МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ
УНИВЕРСИТЕТ РАДИОЭЛЕКТРОНИКИ

В.В. СЕМЕНЕЦ, Г.Ф. КРИВУЛЯ, А.Ф. ГОРБАТЮК,
В.И. ХАХАНОВ, А.В. ШИЛЬВЕРСТ

VHDL

ДЛЯ ПРОЕКТИРОВАНИЯ КОМПЬЮТЕРНЫХ УСТРОЙСТВ

ХАРЬКОВ 2002

УДК 681.51
Б81

В.В. Семенец, Г.Ф. Кривуля, А.Ф. Горбатюк,
В.И. Хаханов, А.В. Шильверст

VHDL для проектирования компьютерных устройств

Харьков: Изд-во Харьковского технического университета радиоэлектроники, 2001. - 164с.

ISBN

Книга включает вопросы проектирования компьютерных устройств с использованием языка HDL, сведения о котором любезно предоставила, в плане обмена научно-технической информацией, фирма ALDEC.

Предназначена для студентов специальностей 7.091501 (Компьютерные системы и сети), 7.091502 (Системное программирования), 7.0915503 (Специализированные компьютерные системы) занимающихся современными методами проектирования средств вычислительной техники.

Табл. 6. Ил. 59. Библиогр. 5.

ASIC	- заказная интегральная схема;
CPU	- центральное процессорное устройство;
IEEE	- международный электротехнический стандарт ;
FPGA	- программируемая пользователем вентильная матрица;
HDL	- язык описания устройства на уровне структуры;
MSB	- старший бит;
PCB	- печатные схемные платы;
RTL	- уровень регисторных передач;
ROM (ПЗУ)	- постоянное запоминающее устройство;
VHDL	- язык описания устройства на уровне структуры для быстрых интегральных схем;
АЛУ	- арифметико-логическое устройство;
ОЗУ	- оперативное запоминающее устройство;
ОП	- оперативная память;
ПЛИС	- программируемая логическая интегральная схема;
ПО	- программное обеспечение;
PC	- персональный компьютер;
САПР	- система автоматизационного проектирования;

СОДЕРЖАНИЕ

ВЕДЕНИЕ

Настоящий курс введен в Харьковском техническом университете радиоэлектроники и в Северодонецком технологическом институте в программу обучения студентов специальностей 7.091501 («Компьютерные системы и сети»), 7.091502 («Технология системного программирования»). Основным толчком для этого стали встречи с президентом фирмы ALDEC Inc. мистером Stanley Hyduke и семинары этой фирмы в Киевском политехническом институте.

Цель курса состоит в том, чтобы обучить VHDL и показать, как VHDL используется практически для проектирования электронных цифровых систем, являясь современным инструментальным средством разработки. Курс содержит достаточно большое количество примеров, что позволяет использовать его и как учебник и как справочник.

VHDL используется для проектирования в течение ряда лет, и предлагаемый курс отражает его возможности в настоящее время. Разработку VHDL инициализировало Министерство обороны США в начале 1980-ых потому что военным США был необходим стандартизированный метод, описывающий электронные системы. VHDL был стандартизирован IEEE в 1987 как IEEE 1076-1987. Последний стандарт VHDL-93 датирован 1993; он не содержит больших изменений по сравнению с VHDL-87, хотя в него были включены новые команды VHDL и атрибуты, прежде всего для моделирования. VHDL имеет подобия с языком программного обеспечения АДА. Возможно, это потому что компания Intermetrics, которая была уполномочена Пентагоном, чтобы определить новый язык, имела большой опыт работы с АДОЙ. VHDL стандарт, точно так же как все стандарты IEEE, должен пересматриваться каждые пять лет [1].

Сегодня, множество компаний работает исключительно с VHDL для цифровых систем. Университеты вводят курсы VHDL для своих студентов, и имеется несколько VHDL групп для поддержки новых пользователей. Маленькие и среднего размера компании используют VHDL благодаря возможности работы с ним на PC. VHDL освобождает проектировщика от необходимости использовать структуры фон Неймана и позволяет ему работать с реальным параллелизмом вместо последовательных машин. Это открывает полностью новые возможности для проектировщика.

VHDL задумывался и создавался для проектирования цифровых электронных схем.

Существуют две важных причины для использования VHDL вместо традиционного схематического проекта – более короткое время разработки электронного проекта и более простое сопровождение.

Однако существуют еще важные причины, которые заставляют обратить внимание на VHDL:

- во-первых, это возможность создавать свои архитектуры компьютерных систем;
- во-вторых, это возможность создавать поведенческие модели сложных компьютерных систем, начиная, например, с печатных плат, с установленными на них микросхемами, что открывает реальную возможность проводить функциональное компьютерное моделирование таких систем;
- в третьих, это нетрадиционные варианты использования VHDL - для проектирования и моделирования технических систем некомпьютерного типа.

В течение последних нескольких лет на рынке имелись имитаторы, которые могут «выполнять» VHDL код как средство проверки поведения. VHDL код может быть преобразован автоматически в FPGA (Программируемая пользователем Вентильная матрица) или вентильную (gate) матрицу. *Это преобразование от кода к технологии называется* ***синтезом***. Основная предпосылка - то, что инструмент синтеза должен генерировать аппаратные средства с тем же самым поведением (функционально) как VHDL код, который моделировался первоначально. Такая программа синтеза как ViewLogic – базируются на PC или рабочие станции, в то время как Synopsys и Autologic сориентированы только на рабочие станции.

Авторы выражают признательность президенту ALDEC Inc. мистеру Stanley Hyduke за любезно предоставленные фирмой материалы по VHDL - книги и программное обеспечение. Неоценимую помощь при постановке данного курса оказали д.т.н., проф. Каневский Ю. С. и к.т.н., с.н.с. Пилипчатин Е.Н.

1 ПОЧЕМУ VHDL? ПОЧЕМУ ПЛИС?

1.1 Традиционное проектирование цифровых схем

Традиционные методы проектирования цифровых схем связаны со схемотехникой и представлением проектируемой схемы на «вентильном уровне». Последовательность такого проектирования представлена ниже:

- формулировка решаемой задачи;
- определение входов и выходов;
- создание таблиц истинности;
- получение булевых выражений;
- проекция булевых выражений на виртуальный «вентильный уровень»;
- моделирование проекта на виртуальном «вентильном уровне»;
- получение бинарного файла, задающего функционирование проектируемого устройства и необходимые начальные условия;
- отладка спроектируемого устройства.

Недостатки такого способа общеизвестны. Несмотря на то, что большинство этапов проектирования может быть автоматизировано, на практике, это проектирование осуществляется в большинстве случаев вручную из-за отсутствия программного обеспечения или высокоразвитых рабочих станций, необходимых для реализации этого программного обеспечения. Существующее программное обеспечение для проекти-рования специализированных больших интегральных схем (СБИС) или иначе технология схем ASIC - Application Specific Integrated Circuits (Приложение Специфических Интегральных схем), нашла широчайшее применение и известно по продукции Intel, AMD, TI, Motorola, японских, южнокорейских и тайваньских фирм. Проектирование таких СБИС осуществляется с использованием мощных рабочих станций Sun, Solaris, Ultra Sparc, HP-UX HP 10.1 and 10.2, HP715, языка проектирования Verilog и программ синтеза типа Synopsys, AutoLogic.

1.2 Проектирование цифровых схем с помощью VHDL

С 70-х годов начали использоваться так называемые программируемые логические интегральные схемы (ПЛИС). Постепенно совершенствуясь, такие схемы в настоящее время по оценкам специалистов занимают не менее 40% общего рынка интегральных схем (ИС). Современные программируемые логические ИС типа FPGA содержат более 1 млн. вентилей/кристалл. Ведущими мировыми производителями ПЛИС являются фирмы Xilinx, Altera, Actel, Amtel и др. Применение таких микросхем целесообразно и экономически обосновано в тех случаях, когда их применение ограничивается количеством до нескольких тысяч штук. Конечно,

при потреблении в миллионы штук по-прежнему целесообразно использование технологии ASIC (заказных ИС).

Использование ПЛИС с большой степенью интеграции вызвало необходимость применения и новой технологии проектирования. В первую очередь можно полагать, что такие микросхемы должны востребоваться небольшими фирмами, производящими небольшие партии сложной цифровой аппаратуры. Использование ПЛИС с большой степенью интеграции в принципе позволяет выполнить всю схему в одном корпусе, устраняя необходимость использования сложных многослойных печатных плат, в том числе и их проектирования и тестирования.

Таким средством проектирования и явился язык VHDL. Общая последовательность проектирования с использованием VHDL показана на рис. 1.1.

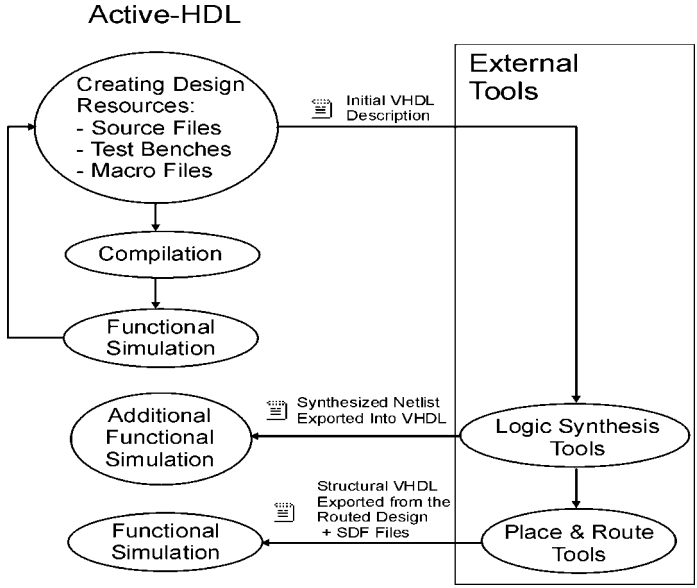


Рисунок 1.1 - Последовательность проектирования с использованием VHDL

Как видно из рисунка, VHDL работает совместно с внешними инструментами синтеза, т.е. **VHDL – это язык проектирования**, а не синтеза. При использовании программ синтеза типа ViewLogic (например, FPGA Express) все проектирование может выполняться на PC компьютерах типа 486 и выше. Это позволяет использовать всю технологию проектирования, тестирования и изготовления спроектированных микросхем не только раз-

личными фирмами разработчиками и изготовителями компьютерной аппаратуры, но и широко использовать эту технологию в учебных заведениях – университетах, институтах и колледжах.

Существуют две важные причины для использования VHDL вместо традиционного схематического проекта – более короткое время разработки электронного проекта и более простое сопровождение. Необходымые для этого инструментальные средства, методология разработки, инструментальные средства синтеза и методология проверки развиваются быстро.

2 Введение в VHDL

2.1 Абстракции языка

VHDL - язык, который может использовать различные уровни абстракции, от описания функций до описания вентилей. Уровни абстракции - средства сокрытия подробностей. Если проектировщик хочет умножить два числа (A = B * C), имеются несколько различных методов описания: использование оператора «*» в VHDL, то есть, a< = b * c; проектирование умножителя на уровне вентилей; проектирование умножителя на уровне размещения. Этим показано, как та же самая функция может быть выполнена в трех различных уровнях абстракции: RTL (Register Transfer Level - уровень межрегистровых пересылок), логическом (уровень вентилей) и уровне размещения. На рис. 2.1 показаны уровни абстракции для аппаратных языков описаний.

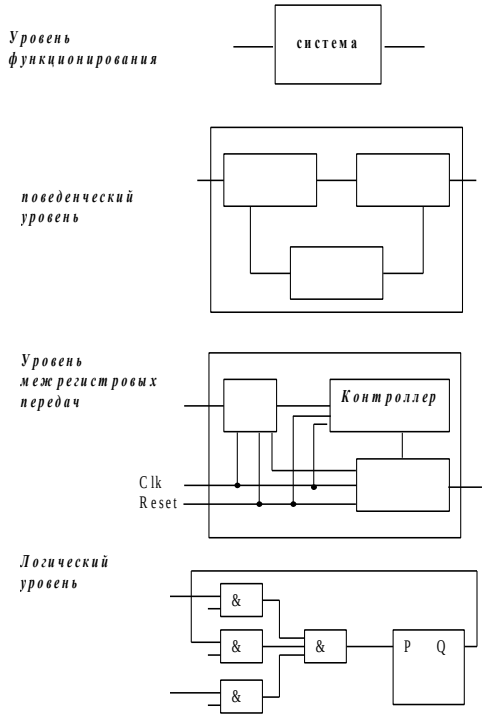


Рисунок 2.1- Урови абстракции VHDL

Алгоритмы могут быть описаны на функциональном уровне, например, алгоритм контроллера может быть описан и имитирован на компьютере.

Архитектура, в поведенческом уровне, не требуется, т.е. реализация регистров, операторов, интерфейс с ОПЕРАТИВНОЙ ПАМЯТЬЮ, и т.д., не определены. Преимущество моделей в этом уровне - то, что модели для моделирования (проверки) могут быть сформированы быстро. Поведенческая модель может быть описана *как функциональные модули и интерфейс между ними*; модули содержат одну или большее количество функций и временных соотношений. В некоторых случаях архитектура может быть определена. Проблемы могут возникнуть, если функции должны совместно использовать некоторые ресурсы (например, умножители), поскольку архитектура будет изменена в уровне RT.

RTL (Уровень межрегистровых пересылок) состоит из языка, который описывает поведение асинхронных и синхронных конечных автоматов, путей данных, операторы (+, *, <, > ..), регистры, и т.д. В VHDL имеется множество различных конструкций языка, которые могут синтезиро-

ваться в этом уровне, например настраиваемые компоненты, реализация, структурный VHDL и перезагрузка. В уровне RT, все регистры определены в VHDL коде.

Логический или уровень вентилей - описание булевой алгеброй или сетью вентилей. Имеются также более низкие уровни типа транзистора (также называемый электрическим) и уровень размещения. Эти уровни не обеспечиваются VHDL. В уровне транзистора имеются модели транзисторов, емкостей и сопротивлений. В уровне размещения, модели сделаны из физического процесса.

Синтез сделан между каждым уровнем, например между RT, и логическим уровнем, VHDL код транслируется в вентили и триггеры, после минимизации. Эти действия создают подробную информацию о технологии с тем, чтобы перейти с RT уровня к уровню вентилей.

Объем информации увеличивается между различными уровнями абстракции. Каждый переход (синтез) генерирует подробную информацию и ее объем и сложность увеличиваются. Чтобы выполнять функцию в ASIC, требуются информации о технологии, монтаже (межсоединениях), информации о вентилях и временных установках, и т.д.

Если проект - в стадии идеи, должен быть выбран поведенческий или функциональный уровень. Если проект имеет жесткие требования эффективности или должен быть очень небольшим, RT уровень - часто самый лучший метод описания, при условии, что используется развитый инструмент синтеза. Трудно достичь лучших результатов в уровне вентилей, хотя это возможно. Уровень Размещения может иногда быть эффективным, особенно для полного заказного ASICs, для достижения максимальной скорости и минимальной площади. Обычно существует обмен между различными требованиями, которые определяют уровень, в котором проект должен быть описан.

Почему различные уровни абстракции используются? Если мы выполняем сравнение с языками программирования для CPU, то мы можем видеть, что они имеют различные уровни абстракции, например микропрограмму, машинный код, ассемблер, C и C++. Если программа разрабатывается с требованием для очень коротких времен выполнения, используется ассемблер. Если, с другой стороны, должна быть разработана сложная программа, она пишется на языках высокого уровня. Это - обычные требования, которые определяют уровень абстракции, в котором информация должна быть описана. То же самое истинно для VHDL. Если требуется короткое время разработки, высокий уровень абстракции должен быть выбран как язык описаний. В практике RT уровень (и части поведенческих) может синтезироваться автоматически к уровню вентилей.

В VHDL возможна поддержка различных уровней описания:

- ASIC (Application Specific Integrated Circuit - Приложение Специфических Интегральных схем) обычно включает вентильную матрицу, стандартную ячейку и полную разработку заказных ИС;
- FPGAs (Программируемые пользователем Вентильные матрицы) включены в ASIC столбец;
- PCB означает Printed Circuit Board – печатные схемные платы. На схемной плате имеется обычно несколько ASICs вместе с микропроцессором и его инфраструктурой.

Это говорит о том, что VHDL может использоваться, чтобы формировать модели из самого высокого уровня абстракции до логического уровня включительно. VHDL идеален для формирования моделей и проектирования ASICs, то есть от функционального уровня до уровня вентилей. Когда это приходит в PCBs, времена моделирование настолько длинны в логическом уровне, что обычно невозможно проверить проект в уровне вентилей. Следовательно, компоненты в уровне PCB должны быть описаны в уровне RT, по крайней мере, чтобы достичь более быстрого времени моделирования. Что касается больших систем, трудно описать сложные функциональные возможности на VHDL.Уровни Абстракции на VHDL языке не должны быть спутаны с иерархиями абстракции проекта.

Имитация (симуляция) моделей - эффективный способ проверки проекта. Компьютерная модель - более или менее подобна действительности. Это меньше похоже на действительность в высоком уровне абстракции (поведенческом) и более похоже в самом низком уровне (размещение).

Компьютерная модель имеет несколько усовершенствований, которые лучше чем физический образец. Установка самого плохого случая для параметров процесса и диапазона температуры обеспечивает лучшую проверку, чем формирование образец. С образцом проверяется только образец.

В статике анализатор или имитатор позволяет проверить/имитировать различные случаи синхронизации:

- **Самый плохой случай:** Самое низкое напряжение (например, 4.5 V), самая высокая температура (например, 125^0 C) и самая медленная характеристика процесса.
- **Типичный случай:** Нормальное напряжение (5.0 V), нормальная температура (например, 25^0 C) и нормальная характеристика процесса.
- **Самый лучший случай:** Самое высокое напряжение (например, 5.5 V), самая низкая температура (например, -55^0 C) и самая быстрая характеристика процесса.

Имитация поведенческих моделей служит для проверки функциональных возможностей. Функциональные возможности и иногда верификация возможны на моделях RTL. Ограничения времени для следующего уровня могут быть установлены для RTL синтеза, но только физическая

модель обеспечивает хороший анализ фактического поведения во времени. Также возможно вычислить / имитировать потребляемую мощность, и т.д., в различных уровнях.

Проблема имитаций, занимающих все большее время с увеличением точности моделей, может означать, что большие проекты невозможно моделировать (потому что моделирование берет слишком длинным) в уровне размещения. Время моделирования также увеличивается, если различные проверки описаны, например тесты установки, хранения и выброса. Если верификация имеет длинные входные строки стимулов для проверки проекта, будет также иметься длительное время моделирования. Проект должен следовательно быть проверен функционально в насколько возможно высоком уровне описания, то есть обычно в RT уровне. Синхронизация (включая установку и требования хранения) анализируются наиболее эффективно и быстро при использовании анализатора синхронизации в уровне вентилей.

Имеются несколько языков, которые используются, чтобы описать электронные проекты. Один популярный язык называется VERILOG. VERILOG используется ниже уровня RT.

