 с соединениями для внешних систем (db).

14. 2. Структура HES Virtex 800 и Virtex 2000E (DB)

Основной частью HES-устройства, выполняющего моделирование, является микросхема FPGA. Для конфигурации FPGA используется управляющий модуль (Control module CPLD). Он передает данные между PCI мостом и микросхемой FPGA, а также выполняет внутреннюю идентификацию функций, управление моделированием и программирование микросхемы FPGA.

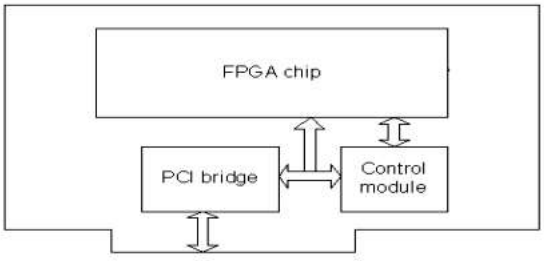
PCI мост предназначен для передачи данных между шиной PCI и платой HES.

Источником синхроимпульсов является кварцевый генератор.

Все платы HES содержат коннектор JP2, который используется для конфигурации плат в соответствии с JTAG стандартом. Микросхема FPGA и управляющий модуль соединены в JTAG цепь.

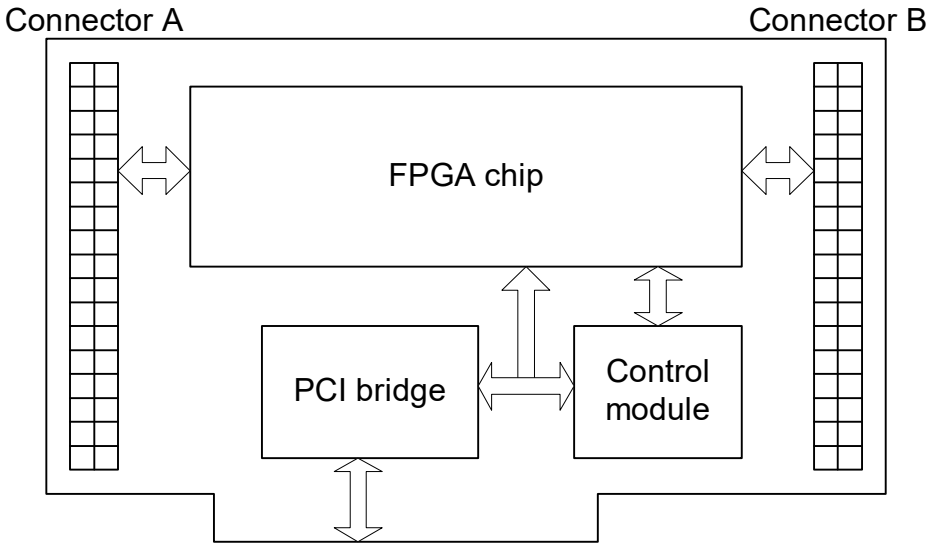
FPGA микросхема на HES-плате определяет устройства HES Virtex 800 и HES Virtex 2000E. На рисунке14.1 приведена структурная схема и описание компонентов платы HES Virtex 800. Рисунок 14.2 содержит соответствующую информацию для HES Virtex 2000E. Существует для типа устройств Virtex 2000E: Virtex2000E и Virtex2000E MB (Mother Board). Различаются они наличием двух слотов –коннекторов (Connector A и Connector B) на платеV2000E MB. Эти слоты используются для соединения с внешними системами. Например, можно моделировать реальное устройство Intel 8253 timer, соединенное через слот. Линейный регулятор поддерживает напряжение: 1.8, 2.5, 3.3 и 5B, которое поступает с PCI соединения. Все эти напряжения подаются на слоты A и B. На рисунке14.3 приведена схема нумерации контактов в слотах A и B.

Рисунок 14.1. Структурная схема платы HES Virtex 800



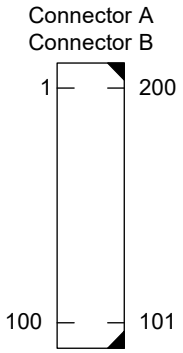
Название модуля	Тип детали	Производитель
Микросхема FPGA	XCV800E-HQ240	XILINX
Управляющий модуль	XC4013XLA −07 PQ208	XILINX
PCI мост	PCI9054	PLX
Регулятор напряжения	1.8V 2.5V 3.3V	NATIONAL SEM
Кварцевый генератор	32MHz	

Рисунок 14.2. Структурная схема платы HES Virtex 2000



Название модуля	Тип детали	Производитель
Микросхема FPGA	XCV2000E-6BG560C	XILINX
Управляющий модуль	XC4013XLA −07 PQ208	XILINX
PCI мост	PCI9054	PLX
Регулятор напряжения	1.8V 2.5V 3.3V	NATIONAL SEM
Кварцевый генератор	32MHz	

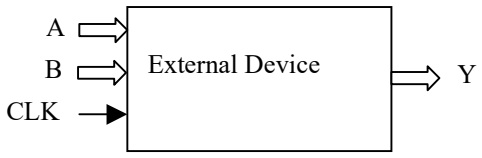
Рисунок 14.3. Нумерация контактов в слотах A и B



14.3. Подготовка проекта для внешнего соединения

Используя слоты, расположенные на Virtex2000E, можно анализировать внешние устройства. Допустим, существует внешнее устройство, которое нужно промоделировать, применяя HES-плату (рисунок 14.4). Это устройство имеет три входа и один выход. Входы – два 8-битных вектора A и B и синхровход. Выход – 8-битный вектор.

Рисунок 14.4. Моделируемое внешнее устройство



Вначале необходимо соединить входные-выходные контакты внешней системы с ножками их разъемов (слоты A и B), расположенных на HES-плате.

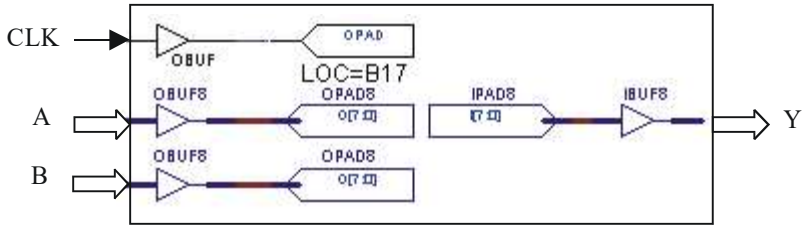
Затем необходимо создать модуль, описывающий внешнюю систему. В нем следует указать только входы и выходы внешней системы.

На языке VHDL такое описание будет иметь вид

```
entity external is
 port map (A,B: in std_logic_vector (7 downto 0);
           CLK: in std_logic;
           Y: out std_logic_vector (7 downto 0));
end entity;
```

Затем следует подготовить EDIF файл, который будет применяться для последующей реализации. Можно использовать Block Diagram Editor из Active-HDL или Xilinx Foundation. Для входов применяется OBUF и OPAD, для выходов – IBUF и IPAD. Затем следует назначить LOC для каждого порта. Пример схемы представлен на рисунке 14.5.

Рисунок 14.5. Моделируемые внешние системы



На рисунке 14.6 показано только одно назначение LOC для CLK порта. Аналогичным образом можно назначить и другие контакты схемы. Можно использовать контакты любого из 6 банков каждого слота.

Затем следует сгенерировать EDIF-файл проекта и подключить его к процессу реализации.

По окончании процесса будет создан .bit файл, необходимый для аппаратного моделирования. Затем нужно подготовить .map файл для модуля, после чего можно моделировать внешнее устройство, используя HES-технологию.

Описанный процесс автоматически выполняет HES Wizard. Следует только выбрать внешний модуль из списка Daughter Board и определить контакты для портов. HES Wizard выполнит процессы синтеза и реализации и подготовит все необходимые файлы для аппаратного моделирования.

14.4. Правила использования и описание HES Wizard

Весь процесс генерации .bit и .map файлов автоматически выполняет TCL скрипт – HES Wizard. При его применении необходимо просто выделить модули для аппаратного моделирования, а скрипт выполнит процесс синтеза и реализации, а также создаст готовые к использованию файлы. HES Wizard удобен и прост в работе. Тем не менее он имеет некоторые ограничения:

1) В VHDL нельзя употреблять расширенные идентификаторы.

2) Wizard позволяет синтезировать модули, написанные на разных языках. Можно написать модуль на Verilog и использовать его для моделирования на VHDL, тем не менее следует избегать употребления ключевых для языков слов (*reg, buf*).

3) Смешанное моделирование не допускается. Следовательно, если HES-модуль для моделирования подготовлен в Verilog, TestBench для проекта также должен быть написан на Verilog-языке.

Правила описания HDL-модулей для HES-симулятора:

1) модули должны быть синтезируемыми для Xilinx FPGA;

2) размер всех моделируемых модулей не должен превышать возможности FPGA;

3) максимальное число входов анализируемых модулей – 1024 для V800 и 8192 для V2000E;

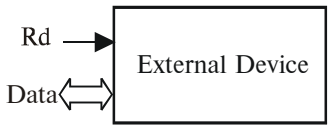
4) максимальное число выходов также – 1024 для V800 и 8192 для V2000E.

Программный симулятор посылает и получает данные от HES-платы. Во время моделирования он посылает тест-векторы на моделируемый модуль, реализованный в FPGA, которое возвращает результаты моделирования. Весь процесс основан на обмене данными между платой HES и программным симулятором.

14.5. Аппаратное моделирование двунаправленных портов

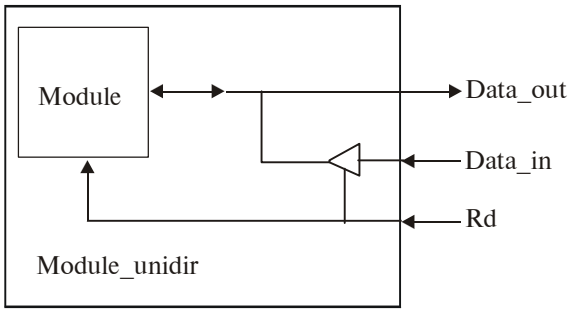
Если в проекте используются двунаправленные порты, их следует разбить на два: входной и выходной, до размещения проекта в аппаратуре. На рисунке 14.6 представлено устройство, содержащее двунаправленный порт *Data,* управляемый входом *Rd*.

Рисунок 14.6. Проект, содержащий двунаправленный порт



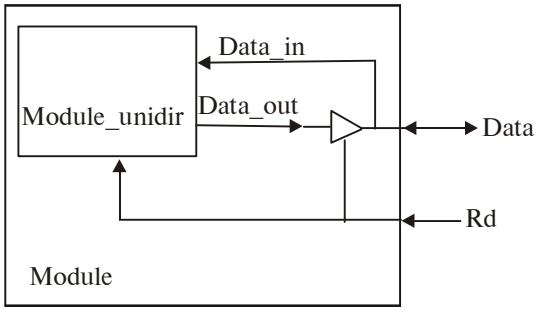
До синтеза и реализации двунаправленный порт делится на вход Data_in и выход Data_out (рисунок 14.7).

Рисунок 14.7. Проект с разделенным двунаправленным портом



Новый модуль Module_unitdir синтезируется и реализуется в аппаратуре. До окончания процесса совместного моделирования вход Data_in и выход Data_out следует преобразовать обратно в двунаправленный порт. Рисунок 14.8 иллюстрирует эту операцию.

Рисунок 14.8. Слияние двунаправленных портов



Реализуемый Module_unidir внутри Module и порты Data_in и Data_out объединяются в двунаправленный порт Data модуля Module.

Весь процесс разделения/слияния двунаправленных портов автоматически выполняется в HES Wizard.

Определенные пользователем типы

Аппаратное моделирование позволяет использовать следующие типы данных:

1) Предопределенные типы: bit, bit_vector, std_logic, std_logic_vector, std_ulogic, std_ulogic_vector, signed, unsigned.

2) Определенные пользователем типы:

– перечисляемые типы, enumeration или enum, например,

```
type week is (Monday, Tuesday);
```

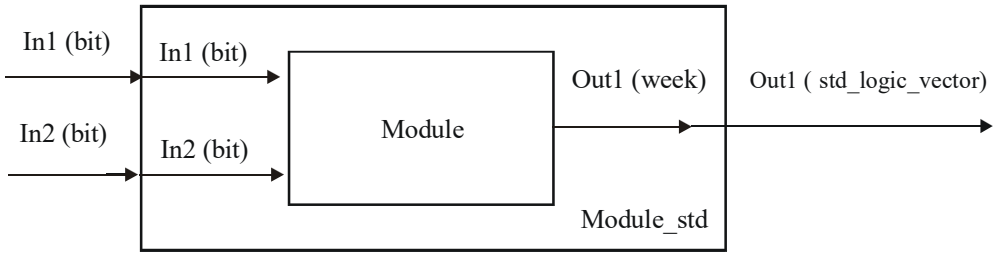
– типы, основанные на integer, например,

```
subtype  myint is integer range 21 downto 8; --or
type digits is range 0 to 9;
```

– массивы предопределенных типов, перечисления и integer;
– записи предопределенных типов, перечисляения и integer.

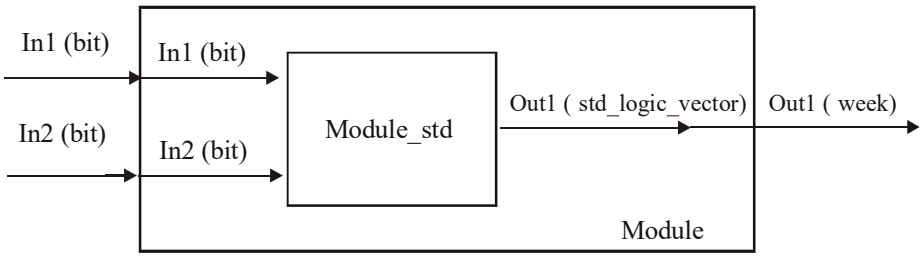
Определенные пользователем типы преобразуются до синтеза в std_logic. На рисунке 14.10 представлен пример такого преобразования. Тип выхода Out1 – Week (Monday,Tuesday). Входы имеют тип bit. Out1 перед синтезом преобразуется в тип std_logic_vector.

Рисунок 14.9. Преобразование определенных пользователем типов



После выполнения процессов синтеза и реализации моделируемые типы преобразуются обратно, поэтому аппаратное и программное моделирование будет давать одинаковые результаты. Процедура преобразования типов, позволяющая моделировать описанные пользователем типы с помощью HES-технологии, выполняется автоматически в HES Wizard.

Рисунок 14.10. Обратное преобразование определенных пользователем типов



Множественная реализация одного и того же модуля в HES-плате

Если в проекте выполняется множественная реализация одного и того же модуля, следует указать, сколько реализаций будет моделироваться в аппаратуре. Сразу после запуска HES Wizard, в окне HES Wizard - Modules Settings необходимо щелкнуть по полю Count и указать, сколько модулей необходимо создать.

Generic-константы для реализуемых модулей

Если в модулях, которые будут анализироваться с помощью аппаратуры, используются Generics,следует помнить, что при аппаратном моделировании будут употребляться начальные значения Generics. Проиллюстрируем это на примере. Допустим, что существует компонент My_module с продекларированным параметром Generic (param:

natural:=5;). При инициализации компонента для param задано значение 3 generic map (param=>3). Теперь значение параметра равно 3. Однако для аппаратного моделирования оно будет оставаться по-прежнему равным 5.

HES-моделирование и моделирование после синтеза

Моделировние проекта с использованием HES-технологии эквивалентно моделированию после синтеза. HES-моделирование можно применять также вместо програмного моделирования временных параметров. Но следует осознавать разницу между этими двумя способами моделирования. Програмное основано на поведенческой модели цифрового проекта, в то время как HES-моделирование основано на реальном устройстве, работающем в компьютере. Выполняя анализ проекта после синтеза, программа моделирования оперирует с сигналами типа STD_LOGIC, имеющими 9 определенных значений.

Однако HES-блок моделирования всегда оперирует только двумя значениями: сильная единица (1) и сильный нуль (0). Поэтому результаты программного и аппаратного моделирования могут быть различными. Это означает некорректность написанного TestBench или наличие ошибок в проекте. В таблице 14.1 представлено преобразование значений типа std_logic в значения, используемые при аппаратном моделировании.

Таблица 14.1. Преобразование значений std_logic для аппаратного моделирования

VHDL	
Программное моделирование	Аппаратное моделирование
'U'	'1' or '0'
'X'	'1' or '0'
'0'	'0'
'1'	'1'
'Z'	'1' or '0'
'W'	'1' or '0'
'L'	'0'
'H'	'1'
'-'	'1' or '0'

14.6. Подготовка TestBench

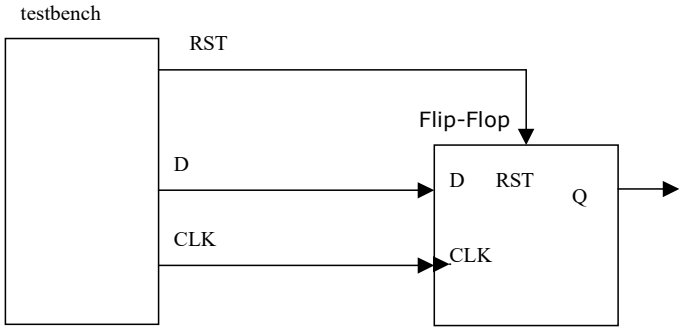
Если TestBench был подготовлен некорректно, это означает, что некоторые временные параметры были упущены, поэтому могут появляться временные нарушения при программном моделировании устройства и некоторые сигналы могут иметь значения Forcing Unknown (X), Uninitialized (U), Weak Unknown 'W'. Поскольку HES-моделирование оперирует только двумя значениями: '1' и '0', нельзя получить информацию о неизвестных и неинициализированных значениях.

Чтобы избежать проблем, связанных с таким ограничением, необходимо следовать двум правилам:

1) Создавая TestBench, следует быть уверенным, что все тестовые последовательности, подаваемые на моделируемое устройство, удовлетворяют временным константам.
2) TestBench должен обеспечивать, чтобы в проекте все сигналы установки были заданы в начале процесса моделирования.

Следующий пример иллюстрирует проблемы, связанные с применением тестовых последовательностей к блоку, моделируемому в HES-технологии.

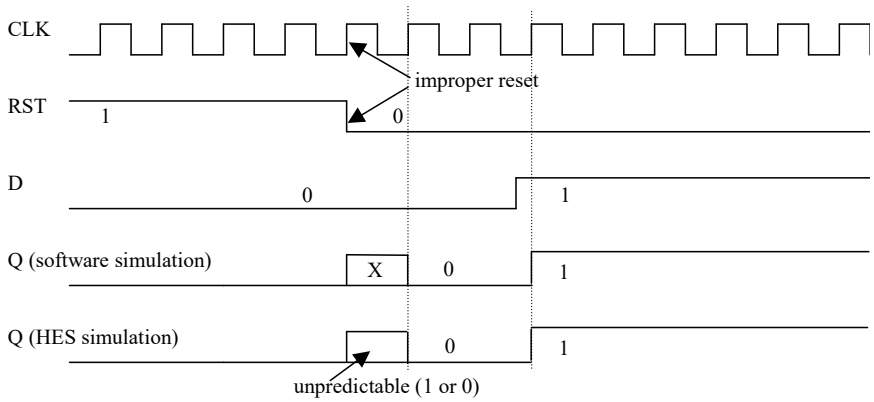
Рисунок 14.11. Триггер, моделируемый с помощью TestBench



Пусть триггер работает по переднему фронту синхроимпульса и имеет асинхронный сброс, активный при RST=1. Сигналы синхронизации и сброса генерируются в TestBench. Переключение сигнала сброса из 1 в 0 происходит одновременно с генерацией переднего фронта синхросигнала. Условие для временных параметров RST входа не выполняется.