Примеры языков, разработанных университетами или центрами исследования: **SLIDE** Structured Language for Interface Description and Evaluation (Parker and Wallace, 1981); **CONLAN** CONsensus LANguage (Piloty *et al.*, 1983); **ISPS** Instruction Set Processor Specification (Barbacci *et al*., 1979); **ADLIB** A Design Language for Indicating Behaviour [Hill *et al.*, 1979]; **HILL** Hierarchical Layout Language (Lengauer and Melhorn, 1984); **OODE** Object Oriented Description Environment for computer hardware (Takeuchi, 1981); **BORIS** Block-ORiented Interacting Simulation system (Decker and Maierhofer, 1984); **SLIDE** Structured Language for Interface Description and Evaluation (Структурный Язык для Описания Интерфейса и Оценки) (Parker and Wallace, 1981); **CONLAN** CONsensus LANguage - ЯЗЫК СОГЛАСИЙ (Piloty *et al*., 1983); **ISPS** Instruction Set Processor Specification -Спецификация Системы команд Процессора (Barbacci *et al*., 1979); **ADLIB** A Design Language for Indicating Behaviour - Язык Проектов для Индикации Поведения [Hill *et al.*, 1979]; **HILL** Hierarchical Layout Language Иерархический Язык Размещений (Lengauer and Melhorn, 1984); **OODE** Object Oriented Description Environment for computer hardware - Объектно ориентированная Среда Описания для компьютерных аппаратных средств (Takeuchi, 1981); **BORIS** Block-ORiented Interacting Simulation system (Блочно-ориентированная Взаимодействующая система Моделирования (Decker and Maierhofer, 1984)).

2.2 Иерархия проекта

Сложные проекты нуждаются в механизме уменьшения их сложности для проектировщика. Трудно понять проект с сотнями компонентов (ОПЕРА-

ТИВНАЯ ПАМЯТЬ, конечные автоматы, управление логикой, и т.д.); проще понять тот же самый проект только с несколькими компонентами.

Использование иерархий для обработки сложности - старый метод. Это не означает, что проект становится менее сложным (иногда это становится более сложным), но становится проще понимание для проектировщика.

Имеются несколько механизмов, чтобы уменьшить сложность:

- абстракции языка используют язык, чтобы описать сложные вопросы, без необходимости описывать мелкие подробности. Функции и процедуры - важные части языка, чтобы обработать сложность;
- иерархия проекта использует компоненты, чтобы скрыть подробности - принцип черный ящик.

Термин «черный ящик» означает, что только вводы / выводы компонент являются видимыми в некотором уровне. Проектировщик решает, сколько различных иерархий должно быть в проекте.

И абстракции языка и иерархия проекта необходимы для уменьшения степени детализации. Иерархия может сравниваться с деревом вверх тормашками. Снаружи на ветвях - письменные компоненты листа в последовательных и параллельных операторах VHDL кода. Далее имеются структурные компоненты, состоящие из других компонентов, написанных в структурном VHDL коде.

Проектирование обычно начинается с определения интерфейса (объект) для корня (или вершины) компонента. Затем компонент разделяется на новые подкомпоненты с соединениями. Новая структура может также называться архитектурой. Общее правило – сводится к тому, чтобы интерфейс между компонентами был небольшим и несложным (насколько это возможно).

2.3 Компоненты

Компоненты - центральное понятие в VHDL. Компоненты используются для создания библиотек компонентов, например микропроцессоров, специальных схем пользователя и других стандартных схем. Если был разработан «хороший» компонент, он может быть сохранен в библиотеке компонентов, допуская многократное копирование, то есть, представляет компонент многократного использования (рис.2.2). На языке информатики, это копирование называется созданием образцов компонента, то есть создает компонент в схематическом (или в текстовом файле).

VHDL является *объектно-базированным* (*object-based*) языком, т.е. отделяет VHDL от *объектно-ориентированных* (*object-oriented*) языков то, что язык не имеет наследования. Обобщенные компоненты и реализация типичны для объектно-базированных языков. Настраиваемые компоненты - компоненты, которые могут изменяться перед реализацией, например

настраиваемый компонент, который справляется с различной длительностью сигналов вывода и ввода.

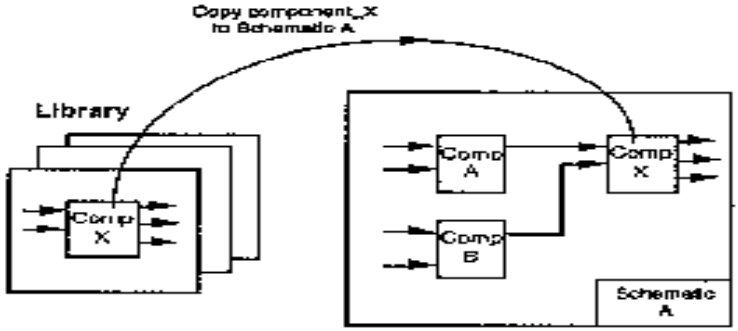


Рисунок 2.2 - Копирование компонента

Внутренняя структура компонента может быть скрыта от проектировщика - принцип черного ящика. В некоторых случаях нет абсолютно никакой потребности знать структуру компонента. Проектировщик обычно только интересуется вводами и выводами, спецификациями функций и времени доступа.

Компонент в библиотеке может состоять из других компонентов. Это возможно для библиотеки, чтобы содержать большие, сложные компоненты: наборы регуляторов, процессоры и схемы связи. Декларация компонента объявляет виртуальный интерфейс проектируемого объекта, который может использоваться в операторе конкретизации компонента.

Упрощенный Синтаксис:

component имя_компонента [**is**]
generic (список_настроек);
port (список_портов);
end component имя_компонента;

Описание:

Компонент представляет пару объекта/архитектуры. Он определяет подсистему, которая может быть конкретизирована в другой архитектуре, ведущей к иерархической спецификации. Конкретизация компонента - подобна включению аппаратного компонента в гнездо в сетевой плате (см. рис. 2.3 в Примере 1). Компонент должен быть объявлен прежде, чем конкретизирован. Декларация определяет виртуальный интерфейс конкретного проектируемого объекта («гнездо») но это непосредственно не указывает проектируемый объект. Привязка проектируемого объекта к данному компоненту может быть задержана и помещена в спецификацию конфигурации или в декларацию конфигурации. Компонент может быть определен

в пакете, проектируемом объекте, архитектуре или декларациях блока. Если компонент объявляется в архитектуре, то он должен быть объявлен перед оператором **begin** архитектуры. В таком случае компонент может использоваться (конкретизироваться) только в архитектуре. Более универсальный подход состоит в том, чтобы объявить компонент в пакете. Такой компонент видим в любой архитектуре, которая использует этот пакет. Родовые и порты компонента - копии настроек и портов объекта, которые представляют компонент.

Пример 1

```
architecture STRUCTURE_2 of EXAMPLE is
 component XOR_4 is
port(A,B: in BIT_VECTOR(0 to 3);
C: out BIT_VECTOR(0 to 3));
end component XOR_4;
signal S1 ,S2 : BIT_VECTOR(0 to 3);
signal S3 : BIT_VECTOR(0 to 3);
begin
X1 : XOR_4 port map(S1 ,S2,S3);
end architecture STRUCTURE_2;
```

Компонент XOR_4 имеет два 4-разрядных входных порта (A и B) и 4-разрядный порт вывода C. Декларация этого компонента расположена в разделе описаний тела архитектуры STRUCTURE_2. Оператор конкретизации компонента назначает метку XI на текущий компонент XOR_4, и это связывает его интерфейс ввода - вывода с сигналами S 1, S2 и S3.

Компоненты - центральная концепция в VHDL, и могут быть полным проектом или маленькой частью системы. Этот раздел исследует части компонента. Компонент сделан из двух основных частей.

Объект (Entity): объявление Порта для вводов и выводов.

Архитектура (Architecture): Структурное или поведенческое описание.

Поведение определено как коллективное имя для функций, операций, поведения и отношений. Поведение может также быть структурным описанием, то есть компонент состоит из других компонентов. Объект может быть расценен как черный ящик с вводами и выводами. Микропроцессор, например, имеет объект, состоящий из данных, адреса, управления и сигналов шины. Поведение должно быть найдено в архитектуре, которая может быть структурным VHDL кодом, то есть это состоит из других компонентов, например, ALU и регистров состояния.

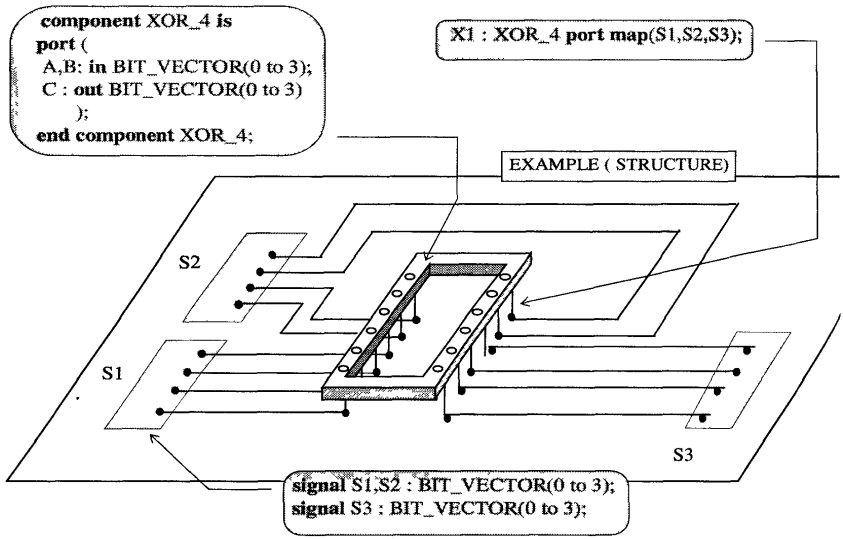


Рисунок 2.3 - Пример декларации компонента и конкретизации

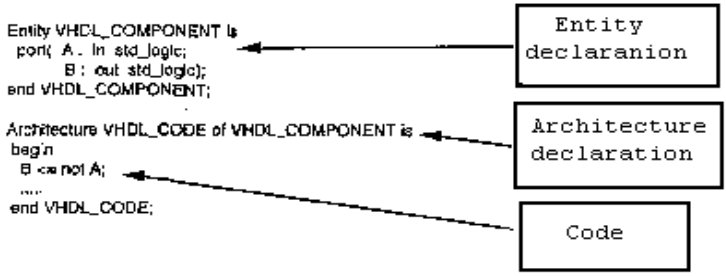


Рисунок 2.4 - Компонент VHDL

Объект называется *vhdl_component* и **architecture** *vhdl_code.*

Два имени определены в объявлении архитектуры: vhdl_component, который описывает, какой объект принадлежит архитектуре, и vhdl_code, который является именем архитектуры.

ОБЪЕКТ

Объявление объекта определяет интерфейс между объектом и средой, в котором он используется. Имя **объекта** - тоже самое как **имя компонента.**

Синтаксис:

Entity <имя_объекта> **is**
Port ([**signal**] <идентификатор>: [<режим>] <тип_идентификатора>;
[**signal**] < идентификатор >: [<режим>] <тип_идентификатора>);
end [<имя_объекта>];

<режим> = in, out, inout, buffer, linkage

in = только для чтения сигнала

out = только для записи сигнала

inout = для записи или чтения сигнала (двунаправленные сигналы)

buffer = для записи и чтения сигнала (но не двунаправленных сигналов)

linkage = используется только в документации.

В следующем примере имеется входной сигнал *a_in* и выходной сигнал *b_out.*

```
Entity vhdl_component is
        port( signal a_in: in  std_logic ;  -- входные сигналы
              signal b_out: out std_logic);  -- выходные сигналы
end vhdl_component;
```

Режим **inout** должен использоваться только в случае двунаправленных сигналов. Если сигнал должен быть перечитан, должен использоваться или режим **buffer**, или внутренний сигнал. Слово **SIGNAL** обычно пропускается в объявлении порта, поскольку не добавляет никакой информации. Режим также быть пропущен. Значит, следующие два примера идентичны.

```
ENTITY EXI IS
        port( signal a,b:  in  std_logic;
              signal c:  out std_logic);
        end exi;

        Entity exi is
        port(a,b:  std_Logic;
c: out std_Logic);
end;
```

Обратите внимание, что VHDL не дифференцирует между символами верхнего регистра и нижнего регистра.

В VHDL-93 стандарте, **end entity** может быть написан вместо **end** только как в VHDL-87 стандарте:

Синтаксис (VHDL-93):

entity<имя_объекта>is

port (**[signal]**< идентификатор>:[<режим>]<тип_идентификатора>;
[signal]< идентификатор>:[< режим>]< тип_идентификатора>);

end [entity] [< имя_объекта >];

Пример (VHDL-93):

```
Entity ex is
        port( a,b: in  std_Logic;
              c:  out std_Logic);
    end entity ex;
```

Архитектура

Формальное Определение:

Архитектура по сути связана с объявлением проектируемого объекта и описывает его внутреннюю организацию и операции. Архитектура используется также для описания режима, потока данных, или структуры проектируемого объекта.

Упрощенный синтаксис:

```
architecture имя_архитектуры of имя_объекта is
        раздел деклараций
        begin
        Сходящиеся инструкции
    end [ architecture ] [имя_архитектуры ];
```

Описание:

Архитектура, присвоенная объекту, описывает внутренние отношения между портами ввода и вывода объекта. Она состоит из двух частей: объявления и параллельных операторов. Первая (декларативная) часть архитектуры может содержать объявления *типов, сигналов, констант,* подпрограмм *(функции* и *процедуры), компонент,* и *групп.*

Параллельные операторы в теле *архитектуры* определяют отношения (связи) между входами и выходами. Эти отношения могут быть определены, используя различные типы инструкций: *параллельное назначение сигнала, обработка инструкции, конкретизация компонента, параллельный вызов процедуры, сгенерировать оператор, параллельный оператор контроля* и *блокировать инструкцию.* Это может быть записано в различных стилях: структурном, потоковом, функциональном или смешанном.

Описание тела **структуры** основано на *конкретизации компонент* и *генерации инструкций*. Это позволяет создавать иерархические проекты, от простых схем до очень сложных компонентов, описывающих полные подсистемы. Подключения среди компонентов создаются через порты. Пример 1 иллюстрирует эту концепцию для BCD декодера.

Потоковое описание построено с параллельными операторами *назначения сигнала*. Каждый из операторов может быть активизирован, когда любой из его входных сигналов изменяет свое значение. В то время как эти операторы описывают поведение схемы, много информации относительно ее структуры также может быть извлечено из формы описания. Пример 2

содержит этот тип описания для того же самого BCD декодера как и в предыдущем примере.

Тело *архитектуры* описывает только ожидаемые функциональные возможности (**поведение)** схемы, без любого прямого признака относительно аппаратного выполнения.

Такое описание состоит только из одного или нескольких *процессов,* каждый из которых содержит последовательные операторы (Пример 3).

Тело *архитектуры* может содержать утверждения, которые одновременно определяют, и поведение и структуру схемы. Такое описание *архитектуры* называют **смешанным** (Пример 4).

Примеры:

Пример1

```
architecture Structure of Decoder_bcd is
signal S: Bit_Vector(0 to 1);
component AND_Gate
port(A,B:in Bit; D:out Bit);
end component;
component Inverter
port(A:in Bit; B:out Bit);
end component;
begin
    Inv1:Inverter port map(A=>bcd(0), B=>S(0));
    lnv2:lnverter port map(A=>bcd(1), B=>S(1));
    A1: AND_Gate port map(A=>bcd(0), B=>bcd(1), D=>led(3));
    A2:AND_Gate port map(A=>bcd(0), B=>S(1), D=>led(2));
    A3:AND_Gate port map(A=>S(0), B=>bcd(1), D=>led(1));
    A4:AND_Gate port map(A=>S(0), B=>S(1), D=>led(0));
    end Structure;
```

Компоненты Инвертор и AND_GATE - текущие под именами *Inv*1, *Inv*2, *A*1, *A*2, *A*3 и *A*4. Подключения среди компонентов возможны при помощи сигналов S (0), S (l) объявленных в декларативной части архитектуры.

Пример 2

```
architecture Dataflow of Decoder_bcd is
begin
    led(3) <= bcd(0) and bcd(1);
    led(2) <= bcd(0) and (not bcd(1));
    led(1) <= (not bcd(0)) and bcd(1);
    led(0) <= (not bcd(0)) and (not bcd(1));
end Dataflow;
```

Здесь одновременно выполнены все четыре утверждения, и каждое из них активизируется индивидуально, когда любой из его входных сигналов изменяет его значение.

Пример 3

```
architecture procedural of Decoder_bcd is
signal S: bit_vector (3 downto 0);
begin
P1: process (bed, S)
    begin
    case bed is
        when “00” => S <= “0001 “ after 5 ns;
        when “01 “ => S <= “0010” after 5 ns;
        when “10” => S <= “0100” after 5 ns;
    when “11 “ => S <= “1000” after 5 ns;
    end case;
 led <= S;
    end process;
    end procedural;
```

Пункт « after 5 ns » здесь позволяет представить задержку времени схемы. Присвоение нового значения к ведомому сигналу будет сделано только после 5 наносекунд моделируемого времени.

Пример 4

```
architecture Mixed of Decoder_bcd is
signal S: Bit_Vector(0 to 2);
component Inverter
port(A: in Bit; B: out Bit);
end component;
begin
    Inv1: Inverter port map (A=>bcd(0), B=>S(0));
    lnv2: Inverter port map (A=>bcd(1), B=>S(1));
    P: process (S, bcd)
begin
    led(0) <= S(0) and S(1) after 5 ns;
    led(1) <= S(0) and bcd(1) after 5 ns;
    led(2) <= bcd(O) and S(1) after 5 ns;
    led(3) <= bcd(O) and bcd(1) after 5 ns;
    end process;
end Mixed;
```

Выше, две компоненты текущего оператора инвертора определяют схему, отвечающую за определение значения сигнала S. Этот сигнал читается поведенческой частью, то есть оператором процесса P. В этом процессе, вычисленные значения и операции присвоены ведомому порту вывода.

Важное замечание:

- Одиночный объект может иметь несколько архитектур, но архитектура не может быть присвоена различным объектам.

- Архитектура не может использоваться без объекта.
- Все декларации, определенные в объекте полностью видимы и доступны в пределах каждой архитектуры, присвоенной этому объекту.
- Различные типы утверждений (то есть процессы, блоки, параллельные назначения сигнала, текущих компонент, и т.д.) могут использоваться в одной и той же архитектуре.

3 ПАРАЛЛЕЛЬНЫЙ VHDL

3.1 Назначение сигнала

Переменные - последовательные объекты, сигналы - сходящиеся объекты. Для аппаратных средств естественнее использовать объекты типа сигнала.

Синтаксис назначения сигнала:
<результирующий сигнал> '<=' <выражение>';'

Пример:

```
Entity NAND_comp is
   port( a,b: in  std_logic;
    c:  out std_logic);
end;
Architecture rtl of NAND_comp        is
begin
c<=a nand b;
end;
```

Вышеупомянутый пример не имеет никакой задержки компонента. Это - нормальный метод описания для VHDL кода, написанного для синтеза. В качестве альтернативы, код couL был написан с задержкой компонента, например 5 ns, используя команду **after** в VHDL:

```
Architecture NAND_beh of NAND_comp is
begin
c<=a nand b after 5 ns;  -- Задержка компоненты = 5 ns
end;
```

Возможно определить несколько значений в одном назначении сигнала. Значения перечисляются одно за другим через запятую. Например:

```
c<= '1',
   ‘0’ after 10 ns,
   b after 20 ns;
```

Сигнал вывода выше будет иметь следующие значения:

```
0 нс  ‘1’
10 нс  ‘0’
20 нс  b  -- значение сигнала b
```

Назначение сигнала выше будет также выполнено снова, как только сигнал изменит значение. Этот метод описания не обеспечивается инструментальными средствами синтеза. Задержка после команды **after** должна быть в порядке возрастания. Следующий пример неправилен:

```
c<= '1',
     '1' after 10 ns,
     '0' after 5 ns;  -- Ошибка: 5 ns меньше чем 10 ns
```

Правильный пример был бы:

```
c<= '1',
   '0' after 5 ns,
   '1' after 10 ns;
```

3.2 Задержка распространения информации

Имеются два типа задержки распространения информации: инерционная и транспортная.

Инерционная задержка предполагает:

- значение по умолчанию в VHDL;
- выбросы не распространяются (если используется **after**);
- часто используется для задержек компонента.

Транспортная задержка предполагает:

- импульсы передаются, независимо от величены задержки;
- используется для задержек связи.

Транспортный режим задержки должен быть описан в VHDL коде. Инерционная задержка является значением по умолчанию.

Пример:

```
ql<=a after 5 ns;  -- инерционная задержка;
q2<=a transport after 5 ns; -- транспортная задержка.
```

Если импульс, скажем, 2 ns происходит в CMOS (Complementary Metal Oxide Semiconductor) компоненте, который имеет задержку 10 ns, этот выброс обычно не видим при выводе. Именно поэтому инерционная задержка обычно используется для моделирования задержек компонента. Проблемы возникают, однако, если в модели компонента с задержкой 10 ns, импульсы на входе равные или больше чем 5 ns, сделать видимыми на выходе. Эта проблема была решена VHDL-93, новой командой: **reject. Reject** может использоваться, чтобы определить длину импульса, при котором компонент должен пропустить импульс или нет.

Пример (VHDL-93):

```
q3<=reject 4 ns inertial a after 10 ns;
```

Вышеупомянутая задержка игнорирует все импульсы на входе, которые меньше чем 4 ns. Импульсы, которые 4 ns или больше, будут видимы при выводе (q3) после 10 ns. Обратите внимание, что **инерционная** задержка должна быть определена, если используется команда **reject.**

Инструментальные средства синтеза не поддерживают данные модели задержек. Они лучше всего используются при создании VHDL модели для моделирования. Однако, инерционная задержка может использоваться в проекте. Но инструмент синтеза игнорирует задержку. Транспортная задержка вызовет ошибки в большой части инструментальных средств синтеза. Пример представлен на рис. 3.1.

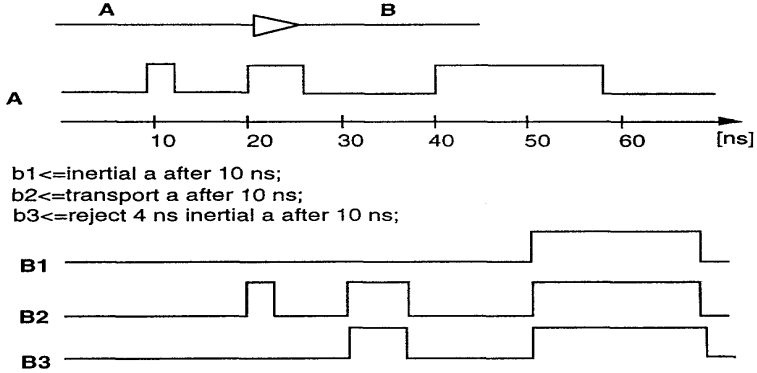


Рисунок 3.1- Фрагменты задержек

3.3 Параллелизм

Аппаратные средства параллельны по своей природе и VHDL - также. Это означает что все параллельные конструкции языка в VHDL коде могут быть выполнены одновременно.

Пример:

ARCHITECTURE EXAMPLE OF EX IS

begin
a<=b;
b<=c;
end example;

это эквивалентно следующему

ARCHITECTURE EXAMPLE2 OF EX IS BEGIN

b<=c;
a<=b;

end example2;

Два примера вышеприведенных идентичны с функциональной точки зрения, потому что сходящиеся команды VHDL управляются событием:

b<=c; -- выполняется когда ***c*** *изменит значение.*

Это означает что, когда входной сигнал *c* изменяет значение, выполняются все строки, в которых *c* справа от символа назначения, то есть b < =c.

Если выполнить синтез для вышеупомянутого примера, то получится результат, показанный на рис.3.2.

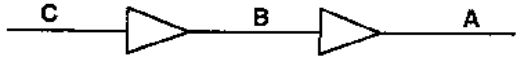


Рисунок 3.2 - Результат синтеза (без оптимизации)

Сигнал *a* не изменяется прежде, чем сигнал *b* не изменит свое значение. Точно так же сигнал *b* не изменяет свое значение прежде сигнала *c*. То же самое в VHDL коде, поскольку выполнение операций управляется событием. Это означает, что все сходящиеся команды VHDL могут быть написаны в любом порядке без изменения функционирования проекта.

Все сходящиеся утверждения могут быть идентифицированы с меткой:
Имя_метки: b<=a;

Эта метка используется только для документирования и не имеет никакого функционального значения. В отличие от некоторых языков программирования (например, Бейсик), невозможно перейти к метке.

3.4 Дельта время

Дельта время используется в VHDL для «стоящих в очереди» последовательных событий. Время между двумя последовательными событиями называется дельта задержкой. Дельта задержка не имеет никакого эквивалента в режиме реального времени, но выполняется, в то время как синхроимпульсы моделирования еще стоят. Величина не назначается сигналу непосредственно, а назначается после задержки дельты в самом начале:

b<=a; - - Сигнал b принимает значение a после одной дельта-задержки.

Пример показан на рис.3.3

В комбинационном логическом блоке, где все элементы имеют задержку 0 ns, все назначения будут в 0 ns, но при этом они могут содержать много дельта задержек. VHDL имитатор считает количество дельта задержек, пока все сигналы не устойчивы. Если VHDL код написан неправильно, имеется риск, что проект будет колебаться *до бесконечности.* Для предотвращения “заедания” VHDL имитаторов, большинство имита-торов останавливается, например, после 1000 дельта времени.

Количество дельта-задержек может обычно устанавливаться. Следующий пример - правильный, но генерирует проект, который колеблется *до бесконечности:*

q<= **not** q;

Сигнал q будет модифицироваться инверсный после дельта время. Это заставляет операцию выполняться снова, и *q* будет модифицироваться после другой дельты - задержки, и т.д. Эту ситуацию можно избежать, если вставить задержку:

q<= not q after 10 ns;

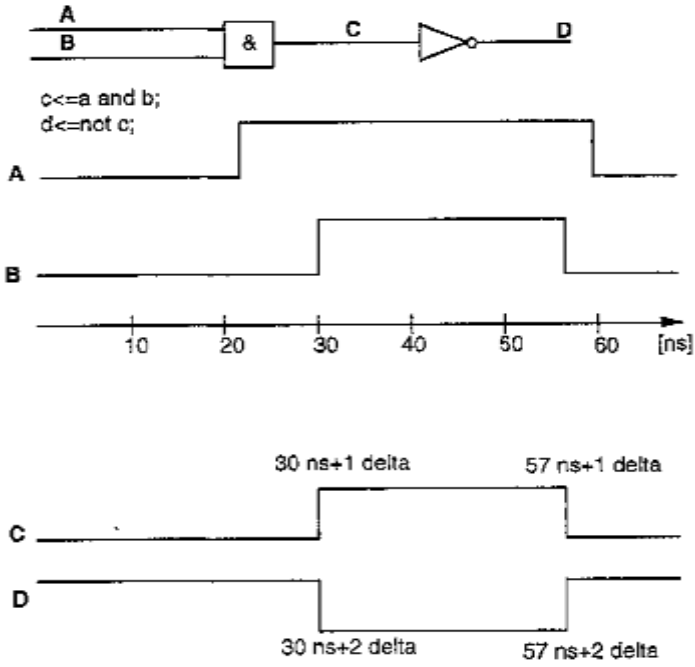


Рисунок 3.3- Пример дельта-задержки

Обратите внимание, что, если вышеупомянутый код синтезируется, это будет вести к асинхронной обратной связи, которая обычно не желательна в проекте.

3.5 Оператор When

Синтаксис:

<выходной сигнал> **<=** <выражение> **[after** <выражение>] **when** <условие> **else**

<выражение> **[after** <выражение>]....;

Допустимо иметь несколько **when else** строк. Сигналу <выходной сигнал> должно всегда назначаться значение независимо от значения <выражение>.

Пример:

```
ENTITY EX IS
        port( a,b,c:  in  std_logic;
              data:  in  std_logic_vector(l downto 0);
              q:  out std_logic);
       end;
       Architecture rtl of ex is
begin q<= a when data=”00” else
         b when data=”11” else c;
        end;
```

В VHDL-93 **else** <выражение> может быть не учтен (это не обеспечивается большей частью инструментальных средств синтеза).

Пример (VHDL-93):

```
ARCHITECTURE RTL OF EX IS
  begin q<= a when data=”00” else
  b when data=”l 1”; - - нет else <выражение>
  end;
```

Команда **when** очень полезна, например при проектировании буфера с тремя состояниями. Пример дан в рис. 3.4.

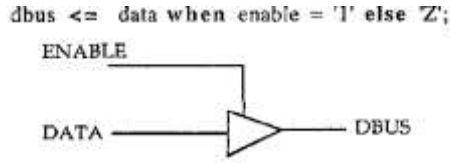


Рисунок 3.4 - Результат синтеза

Если *dbus* и *data* изменить на векторы, всему вектору будет соответствовать буфер с тремя состояниями на выводе. Тогда код в архитектуре должен измениться следующим образом:

dbus <= data **when** enable = ‘1’ **else** (others=>’Z’);

Используя (others=>'Z') назначение, всему вектору, независимо от векторной длины, будет назначено 'Z' значения. Этот способ назначения значения всего вектора очень эффективен и делает код проще для поддержки. Если длина вектора должна быть изменена, только объявление для векторов должно измениться, а не код в архитектуре.

В условии допустимо использовать различные сигналы, делая команду, очень гибкой и полезной для проекта.

Пример:

q<= a **when** en=’0’ **else**
b when data=”11” else
c **when** enable’1’ **else** d**;**

Обратите внимание, что условия проверены в порядке, в котором они перечислены. Т.е., если en= '0', данные = «11» и enable=’1’. Это означает что, *q* будет дано значение сигнала *a* в вышеупомянутом примере. Если строки два и три перенесены, *q* будет дан значение *c* сигнала. Внутри сходящейся команды, например в **when else** порядок является важным.

VHDL синтаксис не дифференцирует между символами верхнего регистра и символами нижнего регистра.

Единственное исключение - внутренние одиночные кавычки (' ') или двойные кавычки (« »), например, когда значение 'Z' назначено сигналу типа std_logic или bit, ' Z ' должен быть верхнего регистра:

sig<='Z'; -- *верная запись*
sig<='z' -- *ошибочная запись*

3.6 Оператор With

Синтаксис:

with <выражение> **select**
<выходной сигнал> <= <выражение> **when** <выбор> ;

Все возможные <выборы> должны перечисляться. Если вы хотите описать вместе все остающиеся варианты, используется **when others**. В этом случае **others** должен быть последний вариант <выбора>.

Пример:

Entity ex is
port(a,b,c: in std_logic;
data: in std_logic_vector(l **downto** 0);
q: out std_logic);
end;

```
ARCHITECTURE RTL OF EX IS
    begin
    with data select q<= a when “00”,
        b when “11”,
        c when others;
    end;
```

По сравнению с командой **when else**, команда **with select** не столь гибка, т.к. разрешено только одно <выражние>. Однако, команда обычно приводит к коду, который является относительно простым, для чтения и структурирования.

Пример:

```
    Entity mux2 is
            port( sel_0,a,b: in  std_logic;
           c:  out std_logic);
    end mux2;
    Architecture mux2_beh of mux2 is
    begin
        c<= a after 10 ns when sel_0=’0’ else
        b after 10 ns;
    end mux2_beh;
    Architecture mux2_beh2 of mux2 is
    begin
with sel_0 select
c<= a after 10 ns when ‘0’,
b after 10 ns when others;
end mux2_beh2.
```

3.7 Настройка

Настройка может использоваться, чтобы ввести информацию в модель, например информацию синхронизации. **Generic** должен быть объявлен в **entity** перед командой **port**. **Generic** может иметь любой тип данных; имеются, однако, некоторые ограничения относительно того, что инструментальные средства синтеза воспримут.

Пример.

```
    Entity generic_ex is
    generic (delay:time:=10 ns);
            port( a,b: in  std_logic;
           c:  out std_logic);
    end generic_ex;
    Architecture generic_beh of generic_ex  is
begin
```

c<=a **and** b **after** delay;
end generic_beh;

Компонент выше имеет **generic** (настраиваемую) задержку. Задержка может быть изменена.

3.8 Конструкция Assert

Конструкция (Assert), делает возможным проверить функцию и ограничения времени на модель внутри VHDL компонента.

Если условие для **assert** не выполнено в течение моделирования проекта VHDL, сообщение в **severity** посылается пользователю (имитатор). Использованием **assert** возможно проверить запрещенные комбинации сигналов или не выполнение ограничений по времени.

Синтаксис:

Assert <условие>
Report <сообщение>
Severity <уровни ошибок>

Если условие не выполнено, assert сообщение посылается пользователю плюс имя процесса (модуля), который был отменен. Имеются четыре различных уровня **severity** (уровни ошибок):

- Note – Примечание;
- Warning – Предупреждение;
- Error - Ошибка;
- Failure - Отказ.

Сообщение и уровень ошибки сообщаются имитатору и обычно отображаются в открытом тексте в окне команды VHDL имитатора. Возможно установить уровень ошибки, в котором VHDL имитатор должен прекратить моделирование. Значение по умолчанию для большинства имитаторов - Error (Ошибка).

Assert может использоваться, чтобы проверить внешние сигналы к компоненту или внутреннему поведению, проверить время или выполнить функциональную проверку. Так как **assert** является и последовательной и параллельной командой, она может использоваться виртуально везде в коде.

Код управления обработкой ошибок используется только в течение VHDL моделирования. Пример кода управления обработкой ошибок следующий:

assert in_0 /= ‘X’ **and** in_l /= ‘X’
report “in is not connected”
severity Error;

Программы управления обработкой ошибок проверяют, что in_0 и in_l соединены, или что они имеют неопределенное значение. Если они не соединены, имитатор остановится и отобразит «in is not connected». Переменная *now is* определена в VHDL стандарте. Она содержит внутреннее абсолютное время имитатора. С командой **assert** можно проверить, что время имитатора не превышает, скажем 900 ns:

```
process(clk)
begin
assert now < 900 ns
report "stopping simulator (max simulation time 900 ns)"
severity Failure;
end process;
```

Если команда **assert** используется в параллельной части VHDL кода, команда будет выполнена, если только событие на ее «список чувствительности» имеет место. Если команда **assert** только проверяет время с переменной *now*, никакое событие не произойдет, и команда **assert** никогда не будет выполнена. Следовательно, команда **assert** в вышеупомянутом примере была помещена в **process**, который выполняется на каждом фронте синхроимпульса.

Пример.

```
ENTITY EX IS
 port( a,b: in  std_logic; q:  out std_logic);
 begin
assert a/='1' or b/='1'
report "a='1' and b='1' at the same time"
severity warning;
 end;
        Architecture
        …
```

Если Вы формируете среду теста, **assert** хороша для проверки ответа из схемы. Если команда **report** используется в последовательной части VHDL кода, команда **assert** может быть не учтена в VHDL-93 стандарте.

Пример (VHDL-87):

```
process(a,b)
begin
if a='1' and b='1' then
assert false
report "a='1' and b='1'";
end if;
            …
 end process;
```

Пример (VHDL-93):

```
process(a.b)
begin
    if a='l' and b='1' then
    report "a='1' and b='1'";
    end if;
          …
end process;
```

3.10 Объекты, классы, типы

Определение объекта в VHDL означает описасание констант, переменных или сигналов**.** Объект содержит значение специфического типа.

Каждый объект имеет тип и класс. Тип указывает, какие данные содержит объект. Класс указывает то, что может быть сделано с объектом. VHDL строго контролирует типы. Это означает, что различные типы не могут быть смешаны без преобразования типов.

Класс Объект Тип Данных

signal a: std_logic

Класс (Class)

Имеются три различных объекта класса:

Constant - константа;

Variable - переменная;

Signal "a wire" – сигнал "провод".

Constant (Константа) не изменяет значение. Она может быть объявлена во всех частях, и может быть любого типа. Например:

constant a: a_type:="1001";

Variable (Переменная) изменяет значение. Может быть объявлена в процессе и подпрограмме, и может быть любого типа. Например:

a:=John;

Signal (Сигнал) может изменять значение относительно времени. Например:

b<=inputl;

Каждый сигнал должен иметь объявленный тип данных, который определяет значения, которые сигнал может принимать. Самые общие типы данных описаны ниже.

Булев тип

Булев тип данных может только быть истинен или ложный. Константы, сигналы и переменные могут быть объявлены как Булев:

```
Architecture rtl of ex is
    begin
```

```
    process(...)
    variable John: boolean;
        begin
John:= a < b;
if John then
            ...
end process;
end;
```

Инструментальные средства Синтеза транслируют значения ложь к «0» и истину к «1». Назначая сигнал, например, std_logic, к значениям “истина” или “ложь” без использования функции преобразования, не разрешается.

Целочисленный тип (Integer)

VHDL-87 стандарт не определяет, какой длины должно быть целое число. Длина - управляемая реализация, то есть зависит от используемого инструмента. Однако, большая часть инструментальных средств использует 32 бита. Это означает что, целое число может измениться между -2147483648 и 2147483647.

Пример 1: **constant** loop_number: integer := 345;

Пример 2: **signal** my_int: integer **range** 0 **to** 255;

Символьный тип (VHDL-87)

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’, ‘1’, ‘m’, ‘n’, ‘o’,
‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’,
‘x’, ‘y’, ‘z’, ‘{‘, ‘|’, ‘}’, ‘~ ‘, DEL);

В VHDL-93, число разрешенных символов увеличилось.

Символьный тип (VHDL-93)

type character is (

nul, soh, stx, etx, eot, enq, ack, bel, bs, ht. If, vt, ff, cr, so, si, dle, del, 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’, ‘0’, ‘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’, ‘f’, ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, ‘z’, ‘{‘, ‘|’, ‘}’, ‘~’, del, cl28, cl29, cl30, cl31, cl32, cl33, cl34, cl35, cl36, cl37, cl38, cl39, cl40, cl41, cl42, cl43, cl44, cl45, cl46, cl47, cl48, cl49, cl50, cl51, cl52, cl53, cl54, cl55, cl56, cl57, cl58, cl59,

Последние 64 символа являются специальными

Тип бит

Бит - тип, предопределенный в стандартном пакете как числовой тип данных, имеющий только два допустимых значения: '0' и ‘1 '.

Синтаксис :

type bit **is (‘**0’,’1’);

Vibit тип

Vibit - тип который использует ViewLogic и является расширением bit типа в стандарте VHDL. Vibit имеет значения 'X', 'Z', '0' или '1'. В течение моделирования, 'X' означает неизвестное.

Пример:

if John(6 **downto** 0) = “1XXXX0X” **then** ...

эквивалентное описание:

if (John(6)=’1’ **and** John(l)) = ‘0’ **then ...**

'Z' Означает высокий импеданс, и в моделировании и синтезе. Вы должны быть внимательным относительно того, что вы синтезируете, например некоторые FPGA. Эта проблема может быть решена мультиплексированием сигнала. Только bit и bit_vector определены в VHDL стандарте. Это ведет к отдельным ограничениям в проекте, если бит и bit_vector используются:

- Невозможно описывать три состояния;
- Невозможно иметь отдельные драйверы для того же самого сигнала;
- Невозможно назначать сигналу значение “X”;

Поскольку бит может принимать только значения ‘0’ и '1', с тремя состояниями и неизвестное, например, не может быть описан, используемый бит или bit_vector. В случае сигнала с тремя состояниями, имеются часто отдельные сигналы, которые могут управлять сигналом. Это означает, что должен быть найден метод для решения, который оценивает значения сигнала, если имеются отдельные драйверы. Это не определено в VHDL стандарте для bit или bit_vector типа данных. Назначение значения

“X” может также быть эффективно в некоторых ситуациях. При проектировании имитационных моделей в VHDL, использование bit и bit_vector также ведет к следующим ограничениям:

- Невозможно описывать подъем (слабая единица);
- Невозможно описывать спуск (слабый нуль);
- Невозможно описывать, что сигнал является неинициализированным.