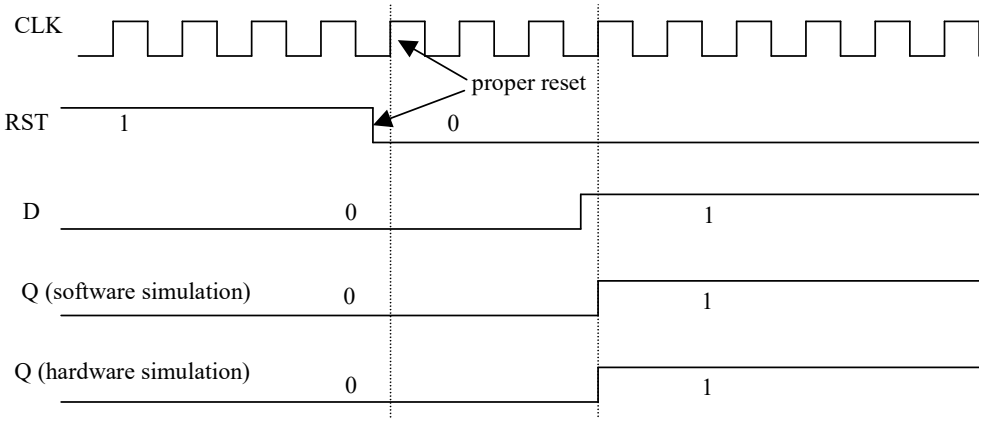
Если триггер моделируется с помощью программы, то в момент, когда RST переключается в 0, на выходе будет значение 'X'. Если же триггер моделируется в HES, то значение сигнала, '0' или '1', на Q будет непредсказуемо (см. рисунок 14.12).

Рисунок 14.12. TestBench не обеспечивает правильные условия сброса



В этом примере TestBench не обеспечивает правильные условия сброса, поэтому результаты программного и аппаратного моделирования различаются. Чтобы избежать этого, необходимо модифицировать TestBench (рисунок 14.13).

Рисунок 14.13. TestBench обеспечивает правильные условия сброса

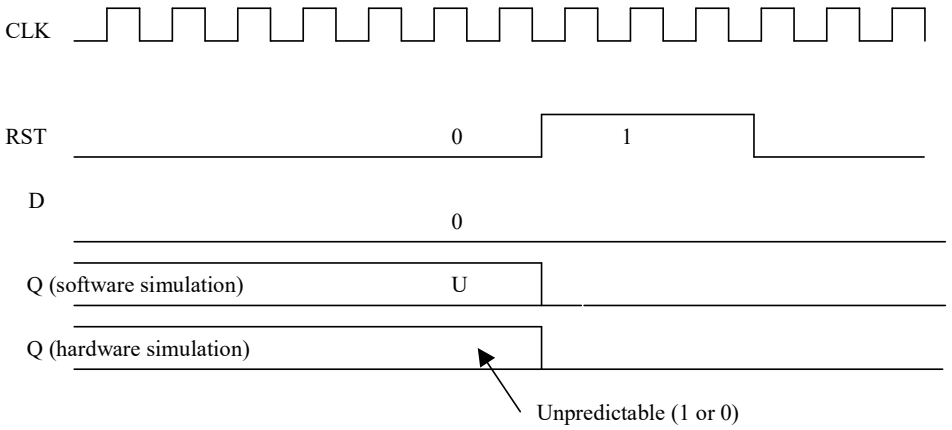


Переключение сигнала сброса из 1 в 0 и передний фронт синхросигнала не должны выполняться в одно и то же время. Все временные параметры должны быть валидными. Тогда TestBench будет обеспечивать правильные условия сброса. Описанное

условие касается не только сигналов сброса или установки, но и всех остальных сигналов триггеров, моделируемых в HES-симуляторе.

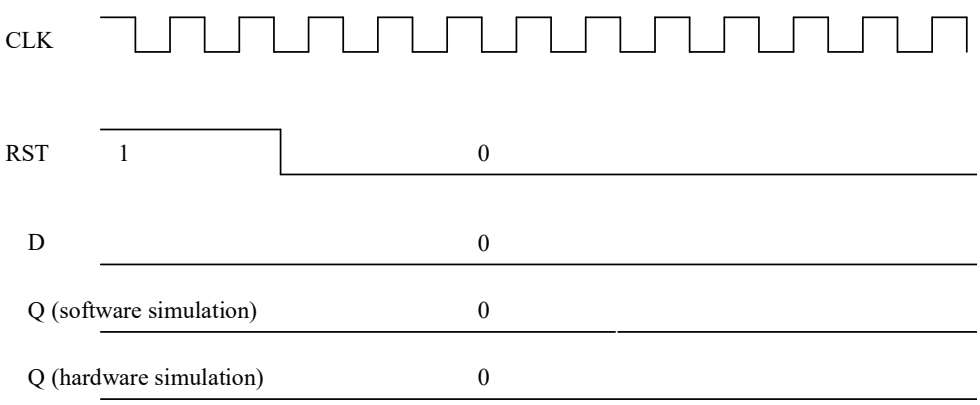
Второй пример иллюстрирует проблемы, связанные с установкой проекта в исходные значения перед началом процесса моделирования. В качестве примера используется предыдущий проект. Сброс триггера не был выполнен до начала моделирования (рис. 14.14). Если бы триггер моделировался программным симулятором, выход Q имел бы значение 'U' до тех пор, пока не появится сигнал сброса. Однако при моделировании HES-симулятором на Q будут непредсказуемые '0' или '1', потому что моделируемый HES-симулятором блок работает как реальное устройство, требующее инициализации. После появления сигнала сброса результаты обоих способов моделирования будут идентичны. Чтобы избежать описаных проблем, все сигналы сброса следует задавать перед началом моделирования.

Рисунок 14.14. Сигнал сборoса не установлен перед началом моделирования



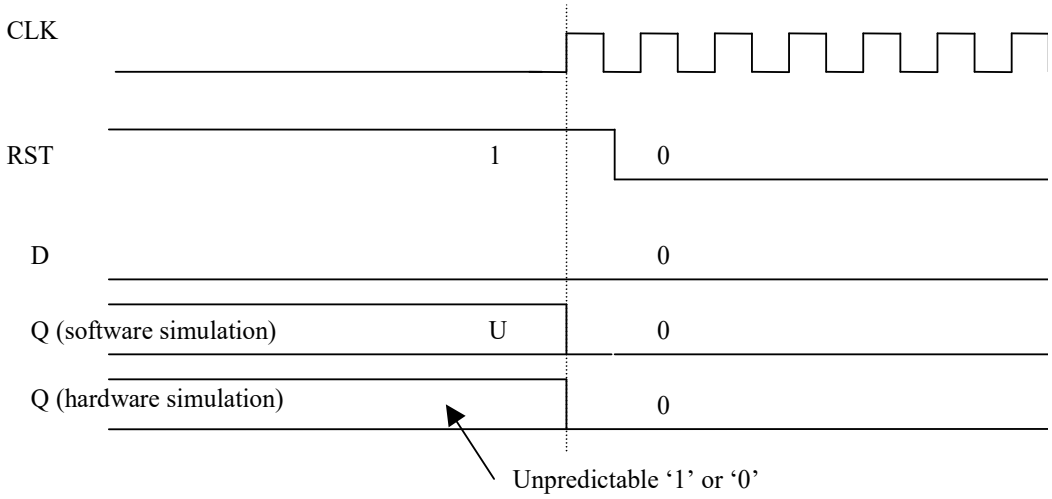
На рисунке14.15 представлены временные диаграммы, соответствующие правильным сигналам установки. Такой тест будет затем полезен для моделирования временных констант проекта.

Рисунок 14.15. Сигнал сброса активен в начале моделирования



Предыдущий пример описывает ситуацию, когда проект инициализируется асинхронным сбросом. Следующий представляет тот же самый проект, но с синхронным сбросом (рисунок 14.16). Триггер будет инициализирован при установке сигнала RST и поступлении переднего фронта синхроимпульса.

Рисунок 14.16. Синхронный сброс



В этом случае до поступления переднего фронта синхроимпульса существует различие между программной обработкой и HES-моделированием. Анализируемый в HES модуль – это реальное устройство, которое инициализируется при установке сигнала сброса, после поступления первого фронта синхроимпульса. В данном случае нет возможности избежать различия между программным и аппаратным моделированием устройства.

Если входные последовательности в TestBench имеют правильные временные константы, проект корректно подготовлен и инициализован, результаты моделирования в HES будут аналогичны результатам анализа проекта после синтеза программным способом. Данные упомянутых способов моделирования цифрового проекта точно сопоставимы по временным характеристикам, но не по дельта-циклам. Другими словами, результаты моделирования этих двух способов будут одинаковы в один момент времени, но номера дельта-циклов в этот момент могут быть разными; кроме того, события конкретных сигналов могут случаться в разные дельта-циклы данного момента времени.

Если записать результаты моделирования в текстовый файл и процедуру записи вызывать в последнем дельта-цикле данного момента моделирования, тогда текстовый файл, полученный после программного моделирования, и текстовый файл, полученный после моделирования HES, будут идентичными.

Допуск небольшой расфазировки синхросигнала при распространении по схеме

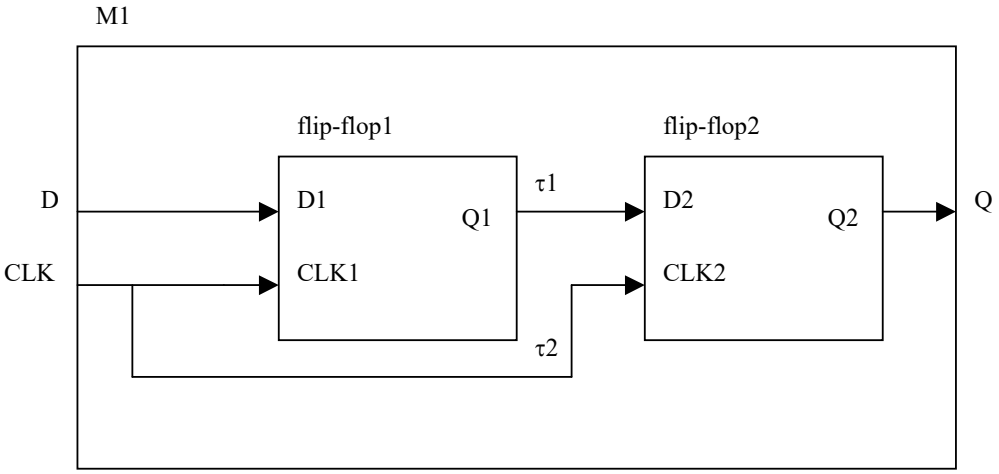
Во время разработки любого цифрового проекта для минимизации расфазировки синхросигнала используются выделенные, с небольшим разбросом, линии синхронизации, которые обеспечивают правильное распределение синхросигнала. Так как HES моделируемая модель работает как реальное устройство, проблема разброса синхросигнала существенна для достижения правильных результатов моделирования. Чтобы избежать разброса синхросигнала, следует придерживаться двух правил:

1. Глобальный буфер BUFG должен быть применен к каждому синхросигналу аппаратурно моделируемого блока.
2. Необходимо избегать внутренней генерации синхросигнала.

Следующий пример иллюстрирует, какую важную роль играют глобальные буферы BUFG в цифровых проектах, точно так же, как и в аппаратурном моделировании.

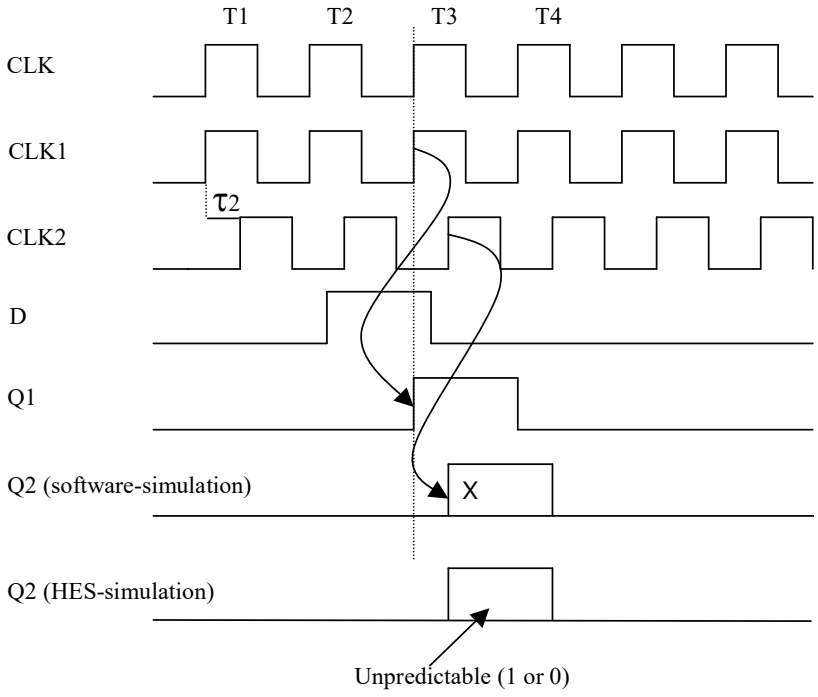
Простой сдвиговый регистр M1 содержит два триггера. Оба имеют синхронизацию по переднему фронту. Необходимо промоделировать модуль M1, при этом буфер BUFG не используется (рис 14.17).

Рисунок 14.17. Моделируемый в HES модуль без глобальных BUFG буферов



Допустим, что проект был реализован и t1 – это время распространения сигнала с выхода Q1 до входа D2, t2 – со входа CLK до входа CLK2. Более того, для наглядности допустим, что задержка с CLK до CLK1 точно такая же, как t1, и равна 0. Тогда при t2>0 результаты моделирования будут неверными (рисунок 14.18).

Рисунок 14.18. Ошибка моделирования из-за расфазировки синхросигнала



Активный фронт синхросигнала достигнет обоих триггеров в разные моменты времени. По этой причине время установки второго триггера во время цикла T3 будет неправильным. В этом случае программное моделирование отметит временное нарушение (timing violation),а аппаратное не даст никаких предупреждений. Второй триггер будет находиться в непредсказуемом состоянии (1 или 0). Использование глобального буфера BUFG гарантирует, что активный фронт синхросигнала достигнет обоих триггеров в один и тот же момент. Таким образом, оба триггера будут иметь правильную синхронизацию. Это правило не является особеностью использования

HES, а представляет собой хороший стиль при проектировании любых цифровых устройств.

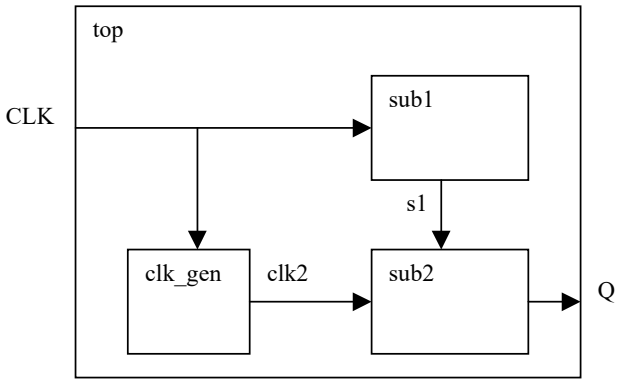
Следующий пример иллюстрирует проблемы проектов, имеющих более одного синхросигнала или генерирующих синхросигнал внутри себя. Приведенный ниже проект состоит из трех модулей (рисунок 14.19):

clock_gen – генерирует синхросигнал для модуля sub2,

sub1 – генерирует сигнал s1 по переднему фронту CLK,

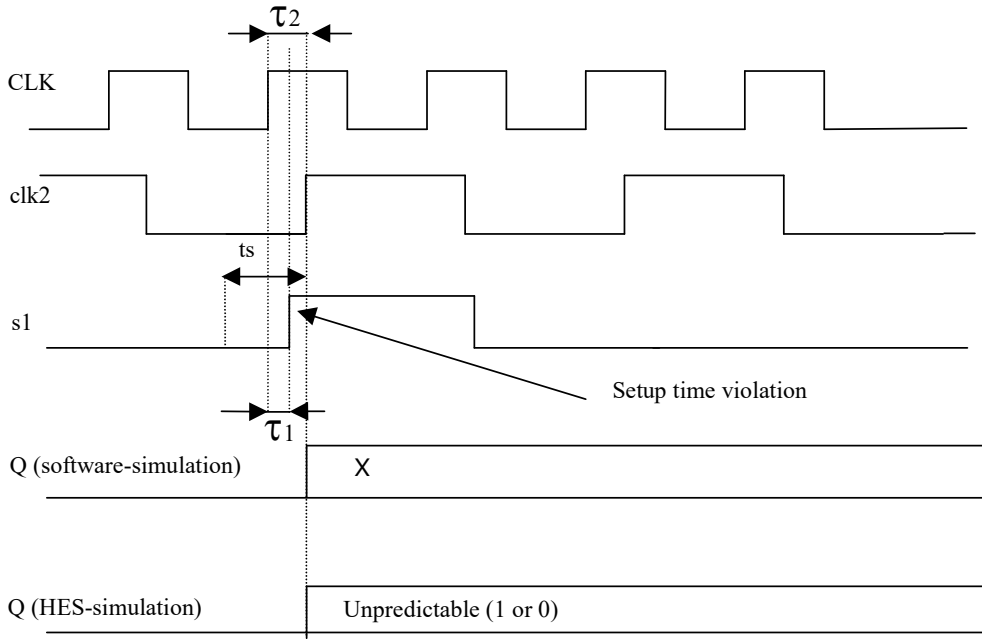
sub2 – считывает сигнал s1 по заднему фронту clk2.

Рисунок 14.19. Проект с внутренней генерацией синхросигнала



Модуль clk_gen, генерирующий синхросигнал clk2, представляет собой простой двоичный счетчик. Пусть τ_1 – задержка модуля sub1, ts – время установки модуля sub 2, τ_2 – задержка модуля clk_gen. Допустим, что задержка распространения сигнала между модулями равна 0. Если $\tau_2 > \tau_1$, происходит нарушение времени установки ts модуля sub2 и на выходе модуля получается неизвестное значение (рисунок 14.20). Только программное моделирование будет обнаруживать неисправность (timing violation), HES-моделирование не выдаст никаких предупреждений.

Рисунок 14.20. Нарушение времени установки



Используя несколько синхросигналов или внутреннюю генерацию синхросигнала, необходимо быть уверенным, что время установки и другие временные константы удовлетворяют требованиям временного соответствия.

ГЛАВА 15
МИКРОКОНТРОЛЛЕР M68HC05

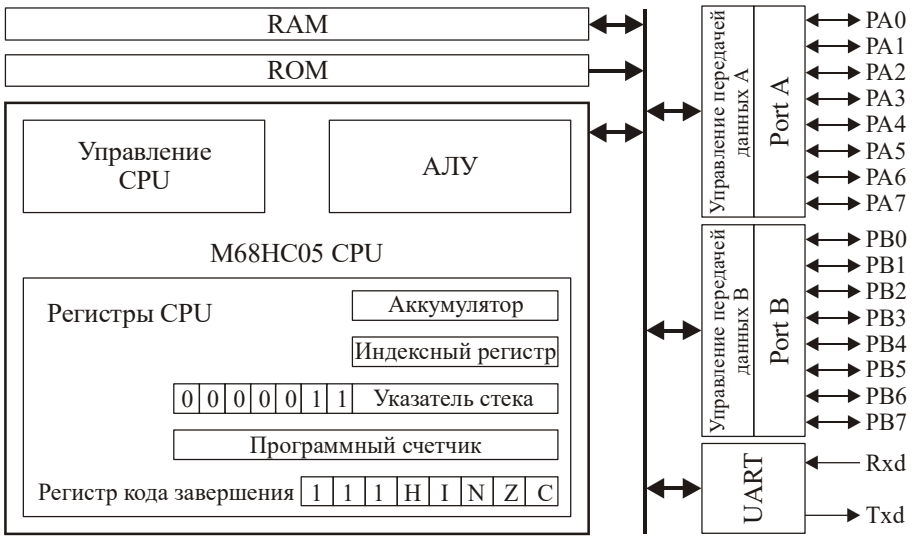
Создается поведенческая модель CPU микропроцессора. Разрабатываются структурные схемы процессора и его VHDL-модель для синтеза. После описания на языке VHDL параллельного порта созданные устройства и UART (спроектированный ранее) используются для создания микроконтроллера M68HC05.

15.1. Структура микроконтроллера

Микропроцессор обычно состоит из CPU, RAM и ROM памяти, различных последовательных и параллельных входных\выходных интерфейсов, размещенных в одной микросхеме. В качестве завершающего примера проектируется микроконтроллер, подобный устройству Motorola MC68HC05, широко используемый в простых управляющих приборах, таких как термостат, контроллер устройства. К таким изделиям, как правило, предъявляются высокие требования по отношению к входным/выходным параметрам и сравнительно небольшие – к быстродействию. При этом низкая стоимость устройства предпочтительнее высокой скорости. Motorola выпускает целое семейство микроконтроллеров 6805, которые различаются в основном количеством и типом памяти, вход/выходными ресурсами.

На рисунке 15.1 представлена упрощенная версия промышленно-используемого микроконтроллера MC68HC05. Его структурная схема включает CPU, RAM, ROM, два 8-битных параллельных вход/выходных порта (Port A и Port B) и UART, реализующий последовательный коммуникационный интерфейс. Реальный микроконтроллер 6805 имеет еще дополнительные последовательный и параллельный порты, встроенный датчик времени. Микросхема устройства также содержит внутренние шины адреса и данных, соединяющие CPU, внутренние RAM и ROM, вход/выходной интерфейс. Далее проектируется CPU, подобный 6805 CPU, который затем интегрируется в систему, представленную на рисунке 15.1.

Рисунок 15.1. Упрощенная структурная схема микроконтроллера M68HC05



Операции процессора 6805 CPU описываются с точки зрения программирования. Шина данных имеет ширину 8 битов, адресная – 13 битов, следовательно, доступное адресное пространство определяется как $2^{13} = 8192$ байт.

На рисунке 15.2 изображена структура регистров MC68HC05C4 для модели программирования. Аккумулятор A, индексный регистр X и регистр кода завершения CCR (the condition code register) имеют длину 8 битов. Левые три бита регистра

CCR постоянно содержат значения 111, остальные биты используются по назначению следующим образом:

C – флаг переноса. Сохраняет перенос или выполняет заем, сформированный в результате арифметической операции.

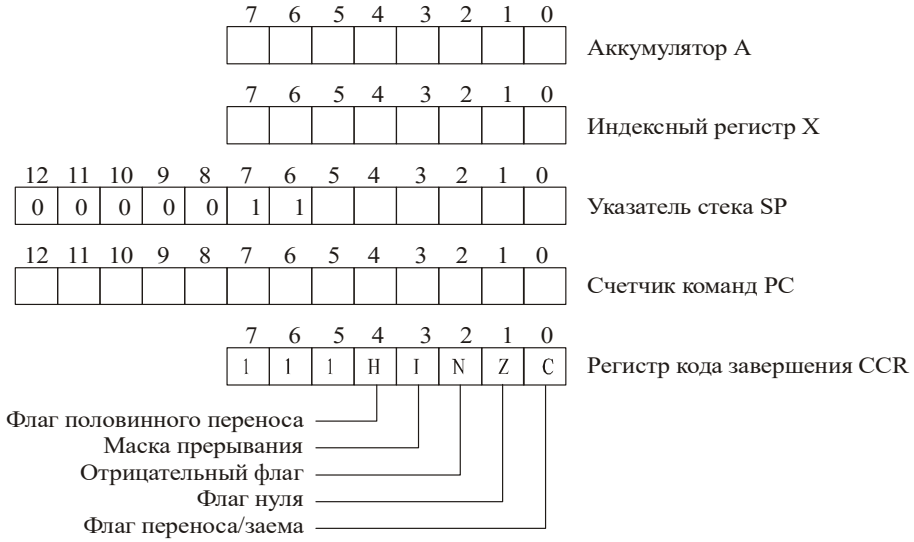
H – флаг половинного переноса. Используется для BCD арифметики (binary-coded decimal arithmetic – двоично-десятичная арифметика).

N – отрицательный флаг. Устанавливается, если результат операции отрицательный.

Z – флаг нуля. Устанавливается в 1, если результат операции равен 0.

I – маска прерывания. Если ее значение равно 1, то запрещается прерывание работы процессора.

Рисунок 15.2. MC68HC05C4 программируемая модель



Счетчик команд PC, длиной 13 битов, адресует в памяти выполняемые команды. В 6805 микропроцессоре часть памяти RAM резервируется под стек, используемый для сохранения адреса возврата стандартных подпрограмм. Содержимое регистров PC, A, X и CCR заносится в стек при выполнении обработки прерывания, однако при программировании нет возможности напрямую обращаться к нему. Его адресация в устройстве MC6805C4 всегда начинается с адреса 0000011111111 и продолжается в порядке убывания. Указатель на стек SP имеет длину 6 битов, следовательно, максимальный размер стека равен 64 байтам. Для адресации памяти через SP используется префикс 0000011 в целях формирования 11-битного адреса. SP всегда указывает на первое пустое место в стеке и уменьшается на единицу после размещения байта в стеке. Поэтому он должен увеличиваться до того, как байт извлекается из стека.

Каждая команда в MC6805 имеет от 1 до 3 байтов. Первый байт – это всегда код, описывающий операцию и способ адресации. Следующие один или два байта обычно содержат информацию об адресе. В таблице 15.1 приведена символика команд и соответствующий им шестнадцатеричный код. Четыре самых старших бита определяют режим адресации. Шестнадцатеричный код, соответствующий этим битам, и названия режимов перечислены в заголовке таблицы. Следующие четыре бита – варианты шестнадцатеричного кода, перечисленные с левой стороны таблицы – определяют операцию. Таким образом, код B4h (h обозначает шестнадцатеричную систему) – это операция AND с прямой адресацией (dir), а FCh соответствует команде перехода JMP с индексной адресацией (ix).

Таблица 15.1. Коды команд микропроцессора MC6805

	Манипуляция битом		Условный перехода	Чтение-изменение-запись					Управление		Регистр-память					
	BTB	BSC	REL	DIRM	INHA	INHX	IXIM	IXM	INH1	INH2	IMM	DIR	EXT	IX2	IX1	IX
Lo\Hi	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	BRSET0*	BSET0*	BRA	NEG	NEG	NEG	NEG	NEG	RTI		SUB	SUB	SUB	SUB	SUB	SUB
1	BRCLR0*	BCLR0*	BRN						RTS		CMP	CMP	CMP	CMP	CMP	CMP
2	BRSET1*	BSET1*	BHI		MUL*						SBC	SBC	SBC	SBC	SBC	SBC
3	BRCLR1*	BCLR1*	BLS	COM	COM	COM	COM	COM	SWI		CPX	CPX	CPX	CPX	CPX	CPX
4	BRSET2*	BSET2*	BCC	LSR	LSR	LSR	LSR	LSR			AND	AND	AND	AND	AND	AND
5	BRCLR2*	BCLR2*	BCS								BIT	BIT	BIT	BIT	BIT	BIT
6	BRSET3*	BSET3*	BNE	ROR	ROR	ROR	ROR	ROR			LDA	LDA	LDA	LDA	LDA	LDA
7	BRCLR3*	BCLR3*	BEQ	ASR	ASR	ASR	ASR	ASR		TAX		STA	STA	STA	STA	STA
8	BRSET4*	BSET4*	BHCC*	LSL	LSL	LSL	LSL	LSL		CLC	EOR	EOR	EOR	EOR	EOR	EOR
9	BRCLR4*	BCLR4*	BHCS*	ROL	ROL	ROL	ROL	ROL		SEC	ADC	ADC	ADC	ADC	ADC	ADC
A	BRSET5*	BSET5*	BPL	DEC	DEC	DEC	DEC	DEC		CLI	ORA	ORA	ORA	ORA	ORA	ORA
B	BRCLR5*	BCLR5*	BMI							SEI	ADD	ADD	ADD	ADD	ADD	ADD
C	BRSET6*	BSET6*	BMC	INC	INC	INC	INC	INC		RSP		JMP	JMP	JMP	JMP	JMP
D	BRCLR6*	BCLR6*	BMS	TST	TST	TST	TST	TST		NOP	BSR*	JSR	JSR	JSR	JSR	JSR
E	BRSET7*	BSET7*	BIL*						STOP*		LDX	LDX	LDX	LDX	LDX	LDX
F	BRCLR7*	BCLR7*	BIH*	CLR	CLR	CLR	CLR	CLR	WAIT*	TXA		STX	STX	STX	STX	STX

* не реализуемые в создаваемой модели команды

В таблице 15.2 определены команды, соответствующие четырем группам. Буквой M обозначены данные, считываемые из памяти или записываемые в нее; буквой R – регистры A, X или M; NZ в последней колонке – обновляемые флаги N и Z. Значение сигнала C формируется каждый раз после его обновления. Исполнительный адрес EA определяет адресный режим. Когда программный счетчик заносится в стек, он делится на старший PCH и младший PCL байты.

Команды из таблицы 15.2, г содержат условия перехода. Если условие имеет ложное значение, то выполняется следующая в списке инструкция, иначе – выбирается инструкция из адресной ветви.

Таблица 15.2. Команды микропроцессора MC 6805

Команды регистр-память			
Символ	Команда	Операция	Флаг
ADD	Сложение	A ← A+M	NZ, C ← перенос
ADC	Сложение с переносом	A ← A+M+C	NZ, C ← перенос
SUB	Вычитание	A ← A-M	NZ, C ← заем
SBC	Вычитание с заемом	A ← A-M-C	NZ, C ← заем
CMP	Сравнение с A	A-M	NZ, C ← заем
CPX	Сравнение с X	X-M	NZ, C ← заем
AND	Операция И	A ← A and M	NZ
BIT	Проверка бита	A and M	NZ
ORA	Операция ИЛИ	A ← A or M	NZ
EOR	Исключающее ИЛИ	A ← A xor M	NZ
LDA	Загрузка A	A ← M	NZ
LDX	Загрузка X	X ← M	NZ
STA	Сохранение A	M ← A	NZ
STX	Сохранение X	M ← X	NZ
JMP	Переход	Переход по EA	
JSR	Переход к подпрограмме	Занесение PC в стек, переход по EA	

а

Окончание таблицы 15.2

Команды чтение-изменение-запись			
Символ	Команда	Операция	Флаг
NEG	Отрицание	R ← 0-R	NZ, C ← заем
COM	Дополнение	R ← not R	NZ, C ← 1
TST	Тест	R - 0	NZ
CLR	Сброс	R ← 0	NZ
INC	Инкремент	R ← R+1	NZ
DEC	Декремент	R ← R-1	NZ
LSR	Логический сдвиг влево	R ← R(6 downto 0)&'0'	NZ, C ← R(7)
ROL	Циклический сдвиг влево	R ← R(6 downto 0)&C	NZ, C ← R(7)
ASR	Арифметический сдвиг вправо	R ← R7&R(7 downto 1)	NZ, C ← R(0)
LSR	Логический сдвиг вправо	R ← '0'&R(7 downto 1)	NZ, C ← R(0)
ROR	Циклический сдвиг вправо	R ← C&R(7 downto 1)	NZ, C ← R(0)

б

Управляющие команды		
Символ	Команда	Операция
TAX	Передача A в X	X ← A
TXA	Передача X в A	A ← X
CLC	Сброс переноса	C ← '0'
SEC	Установка переноса	C ← '1'
CLI	Сброс I	I ← '0'
SEI	Установка I	I ← '1'
RSP	Сброс SP	SP ← "111111"
NOP	Нет операции	
RTI	Возврат из прерывания	Извлечение из стека CCR, A, X, PCH, PCL. Возвращение по адресу из PC
RTS	Возврат из подпрограммы	Извлечение из стека PCH, PCL. Возвращение по адресу из PC
SWI	Программное прерывание	Занесение в стек PCL, PCH, X, A, CCR. Переход по адресу из таблицы векторов прерываний

в

Команды условного перехода		
Символ	Команда	Условие перехода
BRA	Постоянный переход	Всегда
BRN	Переход запрещен	Никогда
BHI	Переход, если больше	(C or Z) = '0'
BLS	Переход, если меньше или равно	(C or Z) = '1'
BCC	Переход, если перенос сброшен	C = '0'
BCS	Переход, если перенос установлен	C = '1'
BNE	Переход, если не равно	Z = '0'
BEQ	Переход, если равно	Z = '1'
BPL	Переход, если плюс	N = '0'
BMI	Переход, если минус	N = '1'
BMC	Переход, если внутр. маска сброшена	I = '0'
BMS	Переход, если внутр. маска установлена	I = '1'

г

Адрес данных (или следующей инструкции для перехода) называется исполнительным адресом EA (effective address). Микроконтроллер 6805 имеет следующие адресные режимы для команд регистр-память (register-memory):

– непосредственный imm (immediate): данные записываются во второй байт команды;

– прямой dir (direct): EA – второй байт команды;

– расширенный ext (extended): EA – второй и третий байты команды (старший байт – первый);

– индексный, без смещения ix (indexed, no offset): EA соответствует X;

– индексный, 1-байтное смещение ixl: смещение описывается вторым байтом команды; EA = X + смещение;

– индексный, 2-байтное смещение ix2, записывается во второй и третий байты команды; EA=X+ offset.

Команды чтение-изменение-запись (read-modify-write) (таблица 15.2, б) используют адресацию: прямую dirm, индексную без смещения ixm, индексную с одним байтом смещения ixim и внутреннюю, inha или inhx (inherent). Для режимов inha и inhx данные содержатся в регистрах A и X соответственно. Режимы dirm, ixm и ixim работают так же, как и dir, ix и ixl, тем не менее они имеют различные имена, поскольку принадлежат различным группам команд. Управляющие команды используют внутреннюю адресацию inhl или inh2, поэтому один байт команды содержит с себе адрес операндов. Отдельная группа команд использует относительную адресацию. Команда содержит оригинальный PC и относительный (второй байт) адрес, который является расширенным по знаку и формирует оригинальный адрес PAC + 2 для получения ветви адресации. В таблице 15.3 объединены адресные режимы MC6805. Длина команды описывается в столбце "Байты".

Таблица 15.3. Режимы адресации

Режим	Имя	Байты	Примеры	Исполнительный адрес
imm	Непосредственный	2	ADD ii	Данные записаны во второй байт команды
dir, dirm	Прямой	2	ADD dd INC dd	EA = dd
ext	Расширенный	3	ADD hh ll	EA = hh ll
ix, ixm	Индексированный, без смещения	1	ADD X INC X	EA = X
ix1, ix1m	Индексированный, один байт смещения	2	ADD ff X INC ff X	EA = ff + X
ix2	Индексированный, два байта смещения	3	ADD ee ff X	EA = (ee ff) + X
rel	Относительный	2	BRA rr	EA = PC + 2 + rr*
inha, inhx	Внутренний	1	INCA INCX	Данные в A Данные в X
inh1, inh2	Внутренний	1	RTI TAX	Код заключает в себе операцию

*rr перед выполнением сложения расширяется по знаку

При появлении прерывания выполняется приостановка программы или ее сброс. Микропроцессор 6805 для поиска адреса подпрограммы прерывания использует таблицу векторов прерывания, которая хранится в верхних адресах памяти. Начальные адреса имеют следующее местоположение в памяти (при этом старший байт записывается первым):

```
сброс                                IFFEh, IFFFh
программное прерывание SWI           IFFCh, IFFDh
внешнее аппаратное прерывание IRQ    IFFAh, IFFBh
последовательное коммуникационное
прерывание SCIJnt                    lFF6h, lFF7h
```

При появлении прерывания IRQ содержимое регистров PCL, PCH, X, A и OCR заносится в стек. Затем в PC загружается адрес IFFAh и IFFBh, по которому процессор переходит к выполнению необходимой подпрограммы прерывания.

15.2. Проектирование CPU микроконтроллера

Ниже проектируется CPU микроконтроллера, подобного 6805 CPU. Некоторые из инструкций (см.таблицу 15.1) будут опущены для упрощения примера. Для некоторых команд временные характеристики будут отличаться от аналогичных параметров оригинального устройства Motorola 6805. С помощью описания CPU, приведенного в подразд. 15.1, пошагово определяются операции, необходимые для реализации различных типов команд и режимов адресации. Затем создается поведенческая VHDL-модель устройства и проводится верификация корректности ее временных параметров. Строится структурная схема CPU и записывается VHDL-модель в терминах уровня регистровых передач. После тестирования последней выполняется синтез схемы устройства. Кроме регистров, используемых в программируемой модели (см. рисунок 15.2), для реализации CPU необходимо еще три дополнительных регистра. Восьмибитный регистр Opcode предназначен для хранения кода команды в момент ее выполнения, 13-битный адресный регистр MAR (memory address register) необходим для хранения исполнительного адреса записываемых или считываемых данных. Также требуется 8-битный регистр Md (memory data register) для хранения данных, считываемых из памяти. Следует определить действия, выполняемые по каждому синхросигналу. В упрощенном процессоре, подобном 6805, внутренняя синхронизация такая же, как у памяти. Следовательно, за один синхротакт можно записать/считать данные из памяти и выполнить внутреннюю операцию в CPU. Каждая команда требует от двух до десяти тактов для выполнения, в зависимости от ее сложности. Первый такт используется для считывания команды. В начале цикла программный счетчик указывает на первый байт команды в памяти, содержащей ее код. Значение из PC передается по адресной шине, а из памяти – по шине данных поступает код команды. После этого последний загружается в регистр Opcode, а значение PC увеличивается на единицу:

```
Opcode ← mem(PC); PC ← PC+1; --mem – массив байтов или память.
```

Таблица 15.4 представляет расписанные по тактам операции для основных команд. MARH – это старший байт регистра MAR, а MARL – младший. В скобках вверху каждой ячейки записывается имя состояния управляющего автомата. ADD – пример типичной команды регистр-память. Команда SUB отличается от ADD только действием, выполняемым на последнем такте. Точно так же команда INC подобна другим командам чтение-изменение-запись. Первый такт каждой команды – считывание ее кода – не приведен в таблице. Во время второго такта выполняются короткие команды, не требующие работы с памятью. Например, INC A увеличивает содержимое регистра A на единицу. Другие команды нуждаются в чтении второго байта из памяти и загрузке его в регистр Md или MAR. В момент, когда байт считывается из памяти, регистр PC увеличивается на 1, таким образом, он будет указывать на следующий байт. Второй такт для команды ADD imm:

```
Md ← mem(PC); -- передача данных из памяти; PC ← PC+1;
```

Во время последнего цикла любой команды ADD данные из памяти добавляются к содержимому регистра A. Эта операция выполняется во время захвата из памяти следующей инструкции, следовательно, ADD imm требует только два такта для ее выполнения. Допускается совмещение операции АЛУ и считывание кода следующей команды из памяти, поскольку в регистре сохраняется старый код операции и новый не будет загружен в него до окончания сихротакта. Для команд с прямой адресацией, таких как ADD dir, прямой адрес считывается из памяти и помещается в регистр MAR на втором такте: MARL ← mem(PC); а MARH устанавливается в 0. В течение третьего такта адрес из MAR поступает на адресную шину, а данные из памяти сохраняются в Md:

```
Md ← mem(MAR);
```

Таблица 15.4. Пошаговая операция для 6805 команд

	2-й такт	3-й такт	4-й такт	5-й такт	6-й такт
ADD imm	**{addr1}** Md ← mem (PC) PC ← PC + 1	(A ← A + Md)*			
ADD dir	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{data}** Md ← mem(MAR)	(A ← A + Md)*		
ADD ix	**{addr1}** MARL ← X	**{data}** Md ← mem(MAR)	(A ← A + Md)*		
ADD ix1	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{addx}** MAR ← MAR + X	**{data}** Md ← mem(MAR)	(A ← A + Md)*	
ADD ext	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{data}** Md ← mem(MAR)	(A ← A + Md)*	
ADD ix2	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{addx}** MAR ← MAR + X	**{data}** Md ← mem(MAR)	(A ← A + Md)*
STA ext	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{data}** mem(MAR) ← A		
INC A	**{addr1}** A ← A + 1				
INC dir	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{data}** Md ← mem(MAR)	**{rd_mod_wr}** Md ← Md + 1	**{writeback}** mem(MAR) ← Md	
INC ix	**{addr1}** MARL ← X	**{data}** Md ← mem(MAR)	**{rd_mod_wr}** Md ← Md + 1	**{writeback}** mem(MAR) ← Md	
INC ix1	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{addx}** MAR ← MAR + X	**{data}** Md ← mem(MAR)	**{rd_mod_wr}** Md ← Md + 1	**{writeback}** mem(MAR) ← Md
JMP dir	**{addr1}** PCL ← mem (PC)				
JMP ix	**{addr1}** PCL ← X				
JMP ix1	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{addx}** PC ← MAR + X			
JMP ext	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** PCL ← mem(PC) PCH ← MARH			
JMP ix2	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{addx}** PC ← MAR + X		
JSR dir	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1 PC ← MAR		
JSR ix	**{addr1}** MARL ← X	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1 PC ← MAR		
JSR ix1	**{addr1}** MARL ← mem(PC) PC ← PC + 1	**{addx}** MAR ← MAR + X	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1 PC ← MAR	
JSR ext	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1 PC ← MAR	

Окончание таблицы 15.4

	2-й такт	3-й такт	4-й такт	5-й такт	6-й такт
JSR ix2	**{addr1}** MARH ← mem(PC) PC ← PC + 1	**{addr2}** MARL ← mem(PC) PC ← PC + 1	**{addrx}** MAR ← MAR + X	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1 PC ← MAR
RTS	**{addr1}** SP ← SP + 1	**{pop2}** PCH ← mem(SP) SP ← SP + 1	**{pop1}** PCL ← mem(SP)		
BRA rel	**{addr1}** Md ← mem(MAR) PC ← PC + 1	**{BRtest}** PC ← PC + sign_ext & Md			
SWI	**{addr1}** Нет действия	**{push1}** mem(SP) ← PCL SP ← SP - 1	**{push2}** mem(SP) ← PCH SP ← SP - 1	**{push3}** mem(SP) ← X SP ← SP - 1	**{push4}** mem(SP) ← A SP ← SP - 1
SWI прод.	(7-й такт) **{push5}** mem(SP) ← CCR SP ← SP - 1	(8-й такт) **{cycle8}** MAR ← vector addr. I ← 1	(9-й такт) **{cycle9}** PCH ← mem(MAR) MAR ← MAR + 1	(10-й такт) **{cycle10}** PCL ← mem(MAR)	
RTI	**{addr1}** SP ← SP + 1	**{pop5}** CCR ← mem(SP) SP ← SP + 1	**{pop4}** A ← mem(SP) SP ← SP + 1	**{pop3}** X ← mem(SP) SP ← SP + 1	6-й и 7-й такты аналогичны такту 3 и 4 команды RTS

1-й такт для всех команд - считывание ее кода

* АЛУ операция выполняется одновременно со считыванием кода следующей команды

Для расширенной адресации, например, ADD ext, старший байт адреса данных считывается из памяти во время второго такта, младший – третьего, а данные – четвертого. Для индексной адресации без смещения, ADD ix , X загружается в MAR в конце второго такта. Индексная адресация с одно- или двухбайтным смещением подобна одно- или двухбитовой расширенной адресации с дополнительным циклом, необходимым для того, чтобы добавить значение из регистра X к MAR.