Чтобы решать эти проблемы, VHDL поставщики создали их собственные типы данных. Это решило вышеупомянутые проблемы, но вызвало новые:

- VHDL код стал зависимым от инструмента.
- Стало невозможным соединить два проекта из двух различных инструментальных средств.

VHDL - язык со строгим контролем типов: не допустимо соединять два сигнала (например, в иерархическом проекте) если они не имеют одинаковый тип данных.

Тип Std_ulogic

Чтобы сделать VHDL эффективной платформой - и имитатор независимым стандартом, IEEE-определенные типы данных представлены: std_logic и std_ulogic. Эти типы данных стали промышленным стандартом для проектов. Std_logic и std_ulogic может принимать точно те же самые значения, различие в том, что std_logic - разрешенный подтип std_ulogic. Std ulogic может принимать следующие значения:

1	'U'	- Инициализация
2	'X'	-- Неопределенное
3	'0'	-- ‘0’
4	‘1’	-- ‘1’
5	‘Z’	-- Высокий импеданс
6	'W	-- Слабое неизвестное
7	•L'	--Слабый 0
8	•H'	-- Слабая 1
9	'-'	-- Безразлично

Std_logic type

Std_ulogic - тип, который объявлен в IEEE пакете std logic 1164. std_ulogic может принимать те же самые значения как std_logic. Различие - что std_logic, определен как:

Подтип std_logic разрешен std_ ulogic;

Факт, что std_logic разрешенное средство, если сигнал управляется отдельными драйверами, вызывается предопределенная функция разрешающей способности, которая решает конфликт и решает, какое значение

сигнала будет дано. Это означает что, std_logic может, например, использоваться для шин с тремя состояниями, в которых отдельные драйверы могут управлять тем же самым проводником на шине (Рис.3.6), но обычно не одновременно. Если сигнал std_ulogic управляется больше чем одним драйвером, это приведет к ошибке (Рис.3.7), так как VHDL не разрешает неразрешенному сигналу управляться больше чем одним драйвером.

```
Entity ex is
  port( d,c,en1,en2:  in  std_logic;
        dbus:         out std_logic);
end;
Architecture rtl of ex is
begin
    dbus<= d when enable1='1' else 'Z';
    dbus<= c when enable2='1' else 'Z';
end ;
```

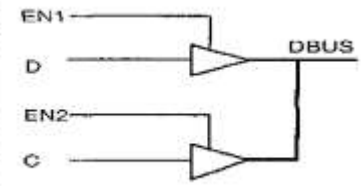


Рисунок 3.6 - Std_logic

```
Entity ex is
  port( d,c,en1,en2:  in  std_ulogic;
        dbus:         out std_ulogic);
end;
Architecture rtl of ex is
begin
    dbus<= d when enable='1' else 'Z';
    dbus<= c when enable2='1' else 'Z';
end ;
```

ERROR

Рисунок 3.7 - Std_ulogic

Это ограничение на std_ulogic приводит к предпочтению типу данных std_logic. Вообще самое простое использовать тот же самый тип данных через проект, поскольку это не требует преобразований типов. Недостаток std_logic - то, что никакой ошибки не выдается, если у вас, по ошибке два драйвера для того же самого сигнала в VHDL коде. Сигналу тогда будет присвоено значение 'X' в моделировании.

Array type

Std_ulogic_ vector определяется как:

type std_ulogic_vector is **array** (natural **range** <>) **of** std_ulogic;

Std_logic_vector определяется как:

type std_logic_vector **is array** (natural **range** <>) **of** std_logic);

Bit_vector определяется как:

type bit_vector **is array** (natural **range** <>) **of** bit;

Vlbit_vector определяется как:

type vlbit_vector **is array** (natural **range** <>) of vibit;

Типы vlbit_vector и vlbit_1d идентичны

Объявление Типа

В архитектуре возможно создавать свои собственные типы.

type <иден> **is** <определяемый_тип> ;

Определяемый_тип = **array** <индексное_значение> of тип_элементов;

Пример:

```
type a_type is array (3 downto 0) of std_logic;
type my_int is integer range 0 to 15;
signal d: a_type;
signal q: my_int;
```

Вышеупомянутое целое число могло также быть определено непосредственно в объявлении сигнала:

```
signal q: integer range 0 to 15;
```

Важно объявить диапазон целого числа, иначе инструмент синтеза примет, что целое число - 32 бит. Если, например, для сохранения целого числа в регистре, будут даны 32 бита, если диапазон не определен, и только 4 для целого числа в примере, приведенном выше. То же самое применяется и к векторам.

```
Entity ex is
 port( a: in  Std_logic_vector(3 downto 0);
 b: in  Bit_vector(0 to 3);
 c: out  Std_ulogic_vector(4 downto 0));
end;
Architecture rtl of ex is
    signal il: std_logic_vector(3 downto 0);
    signal i2: vlbit_vector(3 downto 0);
    signal i3: std_ulogic_vector(3 downto 0);
begin
      . .
end.
```

Допустимо определять векторы **to** и **downto**. Если используется **to**, вектор должен быть определен от 0 до 3. Если используется **downto**, вектор должен быть определен, как 3 **downto** 0. В проектах обычно самое простое использовать то же самое определение последовательно. Кроме того, **downto** лучше подходит нормальному мышлению аппаратного проектировщика, т.к. при этом самый левый бит является старшим - MSB (Most Significant Bit -Старший Бит). Следовательно, большая часть проектов, написанных для синтеза использует **downto** во всех векторных определениях.

Time

Пример:

```
constant delay : time := 0 ns;
```

a <= b **after** delay;

Перечисляемый тип

Возможно определить собственные типы данных в VHDL. В принципе, эти типы данных могут принимать любое значение, что бы то ни было. Конечные автоматы - общая область использования для перечисляемых типов данных.

Пример:

type state_type is (start, idle, waiting, run);
signal state:state_fype;

3.10 Назначение вектора

При проектировании в VHDL, векторы обычно используются как тип данных. **Вектору** может быть назначено значение многими различными способами. Двоичное значение может быть назначено следующим образом:

a_vect<=»10011»;

Отметим, что вектор записывается внутри двойных кавычек, а индивидуальный бит внутри одиночных кавычек.

Если используются логические операторы, например **and**, то результатом будет поразрядное **and.**

Пример.

Architecture rtl of ex is
signal a,b:std_logic_vector(3 **downto** 0);
signal c:std_logic_vector(3 **downto** 0);
begin
a<=”0110”;
b<=”1101”;
c<= a **and** b;
end;

Вектору *c* будет назначено значение»0100»:
«0110» **and** «1101» = «0100»

Требование относительно логических операторов - вектора должны иметь одинаковую длину.

Для bit_vector имеются предопределенные литералы строки битов, которые могут использоваться, чтобы назначить bit_vector следующим образом:

Пример.

Binary B”11000”
Octal 0”456”
Hex X”FFA5”
Decimal 239 только для констант
Real 4.6E-4 не поддерживается синтезом

Преимущество литералов строки битов - то, что VHDL код часто становится более легким для чтения, если вектору назначено шестнадцатеричное значение, чем при двоичной единице. С точки зрения синтеза, однако, не имеется никакого различия.

Если вектору должно быть назначено двоичное значение, ‘B’ должно быть записано перед вектором, точно как показано ранее в главе. Два векторных назначения ниже, идентичны:

```
a_vect<=”10011”;
a_vect<=B”10011”;
```

В битовом векторе для удобочитаемости может использоваться подчеркивание (_).

Пример:

```
a_vect<=B” 1100_0011_0011_1100”;
a_vect<=” 1100001100111100”;
a_vect<=X”C33C”;
a_vect<=X”C3_3C”;
        a_vect<=»1100_0011_0011_1100»;  --ошибочная запись
        a vect<=»C33C»;  --ошибочная запись
```

VHDL - язык со строгим контролем типов. Это означает, что не допустимо назначить bit_vector к std_logic_vector, например, без того, чтобы использовать функцию преобразования.

Пример.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
    Entity ex is
        port( a:  in  std_logic_vector(2 downto 0);
        b:  out bit_vector(2 downto 0));
      end;
      Architecture rtl of ex is
      begin
      b<=to_bitvector(a,0):
      end;
```

Для того чтобы использовать std_logic_vector или std_ulogic_vector при проектировании, должна использоваться функция преобразования.

Пример.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee. std_logic_unsigned. ALL;
Entity ex is
port( a: out std_logic_vector(15 downto 0);
                                        b: out bit_vector(15
                                        downto  0));
```

```
        end;
        Architecture rtl of ex is
begin
    a<=to_stdlogicvector(X"F6");
    b<=X"E4"
    end;
```

Как показывает вышеупомянутый пример, X «F6» преобразовывается в std_logic_vector функцией to_std_logic_vector. Это, потому что X «F6» будет описан как bit_vector, и, поскольку VHDL - язык со строгим контролем типов, bit_vector не может быть назначен к std_logic вектору без предварительного преобразования в std_logic_vector. В этом образце использовалась конверсионная функция to_std_logic_vector, которая была объявлена в пакете IEEE.std_logic_unsigned,. Используемая функция не затрагивает результат моделирования или синтеза.

Если бы в вышеупомянутом примере был назначен сигнал с непосредственным значением X «F6», ошибка произошла бы в трансляции кода VHDL. В VHDL-93 стандарте, константа строки битов была расширена. В VHDL-93 допустимо назначить std_logic_vector и std_ulogic_vector значение непосредственно без конверсионной функции. Это заставит VHDL закодировать более легким для чтения и быстрым для записи кодом:

```
signal a: std_logic_vector(7 downto 0);
                    a<=to_std_logic_vector(X"F4");   -- VHDL-87
                    a<=X"F6";                -- VHDL-93
```

Большая часть инструментальных средств синтеза не поддерживает VHDL-93, это означает, что константы строки битов для std_logic_vector могут быть только использованы для моделирования.

Пример:

```
Architecture rtl of ex is
                    constant myint:integer:=16#FF#;  --myint=255
                    signal intl,int2,int3:  integer range 0 to 1023;
begin
                    intl<=16#FE#;  -- 16#FE# = 254
                    int2<=2#100110#;  -- 2#100110# = 38
                    int3<=8#17#;  -- 8#17# = 15
                    . . .
    end;
```

Если вы хотите назначить значение на бит или часть вектора, это может быть сделано следующим образом:

```
    Architecture rtl of ex is
    signal a_vect: std_logic_vector(4 downto 0);
    signal b_vect: std_logic_vector(0 to 4);
begin
```

```
a_vect(4)<='l';
a_vect(3 downto 0)<="0110";
b_vect(4)<='0';
b_vect(0 to 3)<="1001";
end;
```

Обратите внимание, что, когда сектору массива назначено значение, направление сектора должно быть тем же самым как в объявлении, то есть **downto** или **to.** Также допустимо назначить вектор сектором другого вектора.

Пример:

```
Architecture rtl of ex is
        signal a_vect:  std_logic_vector(4 downto 0);
        signal b_vect:  std_logic_vector(0 to 4);
        signal c:  std_logic;
        begin
        a_vect<="01101";
        b_vect(4)<=c;
        b_vect(0 to 3)<=a_vect(3 downto 0);
end;
```

Тип данных векторов должен быть одинаков с обеих сторон символа назначения, как было уже сказано, и векторные длины также должны быть равны. Следующий пример демонстрирует, как неправильное назначение сигналов.

Пример :

```
Architecture bad of ex is
signal a: std_logic_vector(2 downto 0);
signal b: std_logic_vector(3 downto 0);
begin
 a<=b;  -- ошибка - несоответствие длин векторов ,
end;
```

Вышеупомянутое назначение должно быть перезаписано так, чтобы обе стороны символа назначения имели ту же самую длину, например:

```
a<=b(2 downto 0);
```

Альтернативно (в зависимости от желательной функции):

```
a<=b(3 downto 1);
```

Если индексный порядок для векторов был определен по разному для двух векторов, все еще возможно назначить одному вектору значение другого.

Пример:

```
Architecture rtl of ex is
signal a,b,c: std_logic_vector(2 downto 0);
        signal d:  std_logic_vector(0 to 2);
        a<=d;
        b<=c;
```

end;

В вышеупомянутых векторных назначениях, вектора *a* и *b* приобретают следующие значения:

a(2)<=d(0); b(2)<=c(2);
a(l)<=d(l); b(l)<=c(l);
a(0)<=d(2); b(0)<=c(0);

Амперсант (&) конкатенация средств в VHDL. Конкатенация может быть очень полезна в векторном назначении.

Пример:

```
Architecture rtl of ex is
    signal a:  std_logic_vector( 5 downto 0);
    signal b,c,d: std_logic_vector(2 downto 0);
    begin
       b<='0' & c(l) & d(2);
       a<=c & d;
    end;
```

Если, например, вектор *c* имеет значение «011» и вектор *d* значение «101», вектор *b* приобретает значение '0' и '1' и '1', то есть «011» и вектор *a* значение «011» & «101», то есть «011101». Конкатенация может также использоваться для условных операторов, например:

```
if c & d = "001100" then
    . . .
```

Как предварительно упомянуто, обе стороны назначенного векторного символа должны иметь равную длину. Не допустимо использовать конкатенацию слева, чтобы достичь этого.

Пример ошибочной записи:

```
Architecture bad of ex is signal a:std_logic_vector(2 downto 0);
signal b:std_logic_vector(3 downto 0);
begin
 '0' & a<=b;  -- Ошибка
end;
```

Если Вы выполняете разработку с большими векторами и хотите назначить значение из одинаковых битов, на полный вектор, это может быть сделано следующим образом:

```
    Architecture rtl of ex is
signal a:  std_logic_vector(4 downto 0);
begin
    a<=(others=>'0');
    end;
```

Эта команда идентична a< = «00000»;. Преимущества этой операции в уменьшении записи в случае больших векторных назначений, и что назначение является полностью независимым от векторной длины. Также

допустимо назначить некоторые биты в векторе и затем использовать **others**, чтобы назначить остающиеся биты:

a<=(l=>'1', 3=>'1', **others**>'0'));

Вышеупомянутое назначение означает, что битам 1 и 3 назначены значения '1', а другим битам назначены значения '0'. Также возможна следующее назначение:

a<=(l=>c(2), 3=>c(l), **others**>d(0));

Как альтернатива к вышеупомянутому назначению на вектор *a,* может использоваться конкатенация (предположим, что *a* имеет длину 5 битов):

a<=d(0) & d(0) & c(l) & d(0) & c(2) & d(0);

Недостаток этого метода описания - то, что назначение теперь зависит от длины и должно быть преобразовано(конвертировано), если длина *a* изменена. С точки зрения синтеза, не будет наблюдаться никакой разницы в результате. Оба метода описания поддерживаются большой частью инструментальных средств синтеза.

Иногда не ясно, какой тип данных имеет выражение. Если компилятор не может решить однозначно, какой тип данных имеет выражение или значение, то будет сгенерирована ошибка. Чтобы тип данных был ясным для компилятора, может использоваться спецификатор. При использовании спецификатора, тип данных устанавливается явно, сопровождаемый меткой импульса сигнала времени (') или выражением:

Пример:

ROM_type'(«01»,»10»,»00»);

3.11 Продвинутые типы данных

Под продвинутыми типами данных понимаются многомерные типы данных, подтипы и записи.

Подтипы (Subtypes)

Если объявлено подмножество типа данных, который уже был определен, то должно использоваться объявление подтипа.

subtype <идентификатор> **is** <тип> <ограничения>;

Спецификация необходима для определения значения новых подтипов, то есть подмножества значений базового типа. В качестве альтернативы специфицируется ограниченная длина, например векторная длина, исходного типа. Например:

subtype my_int **is** integer **range** 0 to 3 215;
subtype byte **is** std_logic_vector(7 **downto** 0);
type byte2 **is array** (7 **downto** 0) **of** std_logic;
type byte3 **is** std_logic_vector(7 **downto** 0); -- *Неправильная запись*
subtype byte4 **is array** (7 **downto** 0) **of** std_logic; -- *Неправильная запись*

Как мы видим из вышеупомянутого примера, **array** может только использоваться, чтобы определить новые типы, а не подтипы. Std_logic_vector (7 **downto** 0) не может использоваться в новых объявлениях типа. *Байт* может также быть объявлен как новый тип данных (byte2).

Недостаток использования типа вместо подтипа - то, что объявленный тип становится полностью новым типом, то есть невозможно назначить сектор сигнала с первоначальным типом данных к сигналу с новым типом данных без преобразования сигнала.

Пример неправильной записи:

```
Architecture bad of ex is
type byte is array (7 downto 0) of std_logic;
signal a: byte;
signal c: std_logic_vector(7 downto 0);
begin
a<=c;  -- Ошибка, a и c не принадлежат одному типу данных
end;
```

Если бы *байт* был объявлен как подтип, вышеупомянутый пример был бы передан в VHDL компилятор без ошибки.

Имеются два предопределенных подтипа в VHDL:

Subtype natural **is** integer **range** 0 **to** *верхняя граница* - обычно 2147483647
Subtype positive **is** integer **range** 1 **to** *верхняя граница* - typical 2147483647

С точки зрения синтеза не имеет значения, был ли тип данных объявлен как подтип или отдельный тип.

Обычно используются не больше, чем двух- или трехмерные типы данных. Когда определяются многомерные типы данных, массив используется в объявлении типа.

Пример:

```
type data4x8 is array (0 to 3) of std_logic_vector(7 downto 0);
type data3x4x8 is array (0 to 2) of data4x8;
```

Тип данных data4x8 - двумерный тип данных (4x8 бит), в то время как data3x4x8 - трехмерный тип данных (3x4x8 бит). Объявление типа data4x8, например, могло также быть написано следующим образом:

```
type byte is array (7 downto 0) of std_logic;
type data4x8 is array ( integer range 0 to 3) of byte;
```

Двумерный вектор может быть назначен различными способами. Один массив одновременно может быть назначен, используя индекс, или все двумерные векторы могут быть назначены в один заход, используя агрегат.

Пример:

```
Architecture rtl of ex is
type data4x8 is array (0 to 3) of std_logic_vector(7 downto 0);
signal d,e,f,g,h,i:  data4x8;
```

```
signal bl,b2,b3,b4: std_logic_vector(7 downto 0);
begin
    d(0)<=”01010110”;
    d(l)<=”10101000”;
    d(2)<=”01110110”;
    d(3)<=”10111011”;
e<=(others=> (others=>’0’));
f(0) (0)<=’1’;
f(0) (1)<=’0’;
    . . .
g<=(bl, b2, b3, b4);
h<=(others=>bl);
process(h)
begin
    11: for n in 0 to 3 loop
    i(n)<=h(n);
end loop;
end process;
end;
```

Выбранный метод назначения, будет зависеть от приложения и требуемой удобочитаемости. Большая часть инструментальных средств синтеза, которые поддерживают двумерные типы данных, поддерживает все выше указанные методы.

Записи

Тип записи содержит элементы различных типов данных:

type <индетификатор> **is record**
<декларация полей записи>
end record;

Пример:

```
Architecture beh of ex is
type data_date is record
                year:  integer range 1996 to 2099;
                month:  integer range 1 to 12;
date:           integer range 1 to 31;
                hour:  integer range 0 to 23;
                minute:  integer range 0 to 59;
                second:  integer range 0 to 59;
                data:  std_logic_vector(31 downto 0);
        end record;
        signal d:data_date;
        begin
d.year<=1997;
```

```
d.month<=4;
d.date<=8;
d.hour<=11;
d.minute<=57;
d.second<=22;
d.data<=data_in;
end;
```

В качестве альтернативы, вышеупомянутая **запись** могла быть назначена следующим образом:

```
d<=(1997, 4, 8, 11, 57, 22, data_in);
```

Продвинутые инструментальные средства синтеза поддерживают записи. Но, конечно, имеются некоторые ограничения относительно типов данных, используемых в записях.