Команды STA и STX подобны команде ADD с соответствующим адресным режимом, за исключением последнего такта, в котором данные записываются в память, а не считываются из нее.

Команды чтение-изменение-запись, такие как INC dirm, INC ixm и INC ixim, начинаются так же, как и соответствующая инструкция ADD. Тем не менее, вместо выполнения АЛУ операции и считывания кода следующей команды необходимо добавить два дополнительных такта для операции INC. Во время первого такта сложения выполняется операция INC, во время второго – результат записывается обратно в память.

Команда JMP подобна команде ADD с соответствующим адресным режимом. Только для JMP при вычислении адресации адрес загружается в PC вместо регистра MAR. Например, для JMP ext во время третьего такта выполняются следующие действия:

```
PC ← MARH&mem(PC);
```

Младший байт адреса перехода считывается из памяти, а старший уже находится в MARH. Эти два байта загружаются в PC и следующая команда считывается из сформированного адреса перехода.

Команда JSR подобна команде JMP, но в ней адрес возврата должен быть размещен в стеке. После того как адрес перехода был определен и загружен в MAR, младший и старший байты регистра PC заносятся в стек за два такта:

```
mem(SP) ← PCL;
SP<=SP-1;
```

```
mem(SP) ← PCH;
SP ← SP-1;
```

Изначально SP указывает на первую пустую позицию в стеке, следовательно, PCL записывается в эту позицию и SP увеличивается на 1. В конце второго цикла записи в стек адрес из регистра MAR загружается в PC для выполнения перехода.

Для команды RTS значение в SP увеличивается на 1 во время второго такта. На третьем старший байт адреса возврата считывается из памяти и загружается в PCH; а во время четвертого – младший байт PCL. На следующем такте адрес перехода из PC используется для считывания кода команды.

Для программных прерываний SWI во время второго такта не выполняется никаких действий, за исключением проверки кода команды и перехода к соответствующему следующему состоянию для начала выполнения последовательности стековых операций. Содержимое регистров PCL, PCH, X, A и CCR заносится в стек в течение 3-7 тактов. Адрес вектора прерываний загружается в MAR на 8 такте. Для подготовки выполнения подпрограммы обработки прерываний старший и младший байты адреса загружаются в PC в течение 9-го и 10-го тактов. Возврат для команды прерываний RTI подобен возврату для RTS, только теперь содержимое регистров CCR, A и X извлекается из стека перед выборкой адреса возврата. Последовательность действий для аппаратного прерывания почти такая же, как и для SWI, из нее только исключаются такт считывания команды и 2-й такт.

Следующий пример иллюстрирует последовательность операций выполнения команды ADD с прямой адресацией, за которой следует команда STA также с прямой адресацией. Пусть в памяти, начиная с адреса 300h, сохранены данные: BBh 43h B7h 57h 4Ch; значения в регистрах PC = 300h, A = 12h; ячейка по адресу 0043h содержит 36h. В таблице 15.5 представлена последовательность операций. Все данные записаны в шестнадцатеричном формате. Основные регистры обновляются в конце каждого такта. Например, код BBh считывается по адресу 0300h на первом такте, после чего PC увеличивается на 1 и BBh загружается в регистр Opcode.

Таблица 15.5. Пример пошагового выполнения команды

PC	MAR	Адресная шина	Шина данных	Opcode	A	Md	
0300	xxxx	0300	BB	xx	12	xx	Считывание кода
0301	xxxx	0301	43	BB	12	xx	Получение прямого адреса
0302	0043	0043	36	BB	12	xx	Чтение данных из памяти
0302	0043	0302	B7	BB	12	36	Сложение и считывание кода следующей команды
0303	0043	0303	42	B7	48	36	Получение прямого адреса
0304	0057	0057	48	B7	48	36	Сохранение данных в памяти
0305	0057	0305	4C	B7	48	36	Считывание кода следующей команды

Проектирование управляющего автомата CPU

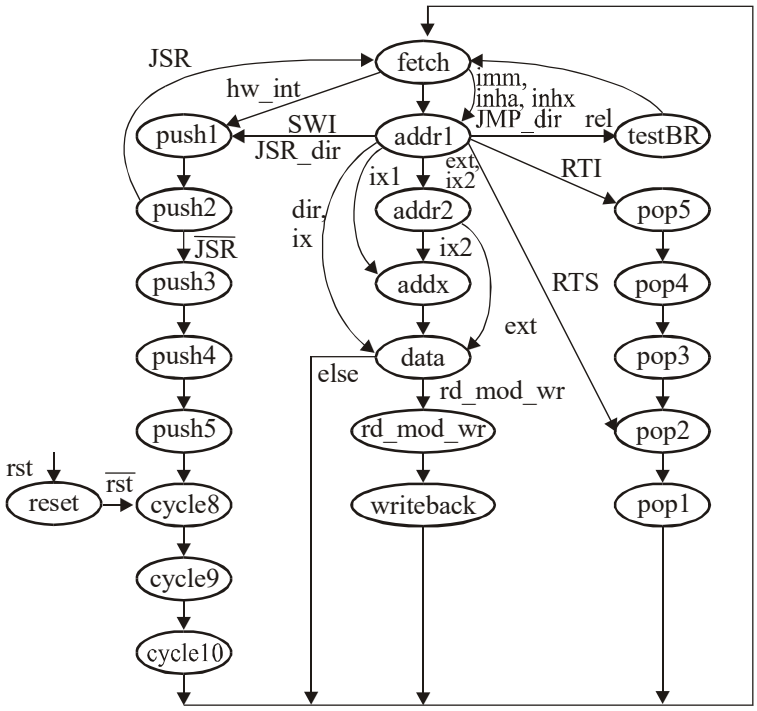
После определения пошаговых операций для выполняемых команд можно спроектировать управляющий автомат для CPU. При этом необходимо каждому действию из таблицы 15.4 поставить в соответствие состояние, имя которого записывается в скобках в верхней строке ячейки. Длительность каждого состояния – один синхротакт. После этого реализация команд рассматривается потактно, для определения действий, выполняемых на каждом шаге. Имя состояния выбирается в зависимости

от выполняемой в нем операции. Например, {addrl} используется для чтения первого байта адреса или смещения, состояние {addx} – для добавления X к MAR, {data} – для чтения данных из памяти, {push1} – для занесения содержимого регистра PCL в стек.

На рисунке 15.3 представлен фрагмент графа состояний управляющего автомата. В большинстве случаев их последовательность определяется режимом адресации. Из групп команд JMP и JSR на рисунке указаны пути только для команд JMP_dir и JSR_dir. Граф включает состояние {reset}, в которое автомат устанавливается после операции сброс. Затем выполняется переход в состояния {cycle8}, {cycle9} и (cycle 10}, для того чтобы получить начальный адрес исполняемого кода из таблицы векторов прерываний.

Выполнение любой команды начинается с состояния {fetch}, в котором ее код считывается из памяти и загружается в регистр Opcode, при этом PC увеличивается на 1 в момент перехода устройства в следующее состояние. Если команда регистр-память, например такая как ADD, предшествует состоянию {fetch}, то операция АЛУ и запись результата в регистр A или X происходят одновременно со считыванием кода следующей команды.

Рисунок 15.3. Фрагмент графа состояний для CPU контроллера



Поскольку код операции недоступен из регистра Opcode до перехода управляющего автоматат в следующее состояние, то у всех команд состояние для второго такта будет одинаковым. Идентификатор такта {addr1}, поскольку в нем часто считывается первый байт адреса. Все действия, перечисленные в таблице 15.4 в колонке "второй такт", выполняются в состоянии {addr1}.

Адресный режим и тип выполняемой команды определяют следующее за {addr1} состояние. Их последовательности для различных режимов перечислены в таблице 15.4. За последним состоянием всегда следует {fetch}.

Разрабатываемая VHDL-модель 6805 CPU основана на таблице 15.4 и рисунке 15.3. В ней для вычисления состояний управляющего автомата используется режим адресации и тип выполняемой команды. Код также описывает действия для каждого состояния. Такая VHDL-модель может быть использована для моделирования поведения CPU и проверки того, что все команды и адресные режимы обрабатываются в соответствии со спецификацией, описанной выше. Для удобства тестирования в архитектуре определен массив памяти mem, который не является компонентом CPU.

Полностью VHDL-код поведенческой модели процессора приведен в подразд.15.4. Он включает следующие основные блоки:

1. Декларация сигналов.
2. Процедура ALU_OP, реализующая операции АЛУ для команд регистр-память и всех операций с двумя операндами.
3. Процедура ALU_1, реализующая операции АЛУ и сдвиговые операции для одноопернадных команд чтение-изменение-запись.
4. Процедура fill_memory для занесения в память тестовых команд и данных.
5. Процесс cpu_cycles, описывающий необходимые для управляющего автомата CPU состояния и выполняемые в каждом из них действия.

После декларации сигналов, соответствующих регистрам PC, MAR, SP, Opcode, используются псевдонимы для разбиения регистров PC и MAR на старший и младший байты. Регистр Opcode делится на четыре младших бита (OP) и четыре старших (mode). Для того чтобы сделать код более понятным, используются константы для связи кода команды и адресного режима с соответствующим им шестнадцатеричным числом из таблицы 15.1. Например:

```
-- четыре младших бита кода описывают операцию
subtype ot is std_logic_vector(3 dovmto 0);
constant SUB: ot:="0000"; constant CMP: ot:="0001";
. . .
-- четыре старших бита кода описывают адресный режим
constant REL: ot:="0010"; constant DIRM: ot:="0011";
. . .
```

Процедура ALU_OP (рисунок 15.4) вызывается в состоянии {fetch} для выполнения команд регистр-память. Она определяет операции АЛУ, основываясь на таблице 15.2, а. Все арифметические операции выполняются над беззнаковыми байтами типа unsigned. Переменная res, длиной 9 битов, позволяет включать перенос в результат АЛУ операции. В зависимости от команды содержимое регистров A или X загружается в res(7 **downto** 0), а перенос C – в res(8). Обновление флагов N и Z основывается на значении переменной res.

Рисунок 15.4. Операции для команд регистр-память

```
-- выполняет АЛУ операции для команд с двумя операндами
procedure ALU_OP
     (Md: in std_logic_vector(7 downto 0);
     signal A, X: inout std_logic_vector(7 downto 0);
     signal N, Z, C: inout std_logic) is
        --результат АЛУ операции
variable res : std_logic_vector(8 downto 0);
       -- обновление флага NZ, выполняемое по умолчанию
variable updateNZ: Boolean:= TRUE;
begin
   case OP is
      when LDA => res = '0'&Md; A <= res (7 downto 0);
      when LDX => res = '0'&Md; X <= res (7 downto 0);
      when ADD => res = ('0'&A) + ('0'&Md);
         C <= res (8); A <= res(7 downto 0);
      when ADC => res = ('0'&A) + ('0'&Md) + C;
         C <= res(8); A <= res(7 downto 0);
      when SUB => res = ('0'&A) - ('0'&Md);
         C <= res(8); A <= res(7 downto 0);
      when SBC => res = ('0'&A) - ('0'&Md) - C;
         C <= res(8) A <= res(7 downto 0);
      when CMP => res = ('0'&A) - ('0'&Md); C <= res(8);
      when CPX => res:= ('O'&X) - ('O'&Md); C <= res(8);
      when ANDa => res := '0'&(A and Md); A <= res(7 downto 0);
      when BITa => res := '0'&(A and Md);
      when ORA => res := '0'&(A or Md); A <= res(7 downto 0);
      when EOR => res := '0'&(A xor Md); A <= res(7 downto 0);
      when others => updateNZ := FALSE;
```

```
   end case ;
   if updateNZ then N <= res (7);
      if res (7 downto 0) = "00000000" then Z <= '1';
         else Z <= '0'; end if;
   end if;
end ALU_OP;
```

Процедура ALU1 реализует операции АЛУ и сдвига для команд чтение-изменение-запись, определенных в таблице 15.2, б. Операндом op1 может быть значение из регистров A, X или Md, в зависимости от адресного режима. Восьмибитный результат каждой операции заносится в переменную res8. Исключением является команда NEG, в которой девятый бит используется для переноса.

Для выполнения процесса тестирования создана процедура fill_memory, которая не является частью CPU и предназначена для занесения в память тестовых команд и данных. Она вызывается после сброса процессора в начальное состояние и выполняет чтение теста из текстового файла. Чтение из памяти подменяется чтением из массива mem. Поскольку последний имеет целочисленную индексацию, всякий раз при обращении к нему вызывается функция CONV_INTEGER для преобразования содержимого регистра PC, MAR или SP в тип integer.

На рисунке 15.5 представлен процесс cpu_cycles, описывающий управляющий автомат с рисунка 15.3 и действия, выполняемые в каждом его состоянии. Изменение состояний и обновление регистров происходит по переднему фронту синхронизации. Если сигнал rst_b = '0', то процесс переходит в состояние {reset}, иначе – оператор case определяет сигнал ST и задает действия, соответствующие каждому состоянию.

Для команд регистр-память в состоянии {fetch} вызывается процедура ALU_OP. Если возникает аппаратное прерывание, то для инициализации записи содержимого регистров в стек автомат переходит в состояние {push1}, иначе – считывается следующая команда и выполняется переход в состояние {addr1}.

Рисунок 15.5. Фрагмент кода процесса cpu_cycles

```
cpu_cycles: process
     variable reg_mem, hw_interrupt, BR: Boolean;
     variable sign_ext: std_logic_vector(4 downto 0);
begin
   reg_mem:= (mode - imm) or (mode - dir) or (mode = ext) or
      (mode = ix) or (mode = ix1) or (mode = ix2);
   hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

   wait until rising_edge(CLK);
   if (rst_b = '0') then ST <= reset; fill_memory(mem);
   else
      case ST is
         when reset => SP <= "111111";
                       if (rst_b = '1') then ST <= cycles; end if;
         when fetch =>
                       if reg_mem then
                          ALU_OP(Md, A, X, N, Z, C); end if;
                       -- завершение предыдущей операции
                       if hw_interrupt then ST <= pushi;
                       else Opcode <= mem(CONV_INTEGER(PC));
                          PC <= PC+1;--считывание следующей команды
                          ST <= addr1;
                       end if;
         when addr1 =>
            case mode is
                  --выполнение операции над A
               when inha => ALU1(A, N, Z, C); ST <= fetch;
                  --выполнение операции над X
               when inhx => ALU1 (X, N, Z, C); ST <= fetch;
                  --получение непосредственных данных
               when imm =>  Md <= mem (CONV_INTEGER (PC));
```

```
                              PC <= PC+1; ST <= fetch;
                  when inhl =>
                     if OP = SWI then ST <= pushi;
                        elsif OP = RTS then ST<= pop2; SP <= SP+1;
                        elsif OP = RTI then ST <= pop5; SP <= SP+1;
                     end if;
                  when inh2 => case OP is
                  when TAX => X <= A;
                  when CLC => C <= '0' ;
                  when SEC => C <= '1';
                  when CLI => I <= '0';
                  when SEI => I <= '1';
                  when RSP => SP <= "111111";
                  when TXA => A <= X;
                  when others =>
                  assert(false)
                              report "illegal instruction, mode = inh2";
               end case;
               ST <= fetch;
            when dir =>
               if OP = JMP then
                     PC <= zero&mem(CONV_INTEGER(PC)); ST <= fetch;
               else MAR <= zero&mem(CONV_INTEGER(PC)); PC <=PC+1;
               -- получение непосредственного адреса
               if (OP=JSR) then ST <= push1; else ST <= data; end if;
end if;
(полностью процесс будет приведен далее)
```

В состоянии {addr1} с помощью условных операторов определяется режим адресации. Для режимов inha и inhx вызывается процедура ALU1, выполняющая заданную операцию над значением из регистра A или X, затем автомат переключается в состояние {fetch}. Для режима imm данные в регистр Md передаются непосредственно из памяти. Для режима inh2 выполняется последовательность операций, определенных в таблице 15.2, в. Для режима dir прямой адрес считывается из памяти, расширяется нулем и для большинства команд загружается в регистр MAR. При выполнении команды JMP_dir прямой адрес записывается в регистр PC, а не в MAR. В подразд.15.4 приведен VHDL-код, описывающий другие адресные режимы, операции для которых перечислены в таблице 15.4.

Команды условного перехода из таблицы 15.2, г выполняются в состоянии {testBR}. Булева переменная BR устанавливается в TRUE или FALSE в зависимости от выполняемой команды и установленного флага. Если BR имеет значение TRUE, то относительный адрес в регистре Md расширяется по знаку и прибавляется к текущему адресу из регистра PC. Ниже приведен фрагмент VHDL-кода для состояния {testBR}:

```
when testBR =>
   case OP is
      when BRA => BR := TRUE;    -- Постоянный переход
      when BRN => BR := FALSE;   -- Переход запрещен
      when BHI => BR := (C or Z) = '0';    -- Переход, если больше
      when BLS => BR := (C or Z) = '1';
                                  -- Переход, если меньше или равно
      when BCC => BR : = C = '0'; -- Переход, если перенос сброшен
      when BCS => BR := C = '1';
                                -- Переход, если перенос установлен
      . . . (остальные ветви инструкций опущены здесь)
   end case;
if Md(7) = '1' then sign_ext:= "11111";
   else sign_ext := zero; end if;
if BR then PC <= PC + (sign_ext&Md); end if;
ST <= fetch;
```

Чтение и запись данных в память выполняется в состоянии {data}. Для команд STA или STX содержимое регистров A или X записывается в память по адресу из

регистра MAR, значения флагов обновляются. Для остальных команд данные из памяти по адресу MAR передаются в регистр Md. Действия, реализуемые в других состояниях, соответствуют приведенным в таблице 15.4.

При возникновении прерывания содержимое регистров PCH, PCL, X, A и CCR заносится в стек в состояниях {push1) - {push5}. Затем в регистр MAR в состоянии {cycle8} заносится адрес вектора прерывания, а в состояниях {cycle9} и {cycle 10} в регистр PC – адрес подпрограммы обработки прерывания.

Разработанныя модель процессора тестируется на корректность выполнения команд из таблицы 15.2 для адресных режимов, перечисленных в таблице 15.3. Для этого в массив памяти загружается информация из файла, выполняются описанные команды, результат выводится в виде временных диаграмм. Синтез схемы CPU из его поведенческой модели невозможен по нескольким причинам. Во-первых, в коде не были описаны адресная шина и шина данных, не был определен интерфейс памяти, не учитывались временные параметры. В частности, чтение из памяти требует, чтобы адрес подавался на шину в начале синхротакта, для того чтобы данные из памяти могли быть считаны в его конце. Во-вторых, при написании VHDL-кода не ставилась задача его эффективной реализации в аппаратуре.