3.12 Псевдоним

Псевдоним может использоваться, чтобы дать альтернативные имена объектам. Псевдоним часто используется, чтобы делать код более легким для чтения.

Пример:

```
alias data: std_logic_vector(7 downto 0) is data_in (15 downto 8);
```

В VHDL-87, **псевдоним** допустим только для объектов. Это ограничение было ослаблено в VHDL-93, и теперь допустимо определять псевдонимы для функций, типов, и т.д.

3.13 Операторы отношения

В VHDL определены несколько операторов отношения:

=, /=, <, >, <=, >=.

Результат операторов отношения, является ‘истина’ или ‘ложь’. Вышеупомянутые операторы могут использоваться непосредственно на целочисленных bit_vectors и std_logic_vectors, например. Операторы = и / = могут также использоваться на всех определяемых типах данных.

Пример:

```
Architecture rtl of ex is
type my_type is (on,off,idle,start);
signal  state1,state2: my_type;
        signal a,b,c:  bit_vector(2 downto 0);
        signal d,e,f,g:  std_logic_vector(2 downto 0);
        signal my_int:  integer range 0 to 15;
begin
a<=b when c=”010” else b;
d<=e when d/=”lll” else “000”;
e<=f when my_int<=12 else “001”;
```

```
f<=”101” when e>=d else “110”;
g<=f when statel=state2 else “OOO”;
end;
```

Все операторы отношения поддерживаются большой частью инструментальных средств синтеза.

3.14 Арифметические операторы

В VHDL используются следующие арифметические операторы:

Символ	Операция
+	сложение
-	вычитание
*	умножение
/	деление
abs	Абсолютное значение
rem	Остаток
mod	Сложение по модулю
**	Возведение в степень

Эти арифметические операторы предопределены для данных типа целочисленных, вещественных, и времени. С другой стороны, они не определены для std_logic_vector и std_ulogic_vector. Если std_logic_vectors должны использовать вышеупомянутые операторы, они должны быть определены для std_logic_vector, например в пакете. Это уже сделано в пакете ieee.std_logic_unsigned.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_lpgic_unsigned.ALL;
Entity ex is
port ( a,b: in  std_logic_vector(2 downto 0);
    c,d: in integer range 0 to 15;
    q1: o u t std_logic_vector(2 downto 0);
    q2: out integer range 0 to 30);
end;
Architecture rtl of ex is
begin
q1<=a + b;
q2<=c + d;
end;
```

Большая часть инструментальных средств синтеза поддерживает «+», «-« и «*» для целых чисел. Однако, они не обеспечиваются для веществен-

ных типов данных и времени. Для std_logic_vector поддерживаются операции «+», «-», «*» и «**».

3.15 Инициализация

В начальный момент времени моделирования, для всех сигналов определяются их начальное значение (инициализация). Для различного типа данных начальные значения могут быть различными. По умолчанию всем сигналам присваивается их значение datatype'left. Бит и std_logic определены следующим образом:

type bit **is** (‘0’, ‘1’);
type std_logic **is** (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W, ‘L’, ‘H’, ‘-‘);

Атрибут ' left обозначает значение первого (левого) значения данных в декларации типа. Это ведет:

bit'left='0'
std_logic'left='U'

Возможно изменить заданное по умолчанию начальное значение в VHDL коде, определяя необходимое для пользователя значение. Начальное значение сигнала устанавливается при объявлении сигнала в объекте или в архитектуре.

Пример :

```
Entity ex is
port(a: in  std_logic:=’0’; b,c: in  std_logic; q:  out bit);
end;
Architecture behv of ex is
signal il:  std_logic:=’1’;
signal i2:  std_logic:=’H’:
     signal i_vect: std_logic_vector(3 downto 0):=”0011”:
     begin
il<=a or b;
     end;
```

В вышеупомянутом примере входные сигналы инициализированы к следующим значениям:

Сигнал	**Значение**
a	‘0’
b	‘U’
c	‘U’
il	‘1’
i2	‘H’
I_vect	«0011»

Если, например, внутреннему сигналу i1 назначено значение ‘1’ в архитектуре, в момент 0 ns + 0 дельта (то есть непосредственно) i1 будет

иметь значение ‘0’ в течение дельта, а затем будет дано значение ‘0’ в момент 0 ns + 1 дельта.

Также допустимо определять начальные значения для сигналов режима out или inout. Если несколько сигналов должны быть инициализированы к тому же самому значению, это может быть выполнено в одной строке:

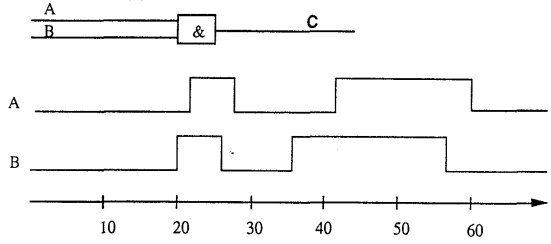
signal a,b: std_logic_vector(2 **downto** 0):=”001”;

Инструментальные средства синтеза не поддерживают начальные значения. Однако, большая часть инструментов принимает их существование в VHDL коде, но игнорирует их с предупреждением.

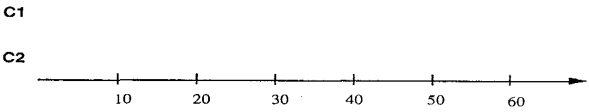
Имеется риск с начальными значениями, что VHDL моделирование и результат синтеза не будут одинаковы. В аппаратных средствах входной сигнал может иметь различное значение при запуске, что может привести к «несоответствию» между VHDL моделированием и моделированием в уровне вентилей. Если используется Std_logic и нет начальной установки, то сигнал будет инициализирован к 'U'. Если нет логического управления std_logic сигналом, другим сигналам, которые находятся в зависимости от сигнала std_logic, будут даны значения 'X'. Эта ошибка будет затем обнаружена в VHDL моделировании.

3.16 Упражнения

1. Что напоминает символ назначения сигнала?
2. Какие два режима задержки определены в VHDL стандарте?
3. Вывести выходные сигналы C1 и C2.



c1<=a and b after 10 ns;
c2<=transport a and b after 10 ns;



4 . Завершить следующие таблицы. Форма волны для сигнала дается в обоих примеры.

Architecture rtl of ex is

begin
b<=a;
c<=b;
end;

	0	**10**	**10+ldelta**	**10+2delta**	**20**	**20+ldelta**	**20+2delta**
A	0	1	1	1	0	0	0
B	0						
C	0						

Architecture rtl of ex is

begin
c<=b;
b<=a;
end:

	0	**10**	**10+ldelta**	**10+2delta**	**20**	**20+ldelta**	**20+2delta**
A	0	,	1	1	0	0	0
B	0						
C	0						

5 . а) Почему следующая строка VHDL неправильна?

q<= a when sel='0' else
b **when** sel='1';

б) Перезапишите код VHDL, без изменения поведения, так, чтобы это было правильно.

с) Действительно ли пример 5 (а) правилен в VHDL-93 стандарте?

6. Какие уровни ошибок существуют в команде **assert** и какие задаются по умолчанию?

7. Перезапишите код VHDL ,чтобы включить команду **assert**. Команда **assert** должна быть выведена «много раз» если вектор равен «111».

Architecture rtl of ex is
signal a: std_logic_vector(2 **downto** 0);
begin
a<=c and b;
end;

8. а) Что означают три объектных класса в VHDL?

б) Почему необходимы типы данных, отличающиеся от bit и bit_vector?

с) Какие два типа данных IEEE определены и какая разница между ними?

9. а) Для чего используется константа строки битов?

б) Для какого типа данных определена в VHDL-87 стандарте константа строки битов?

10. Может ли быть назначено значение целого числа с любым основанием системы счисления, и если да, то как?

11. Что такое - сектор массива?

12. Каково преимущество использования составного назначения для вектора в следующем VHDL коде? -

```
Architecture rtl of ex is
signal a: std_logic_vector(31 downto 0);
begin
a<=(others=>'l');
end;
```

13. Какое значение будет даваться вектору a после следующего назначения?

```
Architecture rtl of ex is
signal a,b: std_logic_vector(4 downto 0);
signal c: std_logic_vector(0 to 1);
begin
a<=(l=>'0', 3=>'1', others=>b(2));
b<=(l=>'1', 3=>'0', others=>c(l);
c<=»10»;
end;
```

14. а) Какое значение сигнала *a* будет установлено по умолчанию?

```
type mytype is ("T, 'R', '0', '1');
signal a: mytype;
```

б) Возможно изменить начальное значение, если да, то как?

4 ПОСЛЕДОВАТЕЛЬНЫЙ VHDL

4.1 Параллельная и последовательная обработка данных

В VHDL имеются конструкции языка, которые могут выполнять и параллельную и последовательную обработку данных.

Параллельные VHDL конструкции

- инструкция Process;
- когда еще инструкция;
- с инструкцией выбора;
- объявление Сигнала;
- инструкция Block.

Последовательные VHDL конструкции

- инструкция If-then-else;
- оператор выбора;
- переменное объявление;
- переменное назначение;
- инструкция Цикла;
- инструкция Возвращения;
- нулевая(пустая) инструкция;
- инструкции ждать.

Имеется также много VHDL команд, которые допустимы и в параллельных и в последовательных частях VHDL:

- назначение Сигнала;
- объявление типов и констант;
- Функция и Оператор контроля команд вызова;
- после задержки;
- сигнал атрибутов.

VHDL - проектировщику следует понимать в каких случаях следует использовать параллельные VHDL – конструкции, а в каких – последовательеные.

4.2 Назначение сигналов и переменных

Сигналы и переменные используются часто в коде VHDL, но они представляют полностью различные выполнения. Переменные используются для последовательного выполнения (подобно «нормальным» программам), в то время как сигналы используются для параллельного выполнения.

Назначения **переменной** и **сигнала** - «: = » и « < = », соответственно. Переменная может быть объявлена и использована только в последовательной части VHDL. Сигналы могут быть объявлены только в параллельной части VHDL, но использоваться могут, и в последовательной и в параллельной частях. Переменная может иметь те же типы данных, что и сигнал. Также допустимо назначать переменной значение сигнала и наоборот, при условии, что они имеют одинаковый тип данных. Следующая программа выполнена последовательно:

```
variable temp_a, diff: integer_type;
temp_a := in_l;
diff := temp_a - 2;
```

Предположим, что программа выполняется каждую дельта t. Табл.4.1 показывает, как изменяются переменные. Дельта t может быть произвольна.

Таблица 4.1- Последовательная инструкция назначения

Variable	T – delta t	T	T + delta t
In 1	2	3	3
Tempa	2	3	3
diff	0	1	1

Во время T-дельта-t, *in_1* имеет значение 2; *temp_a* - значение 2, diff значение (2-2)=0, и т.д. Это - традиционное программирование.

Если выше приведенная программа будет использоваться в параллельной части VHDL, то должны будут использоваться сигналы вместо переменных:

```
signal temp_a, diff : integer_type;
temp_a <= in_l;
diff <= temp_a - 2;
```

Табл. 4.2 показывает, как происходит назначение сигналов.

Таблица 4.2 - Параллельная инструкция назначения

Сигнал	**T-ΔT**	**T**	**T+ΔT**
in_1	2	3	3
temp_a	*Старое значение*	2	3
diff	*Старое значение*	*Старое значение*	0

Большая разница между сигналом и переменной в том, что сигналу назначается значение только после дельта задержки, в то время как переменной присваивают значение немедленно.

Обратите внимание также, что время не затрачивается в последовательной части VHDL. Только, когда используются инструкции wait или end (если используется список чувствительности) в последовательной части, временная синхронизация имитатора может выполняться. Это истинно, даже если имеются тысячи строк в последовательном коде:

Пример :

```
process(a,b ...)
variable c: std_logic_vector(l downto 0);
begin
if a=12 then ...—Строка 1
e<=»01»;  -- Строка 2
if b<=10 then  -- Строка 876
c:=»10»;  -- Строка 2876
end process;
```

Все строки в выше приведенном процессе будут выполнены в течение одной дельты. Строки в последовательном коде VHDL выполняются строка за строкой. Именно поэтому он называется последовательным VHDL. В сходящемся коде VHDL, строки выполняются только тогда, когда в списке чувствительности происходит событие.

Пример :

Sum1 и sum2 - сигналы: *Sum1 и sum2 - переменные:*

```
p0:process                    p1:process
begin                             variable sumi, sum2: integer;
   wait for 10 ns;                          begin
   suml<=suml+l;                            wait for 10 ns;
   sum2<=suml+l;                      suml:=suml+l;
end process;                      sum2:=suml+l;
                                  end process;
```

Таблица 4.3 - Сравнение сигналов с переменными

Time	**Sum 1**	**Sum 2**	**Time**	**Sum** 1	**Sum** 2
0	0	0	0	0	0
10	0	0	10	1	2
10+1 delta	1	1	10+1 delta	1	2
20	1	1	20	2	3
20+1 delta	2	2	20+1 delta	2	3
30	2	2	30	3	4
30+1 delta	3	3	30+1 delta	3	4

Как показывают вышеупомянутые примеры, режим полностью отличен в зависимости от того, используем ли мы сигналы или переменные. Вышеупомянутый пример не может синтезироваться, поскольку используется инструкция **wait for** 10 ns. Если бы это синтезировалось, разность в режимах после синтеза сохранилась бы. Поэтому очень важно знать, когда использовать сходящиеся команды и когда использовать последовательные. Не возможно сказать, вообще сигналы или переменные должны использоваться, решение полностью зависит того, какая функция требуется. Переменные, однако, не могут передавать информацию вне последовательной части VHDL, в которой они объявлены, в этом случае обрабатывается *p1* (процесс из предыдущего примера). Если необходим доступ к значению *sum1* или *sum2* как, например, выходного сигнала или в других частях архитектуры, они должны быть объявлены как сигналы или значение переменной, присвоенной сигналу.

Пример :

```
Entity ex is
        port(suml_sig, sum2_sig:  out integer);
end;
Architecture behv of rtl is
Begin
        pi:process
        variable sum1, sum2: integer;
begin
    wait for 10 ns;
    suml:=suml+l;
    sum2:=suml+l;
```

```
    suml_sig<=suml;
    sum2_sig<=sum2;
end process;
end;
```

Переменные могут хранить только временные значения внутри **процесса, функции** или **процедуры.**

Вышеупомянутое рассуждение относительно переменных было изменено в стандарте VHDL-93. В VHDL-93 были представлены глобальные переменные (**shared variables**), которые могут передавать информацию вне процесса.

Пример (VHDL-93):

```
    Architecture behv of ex is
    shared variable v: std_logic_vector(3 downto  0);
    begin
p0:process(a,b)
          begin
v:= a & b;
          end process;
p1 :process(d)
          begin
    if v=”0110” then c<=d;
    else
    c<=(others=>’0’);
    end if;
    end process;
     . . .
     end;
```

Глобальные переменные не доступны в параллельной части кода VHDL, но только внутри других процессов. И при этом глобальная переменная не может быть включена в список чувствительности процесса. Глобальные переменные могут вызвать недетерминизм. Комитет IEEE для VHDL пытается решить эту проблему. Когда в стандарте VHDL-93 были представлены глобальные переменные, проблема все еще не была решена. Инструментальные средства синтеза также не поддерживают глобальные переменные.

4.3 Оператор процесса

Концепция **процесса** исходит из программного обеспечения и может быть сравнена с последовательной программой. Если в архитектуре имеется несколько процессов, они выполняются одновременно. Процесс может быть в состоянии ожидания или выполнения (рис.4.1).

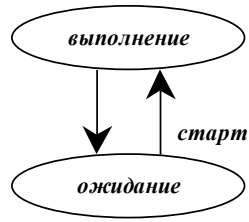


Рисунок 4.1- Состояние процесса.

Если процесс в состоянии ожидания, должно быть удовлетворено условие, например. **wait until** clk='l';. Это означает, что процесс начнется, когда clk имеет высокий уровень. Тогда процесс будет в состоянии выполнения. Как только код выполнился, будет ожидаться следующее увеличение уровня.

Синтаксис процесса:

[<имя_процесса> :]
process [(<список_чувствительности >)]
[<декларативная_часть>]
begin
<тело_процесса>
end process [<имя_процесса >];

Syntax (VHDL-93):

[<имя_процесса> :]
process [(<список_чувствительности >)] **[is]**
. . .
begin
. . .
end process [<имя_процесса>];

Как только процесс начался, требуется время delta t (минимальное разрешение моделирующего устройства) для того, чтобы перейти обратно в состояние ожидания. Это означает, что на исполнение процесса время не затрачивается.

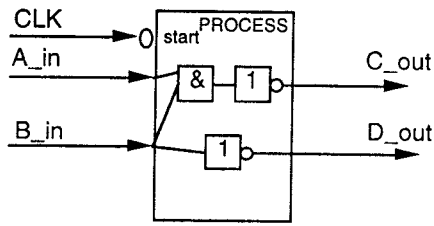


Рисунок 4.2- Малый процесс, начатый clk на низком уровне.

Процесс на рисунке 4.2 может быть описан следующим образом:

```
sync_process:
process
begin
    wait until clk='0';
    c_out <= not (a_in and b_in);
    d_out <= not b_in;
end process sync_process;
```

Процесс запускается, когда сигнал clk станет равен '0'. В процессе 'sync_process' две инструкции назначений значений сигналам, будут выполнены в дельта промежуток времени, который равен разрешению моделирующего устройства. На практике для выполнения операторов потребуются различные отрезки времени. Эта задержка в процессе может быть определена следующим образом:

```
c_out <= not(a_in and b_in) after 20 ns;
d_out <= not a_in after 10 ns;
```

Моделирующее устройство разместит результаты во временную очередь для *c_out* и *d_out*.