Аппаратное проектирование и синтезируемый VHDL-код

Далее выполняется изменение и детализация проекта для его эффективной реализации. Создается структурная схема устройства, основанная на списке регистров из таблицы 15.4 и соответствующем поведенческом VHDL-коде. Затем определяются генерируемые в каждом состоянии управляющие сигналы и переписывается VHDL-код для упрощения генерации этих сигналов.

Устройство делится на три части: управляющий модуль, адресный модуль, вычисляющий передаваемый на шину адрес, и модуль данных, выполняющий арифметические, логические и сдвиговые операции. Адресный модуль реализует функции: прибавить содержимое регистра X к MAR для индексной адресации, прибавить расширенное по знаку значение из регистра Md (SX&Md) к PC для относительной адресации, загрузка адресов в регистр PC и MAR , увеличение на 1 значений в регистрах SP, PC и MAR. На рисунке 15.6 изображен один из возможных вариантов структурной схемы адресного модуля. В нем используется один сумматор и для выбора слагаемых – мультиплексор. Значения в регистры MARH, MARL, PCH и PCL могут быть поданы по шине данных, с выхода сумматора и из таблицы векторов адреса (vector address table – VAN и VAL). Для того чтобы избежать введения дополнительных входов в мультиплексор MUX, регистр PC может загружаться значением MAR при выборе MAR и сигналом 0 со входов сумматора. Также PC или MAR могут получать информацию из регистра X, для этого на мультиплексорах выбираются входы 0 и 0&.X. На рисунке 15.6 не показана дополнительная логика, необходимая для увеличения на 1 значений в регистрах PC, MAR и SP, а также уменьшения на 1 содержимого регистра SP.

Модуль данных может включать два АЛУ. Первое – используется для реализации двухоперандных арифметических операций, таких как ADD и SUB, а второе – для одноопернадных операций, например, INC и COM. Другим вариантом может быть применение одного АЛУ для всех типов арифметических операций. Он и будет использован для реализации модуля данных, потому что позволяет уменьшить количество применяемой аппаратуры и упрощает установку флагов N, Z и C. На рисунке 15.7 представлена структурная схемы модуля данных. При использовании двух операндов: на первый вход может подаваться значение с регистра A или X, что определяет мультиплексор MUX1; на второй вход – с регистра Md мультиплексор MUX2. При выполнении операции с одним операндом мультиплексор MUX1 выбирает один из входов A, X и Md, а MUX2 – 0. Сдвиг реализуется сдвиговым регистром на выходе АЛУ. Для всех операций, кроме сдвига, значение на выход модуля подается через регистр без изменения. Для сдвиговых операций данные с регистров A, X и Md через вход Op1 АЛУ и через АЛУ поступают на

сдвиговый регистр. В такой схеме результат всех операций всегда будет передаваться через один выход shiftout и записываться в регистр A, X или Md. Значение на выходе может быть протестировано для установки флагов N и Z.

Рисунок 15.6. Адресный модуль

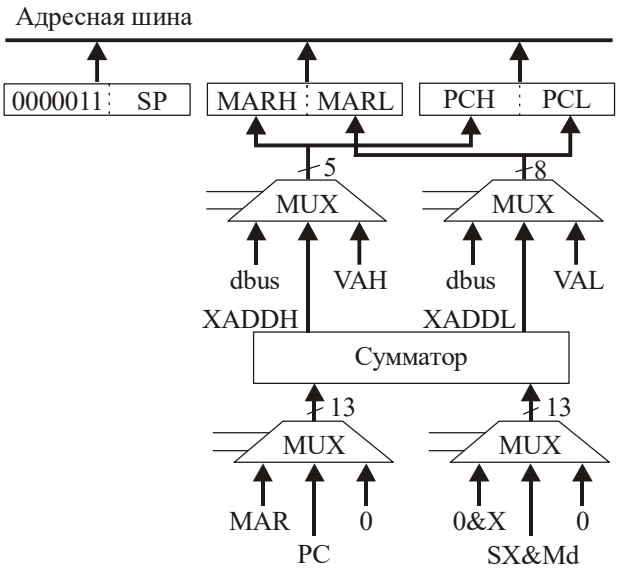
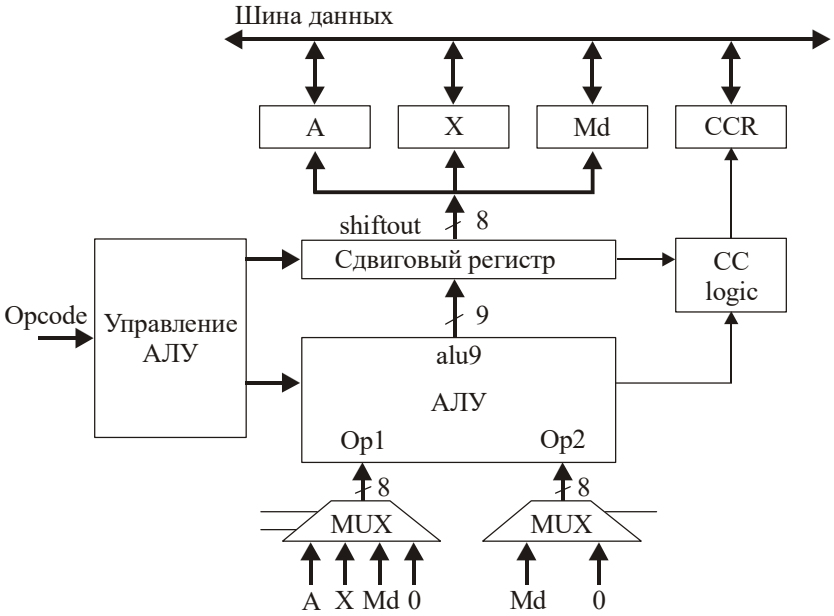


Рисунок 15.7. Модуль данных



Входы мультиплексоров АЛУ описываются следующим образом:

```
-- описание входов мультиплексоров в АЛУ
op1 <=  A when selA = '1'  -- MUX for op1
        else X when selX = '1'
        else Md when selMdl = '1'
        else "00000000";
op2 <=  Md when selMd2 = '1' --MUX for op2
        else "00000000";
```

Для управления АЛУ введены зависящие от выполняемой команды сигналы selA, selX, selMdl и selMd2.

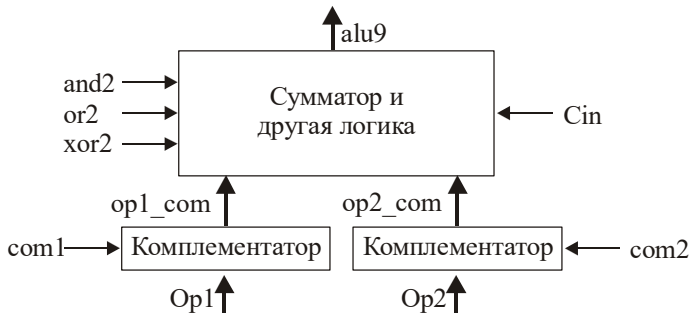
На рисунке 15.8 изображена более подробная схема АЛУ. Все операции, за исключением AND, OR и XOR, могут быть реализованы с использованием сумматора и формирователя дополнительного кода. Последний должен быть подключен к обоим входам сумматора, поскольку операция COM требует формирования дополнительного кода для входа op1, а операция вычитания – дополнительного

кода на входе op2. Когда управляющий сигнал and2 = '1', значение на выходе АЛУ равно op1 **and** op2; если or2 = '1', то op1 **or** op2; если xor2 = '1', то op1 **xor** op2. Иначе – на выход АЛУ подается сумма сигналов opl_com, op2_com и Cin. Ниже представлен VHDL-код, описывающий АЛУ:

```
   -- формирование обратного кода
op1_com <= not op1 when com1='1' else op1;
op2_com <= not op2 when com2='1' else op2;
   -- логические операции
alu9 <= '0'&(op1_com and op2_com) when and2 ='1'
   else '0'&(op1_com or op2_com) when or2 ='1'
   else '0'&(op1_com xor op2_com) when xor2 ='1'
   else ('0'&op1_com) + ('0'&op2_com) + Cin;    -- сумматор
```

Выход alu9, разрядностью в 9 битов, составлен из переноса и 8-битной суммы. При выполнении логических операций перенос устанавливается в '0'. Для контроля за работой АЛУ используются сигналы corn1, com2, Cin, and2, or2 и xor2.

Рисунок 15.8. Схема АЛУ



VHDL-код для сдвигового регистра имеет вид:

```
shiftout <=  shiftin&alu9(7 downto 1) when rsh='1' -- сдвиг вправо
        else alu9(6 downto 0)&shiftin when lsh='1' -- сдвиг влево
        else alu9(7 downto 0);     -- без изменения
```

Если rsh ='1', то значение с выходов АЛУ сдвигается вправо; если lsh = '1', то выполняется сдвиг влево; иначе – сигнал передается через сдвиговый регистр без изменения. В зависимости от типа выполняемой сдвиговой операции переменная shiftin может быть равна '0', С или знаковому биту.

В подразд.15.5 приведен синтезируемый VHDL-код для CPU. Интерфейс CPU6805 содержит адресную шину и шину данных, следовательно, к нему можно подключать память RAM и другие системные компоненты. Архитектура процессора состоит из следующих частей:

1. Декларация сигналов и констант.
2. Параллельные операторы шинного интерфейса, входных мультиплексоров в АЛУ, АЛУ и сдвиговый регистр, адресный сумматор и дешифратор операций.
3. Процесс ALU_control, генерирующий управляющие сигналы для АЛУ и для загрузки регистров A, X и Md.
4. Процесс CPU_control, реализующий управляющий автомат устройства (см. рисунок 15.3) и управляющие сигналы для загрузки регистров.
5. Процесс update_reg, обновляющий содержимое регистров по переднему фронту синхросигнала.

Декларация сигналов для регистров и состояний в архитектуре CPU1 такая же, как и в модели поведенческого уровня. В CPU1 используются параллельные операторы, которые подразумевают тристабильные буферы для управления адресной шиной и шиной данных. Код для интерфейса шины данных:

```
-- управление шиной данных с помощью тристабильных буферов
dbus <= A when A2db='1' else hi_Z;
dbus <= X when X2db='1' else hi_Z;
```

```
dbus <= Md when Md2db='1' else hi_Z;
dbus <= "000"&PCH when PCH2db='1' else hi_Z;
dbus <= PCL when PCL2db='1' else hi_Z;
dbus <= "1110"&CCR when CCR2db='1' else hi_Z;
```

Управляющие сигналы, такие как A2db (значение с A подается на шину) и PCL2db (значение с PCL подается на шину), генерируются в процессе CPU_control.

В поведенческом коде операция OP тестируется в нескольких командах case и в других операторах. Программа синтеза могла бы создавать 4-битный компаратор для каждой проверки значения OP. Чтобы избежать этого, код изменяется – в него добавляется 4x16 дешифратор. Его входом будет OP = Opcode(3 **downto** 0), а выходом – 16-битный вектор opd. Дешифрация реализуется с помощью константы decode типа массив:

```
type decode_type is array(0 to 15) of bit_vector(15 downto 0);
signal opd: bit_vector(15 downto 0);
constant decode: decode_type:=
     X"0001", X"0002", X"0004", X"0008", X"0010", X"0020",
     X"0040", X"0080", X"0100", X"0200", X"0400", X"0800",
     X"1000", X"2000", X"4000", X"8000");
. . .
opd <= decode(TO_INTEGER(Opcode(3 downto 0))); --дешифратор 4-16
```

Если OP = "0000", тогда opd(0) = '1'; если OP = "0001", то opd(l) = '1'. Для того чтобы сделать код более читаемым, для кода команды используется псевдоним opd. Таким образом, SUB – это opd(0), CMP – opd(l). Затем при каждом тестировании кода команды выполняется тестирование одного бита, следствием чего является более эффективная схема. Для адресного режима , наоборот, выбирается неявное декодирование, которое выполняется так же, как и в поведенческом коде.

Процесс ALU_control генерирует сигналы, необходимые для выполнения большинства команд регистр-память, чтение-изменение-запись. В начале процесса управляющие сигналы инициализируются их наиболее часто используемыми значениями. Например, updateNZ и updateC устанавливаются в '1', поскольку флаги N, Z и C обновляются для большинства операций. Управляющие сигналы selA, selMd2 и ALU2A устанавливаются в '1', потому что для большинства операций значения на входы АЛУ поступают с A и Md, а значения с выхода АЛУ сохраняются в A. Для команд reg_mem дополнительные сигналы устанавливаются или сбрасываются по необходимости. Например, для SBC (вычитание с заёмом) com2 устанавливается в '1' для формирования дополнительного кода op2 и Cin <= **not** C, поскольку заем сохраняется в C. Для команды CPX значение из регистра Md должно вычитаться из X, поэтому selX <= '1', selA <= '0' для выбора X как выхода мультиплексора MUX1. Также com2 <='1' и Cin <= '1' в целях прибавления дополнительного кода Md к значению в регистре X. Фрагмент VHDL-кода для процесса ALU_control:

```
--установка управляющих сигналов в состояние по умолчанию
Cin <= '0'; com1 <= '0'; com2 <= '0';
updateNZ<='1';  updateC<='1';
ALU2A <= '1'; ALU2X <= '0'; ALU2Md <= '0';
selA <= '1'; selx <= '0'; selMdl <= '0'; selMd2 <= '1';
. . .
   if SUBC = '1' then com2 <= '1'; Cin<= not C; end if;
   if CPX = '1' then selX <= '1'; selA <= '0'; com2 <= '1';
      Cin <= '1'; ALU2A<='0'; end if;
```

Для команд rd_mod_wr (чтение-изменение-запись) selMd2 <= '0', следовательно, op2 должен быть равен нулю. Для адресного режима inha устанавливаемые по умолчанию значения сигналов selA = '1' и ALU2A = '1' выбирают A для op1 и сохраняют результат в A. Для inhx – selX <= '1' and ALU2X <= '1', следовательно, операция выполняется на X. Для других режимов адресации команд rd_mod_wr selMdl <= '1' и ALU2Md <= '1', поэтому операции выполняются над Md. Другие

управляющие сигналы устанавливаются в 0 или 1 по необходимости. Например, для циклического сдвига ROR1 сигналы rsh <='1' и shiftin <= C. Для команды DEC updateC <= '0', следовательно, флаг Cflag не обновляется. Также com2 <= '1', следовательно, "11111111" (-1) складывается с op1. Для команды CLR and2 <= '1', поэтому op2 = 0 складывается по И с op1, чтобы сформировать на выходе АЛУ нуль, который будет загружен в соответствующий регистр.

Процесс CPU_cycles из поведенческого кода делится на два: CPU_control и update_reg. Процесс CPU_control реализует управляющий автомат, в котором последовательность состояний такая же, как и в процессе CPU_cycles. Тем не менее, вместо того, чтобы описать передачу данных между регистрами непосредственно в CPU_control, этот процесс делается комбинационным, генерирующим управляющие сигналы для загрузки регистров Opcode, PC, MAR и SP для выполнения адресных вычислений, управления интерфейсом адресной шины и шины данных.

На рисунке 15.9 представлен фрагмент процесса CPU_control. VHDL-код для каждого состояния основан на его описании в процессе CPU_cycles. Например, для состояния {fetch} код:

```
Opcode <= mem(PC), PC <= PC+1; ST <= addr1;
```

заменяется на:

```
PC2ab<='1'; db2opcode<='1'; incPC<='1'; nST <= addr1;
```

По сигналу PC2ab данные из регистра PC поступают на адресную шину ab, а по сигналу db2opcode разрешается загрузка регистра Opcode информацией с шины данных db. Управляющие сигналы для обновления регистров A, X, Md и флагов также генерируются в процессе ALU_control. Однако это можно выполнить при осуществлении некоторого условия. Поэтому сигнал обновления устанавливается в '1', когда разрешается обновление регистров A, X,Md и флагов. Это происходит в состоянии {fetch}, если выполняются команды регистр-память, кроме JMP или JSR. Последние операции заменяют вызов процедуры ALU_OP, как это делалось в процессе CPU_cycles.

Рисунок 15.9. Фрагмент кода процесса CPU_control

```
CPU_control: process (ST, rst_b, opd, mode, IRQ, SCint, CCR, MAR,
X, PC, Md)
variable reg_mem, hw_interrupt, BR: boolean;
begin
   nST <= reset; BR := FALSE; wr <= '0'; update <= '0';
   ixaddl <= (others=>'0'); xadd2 <= (others=>'0'); va <= "000";
   db2A <= '0'; db2X <= '0'; db2Md <= '0'; db2CCR <= '0';
   db2opcode <= '0';
    ...  (все управляющие сигналы устанавливаются в '0')
   reg_mem:=  (mode = imm) or (mode = dir) or (mode = ext) or
              (mode = ix) or (mode = ix1) or (mode = ix2);
   hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');
   if (rst_b = '0') then nST <= reset;
   else
   case ST is
      when reset => setSP <= '1';
         if (rst_b = '1') then nST <= cycleS; end if;
      when fetch => if (reg_mem and JMP = '0' and JSR = '0') then
                       update <= '1'; end if;--обновление регистров,
                          -- если выполняется не JMP или JSR команда
                    if hw_interrupt then nST <= push1;
                          --чтение кода команды
                    else PC2ab<='1'; db2opcode<='1'; incPC<='1';
                          nST <= addr1; end if;
      when addr1 =>
         case mode is
```

```
        when inha|inhx => update <= '1'; nST<= fetch;
        when imm => PC2ab<= '1'; db2Md<='l'; incPC<='1';
                    nST <= fetch;
        when inhl => if SWI = '1' then nST <= push1;
                     elsif RTS = '1' then nST <= pop2; incSP<='1';
                     elsif RTI = '1' then nST <= pop5; incSP<='1';
                     end if;
        when inh2 => update <= '1'; nST <= fetch;
        when dir => PC2ab<='l';
             if JMP='1' then db2PCL<='l'; clrPCH<='l'; nST<=fetch;
             else db2MARL<='1'; clrMARH<='1'; incPC <= '1';
                if JSR='1' then nST<=push1;
                else nST<=data; end if;
             end if;
      (полностью процесс приведен ниже)
```

Процесс update_reg обновляет регистры по переднему фронту синхросигнала. Фрагмент из него для изменения значений в регистре PC имеет вид:

```
wait until CLK'event and CLK='1';
if incPC = '1' then PC <= PC + 1; end if;
if xadd2PC = '1' then PC <= xadd; end if;
if db2PCH = '1' then PCH <= dbus(4 downto 0); end if;
if MARH2PCH = '1' then PCH <= MARH; end if;
if MARL2PCL = '1' then PCL <= MARL; end if;
if drPCH = '1' then PCH <= "00000"; end if;
if db2PCL = '1' then PCL <= dbus; end if;
if X2PCL = '1' then PCL <= X; end if;
```

В данном коде нет явного определения мультиплексора. Тем не менее программа синтеза создает мультиплексор, если значение в регистр может загружаться из различных источников. Если такая программа имеет хороший алгоритм оптимизации, в результате может быть получена эффективная реализация схемы. Иначе, чтобы гарантировать качественный результат синтеза, следует явно описать мультиплексоров в VHDl-коде.

Код для обновления регистров A, X, Md и флагов имеет вид:

```
if (update = '1') then
   if (ALU2A = '1') then A <= Shiftout; end if;
   if (ALU2X = '1') then X <= Shiftout; end if;
   if (ALU2Md = '1') then Md <= Shiftout; end if;
   if updateNZ='1' then N <= Shiftout(7);
      if Shiftout = "00000000" then Z <= '1'; else Z <= '0';
      end if;
   end if;
   if updateC='1' then C <= newC; end if;
end if;
```

Сигналы обновления генерируются в процессе CPU_control, а другие управляющие сигналы – в ALU_control.

После создания VHDL-модель тестируется с использованием TestBench.

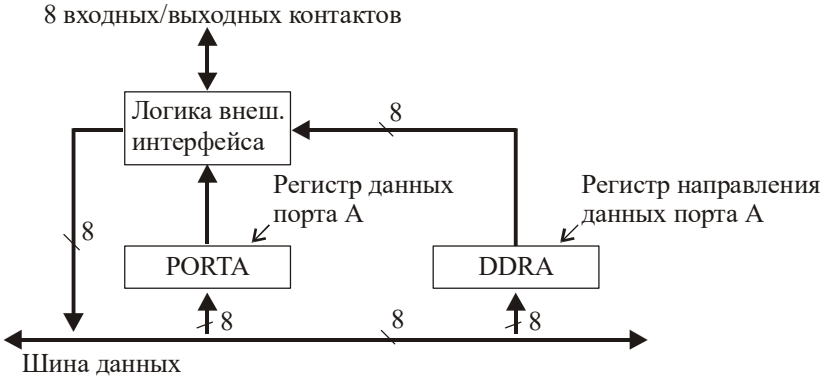
15.3. Завершение проекта микроконтроллера

Завершение проектирования микроконтроллера, изображенного на рисунке 15.1, основывается на рассмотренных проектах UART и CPU. После разработки модели параллельного порта все эти устройства используются для формирования микроконтроллера.