```
A_in = 1
B_In =  0
                C_out  =1 time=x +20ns
                D_out  = 1 time=x +10ns
```

Если *a_in* или *b_in* изменены, и clk низкого уровня, событие будет занесено в очередь относительно того времени, когда произошло изменение. Если это сравнивается с программной областью, имеется по крайней мере еще одно состояние, а именно состояние готовности. Готовность - состояние, в котором находится процесс, когда ему требуется ресурс. В электронике эта часть опущена, поскольку процесс уже имеет доступ к ресурсу. Следующая часть, в которой имеется различие - то, что процесс находится в состоянии исполнения в течение времени delta t. Время delta t используется, чтобы дать возможность моделирующему устройству обработать параллелизм. В действительности, время delta t равно нулю.

Существует два типа процессов в VHDL:

- Комбинационные процессы ;
- Синхронизированные процессы.

Комбинационные процессы используются для проектирования комбинационной логики в аппаратных средствах ЭВМ. Результатом синхронизированных процессов являются триггеры и возможно комбинационная логика.

В комбинационном процессе все входные сигналы должны содержаться в списке чувствительности (все направо от < = или в выражении if / case). Если сигнал не учтен, процесс не будет вести себя подобно комбина-

ционной логике в аппаратных средствах ЭВМ. В реальных аппаратных средствах, выходные сигналы могут изменяться, если один или большее количество входных сигналов комбинационного блока изменены. Если один из этих сигналов опущен из списка чувствительности, процесс не будет инициирован, когда значение опущенного сигнала изменено, так что в итоге новое значение не будет назначено выходным сигналам процесса. Стандарт VHDL допускает, чтобы сигналы были опущены из списка чувствительности. Только в конструкции имитационных моделей, а не для синтеза. Если сигнал опущен из списка чувствительности в конструкции VHDL для синтеза, VHDL моделирование и синтезируемые аппаратные средства ЭВМ будут вести себя по-другому. Это - серьезная ошибка, поскольку код VHDL, как предполагается, описывает аппаратные средства ЭВМ функционально.

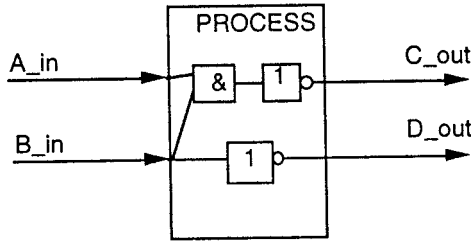


Рисунок 4.4 - Комбинационный процесс.

Процесс на рис.4.4 может быть описан следующим образом:

```
comb_process:process(a in.b in)
begin
    c_out <= not(a_in and b_in) after 20 ns;
    d_out <= not b_in after 10 ns;
    end process comb_process;
```

В скобках после оператора процесса располагается список сигналов, которые запускают процесс, если было изменение. Если мы опускаем *a_in* из списка чувствительности, *c_out* сохранит его значение и не модифицирует его, пока значение *b_in* не изменится. Это приведет к рассогласованию между моделированием VHDL и аппаратными средствами ЭВМ, описанными выше.

Комбинационные процессы хороши для проектирования потока данных, например, тракты данных в CPU.

Пример:

```
        process (a,b,c)
begin
        d <= (a and b) or c;
    end process;
```

Результат синтеза показан на рис.4.5.

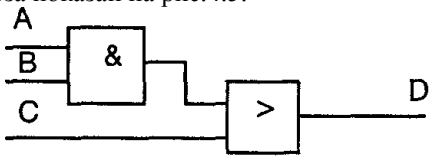


Рисунок 4.5 - Результат синтеза

В случае проектирования комбинационных процессов, всем выходным сигналам процесса присваиваются значения, каждый раз по выполнении процесса. Если это условие не удовлетворено, сигнал сохраняет свое значение. Инструмент синтеза будет воспринимать и решать это требование, создавая триггер на выходе.

Функционально, код VHDL и аппаратные средства ЭВМ будут идентичны. Цель комбинационного процесса состоит в том, чтобы генерировать комбинационную логику. Если введены триггеры, временные характеристики ухудшаются, и число вентилей увеличится. Решение является простым: занести все сигналы, которые «считываются» внутри процесса, в список чувствительности для комбинационных процессов.

Синхронизированные процессы - синхронные, и несколько таких процессов могут запускаться одними и теми же синхроимпульсами. Известно, что никакой синхронизированный процесс не может начаться, пока не приходит фронт синхроимпульса *(clk).* Выходной сигнал *d_ out* процесса А, связан со входом процесса В. Требование состоит в том, что *d_out* должен быть устойчив прежде, чем синхроимпульсы начинают процессы. Самое длинное время, которое потребуется процессу В, чтобы дать сигналу *d_out* устойчивое значение определяет самый короткий период для синхроимпульса. В моделирующих устройствах VHDL это время равно 10 ns.

Рассмотрим пример (рис.4.6)

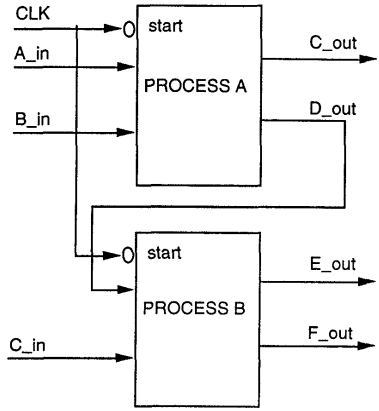


Рисунок 4.6 - Процессы, синхронизированные
одним и тем же синхросигналом

Этот пример можно выразить следующей VHDL-программой:

```
A_process:process
begin
        wait until clk='0';
    c_out <= not(a_in and b_in);
    d_out <= not b_in after 10 ns;
end process A_process;
B_process: process
Begin
wait until clk='0';
        e_out <= not(d_out and c_in);
f_out <= not c_in;
end process B_process;
```

Эти два процесса могут формировать **компонент** (рис. 4.7).

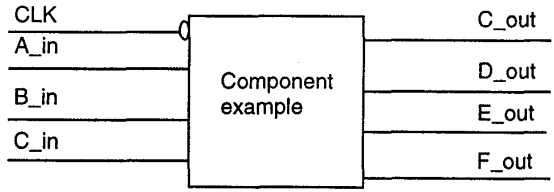


Рисунок 4.7 - Процессы A и B, скрытые в компоненте

```
Entity comp_ex is
                port( clk, a_in, b_in, c_in : in      std_logic;
                 d_out                    : out     std_logic;
```

```
                    c_out, e_out, f_out  : out        std_logic);
end,
        Architecture rtl of comp_ex  is
begin
        A_process:process
    begin
    end process A_process;
     B_process: process
    begin
    end process B_process;
     end;
```

Синхронизированные процессы приводят к тому, что все сигналы, назначенные внутри процесса, в результате порождают триггер**.** Пример синхронизированного процесса:

```
example:process
begin
        wait until clk='1';
    dout <= din;
    end process example;
```

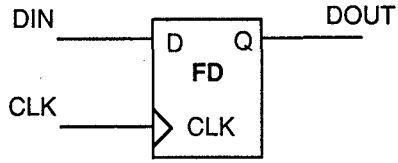


Рисунок 4.8 - Результат синтеза

Предшествующий пример показывает, как синхронизированный процесс преобразован в триггер. Как быстро при синтезе *din* будет преобразован в *dout*, зависит от технических характеристик триггера

Переменные могут также вызывать триггер в синхронизированном процессе. Если переменная считывается прежде, чем ей назначено значение, это приведет к триггеру для переменной.

Пример:

```
    process
    variable count: std_logic_vector( 1  downto 0);
begin
    wait until clk='1';
    count:=count + 1;
    if count="11"  then  q<='1'
    else q<='0';
    end if;
    end process;
```

В вышеупомянутом примере результат синтеза создаст три триггера. Один для сигнала *q* и два для переменной ***count..*** Это потому что мы считываем переменную *count* прежде, чем мы назначаем ей значение (*count*: = *count* + 1;).

Если сигналу не назначено значение в синхронизированном процессе, то сигнал сохранит старое значение. Результат синтеза при таком описании приведет к обратной связи. Выходной сигнал триггера через мультиплексор попадает обратно на вход. Этот метод приводит к схеме, остающейся синхронной и контролируемой. Альтернативно, синхронизация могла бы задаваться сигналом *en* (рис.4.9). Однако, эта альтернатива намного хуже, чем первая с точки зрения методологии конструирования и контролируемости.

Пример:

```
process
begin
wait until clk='1';
if en='1' then
q<=d;
end if;
end process;
```

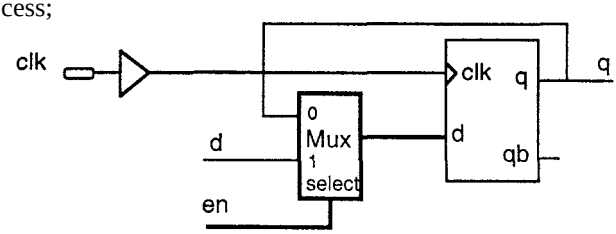


Рисунок 4.9 - Синхронная схема.

Пример, который не очень хорош, но который является все еще допустимым кодом VHDL:

```
clk2<=clk and en;
process
begin
wait until clk2='1';
q<=d;
    end process;
```

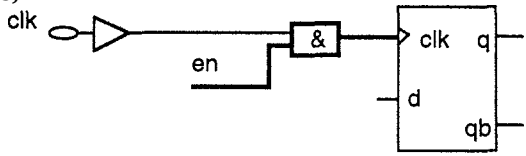


Рисунок 4.10 - Вентильная синхронизация.

Синхронизированный процесс может приводить как к триггерам, так и к комбинационным логическим элементам. Это приводит к меньшему и более легко читаемому коду, и иногда к лучшему результату синтеза.

Пример:

```
        process(clk.resetn)
        begin
if reset='1' then
        q<=(others=>'0');
elsif clk'event and clk='1' then
        if en='l' then
    q<=a + b;
end if;
end if;
    end process;
```

Все логические элементы, вызванные назначением сигналов в синхронизированном процессе, заканчиваются слева от триггера (перед входом триггера). Поэтому вышеупомянутый пример будет иметь сумматор и мультиплексор на входе триггера.

Это приводит к логической схеме, показанной на рис.4.11, то есть логический блок Bl включен в синхронизированный процесс, который описывает триггер FD1. Код VHDL, который описывает логический блок B2, с другой стороны, должен быть включен в комбинационный процесс.

Однако возможно логический блок B2 также включать и в синхронизированный процесс, но это не желательно.

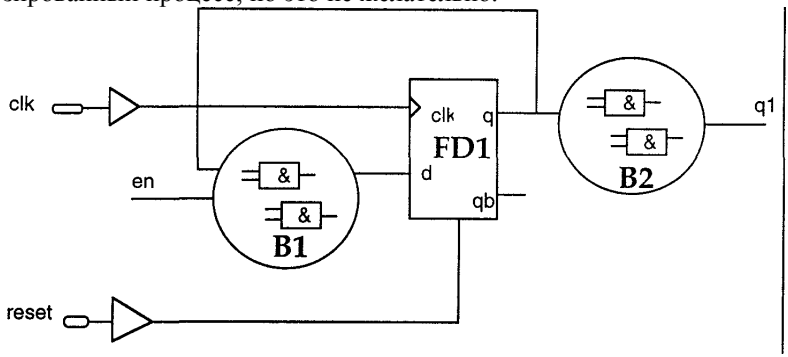


Рисунок 4.11 - Синхронизированный / комбинационный логический блок.

Код VHDL для рис 4.11 должен быть структурирован следующим образом:

```
Architecture rtl of ex is
signal q: std_logic;
begin
FD1_B1 :process(clk,reset)
```

```
    begin
if reset='1' then  q<='0';
elsif clk'event and clk==T then
if en='l' then
          q<=....  - булево выражение(Bl);
          end if;
end if
          end process;
          ql<=q and ...  -- булево выражение (B2)
          end;
```

Логический блок B2 мог быть описан и отдельно, в комбинационном процессе.

4.4 Условный оператор if

Условный оператор **if** выбирает одну или ни одной инструкции из последовательности инструкций. Выбор зависит от одного или нескольких условий. Команда **if** сходна с командой **when else** в параллельной части VHDL.

Синтаксис:

if <условие> **then** <последовательность операций>
[**elsif** <условие> **then** < последовательность операций >] ....
[**else** < последовательность операций >]
end if;
<условие>= <выражение> [<операция отношения> <выражение>]..
< операция отношения >= ('=', '/=', '<', '<=', '>', '> =')

Пример:

```
if sel = '1' then c<=b;
else c<=a;
end if;
```

Результат синтеза показан на рис 4.12.

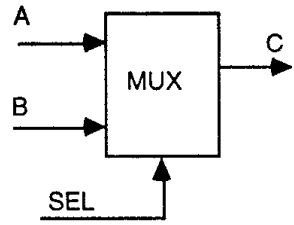


Рисунок 4.12 - Результат синтеза

Пример:

```
if sel = '1' then  c<='1'; …
elsif  John = "1001" then
```

```
    c <= '0'; ...
    elsif  David > John then
           c<='l';...
else
           c <= 'X';...
end if;
```

Допускается несколько конструкций типа **elsif,** но **else** допускается только одно**.** Обратите внимание на **ELSIF** и **END IF.**

Пример:

```
Entity ex_if is
  port( ssel:  in  std_logic;
  a,b,syn: in  std_logic;
  sout:  out std_logic);
end;
Architecture rtl of ex_if is
begin
    process(ssel,syn,a,b)
    begin
    if ssel='0' and syn=T then sout<=a;
    else
    if ssel='l' and syn='0' then sout<=b;
    else
    sout<='0';
    end if;
    end if;
end process;
 end;
```

4.5 Оператор выбора case

Оператор **case** соответствует команде **with select** параллельной части VHDL. Использование оператора выбора часто делает код VHDL более читабельным.

Синтаксис:

case < выражение > **is**
when < вариант > => < список операций>;
when < вариант > => < список операций >);
. . .
[when others => [<список операций >] ;] **end case;**
< вариант >= < вариант > [| < вариант > [...]]
| = or

Все варианты в операторе **case** должны быть перечислены. Не допустимо чтобы один и тот же вариант, был включен в описание нескольких

инструкций **when**, то есть был перечислен несколько раз. Это также означает, что варианты не должны перекрываться. Если значение выражения содержит много значений, может быть трудно перечислить все из них. В этом случае, может использоваться вариант **others**.

Пример:

```
Entity case_ex2 is
port( a: in integer range to30;
     b: out integer range Oto6);
end;
Architecture rtl of case_ex2       is
begin
pl:process(a)
          begin
case a is
                    when 0  =>       b<=3;
                    when 1 I 2=>    b<=2;
                    when others => b<=0;
    end case;
    end process;
end;
```

Тип данных std_logic обычно используется при конструировании. Так как std_logic может принимать девять различных значений, все значения должны быть охвачены в операторе выбора **case**.

Пример:

```
Entity case_ex3 is
port( a,b,sel: in  std_logic;
  c:  out std_logic);
end;
Architecture rtl of case_ex3 is
begin
          pl:process(sel,a,b)
          begin
case sel is
    when '0' =>  c<=a;
    when others=> c<=b;
     end case;
end process;
end;
```

Если входной сигнал *sel* в вышеупомянутом примере принимает значение отличное от '0', сигналу *c* будет назначено значение сигнала *b*. В VHDL моделировании оно может быть одним из других восьми значений, которые может принимать сигнал типа std_logic. В аппаратных средствах

ЭВМ это может быть только значение '1'. Инструмент синтеза понимает, что аппаратные средства ЭВМ могут только принимать значения '0' и '1', который ведет к правильному результату синтеза, то есть мультиплексору. Также допустимо определить диапазоны в списке вариантов. Это можно сделать, используя **to** или **downto.**

Пример:

```
Entity case_ex4 is
     port( a: in  integer range 0 to 30;
     q: out integer range 0 to  6);
end;
Architecture rtl of case_ex4 is
begin
          pl:process(a)
begin
                                                  case a is
                              when 0  => q<=3;
                              when 1 to 17  => q<=2;
    when 23 downto 18=> q<=6;
     when others  => q<=0;
     end case;
end process;
end;
```

Не допустимо указывать диапазон для векторов, поскольку для них не существует понятия диапазона.

Пример (неправильно):

```
     Entity case_ex5 is
 port( a:  in  std_logic_vector(4 downto 0);
             q: out std_logic_vector(2  downto 0));
end;
          Architecture rtl of case_ex5 is
begin
          pl:process(a)
          begin
    case a is
    when “00000”  => q<=”011”;
    when “00001” to “11110”=> q<=”010”; -- Ошибка
    when others  => q<=”000”;
    end case;
    end process;
end;
```

Если требуется указание диапазона, std_logic_vector сначала должен быть преобразован к целому числу. Это может быть сделано, путем объявле-

ния переменной типа *integer* в процессе и последующим преобразованием вектора с использованием конверсионной функции conv_integer. Так же поступают, если необходимо использовать вектор, как вариант целого числа.

Пример:

```
    Entity case_ex6 is
      port( a:  in  std_logic_vector(4 downto 0);
      q:  out std_logic_vector(2 downto 0));
    end;
    Architecture rtl of case_ex6    is
begin
    pl:process(a)
    variable int: integer range 0 to 31;
    begin
    int:=conv_integer(a);
    case int is
        when 0 =>  q<=”011”;
        when 1 to 30=> q<=”010”;
        when others => q<=”000”;
    end case;
    end process;
    end;
```

На результат синтеза не воздействуют при помощи конверсионной функции. Решение относительно того, должна ли конверсионная функция использоваться, должно быть основано на документационном аспекте и на том, не проще ли записать код VHDL, если в списке выбора используется целое число.

Если вы хотите включить несколько векторов в список вариантов в инструкции **case**, не допустимо использовать конкатенацию для объединения векторов в один. Это потому, что <выражение> оператора **case** должно быть статическим.

Пример неправильного кода:

```
Entity case_ex7 is
port( a,b: in std_logic_vector(2 downto       0);
  q: out std_logic_vector(2 downto       0));
end;
Architecture rtl of case_ex7 is
begin
pl :process(a,b)
     begin
                    case a & b is  -- Ошибка
                              when “000000” => q<=”011”;
                              when “001110”=> q<=”010”;
```

```
        when others => q<="000";
    end case;
end process;
end;
```

Решение должно или представлять в процессе переменную, которой назначена величина, равная *a* & *b*, или использовать спецификатор для подтипа.

Пример:

```
Architecture rtl of case_ex8 is
begin
pl:process(a,b)
    variable int: std_logic_vector(5 downto 0);
    begin
    int:=a & b;
    case int is
        when "000000" => q<="011";
        when "001110" => q<="010";
        when others => q<="000";
    end case;
    end process;
end;
```

Пример:

```
Architecture rtl of case_ex9 is
begin
pl:process(a,b)
    subtype mytype is std_logic_vector(5 downto 0);
    begin
        case mvtype'(a & b) is
        when "000000" => q<="011"
        when "001110" => q<="010"
        when others =>  q<="000"
     end case;
     end process;
end;
```

Допустимо в выражении выбора непосредственно указывать подтип.

4.6 Множественное назначение

В параллельной части VHDL вам всегда дано задающее устройство для назначения каждого сигнала, которое обычно не желательно. В последовательной части VHDL возможно назначить тот же самый сигнал несколько раз в том же самом процессе без учета нескольких задающих устройств для сигнала. Этот метод назначения сигнала может быть использован для того, чтобы значение сигналам назначать в процессе по умолча-

нию. Тогда это значение может быть перезаписано другим сигналом назначения. Следующие два примера являются идентичными в отношении и моделирования VHDL и результата синтеза.

Пример:

```
        Architecture rtl of exi is
        begin
        pl:process(a)
        begin
        case a is
    when “00” =>  ql<=’1’;
      q2<=’0’;
      q3<=’0’;
    when “10” =>  ql<=’0’;
      q2<=’1’;
      q3<=’1’;
    when others =>  ql<=’0’;
      q2<=’0’;
      q3<=’1’
end case;
    end process;
    end;
```

Пример:

```
        Architecture rtl of ex2 is
begin
        pl:process(a)
        begin
    q1<=’0’;
    q2<=’0’;
    q3<=’0’;
    case a is
      when “00” =>  ql<=’1’;
      when “10” =>  q2<=’1’;
      q3<=’1’;
    when others => q3<=’1’;
    end case;
    end process;
    end;
```

Если мы сравним вышеупомянутые примеры, мы увидим, что в примере 2 имеется меньшее количество назначения сигналов. Тот же самый принцип может применяться, например, при использовании **if-then-else**. Объяснение этого состоит в том, что время внутри процесса не затрачивается, то есть назначения сигнала только перезаписывают друг друга в по-

следовательном коде VHDL. С другой стороны, если тот же самый сигнал назначен в различных процессах, сигналу будет соответствовать несколько задающих устройств.

4.7 Нулевая инструкция null

В VHDL имеется инструкция, которая означает «не делать ничего». Эта команда, например, может использоваться, если заданные по умолчанию сигнальные назначения использовались в процессе, а альтернатива в операторе **case** не должна изменить это значение.

Пример:

```
        Architecture rtl of ex is
        begin
        pl:process(a)
        begin
    q1<='0';
    q2<='0';
    q3<='0';
    case a is
    when "00" => ql<='1';
    when "01" =>       q2<='1';
    q3<='1';
    when others => null:
    end case;
    end process;
    end;
```

В вышеупомянутом примере инструкция **null** может быть опущена, но читаемость кода улучшается, если команда **null** присутствует. Если **null** опускается, существует опасность того, что если кто - то еще прочитает код VHDL, он будет неуверен в том, не забыл ли VHDL проектировщик сделать сигнальное назначение и должна ли строка быть пустой.

4.8 Оператор задержки

Имеется четыре различных пути описания оператора задержки в процессе:

1. process(a,b);
2. wait until a=l;
3. wait on a,b;
4. wait for 10 ns;

Первый и третий способы идентичны, если wait on a, b помещен в конце процесса. Следующие два процесса идентичны.

Пример 1:	***Пример 2:***
Process(a,b)	**Process**

```
begin                                   begin
   if a>b then                          if a>b then
      q<='1';                           q<='1';
    else                            else
     q<='0';                            q<='0';
    end if;                             end if;
end process;                            wait on a,b;
                                    end process;
```

В примере 1 процесс будет вызываться каждый раз, когда сигналы a или b изменяют значение (a'event или b'event). Чтобы быть идентичным примеру 1, строка **wait on** a,b; должна быть помещена в конце примера 2, потому, что никакие процессы не запускаются, пока не достигнут оператора задержки. Если бы использовался список чувствительности, согласно VHDL стандарту, размещение **wait on** по-прежнему было бы в конце процесса. Если **wait on** помещен куда-нибудь еще, значение выходного сигнала будет отлично, когда начнется моделирование (см. пример задержки ниже).

Если в процессе используется список чувствительности, не допустимо использовать команду **wait**. Однако допустимо иметь несколько команд ожидания в одном и том же процессе.

Строка **wait until** *a*='1'**;** означает, что, для оператора **wait** условие должно быть удовлетворено, и тогда продолжится выполнение кода, необходимо чтобы сигнал нес сообщение, то есть изменил значение, и его величина должна стать равной '1', то есть высокий фронт для сигнала *a.*

Wait on *a, b*; выполниться, когда или сигнал *a*, или *b* изменят свое значение.

Wait for 10 ns; означает, что моделирующее устройство будет ждать 10 нс, прежде чем продолжить выполнение процесса. Также допустимо использовать команду **wait for** следующим образом:

```
constant period:time:=10 ns;
wait for 2 * period;
```

Альтернативы 2, 3 и 4 могут также быть объединены в: **wait on** a **until** b='1' for 10 ns, но способ 1 никогда не должен быть объединен с 2, 3 или 4. Предположим, что используется следующая команда:

```
wait until a='1' for 10 ns;
```

Условие ожидания будет удовлетворено, когда сигнал *a* изменит значение или после ожидания более чем 10 нс. Поэтому условие может быть расценено как условие **или**, то есть ждать, пока сигнал *a* не изменится либо продолжать через 10 нс, если сигнал все еще не изменился.

Ниже следует несколько примеров различных команд задержки и результаты их моделирования на рис 4.13. Чтобы гарантировать лучшее понимание примера, повторим необходимый материал VHDL:

Во время 0 всем сигналам назначены значения type'left. Это означает, что каждому сигналу дают значение, являющееся самым левым в объявленном типе данных. Таким образом, сигналу типа **bit** дают значение '0', поскольку bit объявлен как тип - ('0', '1');. Все процессы также начнут выполняться во время 0. Затем все процессы будут выполнять строку за строкой, пока не будет достигнут оператор задержки.

Как мы видим в примере задержки ниже, примеры 1 и 2 идентичны. В примере 3 wait on, не был помещен в конце процесса, что привело к отличному, от предыдущих примеров, значению для выходного сигнала c3. В примере 4 условие ожидания истинно только в том случае, если сигнал имеет положительный фронт и дается значение '1'. Инверсия '1' -это '0', так что c4 будет назначено значение '0' во время 20 ns и 47 ns. В примере 5, который является наиболее комплексным, условие ожидания - true после 10 ns или при изменении сигнала *a*.

Примеры (a,c1..c5=bit):

Пример 1

```
process(a)
        begin
        cl<=nota;
    end process;
```

Пример 2

```
process
        begin
c2<=nota;
    wait on a;
    end process;
```

Пример 3

```
process
        begin
wait on a;
        c3<=not a;
        end process;
```

Пример 4

```
process
        begin
wait until a='1';
                                                c4<=not  a;
        end process;
```

Пример 5

```
process
        begin
c5<=not a;
```

wait until a='1' for 10 ns;
end process;

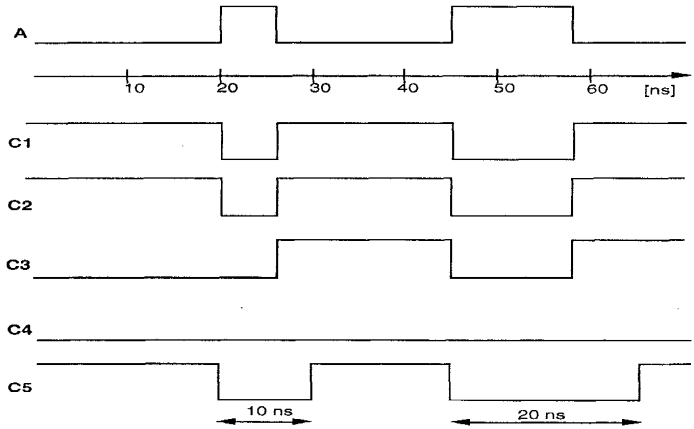


Рисунок 4.13 - Результаты моделирования

Инструментальные средства синтеза не поддерживают использование команды **wait for** 10 ns;. Этот метод описания при синтезе приводит к ошибке. Более того, большинство инструментальных средств синтеза допускают только список чувствительности, то есть может синтезироваться пример 1, но не пример 2 или пример 3. Некоторые расширенные системы, однако, могут синтезировать пример 2. Пример 4 - синхронизированный процесс и после синтеза приводит к D-триггеру. Примеры 3 и 5 могут использоваться только для моделирования, а не для конструирования. Как упоминалось ранее, допустимо иметь несколько задержек внутри одного и того же процесса.

Пример:

```
Process
    begin
    wait until clk='1';
        …
wait until clk='1';
        …
wait until clk='1';
        …
end process;
```

Этот метод описания воспринимается только несколькими инструментальными средствами синтеза, которые также требуют, чтобы в про-

цессе использовался только один синхросигнал (wait until clk='l';) и чтобы один и тот же уровень синхроимпульса использовался во всех командах задержки. Результат синтеза будет отражать состояние модели, при помощи команд, которые прибывают после первого оператора задержки, выполняемого после первого фронта синхроимпульса , команд, которые прибывают после второго оператора задержки после второго фронта синхроимпульса, и так далее. Поскольку полная команда процесса должна быть расценена как бесконечный цикл, то когда будет достигнута строка **end process**;, «программный указатель» перейдет опять к началу процесса и продолжит выполнение.

Команда задержки - последовательная команда несмотря на факт, что не допустимо использовать задержку в функциях. С другой стороны, команда wait может использоваться в процедурах и процессах.

4.9 Операторы цикла

Имеются две различные инструкции для описания цикла в последовательном VHDL:

- For loop;
- While loop.

Оператор For loop

Синтаксис:

[Метка:] **for** <идентификатор> **in** <дискретная область> **loop**
<последовательность операций>
end loop [Метка];

Команда цикла очень полезна, когда приходится использовать одно- или двумерные вектора.

Пример:

```
Entity ex is
   port(a,b,c: in   std_logic_vector(4 downto  0);
   q:  out     std_logic_vector(4 downto 0));
end;
Architecture rtl of ex is
begin
   process(a,b,c)
   begin
   for i in 0 to 4 loop
   if a(i)='1'  then
q(i)<=b(i);
else
q(i)<=c(i);
end if;
```

```
        end loop;
end process;
end;
```

Не допустимо изменять значение величины индексной переменной в процессе. Расширенные служебные программы поддерживают команду **for loop**. Они требуют, чтобы был установлен диапазон изменения индексной переменной.

Диапазон в **for loop** может быть опущен, таким образом цикл превращается в бесконечный. Однако цикл можно прервать, используя инструкцию **exit** внутри него. Также каждой команде цикла loop можно сопоставлять метку.

Пример:

```
Architecture rtl of ex is
begin
    process
    begin
    reset loop: loop
        q<=(others=>'0');
        wait until clk='1';
        exit reset loop when resetn='0';
    main loop: loop
    wait until clk='l';
    exit reset loop when resetn='0';
    q<=a+b;
    wait until clk='l';
    exit reset loop when resetn='0';
    while en='0' loop
    exit reset loop when resetn='0';
    end loop:
    wait until clk='l';
    exit reset loop when resetn='0';
    q<=b+c;
    end loop;
    end loop;
    end process,;
end;
```

Этот метод описания широко используется в поведенческом синтезе.

Оператор while loop

Синтаксис:

[Метка:] **while** <условие> **loop**
<последовательность операций>
end loop [Метка];

Пример:

```
    process(a,b,resetn)
    variable i: integer range 0 to 31;
    begin
    if resetn='0' then
    i:=0;
    q<=(others=>'0');
    else
    while q<12 loop
        if i=31 then
        exit
        else
        i<=i+l;
        q<=a(i) + b(i) +q;
end if;
                                                    end loop;
end if;
end process;
```

Только расширенные инструментальные средства синтеза поддерживают цикл с условием продолжения (**while loop**). Вышеупомянутый пример может только моделироваться, но не синтезироваться.

4.10 Отложенный процесс

Отложенный процесс - новая концепция, которая была представлена в стандарте VHDL-93. При использовании отложенного процесса, возможно заставить процесс выполняться в последний дельта-цикл отведенного времени. Отложенный процесс обычно инициируется списком чувствительности или инструкцией задержки. Разница в том, что отложенный процесс не начинает выполняться в дельта событие, которое инициирует процесс, а ждет, пока все сигналы установятся, то есть не имеется больше дельта перечислений в рассматриваемое время. Это означает, что процесс будет выполнен как последнее событие в это время. Если используются несколько отложенных процессов, VHDL стандарт -93 не определяет, какой должен выполнять сначала. Поэтому использования нескольких отложенных процессов следует избегать.

Отложенный процесс не поддерживается при синтезе. Прежде всего это добавление к новому стандарту было важным для проектирования имитационных моделей, по причине того, что в VHDL не допустима запись: **wait for** 0 ns;. С **отложенным процессом** теперь возможно гарантировать, что сигналам будут присвоены значения в самом последнем дельта цикле, облегчая возможность описания некоторых имитационных моделей.

Пример:

```
    Architecture behv of ex is
        begin
        process
        begin
            …
end process;
        postponed process (a.b)
begin
        if a>b then
    q<=’l’ after 5 ns;
        else
q<=’0’ after 5 ns;
        end if;
        end process;
    end;
```

Все назначения сигналам в **отложенном процессе** должны иметь задержки типа **after**.

4.11 Предопределенные сигнальные атрибуты

Сигнальные атрибуты очень полезны, когда приходится создавать VHDL модели. В синтезируемом VHDL коде нужно только использовать атрибут 'even, чтобы проверить, имеется ли фронт синхроимпульса. Некоторые инструментальные средства синтеза также используют ' last_value, чтобы решить, имеется ли фронт синхроимпульса. Атрибут - информация, которая может быть получена из блоков, массивов, сигналов и типов.

Синтаксис:

<имя>' атрибут

Самые обычные сигнальные атрибуты следующие:

<сигнал>'event

возвращает значение true или false, в зависимости от того, произошло ли событие в настоящем периоде дельта времени.

<сигнал>'active

возвращает значение true или false в зависимости от того, произошло ли действие в настоящем периоде дельта времени.

<сигнал>’stable(t)

возвращает значение сигнала true или false в зависимости от того, помещается ли событие в (t) единицах времени.

<сигнал>’quiet(t)

возвращает значение true или false в зависимости от того, располагается ли действие в (t) единицах времени.

<сигнал>'transaction

возвращает событие всякий раз, когда имеется действие на сигнал.

<сигнал>’delay(t)

возвращается сигнал, задержанный на (t) единиц времени.

<сигнал>'last_event

возвращает количество времени, прошедшее с момента последнего события.

<сигнал>'last_active

возвращает количество времени, прошедшее с момента последнего действия.

<сигнал>'last_value

возвращает значение, равное предыдущему значению.

<сигнал>'range

возвращает диапазон сигналов, например:

signal a:std_logic_vector(3 **downto** 0);

for a’range loop - - ‘range = 3 downto 0

. . .

end loop;

Примеры атрибутов для типов:

<тип>'left

возвращает левое число массива.

<тип>'right возвращает правое число массива.

<тип>'high

возвращает максимальное число массива.

<тип>'low

возвращает минимальное число массива.

<тип>'length

возвращает длину типов данных.

Пример:

subtype my_typel **is** integer **range** 15 **downto** 0;

subtype my_type2 **is** integer **range 0 to** 15;

my_typel’left = 15 my_type2’left = 0

my-type 1’right = 0 my_type2’right = 15

my_typel’high = 15 my_type2’high = 15

my_typel’low = 0 my_type2’low = 0

my _typel ‘length = 16 my_type2’length = 16

<тип>'succ(значение данных)

возвращает последующее значение в типе данных, соответствующему значению данных.

<тип>'pred(значение данных)

возвращает предыдущее значение в типе данных соответствующему значению данных.

<тип>'rightof(значение данных)

возвращает значение справа от значения данных.

<тип>'leftof(значение данных)

возвращает значение слева от значения данных.

Пример:

```
type my_type3 is (John, Tom, Petra);
type my_type4 is my_type3 range Tom downto John;
    my_type3'succ(John) = Tom
    my_type4'succ(John) = Error
    my_type3'pred(John) = Error
    my_type4'pred(John) = Tom
    my_type3'rightof(John) = Tom
    my_type4'rightof(John) = Error
    my_type3'leftof(John) = Error
    my_type4'leftof(John) = Tom
```

4.12 Описание синхронизации

Имеется много различных методов описания синхроимпульса для триггера в синхронизированном процессе. Самые общие (обычные) методы рассмотрены ниже.

```
Alt 1: process(clk)
begin
        if clk'event and clk='1'  then
    q<=d;
        end if;
end process;
Alt 2: process(clk)
begin
        if clk='l' then
    q<=d;
    end if;
    end process;
    Alt 3: process(clk)
    begin
        if clk'event and clk='1'  and clk'last_value='0' then
    q<=d;
        end if;
end process;
Alt 4: process begin
    wait until clk='1';
    q<=d;
end process;
```

Alt 5: process
begin
wait until prising(clk);
q<=d;
end process;

Выбор метода описания зависит от инструмента синтеза, который должен использоваться. Табл.4.4 суммирует данные о четырех самых обычных инструментальных средствах синтеза и том, что они поддерживают.

Таблица 4.4 - Различные описания синхронизации и что они поддерживают

	Synopsys	**Autologic 1**	**Autologic 2**	**View Logic**
Alt 1	Yes	No	Yes	Yes
Alt 2	Yes	No	Yes	No
Alt 3	No	Yes	Yes	No
Alt 3	Yes	Yes	Yes	Yes
Alt 5	No	Yes	Yes	Yes

Только Alt3 гарантирует, что фронт синхроимпульса будет идти от '0' до '1' в VHDL моделировании. Другие процессы принимают фронт синхроимпульса от 'X' до '1', например.

4.13 Асинхронный и синхронный сброс

Имеются два различных способа сброса триггера: асинхронный или синхронный сброс.

Асинхронный сброс

Триггер с асинхронным сбросом будет сброшен, как только сброс активизирован, независимо от синхросигналов. Это означает это, что асинхронный сброс должен быть включен в список чувствительности также как синхросигнал (alt 1). С точки зрения документации легко увидеть, какой сброс имеет синхронизированный процесс, асинхронный или нет: вы только должны проверить действительно ли имеется сброс в списке чувствительности. Общий метод описания, который поддерживается большой частью инструментальных средств синтеза.

С точки зрения методологии конструирования и документации только один сигнал должен использоваться для синхронизации или сброса. Инструментальные средства синтеза не поддерживают несколько синхроимпульсов в одном и том же процессе. Если необходимо несколькими сигналами сбросить триггер, то для того, чтобы можно было синтезировать модель, они должны быть пропущены вместе вне синхронизированного процесса.

Синхронный сброс

При использовании синхронного сброса триггер может быть сброшен только при активном фронте синхроимпульса. Это означает, что, если в синхронизированном процессе используется список чувствительности, только синхросигнал должен быть включен в список чувствительности (alt 1). Общий метод описания, который поддерживается большой частью инструментальных средств синтеза:

```
Alt 1:  process(clk)
begin
        if clk'event and clk='1' then
    if reset='1' then
    data<="00";
    else
    data<=not in_data;
    end if;
    end if;
    end process;
```

Альтернативный метод описания, поддерживаемый View Logic :

```
    Alt 2:  process
    begin
    wait until prising(clk);
        if reset='1' then
data<="00";
        else
    data<=not in_data;
    end if;
    end process;
```

Результат синтеза обоих методов показан на рис. 4.14.

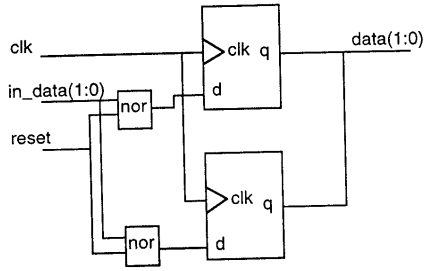


Рисунок 4.14 - Синхронный сброс

4.14 Защелки

Если в VHDL должны быть описаны защелки, обычно используется процесс. Нормальная защелка (прозрачная защелка) передает значение от d-входа q-выходу, когда отпирающий сигнал активен. Если отпирающий сигнал деактивирован, выход будет хранить значение. В VHDL защелки

описываются при помощи комбинационных процессов. Все сигналы, которые считываются в процессе, должны быть в списке чувствительности. Выходу защелки должно быть назначено значение только тогда, когда один из сигналов из списка чувствительности активен. Это приводит к защелкам после синтеза.

Пример:

```
        process(enable,d_in)
begin
        if enable='1' then
    q<=d_in;
    end if;
    end process;
```

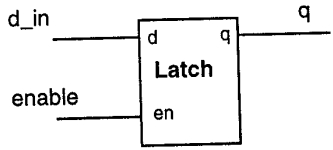


Рисунок 4.15 - Защелка

4.15 Упражнения

1. Какие два главных типа процессов вы знаете?
2. Какую логику они представляют?
3. Что характеризует неполный комбинационный процесс?
4. Как можно избежать неполных комбинационных процессов?
5. Почему существуют различные методы описания фронта синхроимпульса?
6. Приведите пример по крайней мере трех альтернатив.
7. Сколько триггеров вызывает следующий код VHDL?

а)

```
  Architecture rtl of ex is
  signal a,b: std_logic_vector(3 downto 0);
  begin
  process(clk)
  begin
if clk='1' and clk'event then
        if q(3)/='1' then
q<=a + b;
  end if;
  end if;
  end process;
  end;
```

б)

```
Architecture rtl of ex is
signal a,b: std_logic_vector(3 downto 0);
begin
process(clk)
variable int: std_logic_vector(3 downto 0);
begin
if clk='l' and clk'event then
if int(3)/='1' then
    int:=a + b;
    q<=int;
end if;
end if;
end process;
end;
```

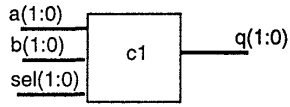
8. Каковы два пути сброса триггера?
9. Если для синхронизированного процесса используется список чувствительности, какой тип возврата должен иметь сигнал возврата в списке чувствительности?
10. Скольким триггерам будет дан синхронизированный сброс и скольким асинхронный в следующем VHDL коде ?

```
    process(clk,resetn)
begin
    if resetn='0' then
  ql<='0';
q2<='0';
    elsif clk'event and clk='1' then
ql<=a and b;
2<=c;
end if;
end process;
process(clk) begin
if clk='1' then
if resetn='0' then
q3<='0';
else
q3<=q2;
end if;
end if;
end;
```

11. Сколько защелок произведет следующий код VHDL?