Каждый параллельный порт имеет восемь двунаправленных вход/выходных контактов и два 8-битных регистра, как это изображено на рисунке 15.10. Состояние регистра направления данных DDRA (data direction register) определяет вход и выходы устройства. Если значение бита в DDRA равно '1', то соответствующий ему контакт будет выходом, иначе – входом. Регистры портов являются отображаемы-

ми в памяти, следовательно, процессор может загружать данные в него и считывать их. В режиме записи данные заносятся в регистр PORTA, а затем передаются через любые контакты, отмеченные как выходы. В режиме чтения из PORTA данные снимаются с вход\выходных контактов. Если контакт запрограммирован как вход, то с него выполняется считывание данных, если как выход, считываемые данные такие же, как и в регистре PORTA.

Рисунок 15.10. Структурная схема параллельного порта



На рисунке 15.11 представлен VHDL-код для параллельного порта. Управляющие сигналы для чтения и записи в регистры реализуются в помощью параллельных операторов. Если порт выбран для чтения или записи, то сигнал Port_Sel = '1'. Единственный адресный бит ADDR0 служит для выбора регистра PORTA или DDRA. Оператор generate имеет метку Portbits и используется для генерации логики, связанной с каждым битом порта. Процесс обновляет регистры портов по переднему фронту синхросигнала.

Рисунок 15.11. VHDL-модель параллельного порта

```
library ieee;
use ieee.std_logic_1164.all;
entity PORT_A is
   port( clk, rst_b, Port_Sel, ADDRO, R_W: in std_logic;
         DBUS: inout std_logic_vector(7 downto 0);
         PinA: inout std_logic_vector(7 downto 0));
end PORT_A;

architecture port1 of PORT_A is
   signal DDRA, PORTA: std_logic_vector(7 downto 0);
   signal loadDDRA, loadPORTA, ReadPORTA, ReadDDRA: std_logic;
begin
loadPORTA <= '1' when (Port_Sel='1' and ADDRO='0' and R_W='1') else '0';
loadDDRA <= '1' when (Port_Sel= ' 1' and ADDRO='l' and R_W= '1') else '0';
ReadPORTA <= '1' when (Port_Sel='1' and ADDRO='O' and R_W='0') else '0';
ReadDDRA <= '1' when (Port_Sel='1' and ADDRO='l' and R_W= '0') else '0';

-- логика интерфейса контактов
Portbits: for i in 7 downto 0 generate
     -- установка состояния внешних контактов
     PinA(i) <= PORTA(i) when DDRA(i) = '1' else 'Z';
     --чтение регистра направления данных
     DBUS(i) <= DDRA(i) when (ReadDDRA = '1')
              else PinA(i) when (ReadPORTA = '1') else 'Z';
   end generate;
process (clk, rst_b) -- процесс выполняет запись данных в порт
   begin
      if (rst_b = '0') then DDRA <= "00000000";  -- конфигурация
                                          -- всех контактов входами
      elsif (rising_edge(clk)) then
         if (loadDDRA = '1') then DDRA <= DBUS; end if;
         if (loadPORTA = '1') then PORTA <= DBUS; end if;
      end if;
```

```
  end process;
end porti;
```

Далее необходимо объединить все компоненты вместе для формирования микроконтроллера. В VHDL-модели, представленной на рисунке 15.12, реализуется копия компонента cpu6805, две копии компонента PORT_A (параллельный порт), одна – UART, low RAM и high RAM. Для того чтобы выполнить синтез устройства для одной микросхемы Xilinx 4020E FPGA или Altera FLEX 10K20 CPLD, уменьшается размер памяти. Для FPGA изначально используется одна память 32 x 8 RAM для нижней области с прямой адресацией и вторая память 32 x 8 RAM для верхней области, в которой применяется расширенная адресация. Компонент ram32X8_io использует восемь ячеек CLB, сконфигурированных для работы в режиме памяти (чтение/запись). Для создания тристабильной вход\выходной линии используется тристабильный буфер каждого CLB.

Устройство имеет следующую карту памяти:

- PortA имеет адресацию 0000h-0001h.
- PortB – 0002h-0003h.
- SCI – 0004h-0007h.
- Нижняя RAM – 0020h-003Fh.
- Верхняя RAM – IFE0h-lFFFh.

Дешифратор адреса использует условные операторы назначения сигнала.

Сигнал wr для CPU при записи содержимого регистров в стек устанавливается для нескольких последовательных синхроциклов. Для внесения данных в память по байту за каждый синхротакт вместе с сигналом записи генерируется сигнал разрешения записи we. Запись в модуль памяти выполняется по заднему фронту синхросигнала. Для того чтобы избежать сохранения в памяти ложных сигналов, сигнал разрешения записи we и выбора памяти cs следует установить в 1 в течение второй половины синхротакта, когда адресная шина и шина данных должны быть стабильными.

```
csl <= SelLowRam and not clk; -- выбор второй половины синхротакта
we <= wr and not clk;   -- сигнал разрешения записи во второй
                           половине синхротакта
```

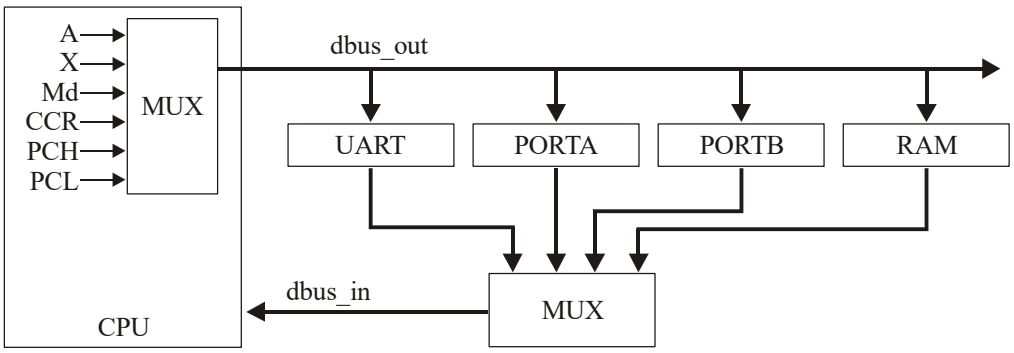
Рисунок 15.12. VHDL-модель микроконтроллера 6805

```
library ieee;
use ieee.std_logic_1164.all;

entity m68hc05 is
  port(clk, rst_b, irq, RxD: in std_logic;
     PortA, PortB: inout std_logic_vector(7 downto 0);
     TxD : out std_logic);
end m68hc05;

architecture M6805_64 of m68hc05 is
  component cpu6805
    port( clk, rst_b, IRQ, SCint: in std_logic;
          dbus: inout std_logic_vector(7 downto 0);
          abus: out std_logic_vector(12 downto 0);
          wr: out std_logic);
  end component;
  component ram32x8_io
    port( addr_bus: in std_logic_vector(4 downto 0);
          data_bus: inout std_logic_vector(7 downto 0);
          cpu_wr: in  std_logic);
  end component;
  component PORT_A
    port( clk, rst_b, Port_Sel, ADDR, R_W: in std_logic;
          DBUS: inout std_logia.Lvector(7 downto 0);
          PinA: inout std_logic_vector(7 downto 0));
```

```
   end component;
   component UART
      port( SCI_sel, R_W, elk, rst_b, RxD: in std_logic;
            ADDR: in std_logic_vector(1 downto 0);
            DBUS: inout std_logic_vector(7 downto 0);
            SCI_IRQ, TxD: out std_logic) ;
   end component;
signal SCint, wr, cs, we: std_logic;
signal SelLowRam, SelHiRAM, SelPA, SelPB, SeISC: std_logic;
signal addr_bus: std_logic_vector(12 downto 0):= (others => '0')
signal data_bus: std_logic_vector(7 downto 0):= (others => '0')
begin
CPU: cpu6805
   port map (clk, rst_b, irq, SCint, data_bus, addr_bus, wr);
PA: PORT_A
   port map (clk, rst_b, SelPA, addr_bus(0), wr, data_bus, PortA);
PB: PORT_A
   port map (clk, rst_b, SelPB, addr_bus(0), wr, data_bus, PortB);
Uarti: UART
   port map ( SeISC, wr, clk, rst_b, RxD, addr_bus(1 downto 0),
              data_bus, SCint, TxD);
LowRAM: ram32X8_io
   port map (addr_bus(4 downto 0), data_bus, csl, we);
HiRAM:  ram32x8_io
   port map (addr_bus(4 downto 0), data_bus, cs2, we);
-- интерфейс памяти
csl <= SelLowRam and not clk;   -- выделить память во второй
                                -- половине синхротакта
cs2 <= SelHiRam and not clk;
we <= wr and not clk;   -- сигнал разрешения записи во второй
                        -- половине синхротакта
-- адресный дешифратор
SelPA <= '1' when addr_bus(12 downto 1)= "000000000000" else '0';
SelPB <= '1' when addr_bus(12 downto 1)= "000000000001" else '0';
SeISC <= '1' when addr_bus(12 downto 2)= "00000000001"  else '0';
SelLowRam <= '1' when addr_bus(12 downto 5)= "00000001" else '0';
--         32 <= addr <= 63
SelHiRam <= '1' when addr_bus(12 downto 5) = "11111111" else '0';
--        addr >= 8160 (IFEOh)
end M6805_64;
```

Для того чтобы синтезировать схему микроконтроллера для FLEX 10K20, необходимо внести некоторые изменения в VHDL-код. Поскольку 10K20 не поддерживает внутренние тристабильные двунаправленные шины, изменяется структура шины процессора. Для выбора данных, поступающих или выходящих из CPU, используется мультиплексор, как это показано на рисунке 15.13. Также изменяются компоненты памяти, для нее применяются четыре блока 10K20 EAB, сконфигурированных как 1024 x 8 RAM.

Рисунок 15.13. Схема мультиплексирующей шины данных



15.4. Поведенческая VHDL-модель M6805 CPU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.CONV_STD_LOGIC_VECTOR;
use std.textio.all;

entity M6805 is
  port(clk: in std_logic;
     rst_b: in std_logic; -- активный по нулю сигнал сброса
     IRQ, SCint: in std_logic); -- сигналы аппартного прерывания
end M6805;

architecture behv of M6805 is

type RAMtype is array (0 to 8191) of std_logic_vector(7 downto 0);
signal mem: RAMtype:= (others=> (others=> '0'));
signal memory: std_logic_vector(7 downto 0);
signal Opcode, A, X, Md:std_logic_vector(7 downto 0):=(others=>'0');
alias OP: std_logic_vector(3 downto 0) is Opcode(3 downto 0);
alias mode: std_logic_vector(3 downto 0) is Opcode(7 downto 4);
type state_type is ( reset, fetch, addrl, addr2, addX, data,
                     rd_mod_wr, writeback, testBR, push1, push2,
                     push3, push4, push5, cycle8, cycle9,cycle10,
                     pop5, pop4, pop3, pop2, pop1);
signal ST: state_type;

signal CCR: std_logic_vector(3 downto 0);
alias I: std_logic is CCR(3);
alias N: std_logic is CCR(2);
alias Z: std_logic is CCR(1);
alias C: std_logic is CCR(0);

signal PC, MAR: std_logic_vector (12 downto 0) ;
alias PCH : std_logic_vector(4 downto 0) is PC(12 downto 8);
alias PCL : std_logic_vector(7 downto 0) is PC(7 downto 0);
alias MARH : std_logic_vector(4 downto 0) is MAR(12 downto 8);
alias MARL : std_logic_vector(7 downto 0) is MAR(7 downto 0);
signal SP: std_logic_vector(5 downto 0);
constant zero: std_logic_vector(4 downto 0) := "00000";
-- четыре младших бита кода команды
subtype ot is std_logic_vector(3 downto 0);
constant SUB: ot:="0000"; constant CMP: ot:="0001";
constant SBC: ot:="0010"; constant CPX: ot:="0011";
constant ANDa: ot:="0100"; constant BITa: ot:="0101";
constant LDA: ot:="0110"; constant STA: ot:="0111";
constant EOR: ot:="1000"; constant ADC: ot:="1001";
constant ORA: ot:="1010"; constant ADD: ot:="1011";
constant JMP: ot:="1100"; constant JSR: ot:= "1101";
constant LDX: ot:="1110"; constant STX: ot:="1111";

constant RTI: ot:="0000"; constant RTS: ot:="0001";
constant SWI: ot:="0011"; constant TAX: ot:="0111";
constant CLC: ot:="1000"; constant SEC: ot:="1001";
constant CLI: ot:="1010"; constant SEI: ot:="1011";
constant RSP: ot:="1100"; constant TXA: ot:="1111";

constant NEG: ot:="0000"; constant COM: ot:="0011";
constant LSR: ot:="0100"; constant RORx: ot:="0110";
constant ASR: ot:="0111"; constant LSL: ot:="1000";
constant ROLx: ot:="1001"; constant DEC: ot:="1010";
constant INC: ot:="1100"; constant CLR: ot:="1101";
constant TST: ot:="1111";

constant BRA: ot:="0000"; constant BRN: ot:="0001";
```

```
constant BHI: ot:="0010"; constant BLS: ot:="0011";
constant BCC: ot:="0100"; constant BCS: ot:="0101";
constant BNE: ot:="0110"; constant BEQ: ot:="0111";
constant BPL: ot:="1010"; constant BMI: ot:="1011";
constant BMC: ot:="1100"; constant BMS: ot:= "1101";

-- четыре старших бита кода команды
constant REL: ot:="0010"; constant DIRM: ot:="0011";
constant INHA: ot:="0100"; constant INHX: ot:="0101";
constant IX1M: ot:="0110"; constant IXM: ot:="0111";
constant INH1: ot:="1000"; constant INH2: ot:="1001";
constant IMM: ot:="1010"; constant DIR: ot:="1011";
constant EXT: ot:="1100"; constant IX2: ot:="1101";
constant IX1: ot:="1110"; constant IX: ot:="1111";

procedure ALU_OP  -- выполняет операции АЛУ
     (Md : in std_logic_vector(7 downto 0);
   signal A, X: inout std_logic_vector(7 downto 0);
   signal N, Z, C: inout std_logic) is
     -- результат АЛУ-операции
   variable res : std_logic_vector(8 downto 0);
     -- выполняемое по умолчанию обновление флага
   variable updateNZ : Boolean := TRUE;
begin
case OP is
   when LDA =>  res:= '0'&Md; A <= res(7 downto 0);
   when LDX =>  res:= '0'&Md; X <= res(7 downto 0);
   when ADD =>  res:=('0'&A) + ('0' & Md);
                C <= res(8); A <= res(7 downto 0);
   when ADC =>  res:=('0'&A) + ('0'&Md);
                C <= res(8); A <= res(7 downto 0);
   when SUB =>  res:=('0'&A) - ('0'&Md);
                C <= res(8); A <= res(7 downto 0);
   when SBC =>  res:=('0'&A) - ('0'&Md)-C;
                C <= res(8); A <= res(7 downto 0);
   when CMP =>  res:=('0'&A) - ('0'&Md); C <= res(8);
   when CPX =>  res:=('0'&X) - ('0'&Md); C <= res(8);
   when ANDa => res :='0'&(A and Md); A <= res(7 downto 0);
   when BITa => res :='0'&(A and Md);
   when ORA =>  res := '0'&(A or Md); A <= res(7 downto 0);
   when EOR =>  res := '0'&(A xor Md); A <= res(7 downto 0);
   when others => updateNZ:= FALSE;
end case;

   if updateNZ then N <= res(7);
     if res(7 downto 0) = "00000000" then Z <= '1';
     else Z <= '0'; end if;
   end if;
end ALU_OP;

Procedure ALU1  -- выполняет операции над одним операндом
     (signal op1: inout std_logic_vector(7 downto 0);
     signal N, Z, C : inout std_logic) is
variable res9 : std_logic_vector(8 downto 0);
variable res8 : std_logic_vector(7 downto 0);
begin
   case OP is
     when NEG =>  res9 := not('0'&op1) + 1;
                  C <= res9(8); res8:= res9(7 downto 0);
     when COM =>  res8:= not op1; C <= '1';
     when LSR =>  res8:= '0'&op1(7 downto 1); C <= op1(0);
     when RORx => res8:= C&op1(7 downto 1); C <= op1(0);
     when ASR =>  res8:= op1(7)&op1(7 downto 1); C <= op1(0);
     when LSL =>  res8:= op1(6 downto 0)&'0'; C <= op1 (7);
     when ROLx => res8:= op1(6 downto 0)&C; C <= op1(7);
     when DEC =>  res8:= op1 - 1;
     when INC =>  res8:= op1 + 1;
```

```
      when CLR =>  res8:= "00000000";
      when TST =>  res8:= op1; C <= '0';
      when others => assert (false) report "illegal opcode";
   end case;
op1 <= res8; N <= res8 (7);
   if (res8 = "00000000") then Z <= '1';
   else Z <= '0'; end if;
end ALU1;

Procedure fill_memory (signal mem: inout RAMType) is
   type HexTable is array(character range <>) of integer;
-- шестн. символы: 0, 1, ... A, B, C, D, E, F (только верхний регистр)
constant lookup: HexTable('0' to 'F'):= (0, 1, 2, 3, 4, 5, 6, 7,
      8, 9, -1, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
-- файл открывается для чтения
   file infile: text open read_mode is "meml.txt";
   variable buff: line;
   variable addr_s: string(4 downto 1);
   variable data_s: string(3 downto 1);  --data_s(l) содержит пробел
   variable addr, byte_cnt: integer;
   variable data: integer range 255 downto 0;
begin
   while (not endfile(infile)) loop
      readline (infile, buff);
      read (buff, addr_s);    -- прочитать шестн. адрес
      read(buff, byte_cnt);  -- прочитать число байтов для считывания
      addr:= lookup(addr_s(4))*4096 + lookup(addr_s(3))*256
         + lookup(addr_s(2))*16 + lookup(addr_s(1));
      readline (infile, buff);
      for i in 1 to byte_cnt loop
         -- чтение 2-х шестнадцатеричных цифр и пробела
         read (buff, data_s);
         data:= lookup(data_s(3))*16 + lookup(data_s(2));
         mem(addr) <= CONv_STD_LOGIC_VECTOR(data, 8);
         addr:= addr + 1;
      end loop;
   end loop;
end fill_memory;

begin
cpu_cycles: process
      variable reg_mem, hw_interrupt, BR: Boolean;
      variable sign_ext: std_logic_vector(4 downto 0);
   begin
      reg_mem:= (mode = imm) or (mode = dir) or (mode = ext) or
               (mode = ix) or (mode = ix1) or (mode = ix2);
      hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');
      wait until rising_edge(CLK);
      if (rst_b = '0') then ST <= reset; fill_memory(mem);
      else
      case ST is
         when reset => SP <= "111111";
            if (rst_b =  '1') then ST <= cycle8; end if;
         when fetch =>
            if reg_mem then ALU_OP(Md, A, X, N, Z, C); end if;
            -- завершение предыдущей операции
            if hw_interrupt then ST <= push1;
               else Opcode <= mem(CONV_INTEGER(PC));
                    PC <= PC+1;   --считывание команды
                    ST <= addrl; end if;
         when addrl =>
            case mode is
            when inha =>
                 ALU1(A, N, Z, C); -- операция над регистром A
                 ST <= fetch;
            when inhx =>
                 ALU1(X, N, Z, C); -- операция над регистром X
```

```
            ST <= fetch;
         when imm => Md <= mem(CONV_INTEGER(PC));
            -- получение непосредственных данных
            PC <= PC+1; ST <= fetch;
         when inh1 =>
            if OP = SWI then ST <= push1;
              elsif OP = RTS then ST<= pop2; SP <= SP+1;
              elsif OP = RTI then ST <= pop5; SP <= SP+1;
            end if;
         when inh2 =>
            case OP is
              when TAX => X <= A;
              when CLC => C <= '0';
              when SEC => C <= '1';
              when CLI => I <= '0';
              when SEI => I <= '1';
              when RSP => SP <= "111111";
              when TXA => A <= X;
              when others => assert (false)
                report "illegal opcode, mode = inh2";
            end case;
            ST <= fetch;
         when dir =>
           if OP = JMP then PC <= zero&mem(CONV_INTEGER(PC));
                            ST <= fetch;
           else MAR <= zero&mem(CONV_INTEGER(PC)); PC <=PC+1;
              -- получение прямого адреса
              if (OP=JSR) then ST <= push1;
              else ST <= data; end if;
           end if;
         when dirM => MAR <= zero&mem(CONV_INTEGER(PC));
            PC <= PC+1; ST <= data;
         when ix =>
            if OP = JMP then PC <= zero&X; ST <= fetch;
            else MAR <= zero&X;
              if (OP=JSR) then ST <= push1;
                else ST <= data; end if;
            end if;
         when ixm => MAR <= zero&X; ST <= data;
         when ext | ix2 => MARH<=mem(CONV_INTEGER(PC))(4 downto 0);
              -- получение старшего байта
            PC <= PC+1; ST <= addr2;
         when ix1 | ix1m => MAR <= zero&mem(CONV_INTEGER(PC));
              --получение смещения
            PC <= PC+1; ST <= addX;
         when rel => Md <= mem(CONV_INTEGER(PC));
              --получение смещения
            PC <= PC+1; ST <= testBR;
         when others => ST <= fetch;
            assert(false) report "address mode not implemented";
      end case;
   when addr2 =>
      if (mode = ix2) then MARL <= mem(CONV_INTEGER(PC));
        PC <= PC+1; -- получение младшего байта
        ST <= addX;
      elsif OP=JMP then PC <= MARH&mem(CONV_INTEGER(PC));
                        ST <= fetch;
      else MARL <= mem(CONV_INTEGER(PC)); PC <= PC+1;
        -- получение младшего байта
        if OP=JSR then ST <= push1; else ST <= data; end if;
      end if;
   when addX => if OP=JMP then PC <= MAR + (zero&X);
                               ST <= fetch;
      else MAR <= MAR + (zero&X);
        if OP=JSR then ST <= push1; else ST <= data; end if;
      end if;
   when data =>
```

```
      if OP = STA then mem(CONV_INTEGER(MAR)) <= A; N <= A(7);
        if (A = "00000000") then Z<='1'; else Z<='0'; end if;
      elsif OP = STX then mem(CONV_INTEGER(MAR))<= X; N<=X(7);
        if X = "00000000" then Z <= '1'; else Z <='0'; end if;
      else Md <= mem(CONV_INTEGER(MAR)) ;
      end if;
      if ((mode = dirM) or (mode = ixm) or (mode = ix1m)) then
        ST <= rd_mod_wr; else ST <= fetch; end if;
   when rd_mod_wr => ALU1(Md, N, Z, C); ST <= writeback;
   when writeback => mem(CONV_INTEGER(MAR)) <= Md; ST <= fetch;
   when testBR =>
      case OP is
        when BRA => BR := TRUE;
        when BRN => BR := FALSE;
        when BHI => BR := (C or Z) = '0';
        when BLS => BR := (C or Z) = '1';
        when BCC => BR := C = '0';
        when BCS => BR := C = '1';
        when BNE => BR := Z = '0';
        when BEQ => BR := Z = '1';
        when BPL => BR := N = '0';
        when BMI => BR := N = '1';
        when BMC => BR := I = '0';
        when BMS => BR := I = '1';
        when others => assert(false)
                       report "illegal branch instruction";
      end case;
      if Md(7) = '1' then sign_ext:= "11111";
        else sign_ext:= zero; end if;
      if BR then PC <= PC + (sign_ext&Md); end if;
      ST <= fetch;
   when push1 =>
      mem(CONV_INTEGER("0000011"&SP)) <= PCL;
        -- занести в стек LO байт
      SP <= SP - 1; ST <= push2;
   when push2 =>
      mem(CONV_INTEGER("0000011"&SP)) <="000"&PCH;
        -- занести в стек HI байт
      SP <= SP - 1;
      if (hw_interrupt or OP = SWI) then ST <= push3;
        else PC <= MAR; ST <= fetch; end if; -- JSR
   when push3 =>
      mem(CONV_INTEGER("0000011"&SP)) <= X; -- занести в стек X
      SP <= SP - 1; ST <= push4;
   when push4 =>
      mem(CONV_INTEGER("0000011"&SP)) <= A; -- занести в стек A
      SP <= SP - 1; ST <= push5;
   when push5 =>
      mem(CONV_INTEGER("0000011"&SP))<= "0000"&CCR;
      -- занести в стек CCR
      SP <= SP - 1; ST <= cycle8;
   when cycle8 => I <= '1'; ST <= cycle9;
      if OP = SWI then
        MAR <= "1111111111100";
        -- получение адреса вектора прерывания
      elsif IRQ = '1' then MAR <= "1111111111010";
      elsif SCint = '1' then MAR <= "1111111110110";
      else MAR <= "1111111111111"; -- сброс адресного вектора
      end if;
   when cycle9 => PCH <= mem(CONV_INTEGER(MAR))(4 downto 0);
        -- получение старшего байта
      MAR <= MAR + 1; ST <= cycle10;
   when cycle10 => PCL <= mem(CONV_INTEGER(MAR)); ST <= fetch;
      --получение младшего байта
   when pop5 => CCR<=mem(CONV_INTEGER("0000011"&SP))(3 downto 0);
      -- извлечение из стека CCR
      SP <= SP + 1; ST <= pop4;
```

```
      when pop4 => A <= mem(CONV_INTEGER("0000011"&SP));
         -- извлечение из стека A
         SP <= SP + 1; ST <= pop3;
      when pop3 => X <= mem(CONV_INTEGER("0000011"&SP));
         -- извлечение из стека X
         SP <= SP + 1; ST <= pop2;
      when pop2 => PCH<=mem(CONV_INTEGER("0000011"&SP))(4 downto 0);
         -- извлечение из стека HI байта
         SP <= SP + 1; ST <= pop1;
      when pop1 => PCL <= mem(CONV_INTEGER("0000011"&SP));
         -- извлечение из стека LO байта
         ST <= fetch;
      when others => null;
   end case;
   end if; -- if (rst_b = '1')
   end process;
end behv;
```

15.5. Синтезируемая VHDL-модель M6805 CPU

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity cpu6805 is
   port(clk, rst_b, IRQ, SCint: in std_logic;
      dbus: inout std_logic_vector(7 downto 0);
      abus: out std_logic_vector(12 downto 0);
      wr: out std_logic);
end cpu6805;

architecture cpul of cpu6805 is
-- определение регистров
signal Opcode : bit_vector(7 downto 0) := (others => '0');
signal A, X, Md : std_logic_vector(7 downto 0) := (others => '0');
alias mode : bit_vector(3 downto 0) is Opcode(7 downto 4);
signal CCR: std_logic_vector(3 downto 0); -- CCR = I N Z C
alias I: std_logic is CCR(3); alias N: std_logic is CCR(2);
alias Z : std_logic is CCR(1); alias C : std_logic is CCR(0);
signal PC, MAR, xadd, xadd1, xadd2: std_logic_vector (12 downto 0);
signal SP: std_logic_vector(5 downto 0);
alias PCH: std_logic_vector(4 downto 0) is PC(12 downto 8);
alias PCL: std_logic_vector(7 downto 0) is PC(7 downto 0);
alias MARH: std_logic_vector(4 downto 0) is MAR(12 downto 8);
alias MARL: std_logic_vector(7 downto 0) is MAR(7 downto 0);
signal va : std_logic_vector(2 downto 0);

type state_type is (reset, fetch, addr1, addr2, addX, data,
rd_mod_wr, writeback, testBR, push1, push2, push3, push4, push5,
cycle8, cycle9, cycle10, pop5, pop4, pop3, pop2, pop1);
signal ST, nST: state_type;
type decode_type is array(0 to 15) of bit_vector(15 downto 0);
signal opd: bit_vector(15 downto 0);
constant decode: decode_type:=
(X"0001", X"0002", X"0004", X"0008", X"0010", X"0020", X"0040",
X"0080", X"0100", X"0200", X"0400", X"0800", X"1000", X"2000",
X"4000", X"8000");
alias SUB: bit is opd(0); alias CMP: bit is opd(1);
alias SUBC: bit is opd(2); alias CPX: bit is opd(3);
alias ANDa: bit is opd(4); alias BITa: bit is opd(5);
alias LDA: bit is opd(6); alias STA: bit is opd(7);
alias EOR: bit is opd(8); alias ADC: bit is opd(9);
alias ORA: bit is opd(10); alias ADD: bit is opd(11);
alias LDX: bit is opd(14); alias STX: bit is opd(15);
```

```
alias BRA: bit is opd(0); alias BRN: bit is opd(1);
alias BHI: bit is opd(2); alias BLS: bit is opd(3);
alias BCC: bit is opd(4); alias BCS: bit is opd(5) ;
alias BNE: bit is opd(6); alias BEQ: bit is opd(7);
alias BPL: bit is opd(10); alias BMI: bit is opd(11);
alias BMC: bit is opd(12); alias BMS: bit is opd(13);
alias BIL: bit is opd(14); alias BIH: bit is opd(15);
alias TAX: bit is opd(7); alias CLC: bit is opd(8);
alias SEC: bit is opd(9); alias CLI: bit is opd(10);
alias SEI: bit is opd(11); alias RSP: bit is opd(12);
alias NOP: bit is opd(13); alias TXA: bit is opd(15);
alias NEG: bit is opd(0); alias COM: bit is opd(3);
alias LSR: bit is opd(4); alias ROR1: bit is opd (6);
alias ASR: bit is opd(7); alias LSL: bit is opd (8);
alias ROL1: bit is opd(9); alias DEC: bit is opd(10);
alias INC: bit is opd(12); alias TST: bit is opd (13);
alias CLR: bit is opd(15); alias RTI: bit is opd(0);
alias RTS: bit is opd(1); alias SWI: bit is opd(3);
alias JMP: bit is opd(12); alias JSR: bit is opd(13);

--определяется адресный режим по четырем старшим битам кода команды
subtype ot is bit_vector (3 downto 0);
constant REL: ot:="0010"; constant DIRM: ot:="0011";
constant INHA: ot:="0100"; constant INHX: ot:="0101";
constant IX1M: ot:="0110"; constant IXM: ot:="0111";
constant INH1: ot:="1000"; constant INH2: ot:="1001";
constant IMM: ot:="1010"; constant DIR: ot:="1011";
constant EXT: ot:="1100"; constant IX2: ot:="1101";
constant IX1: ot:="1110"; constant IX: ot:="1111";
signal shiftout,op1_com,op2_com,op1,op2:std_logic_vector(7 downto 0);
signal alu9: std_logic_vector(8 downto 0);
  alias Cout : std_logic is alu9(8);
signal shiftin, Cin, newC : std_logic;
signal com2, and2, or2, xor2, rsh, lsh, clear: std_logic;
signal  incPC, xadd2PC, db2PCH, MARH2PCH, MARL2PCL, clrPCH,
        db2PCL, X2PCL, db2opcode, incSP, decSP, setSP, setSP1,
        setI, setI1,clrI, xadd2MAR, va2MAR, db2MARH, clrMARH,
        db2MARL, incMAR, X2MARL, ALU2A, db2A, ALU2X, db2X, db2Md,
        db2CCR, updateNZ, updateC, ALU2Md, Md2db, A2db, X2db,
        PCH2db, PCL2db, CCR2db, PC2ab, MAR2ab, SP2ab,
        com1, selA, selx, selMd1, selMd2, update: std_logic;
signal setC, drC : std_logic;
constant hi_Z: std_logic_vector(7 downto 0):= (others => 'Z');
constant hi_Z13: std_logic_vector(12 downto 0):= (others => 'Z');
constant zero: std_logic_vector(4 downto 0):= "00000";

begin
-- управление шиной данных с помощью тристабильных буферов
dbus <= A when A2db= '1' else hi_Z;
dbus <= X when X2db= '1' else hi_Z;
dbus <= Md when Md2db= '1' else hi_Z;
dbus <= "000"&PCH when PCH2db='1' else hi_Z;
dbus <= PCL when PCL2db= '1' else hi_Z;
dbus <= "1110"&CCR when CCR2db= '1' else hi_Z;

-- управление адресной шиной
abus <= MAR when MAR2ab= '1'
     else "0000011"&SP when SP2ab= '1'
     else PC;

-- определение операндов – входов АЛУ
op1 <= A when selA = '1'     -- MUX for op1
     else X when selx = '1'
     else Md when selMd1 = '1'
     else "00000000";
op2 <= Md when selMd2 = '1'     -- MUX for op2
     else "00000000";
```

```
-- операции АЛУ и сдвига
op1_com <= not op1 when com1='1' else op1;
          -- формирование обратного кода
op2_com <= not op2 when com2='1' else op2;
-- логические операции или сложения
alu9 <= '0'&(op1_com and op2_com) when and2= '1'
     else '0'&(op1_com or op2_com) when or2= '1'
     else '0'&(op1_com xor op2_com) when xor2= '1'
     else ('0'&op1_com) + ('0'&op2_com) + Cin;
shiftout <= shiftin&alu9(7 downto 1) when rsh= '1'-- сдвиг
     else alu9(6 downto 0)&shiftin when lsh= '1'
     else "00000000" when clear = '1'
     else alu9(7 downto 0);
newC <= alu9(0) when rsh= '1' -- перенос
     else alu9(7) when lsh= '1'
     else '1' when setC = '1'
     else '0' when drC = '1'
     else Cout xor (com1 or com2);
xadd <= xadd1 + xadd2; -- сложение адреса
-- дешифрация команды
opd <= decode(CONV_INTEGER(TO_STDLOGICVECTOR(Opcode(3 downto
0))));

ALU_control: process (opd, mode, C, alu9)
-- процесс генерирует управляющие сигналы для операций АЛУ
  variable reg_mem, rd_md_wr: boolean := FALSE;
begin
  Cin <= '0'; shiftin<= '0'; and2 <= '0'; or2 <= '0'; xor2 <= '0';
  rsh<= '0'; lsh<= '0'; com1 <= '0'; com2 <= '0'; updateNZ<= '1';
  updateC<= '1' ; clrI <= '0'; setC <= '0'; drC <= '0'; clear <= '0';
  setI1<='0'; setSP1<='0'; ALU2A<='1'; ALU2X<='0'; ALU2Md<='0';
  selA <= '1'; selX <= '0'; selMd1 <= '0'; selMd2 <= '1';

  reg_mem:=(mode = imm) or (mode = dir) or (mode = ext) or
          (mode = ix) or (mode = ix1) or (mode = ix2);
  rd_md_wr:= (mode = dirM) or (mode = inha) or (mode = inhx) or
          (mode = ix1m) or (mode = ixm);

  if reg_mem then
          -- управляющие сигналы для операций регистр-память
    if ADD= '1' then null; end if; -- используемый по умолчанию
    if ADC= '1' then Cin<=C; end if;
    if SUB= '1' then com2<= '1'; Cin<= '1'; end if;
    if SUBC= '1' then com2<= '1'; Cin<= not C; end if;
    if CMP= '1' then com2 <= '1'; Cin <= '1'; ALU2A<= '0'; end if;
    if CPX = '1' then selX <= '1'; selA <= '0'; com2 <= '1';
                      Cin <= '1'; ALU2A<= '0'; end if;
    if ANDa= '1' then and2<= '1'; updateC<= '0'; end if;
    if BITa='1' then and2<='1'; ALU2A<='0'; updateC<='0'; end if;
    if ORA= '1' then or2 <= '1'; updateC<= '0'; end if;
    if EOR= '1' then xor2<= '1'; updateC<= '0'; end if;
    if LDA = '1' then updateC<= '0'; selA <= '0'; end if;
    if LDX = '1' then selA <= '0' ;
                  updateC<= '0'; ALU2A<= '0'; ALU2x<='1'; end if;
    if ((STA or STX) = '1') then ALU2A <= '0';
       updateC <= '0'; end if; -- только для обновления флага NZ
  end if;

  if rd_md_wr then
    -- управляющие сигналы для операций rd_md_wr и сдвига
    selMd2 <= '0'; -- op2 всегда равен 0 для rd_md_wr
    if (mode /= inha) then
       ALU2A <= '0'; selA <= '0'; -- по умолчанию выключено
       if mode = inhx then ALU2X <= '1'; selX <= '1'; -- op1 = X
          else ALU2Md <= '1'; selMd1 <= '1'; end if; -- op1 = Md
    end if;
```

```
      if NEG= '1' then Cin<= '1'; com1 <= '1'; end if;
      if COM= '1' then com1 <= '1'; end if;
      if DEC = '1' then updateC<= '0';
                        com2 <='1'; end if; -- op2_com = -1
      if LSR = '1' then rsh<= '1'; end if;
      if ROR1 = '1' then rsh<= '1'; shiftin<=C; end if;
      if ASR = '1' then rsh<= '1'; shiftin<=alu9(7); end if;
      if LSL = '1' then lsh<= '1'; end if;
      if ROL1 = '1' then lsh<= '1'; shiftin <= C; end if;
      if INC = '1' then Cin<= '1'; updateC<= '0'; end if;
      if CLR = '1' then and2 <= '1'; updateC<= '0'; end if;
                    -- выполнение операции and с 0 для обнуления
      if TST = '1' then updateC<= '0'; end if;
   end if; -- rd_md_wr

   if (mode = inh2) then
      selMd2 <= '0'; updateC <= '0'; updateNZ <= '0'; -- op2=0
-- значение в регистре A всегда обновляется, поскольку ALU2A = '1'
      if TAX= '1' then ALU2X<= '1'; end if;
      if TXA= '1' then selX <= '1'; selA <= '0'; end if;
      if CLC= '1' then drC <= '1'; updateC <= '1'; end if;
      if SEC= '1' then setC <= '1'; updateC <= '1'; end if;
      if CLI= '1' then clrI<= '1'; end if;
      if SEI= '1' then setI1<= '1'; end if;
      if RSP= '1' then setSP1<= '1'; end if;
end if;
end process;

CPU_control: process (ST,rst_b,opd, mode, IRQ,SCint,CCR,
MAR,X,PC,Md)
      -- управляющий автомат процессора
   variable reg_mem, hw_interrupt, BR : boolean;
begin
   nST <= reset; BR:= FALSE; wr <= '0'; update <= '0';
   xadd1 <= (others => '0'); xadd2 <=(others => '0'); va <= "000";
   db2A<='0'; db2X<='0'; db2Md<='0'; db2CCR<='0'; db2opcode<='0';
   incPC <= '0'; xadd2PC <= '0'; db2PCH <= '0'; MARH2PCH <= '0';
      MARL2PCL <= '0';
   clrPCH <= '0'; db2PCL <= '0'; X2PCL <= '0'; xadd2MAR <= '0' ;
      va2MAR <= '0';
   db2MARH <= '0'; clrMARH <= '0'; db2MARL <= '0'; X2MARL <= '0' ;
      incMAR <= '0';
   A2db<='0'; X2db <='0'; CCR2db <='0'; PCH2db <='0'; PCL2db<='0';
   Md2db<='0'; MAR2ab <='0'; PC2ab <='0'; SP2ab <='0'; incSP<='0';
   decSP <= '0'; setI <= '0'; setSP <= '0';
   reg_mem:= (mode=imm) or (mode=dir) or (mode=ext) or (mode=ix)
           or (mode=ix1) or (mode=ix2);
   hw_interrupt := (I = '0') and (IRQ = '1' or SCint = '1');

   if (rst_b = '0') then nST <= reset;
   else
      case ST is
         when reset => setSP <= '1' ;
            if (rst_b = '1') then nST <= cycle8; end if;
         when fetch =>
            if (reg_mem and JMP = '0' and JSR = '0')
            then update <= '1'; end if; -- обновление регистров,
                                    если операция не JMP или JSR
            if hw_interrupt then nST <= push1;
                 -- чтение кода следующей команды
               else PC2ab<='1'; db2opcode<='1'; incPC<='1';
                  nST <= addr1; end if;
         when addr1 =>
            case mode is
               when inha | inhx => update <= '1'; nST<= fetch;
               when imm => PC2ab<= '1'; db2Md<= '1'; incPC<= '1';
                  nST <= fetch;
```

```
      when inh1 =>
         if SWI = '1' then nST <= push1;
         elsif RTS = '1' then nST <= pop2; incSP<= '1';
         elsif RTI = '1' then nST <= pop5; incSP<= '1';
         end if;
      when inh2 => update <= '1'; nST <= fetch;
      when dir => PC2ab<= '1';
         if JMP= '1' then db2PCL<= '1' ; clrPCH<= '1';
                          nST<=fetch;
         else db2MARL<= '1'; clrMARH<= '1'; incPC <= '1';
            if JSR= '1' then nST<=push1;
                     else nST<=data; end if;
         end if;
      when dirM => PC2ab<='1'; db2MARL<='1'; clrMARH<='1';
         incPC<= '1'; nST<=data;
      when ix =>
            if JMP= '1' then X2PCL<= '1'; clrPCH<= '1';
                    nST<=fetch;
            else X2MARL<= '1'; clrMARH<= '1';
               if JSR= '1' then nST<=push1;
                  else nST<=data; end if;
            end if;
      when ixm => X2MARL<= '1'; clrMARH<= '1'; nST<=data;
      when ext | ix2 => PC2ab<= '1'; db2MARH<= '1';
            incPC<= '1'; nST<=addr2;
      when ix1 | ix1m => PC2ab<= '1'; db2MARL<= '1';
            clrMARH<= '1'; incPC<= '1'; nST<=addX;
      when rel => PC2ab<='1'; db2Md<= '1';
            incPC<= '1'; nST<=testBR;
      when others => null;
   end case;
when addr2 => PC2ab<= '1';
   if (mode = ix2) then db2MARL<= '1';incPC<= '1';
           nST <= addX; -- все[ix2]
   elsif (JMP = '1') then db2PCL<= '1'; MARH2PCH<= '1';
           nST <= fetch;
      -- JMP [ext]
   else db2MARL<= '1';incPC<= '1'; -- JSR/остальные [ext]
      if (JSR = '1') then nST<=push1;
           else nST <= data; end if;
   end if;
when addX => xadd1<=MAR; xadd2<=zero&X;
   if JMP= '1' then xadd2PC<= '1'; nST <= fetch;
else xadd2MAR<= '1';
   if JSR= '1' then nST<= push1; else nST<=data; end if;
end if;
when data => MAR2ab<= '1';      -- nST <:= fetch;
   if STA = '1' then wr<= '1'; A2db<= '1';
         update <= '1';          -- Обновление флага NZ
   elsif STX = '1' then wr<= '1'; X2db<= '1';
         update <= '1';          -- Обновление флага NZ
   else db2Md <= '1';
   end if;                       -- чтение с шины данных
   if ((mode = dirM) or (mode = ixm) or (mode = ix1m))
      then nST <= rd_mod_wr; else nST <= fetch; end if;
when rd_mod_wr => update <= '1' ;
   nST <= writeback;             -- обновление Md
when writeback => wr<= '1'; MAR2ab<= '1';
   Md2db<= '1';          -- запись содержимого Md в память
   nST <= fetch;
when testBR =>
   if BRA = '1' then BR:= TRUE; end if;
   if BRN = '1' then BR:= FALSE; end if;
   if BHI = '1' then BR:= (C or Z) = '0'; end if;
   if BLS = '1' then BR:= (C or Z) = '1'; end if;
   if BCC = '1' then BR:= C = '0'; end if;
   if BCS = '1' then BR:= C = '1'; end if;
```

```
            if BNE = '1' then BR:= Z = '0'; end if;
            if BEQ = '1' then BR:= Z = '1'; end if;
            if BPL = '1' then BR:= N = '0'; end if;
            if BMI = '1' then BR:= N = '1'; end if;
            if BMC = '1' then BR:= I = '0'; end if;
            if BMS = '1' then BR:= I = '1'; end if;
                 -- установка входов сумматора адреса
            xadd1<=PC; xadd2<=Md(7)&Md(7)&Md(7)&Md(7)&Md(7)&Md;
                 -- расширение по знаку Md
            if BR then xadd2PC<= '1'; end if; -- PC <= xadd1+xadd2;
                 nST <= fetch;
          when push1 =>wr<= '1'; SP2ab<= '1'; PCL2db<= '1';
            decSP <= '1'; nST <= push2;
          when push2 => wr<='1'; SP2ab<='1'; PCH2db<='1'; decSP<='1';
            if (hw_interrupt or SWI = '1') then nST <= push3;
            else MARH2PCH <= '1'; MARL2PCL <= '1'; nST <= fetch;
            end if;
               -- PC <= MAR (execute JSR)
          when push3 => wr<= '1'; SP2ab<= '1'; X2db<= '1' ;
            decSP <= '1'; nST <= push4;
          when push4 =>wr<= '1'; SP2ab<= '1' ; A2db<= '1';
            decSP <= '1'; nST <= push5;
          when push5 => wr<= '1'; SP2ab<= '1'; CCR2db<= '1';
            decSP <= '1'; nST <= cycle8;
          when cycle8 =>
                 -- 3 адреса вектора прерывания
            if SWI = '1' then va <= "110";
            elsif IRQ = '1' then va <= "101";
            elsif SCint = '1' then va <= "011";
                 -- значение по умолчанию для сброса вектора
            else va <= "111";
            end if;
            va2MAR <= '1'; setI <= '1'; nST <= cycle9;
          when cycle9 => MAR2ab <= '1';
            db2PCH <= '1';             -- get вектор прерываний
            incMAR <= '1'; nST <= cycle10;
          when cycle10 => MAR2ab <= '1'; db2PCL <= '1';
            nST <= fetch;
          when pop5 => SP2ab<= '1';
            db2CCR<= '1';              -- восстановление регистров
            incSP<= '1'; nST <=pop4;
          when pop4 => SP2ab<= '1'; db2A<= '1' ;
            incSP<= '1'; nST <=pop3;
          when pop3 => SP2ab<= '1'; db2x<= '1';
            incSP<= '1'; nST <=pop2;
          when pop2 => SP2ab<= '1'; db2PCH<= '1';
            incSP<= '1'; nST <=pop1;
          when pop1 => SP2ab<= '1'; db2PCL<= '1';
            nST <=fetch ;
        end case;
      end if;                          -- если rst_b = '0'
    end process;

    update_reg: process
    begin
    wait until CLK 'event and CLK= '1';
      ST <= nST;
      if incPC = '1' then PC <= PC + 1; end if;
      if xadd2PC = '1' then PC <= xadd; end if;
      if db2PCH = '1' then PCH <= dbus(4 downto 0); end if;
      if MARH2PCH = '1' then PCH <= MARH; end if;
      if MARL2PCL = '1' then PCL <= MARL; end if;
      if clrPCH = '1' then PCH <= "00000"; end if;
      if db2PCL = '1' then PCL <= dbus; end if;
      if X2PCL = '1' then PCL <= X; end if;
      if db2opcode= '1' then Opcode <= TO_BITVECTOR(dbus); end if;
      if incSP = '1' then SP <= SP+1; end if;
      if decSP = '1' then SP <= SP - 1; end if;
```

```
    if (setSP = '1' or setSP1 = '1') then SP <= "111111"; end if;
    if xadd2MAR = '1' then MAR <= xadd; end if;
    if va2MAR = '1' then MAR <= "111111111"&va&'0'; end if;
    if db2MARH = '1' then MARH <= dbus(4 downto 0); end if;
    if clrMARH = '1' then MARH <= "00000"; end if;
    if db2MARL = '1' then MARL <= dbus; end if;
    if X2MARL = '1' then MARL <= X; end if;
    if incMAR = '1' then MAR(0) <= '1'; end if;
  -- в этот момент бит MAR(0)всегда равен '0', следовательно,
  -- нет необходимости в устройстве увеличения на 1

    if db2A= '1' then A <= dbus; end if;
    if db2X= '1' then X<=dbus; end if;
    if db2Md = '1' then Md <=dbus; end if;
    if db2CCR = '1' then CCR<= dbus(3 downto 0); end if;
    if (update = '1') then
       if (ALU2A = '1') then A <= Shiftout; end if;
       if (ALU2X = '1') then X <= Shiftout; end if;
       if (ALU2Md = '1') then Md <= Shiftout; end if;
       if updateNZ= '1' then N <= Shiftout(7);
          if Shiftout = "00000000" then Z <= '1';
             else Z <= '0';  end if;
       end if;
       if updateC= '1' then C <= newC; end if;
    end if;
    if (setI = '1' or setI1= '1') then I <= '1'; end if;
    if clrI = '1' then I <= '0'; end if;
  end process;

  end cpu1;
```

Выводы. Созданы VHDL-модели интерфейса UART и микроконтроллера. Процесс проектирования микроконтроллера включает следующие шаги:

1. Определение структуры регистров, множества команд и адресных режимов.
2. Создание таблицы, описывающей передачу данных между регистрами.
3. Проектирование управляющего автомата.
4. Написание поведенческой VHDL-модели, основанное на результатах выполнения пунктов 1, 2, 3. Тестирование поведения устройства, чтобы гарантировать его соответствие техническому заданию.
5. Разработка структурных схем процессора и его основных компонентов, определение управляющих сигналов.
6. Модификация VHDL-кода, исходя из результатов, полученных при выполнении пункта 5. Тестирование созданной модели.
7. Создание схемы процессора с помощью программы автоматического синтеза .
8. Программирование полученным кодом реальной микросхемы и верификация разработанного устройства.

Несмотря на то, что проект спроектирован для реализации на PLD, его можно исполнить в виде других электронных устройств, например ASIC, однако для этого может потребоваться внесение небольших изменений в VHDL-код.

15.6. Задачи

15.6.1. Внести необходимые изменения в VHDL-код приемника UART, чтобы в нем можно было использовать 16X-битную синхронизацию вместо 8X-битной. Использование более высокой частоты квантования позволяет улучшить защищенность от шумов приемника.

15.6.2. Написать VHDL- TestBench для UART. Протестировать ситуацию возникновения ошибок переполнения, кадрирования, вызывающих фальшстарт шумов, изменение частоты BAUD. Промоделировать VHDL-код.

Написать простейший TestBench для выполнения теста обратной передачи с TxD, подключенного внешне к RxD. Просинтезировать TestBench вместе с UART,

загрузить в целевое устройство и протестировать корректность выполнения аппаратурой основных операций.

15.6.3. Выполнить необходимые изменения VHDL-кода для внесения возможности реализации проверки паритета для UART, описанного ранее. Добавить к регистру SCCR два бита P_1P_2 для выбора режима проверки:

```
P1P2 = 00   8-битные данные, без бита проверки
P1P2 = 01   7-битные данные, 8-й бит – проверка на четность
P1P2 = 10   7-битные данные, 8-й бит – проверка на нечетность
P1P2 = 11   7-битные данные, 8-й бит всегда содержит '0'
```

В зависимости от режима трансмиттер должен генерировать бит четности, нечетности или '0'. Приемник должен проверять бит паритета. Если последний неправильный – устанавливается флаг ошибки PE в регистре SCSR.

15.6.4. Команда BSR подобна JSR, только она использует относительную адресацию. Внести команду BSR в таблицу 15.4 и в VHDL-код из подразд. 15.4 и 15.5.

15.6.5. Команда BSET n (установка бита) имеет два бита и всегда использует прямую адресацию. Она устанавливает бит n ($0 < n < 7$) в определенном месте памяти. Команда BCLR n (сброс бита) подобна предыдущей, за исключением того, что последняя очищает бит n. Внести команды BSET и BCLR в таблицу 15.4 и в VHDL-код из разделов 15.4 и 15.5.

15.6.6. Команда BRSET n (выбор команд, если бит установлен) имеет 3 бита. В байте 2 записан прямой адрес, а в байте 3 – относительный (rel) для выбираемых команд. BRSET считывает данные из памяти и устанавливает C равным биту n. Если этот бит установлен, следующая команда считывается по адресу PC + 3 + rel. Команда BRCLR n (выбор команд, если бит сброшен) подобна предыдущей, за исключением того, что в последней переход выполняется, если бит n сброшен. Внести команды BRCLR и BRSET в таблицу 15.4 и в VHDL-код.

15.6.7. Добавить команду WAIT (ожидание прерывания), которая очищает I, а затем останавливает процессор до возникновения аппаратного прерывания. Внести команду WAIT в таблицу 15.4 и в VHDL-код из подразд. 15.4 и 15.5.

15.6.8. Синтезировать VHDL-код процессора (подразд. 15.5), используя различные параметры оптимизации (по площади или по скорости). Сравнить результаты.

15.6.9. Изменить VHDL-модель процессора (подразд. 15.5), так чтобы при реализации она требовала меньшее число логических элементов. Затем выполнить следующие действия:

а) Записать код, в котором для выбора режима используется простой декодер. Заменить все проверки "mode = ..." на тестирование одного бита с выхода дешифратора. Изменить для этого псевдонимы (aliases) режимов адресации.

б) Явно сгенерировать управляющие сигналы для мультиплексора в адресном модуле (рисунок 15.6). Написать VHDL-код для to infer these MUXes.

в) Получить логические уравнения для каждого бита блока "ADDER and logic operations" (рисунок 15.8). Использовать эти уравнения в VHDL-коде.

15.7.10. Переписать VHDL-код для параллельного порта (рисунок 15.11), используя процесс, вместо параллельных операторов для декодирования и pin interface logic. Сравнить результаты синтеза.

СПИСОК ЛИТЕРАТУРЫ

1. *Соловьев В.В*. Проектирование цифровых схем на основе программируемых логических интегральных схем.– Горячая линия-Телеком.– 2001.–636с.

2. *Charles H. Roth, Jr*. Digital Systems Design UsingVHDL.– PWS Publishing Company,20 Parkl Plaza, Noston, MA 02116 ISBN.– 470p.

3. *Ashenden, Peter J*. The designer's guide to VHDL.– San Francisco, California: Morgan Kaufmann Publishers.– 1996. – 688 c.

4. *Самофалов К.Г., Корнейчук В.И., Тарасенко В.П*. Цифровые электронные вычислительные машины.– Киев: Выща шк., 1983.– 455с.

5. *Active-HDL* Series Book #1- #4: VHDL Reference Guide.– ALDEC.– 1998.– 206 p.

6. *Глушков В.М*. Кибернетика. Вопросы теории и практики. – М.: Наука, 1986. – 488с.

7. *IEEE* Standard VHDL Language Reference Manual.– New York: IEEE Std 1076-1993 Erschienen August 1994, IEEE, Taschenbuch ISBN: 1559373768, 1994.– 186p.

8. *Bhasker, J*. A VHDL Synthesis Primer. – Allentown: Star Galaxy Publishing, 1998. – 296p.

9. *IEEE* Standard Multivalue Logic System for VHDL Model Interoperability.– New York: The Institute of Electrical and Electronics Engineers.– (Std_logic_1164).– ISBN 1-55937-299-0.1993.– 24 p.

10. *IEEE* 1029.1-1991. IEEE Standard for Waveform and Vector Exchange (WAVES) (ANSI).– ISBN 1-55937-195-1.–96 p.

11. *IEEE* Standard VHDL Analog and Mixed-Signal Extensions.– IEEE Std 1076.1-1999.

12. *Майоров С.А., Новиков Г. Н*. Структура электронных вычислительных машин.– Л.: Машиностроение, 1979.– 384 с.

13. *Bergeron, Janick*. Writing testbenches: functional verification of HDL models.– Boston: Kluwer Academic Publishers, 2001.– 354 p.

14. *XILINX Inc*. The Programmable Logic Data Book, 1999. (http://www.xilinx.com).

15. *Справочная* система ПО. Synplify Pro User Guide and Tutorial фирмы Sinplicity.

16. *Справочная* система ПО. Hardware Design Verification Maner компании ALATEK INC.

17. *Хаханов В.И., Хак Х.М. Джахирул, Масуд М.Д. Мехеди.* Модели анализа неисправностей цифровых систем на основе FPGA, CPLD // Технология и конструирование в электронной аппаратуре.–2001.–№ 2.– С.3-11.

18. *Хаханов В.И.* Кубическое моделирование неисправностей и генерация тестов для цифровых систем.– В кн.: Ежегодный отчет ХТУРЭ.– 1999-2000.– С.139-146.

19. *Хаханов В.И., Сысенко И.Ю., Хак Х.М. Джахирул, Масуд М.Д. Мехеди.* Кубическое моделирование неисправностей цифровых проектов на основе FPGA, CPLD // Радиоэлектороника, информатика, управление.– 2001.– № 1.– С.123-129.

20. *Menon P.R., Chappel S.G.* Deductive failt simulation with functional blocks // IEEE Trans. on Computers.– 1978.Vol. C.27, No 8.– P.689-695.

21. *Levendel Y.H., Menon P.R.* Comparison of fault simulation methods – Treatment of unknown signal values // Journal of digital systems.– 1980.Vol. 4.– P.443-459.

22. *Chang H.Y., Chappel S.G., Elmendorf C.H., Smidt L.D.* Comparison of parallel and deductive fault simulation Methods // IEEE Trans. on Computers.– 1974.Vol. C23, No 11.– P.1132-1138.

23. *Abramovici M., Breuer M.A. and Friedman A.D*. Digital System Testing and Testable Design.– Computer Science Press.– 1998.– 652 p.

24. *Хаханов В.И.* Техническая диагностика элементов и узлов персональных компьютеров.–К.: ІЗМН.– 1997.– 308с.

25. *Бондаренко М.Ф., Кривуля Г.Ф., Рябцев В.Г., Фрадков С.А., Хаханов В.И.* Проектирование и диагностика компьютерных систем и сетей.– К.: НМЦ ВО.– 2000.– 306 с.

26. *Active*-VHDL Series. Book #1 - #4.– Reference Guide.– ALDEC Inc.– 1998.– 206 p.

27. К*ондратенко Ю.П., Сидоренко С.А., Подопригора Д.М*. Поведенческий синтез цифровых устройств в среде Active-HDL.– Николаев: МФНаУКМА.– 2002.– 116с.

28. *Active-HDL* User's Guid. Second Edition.– Copyright.– Aldec Inc.– 1999.– 213p.

29. *Дрозд А.В.* Контроль по модулю вычислительных устройств.– Одесса: ОПУ.– 2002.– 144 с.

30. *Закревский А.Д.* Параллельные алгоритмы логического управления.– Минск: ИТК НАНБ.– 1999.– 202с.

31. *Marina Brik*. Investigation and Development ot Test Generation Methods for Control Part of Digital System. Dissertation of the Degree of Doctor of Science in Computer Engineering.– Tallinn: TTU.– 2002.– 160p.

32.*Ubar.R.* Test Synthesis with Alternative Graphs.– IEEE Design&Test of Computers.– Spring.– 1996.– P.48-57.

33. *Хаханов В.И., Сысенко И.Ю., Колесников К.В.* Дедуктивно-параллельный метод моделирования неисправностей на реконфигурируемых моделях цифровых систем// Радиоэлектроника и информатика.– 2002.– №1.– С.98-105.

34. *Хаханов В.И., Колесников К.В., Хаханова А.В*. BDP-метод моделирования неисправностей для синтеза тестов цифровых проектов// Радиоэлектроника и информатика.– 2002.– №2.– С.60-66.

35. 6. *Hahanov V.I., Babich A.V., Hyduke S.M.* Test Generation and Fault Simulation Methods on the Basis of Cubic Algebra for Digital Devices.– Proceedings of the Euromicro Symposium on Digital Systems Design DSD2001.– Warsaw, Poland.– September, 4-6.– 2001.– P. 228-235.

36. *Бибило П.Н.* Синтез логических схем.– М.: Солон-Р.– 2002.–384с.

37. *Karen Parnell, Nick Mehta.* Programmable Logic Design Quick Start Hand Book. – ©Xilinx, January 2002.– 201 p. (http://www.xilinx.com).

38. *Samir Palnitkar*. Verilog HDL. A guide to digital design and synthesis.– SunSoft Press.– 1996.– 396p.

39. *Стешенко В.Б.* EDA.Практика автоматированного проектирования радиоэлектронных устройств.– М.: "Нолидж", 2002. – 768 с.

Содержание

Учебное пособие

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ СИСТЕМ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА VHDL

СЕМЕНЕЦ Валерий Васильевич
ХАХАНОВА Ирина Витальевна
ХАХАНОВ Владимир Иванович

Редактор *О.П. Гужва*
Дизайн-макетування *І.В. Хаханова*
Художник *Е.Б. Лук'янченко*
Комп'ютерний набір та верстка ХНУРЕ
Підписано до друку 27.02.2003. Формат $60\times84^{1}/_{8}$.
Умов. друк. арк. 57,2. Облік.-вид. арк. 55,8. Зам. №371. Тираж 500 прим. Ціна договірна.

ХНУРЕ. 61166, Харків, просп. Леніна, 14.

Надруковано у видавництві ЧП "Степанов В.В."
61168, Харків, вул. акад. Павлова, 311