```
        Architecture rtl of ex is
signal a,b,c,d,e: std_logic_vector(l downto 0);
        begin
    process(c,d,e,en)
    begin
    if en='1' then
    a<=c;
    b<=d;
    else a<=e;
        end if;
end process;
        end;
```

12. Спроектируйте компонент со следующими входами и выходами:



Компонент должен иметь следующий режим:

Sel	*q*
00	a **xor** b
01	a **or** b
10	a **nor** b
11	a **and** b
others	**"XX"**

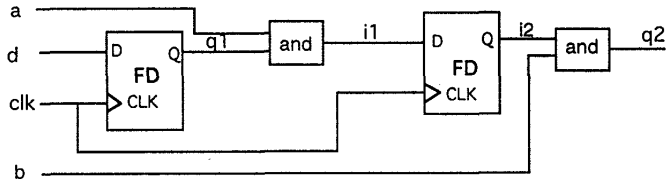
a) Используйте условный оператор **if**.

b) Используйте оператор выбора **case**.

c) Используйте инструкцию **when else**.

d) Если вышеупомянутый пример синтезируется, которая из альтернатив генерирует наименьшее количество логики?

13. Предположим, что должна быть осуществлена следующая логика:



(a) Которая из альтернатив является правильно синтезированным VHDL кодом для вышеупомянутого схемного решения?

1) **Architecture** rtl of ex is
std_logic; **signal** il,i2: std_logic;
begin
process
begin
wait until clk='1';
il<=d **and** a;
q2<=il **and** b;
end process;
end;
end;

2) **Architecture** rtl **of** ex is **signal** il:
begin
process
begin
wait until clk='1';
il<=d **and a;**
i2<=il;
end process;
q2<=i2 **and b;**

3) Architecture rtl of ex is
signal ql, il, i2: std_logic;
begin
process
begin
wait until clk='1';
ql<=d;
il<=a **and** ql;
i2<=il;
q2<=i2 and b;
end process;
end;

4) Architecture rtl of ex is
signal ql, i2: std_logic;
begin
process
begin
wait until clk='1';
ql<=d;
i2<=ql **and** a**;**
end process;
q2<=i2 and b;
end;

5) Architecture rtl of ex is
signal ql, il, i2: std_logic;
begin
process
begin
wait until clk='1';
then
ql<=d;
i2<=il;
end process;
q2<=i2 **and** b**;**
il<=ql **and** a;
end;

6) Architecture rtl of ex is
signal ql, i2: std_logic;
begin
process(clk)
begin
if clk'event **and** clk='1''
ql<=d;
i2<=ql **and** a**;**
end if;
end process;
q2<=i2 **and** b**;**
end;

b) Рисует результат синтеза (в принципе) для всех альтернатив приведенных выше.

5 БИБЛИОТЕКИ, ПАКЕТЫ, ПОДПРОГРАММЫ

5.1 Библиотеки

Когда VHDL компонент откомпилирован, он сохраняется в библиотеке проекта, по умолчанию. Библиотека проекта - не имя каталога на ПЕРСОНАЛЬНОМ КОМПЬЮТЕРЕ или рабочей станции, на которой транслируется проект, а логическое имя. В системе должен иметься указатель, который определяет физический адрес библиотеки компонент, то есть на какой каталог он указывает. VHDL инструментальные средства обычно определяют библиотеку компонент автоматически.

Все откомпилированные компоненты сохраняются в библиотеке. Пакеты также обычно сохраняются в библиотеке. **Пакет** может содержать функции, процедуры, константы и типы. Чтобы использовать компоненты и пакеты не от рабочей библиотеки, библиотеки и пакеты должны сначала быть определены в коде VHDL, используя следующие команды:

Library <имя библиотеки>;

Use < имя библиотеки >.<имя пакета>.ALL;

VHDL стандарт определен таким способом, что проект и его библиотека являются всегда видимой. Эти две библиотеки, поэтому обычно не определяюся в коде VHDL. Все предопределенные типы данных и функции в VHDL стандарте доступны в пакете, называемом стандартом. Этот пакет расположен в std библиотеке. Именно в этом пакете определены bit, bit_vector, character, time и integer. Пакет тоже всегда видим согласно VHDL стандарту и также не определяется в VHDL коде. Это означает, что следующие «невидимые» строки всегда включаются в каждую часть VHDL кода:

Library work;

Library std;

Use std.standard.**ALL**;

Если должны использоваться другие библиотеки и пакеты, они должны быть определены в VHDL коде, перед объектом. Типы данных std_logic, std_ulogic, std_logic_vector и std_ulogic_vector определены в пакете, называемом std_logic_1164. Этот пакет определен в IEEE библиотеке. Это означает, что, если эти типы данных должны исполь-зоваться, следующие строки должны всегда включаться перед каждым объектом:

Library ieee;

Use ieee.std_logic_1164.ALL;

entity ...

5.2 Пакеты

В случае больших проектов, часто имеются функции, типы данных и константы, которые должны использоваться в нескольких компонентах и несколькими проектировщиками. Константа, например, могла бы задавать размер шины в системе. Функция может задавать вычисление контрольной суммы пересылаемых данных в системе. Если менеджер проекта изменяет размер шины или вычисление контрольной суммы, предпочтительно, чтобы изменение было сделано только однажды. Это возможно сделать при помощи пакетов. Если каждый проектировщик, который должен использовать общедоступную константу или функцию для проекта, использует указатель (Use) к пакету, все проекты могут быть модифицированы, изменяя только общедоступный пакет для проекта.

Пример:

Library ieee;

Use project.proj_pack.**ALL**

Все типы данных, константы и подпрограммы, определенные в пакете, могут также многократно использоваться в будущих проектах просто, определяя пакет, используя команду Use.

Пакет может содержать функции, процедуры, типы, константы, атрибуты и компоненты. Пакет состоит из двух частей:

- Объявление
- Тело

Объявление состоит из функции или объявлений процедуры, компонентов, констант и типов, которые используются. Объявление в пакете может быть сравнено с объектом, то есть объявление определяет то, что должно быть «внешне» видимым, так же, как в объекте. Различие - то, что объект определяет, какие сигналы должны экспортироваться вне компонента, в то время как объявление пакета определяет, какие подпрограммы, константы и типы данных должны экспортироваться вне пакета.

Тело состоит из программы, включающей в себя функции и процедуры, которые были объявлены в объявлении пакета. Если внутренние подпрограммы созданы в теле, они будут невидимы внешней стороной пакета, то есть вы не сможете использовать внутреннюю подпрограмму в вашем собственном коде VHDL. Тело может быть сравнено с архитектурой в VHDL компоненте. Архитектура описывает поведенческие свойства компонента. Тело пакета описывает поведение объявленных подпрограмм из объявления пакета.

Синтаксис для пакета:

package <имя> is

[декларация экспортируемых подпрограмм]

[декларация экспортируемых констант]

[декларация экспортируемых компонент]
[декларация экспортируемых типов]
[декларация атрибутов]
[спецификация атрибутов]
end [<имя>];
package body <имя> is
[тела экспортируемых подпрограмм]
[декларация внутренних подпрограмм]
[тела внутренних подпрограмм]
[декларация внутренних констант]
[декларация внутренних типов]
end [<имя>];

Пример:

```
package mypack is
function minimum (a,b:in std_logic_vector) return std_logic_vector;
constant maxint: integer := 16#FFFF#;
type arithmetic_mode_type is (signed,     unsigned);
end mypack;
package body mypack is
function minimum (a,b:in std_logic_vector) return std_logic_vector is
begin
if a<b then return a else return b;  end if;
end minimum;
end mypack;
```

Команда Use позволяет использоваться позже готовому пакету. Эта команда должна прибыть перед объявлением объекта, но после библиотеки.

Пример:

```
Library ieee;
Library mylib;
Use ieee. std_logic_l 164. ALL;
Use mylib.mypack.ALL;
entity <name> is
 port( a: i n   std_logic;
 q: out std_logic;
. . .
```

Вышеупомянутый пример предполагает, что пакет откомпилирован к библиотеке mylib. Система должна содержать указатель, который определяет, где библиотека mylib расположена, то есть какому каталогу это соответствует. Пример также определяет *mypack. ALL.* **ALL** означает, что все подпрограммы в пакете могут использоваться.(!) Если только одна или некоторые из подпрограмм используется, например, только функция *minimum,* должно следующее быть написано:

Use mylib.mypack.minimum;

Если должны использоваться несколько различных функций из различных пакетов, может быть написано следующее:

Library mylib;
Library ieee;
Use mylib.mypack.minimum;
Use ieee.std_logic_l164.**ALL**;

5.3 Подпрограммы

Имеются два типа подпрограмм:

- Функции
- Процедуры

Функции производят только одно значение, принимая во внимание, что процедура используется для определения алгоритма и может производить несколько значений или ни одного. Подпрограммы могут быть определены в трех местах в коде VHDL:

- пакет;
- архитектура;
- процесс.

Поскольку в пакете та же самая подпрограмма может многократно использоваться в нескольких проектах, поэтому рекомендуется, чтобы все подпрограммы были определены в пакете.

Внутренние подпрограммы VHDL кода последовательны. Это означает, что только последовательные команды VHDL, например оператор **if-then-else** и команды **case**, могут использоваться в подпрограммах. Параллельные команды (**process, when else**) не возможны в подпрограммах. Объявление пакета между подпрограммой и **begin** – последовательная декларация. Это означает, что только переменные, не сигналы, могут быть объявлены внутри подпрограмм. Обратите внимание, однако, что не допустимо использовать последовательную команду, **wait** внутри функции.

Процедуры

Процедура может изменять свой параметр. Она принимает константу, переменную и сигнал как объектный класс. Они могут иметь три различных режима: **in, out** и **inout.** Если никакой режим не определен, это значение по умолчанию **in**. Если режим **out** или **inout,** объектный класс должен быть переменная или сигнал. Если объектный класс не был определен, он принимается как переменная или константа в зависимости от режима (**in, out**, **inout**). Другими словами, ввод (режим **in**) принимается по умолчанию как класс констант, а вывод (режим **out** или **inout**) как класс

переменной. Переменные параметры не разрешаются в параллельной команде вызова процедур.

В разделе декларации пакета:

procedure <имя>
[([объектный класс] имя параметра (,имя параметра):
[режим] type [(диапазон [, диапазон] }];
{ [объектный класс] имя параметра {, имя параметра}:
[режим] type [(диапазон [,диапазон])] })];

В теле пакета:

procedure <имя>
[([объектный класс] имя параметра {, имя параметра }:
[режим] type [(диапазон [,диапазон] }];
{ [объектный класс] имя параметра {, имя параметра):
[режим] type [(диапазон [,диапазон])] })] **is**
[декларация констант]
[декларация переменных]
[декларация типов]
begin
последовательность операций
end [<имя>];

Пример процедуры

```
procedure ff ( signal clk, data : in std_logic;
               signal q:   out   std_logic) is
               begin
               loop
               wait until clk='1';
               q <= data;
               end loop;
               end ff;
```

Пример неправильного описания:

```
Architecture rtl of ex2 is
procedure calc (a, b:      in integer;avg, max:out integer)
begin
avg :=(a+b)/2;
if a>b then
max:=a;
else
max:=b;
end if;
end calc;
begin
calc (dl,d2,ql,q2);  -- Ошибка, ql,q2 не переменные
```

```
    process(d3,d4)
    variable a,b: integer;
    begin
    calc(d3,d4,a,b); -- Правильный вызов, a и b переменные
    q3<=a;
    q4<=b;
                ...
     end process;
                ...
     end;
```

Пример 3 (корректного вызова процедуры):

```
    Architecture rtl of ex3 is
    procedure calc ( a, b: in integer;
signal avg, max: out integer) is
    begin
    avg<=(a+b)/2;
    if a>b then max<=a;
    else max<=b;
    end if;
    end calc;
    begin
    calc(dl,d2,ql,q2); --Параллельный вызов процедуры
    process(d3,d4) begin
    calc(d3,d4,q3,q4); -- Последовательный вызов
                                ...
    end process;
    end;
```

Функции

Функция не может изменять свой параметр и может только использовать параметры типа константы или сигналы в режиме **in**. Если класс объекта или режим не определены, он принимается типа константы и/или режима **in**.

В разделе декларации пакета:

```
function <имя>
[([объектный класс] имя параметра {, имя параметра }:
[IN ] тип [ (диапазон [, диапазон] ) ]
{[объектный класс] имя параметра {,имя параметра}:
[IN ] тип [ (диапазон [,диапазон] ) ] } ) ]
return тип;
```

В теле пакета:

```
function <имя>
[( [объектный класс] имя параметра {, имя параметра }:
```

[IN] тип [(диапазон [,диапазон])]
{ [объектный класс] имя параметра {, имя параметра }:
[IN] тип [(диапазон [,диапазон])] })]
return тип is
[декларация констант]
[декларация переменных]
[декларация типов]
begin
последовательность операций
return(выражение);
end [<имя>];

Как упомянуто предварительно, подпрограммы могут быть определены в пакете, архитектуре и процессе. Ниже - примеры того, как функция определена и используется в трех местах.

Пример 1: Пакет

```
package mypack is
function max <a,b:in std_logic_vector) return
std_logic_vector;
end;
package body mypack is
function max (a,b:in std_logic_vector) return
std_logic_vector is
begin
if a>b then return a;
else return b;
end if;
end;
end;
```

Функция, *max* в пакете *mypack* может тогда использоваться свободно в архитектуре, если строка **Use** work.mypack. **ALL** записана в начале VHDL кода.

```
 Use work.mypack.ALL;
        ...
Entity ex is
port(...
end;
Architecture rtl of ex is
Begin
        ...
q<=max(dl,d2); -- Параллельный вызов процедуры
process (data,g)
begin
data_out<=max(data,g); -- Последовательная вызов процедуры
```

```
    end process;
          ...
     end;
```

Пример 2: Архитектура

```
          Architecture rtl of ex2 is
function max (a,b: in  std_logic_vector) return
                                                    std_logic_vector is
    begin
    if a>b then return a;
    else return b;
    end if;
    end max;
              …
    begin
    q<=max(dl,d2); -- Параллельный вызов процедуры
    process(data.g)
    begin
    data_out<=max(data,g); -- Последовательный вызов процедуры
    end process;
    end;
```

Пример 3: Процесс

```
    Architecture rtl of ex3 is
    begin
    process(data,g)
        function max (a,b: in std_logic_vector) return
        std_logic_vector is
        begin
        if a>b then return a;
        else
        return b;
        end if;
        end max;
          begin  -- Начало процесса
data_out<=max(data,g);
          end process;
            …
     end;
```

Все функции и процедуры обычно пишутся без определения длины векторов для входных и выходных параметров. Это означает, что подпрограмма может вызываться с любой длиной вектора.

Пример:

```
    Library ieee;
```

```
Use ieee.std_logic_1164.ALL;
Use work.mypack.ALL;
    Entity ex is
          port( a,b: in  std_logic_vector(3 downto 0);
          c,d: in  std_logic_vector(5 downto 0);
          ql: out std_logic_vector(3 downto 0);
          q2: out std_logic_vector(5 downto 0))
         end;
        Architecture rtl of ex is
        begin
            ql<=max(a,b); -- Длина вектора = 4 бит
            q2<=max(c,d); -- Длина вектора = 6 bits
        end;
```

Как показывают вышеприведенные примеры, допустимо иметь несколько операторов **return** в функции. Единственное требование – только один оператор **return** может быть выполнен. В принципе, все инструментальные средства синтеза поддерживают и процедуры, и функции. Более простые инструментальные средства синтеза могут иметь ограничение, что только один оператор **return** может использоваться и что он должен появиться в конце функции. Чтобы удовлетворять это ограничение, в функции должна быть объявлена внутренняя переменная для временного хранения значения, которое будет возвращаться.

Пример

```
    function max (a,b: in integer) return integer is
    variable int:integer;
    begin
if a>b then
                                                    int:=a; -- Сохранить возвращаемое значение
else
                                                    int:=b;  -- Сохранить возвращаемое значение
        end if;
        return int;  -- Возвратить значение
     end max;
```

Однако, большинство инструментальных средств синтеза поддерживает несколько операторов **return**.

Как и в ADA, в VHDL возможно определить полный путь файлов для функции, когда она вызывается. Если определен полный путь файлов, это может облегчить понимание того, где объявлена функция. Должно быть определено следующее:

<библиотека>.<пакет>.функции >

Пример:

q<=work.mypack.max(a,b);

Если используется полный путь файлов для подпрограммы, пакет не должен быть объявлен с помощью **Use** library.package.ALL в начале VHDL кода.

Функции разрешения

Типы данных std_logic и std_logic_vector, известны как разрешенные типы данных. Это означает, что тип данных std_logic определен следующим образом в пакете std_logic_1164:

subtype std_logic **is** resolved std_ulogic;

Функция разрешения вызывается, когда имеются несколько источников сигнала, влияющие на тот же самый сигнал std_logic. Функция разрешения возвратит значение, которое применяется, когда имеются несколько источников сигнала (драйверов для сигнала). Функция определена в пакете std_logic_1164 следующим образом:

function resolved (s : std_ulogic_vector) **return** std_ulogic **is**
variable result : std_ulogic := 'Z'—по умолчанию самое слабое состояние
begin
if (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 resolved;
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	H	W	X	H
U	X	X	X	X	X	X	X	X	-

Пример функции разрешения показан на рис. 5.1.

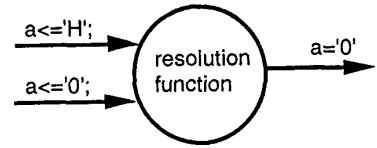
Architecture rtl **of** ex **is**
signal a,b: std_logic;
begin
a<=’H’;
a<=’0’;
end;
	

Рисунок 5.1- Вызов функции разрешения

5.4 Перезагрузка

Так как VHDL - язык со строгим контролем типов, то нельзя вызвать функцию, которая ожидает bit_vector, тогда как входной параметр - std_logic_vector. Чтобы решить эту проблему и иметь возможность вызвать функцию *max*, например, с различными типами данных, функция *max* должна быть перезагружена. Перезагрузка означает, что та же самая функция определена несколько раз с различными типами данных как аргумент для функции

Пример:

```
    Library ieee;
    Use ieee.std_logic_1164.ALL;
    Package ex_pack is
    function max(a,b: in std_logic_vector) return
    std_logic_vector;
        function max(a,b: i n vlbit_vector)  return      vlbit_vector;
        function max(a,b: in integer)   return integer;
    end;
    Package body ex_pack is
    function max(a,b: in std_logic_vector) return
    std_logic_vector is
begin
if a>b then
    return a;
    else
    return b;
    end if;
        end;
function max(a,b:in vlbit_vector) return vlbit_vector is
begin
    if a>b then return a;
          else return b;
         end if;
```

```
        end;
        function max(a,b:in integer) return integer is
begin if a>b then return a;
    else return b;
    end if;
    end;
    end;
```

Пример запроса перезагруженной функции *max* выглядит следующим образом:

```
Library ieee;
    Use ieee. std_logic_l 164. ALL;
    Use work.ex_pack.ALL;
        Entity ex is
port( al,bl  in  std_logic_vector(3 downto 0);
    a2,b2  in  vlbit_vector(4 downto 0);
    a3,b3  in  integer;
                    cl  out std_logic_vector(3 downto 0);
                    c2  ou t vlbit_vector(4 downto 0);
                    c3  out integer);
         end;
       Architecture ex_beh of ex is
begin
    cl<=max(al,bl); -- Function max(a,b:std_logic_vector)
    c2<=max(a2,b2); --Function max(a,b:vlbit_vector)
    c3<=max(a3,b3); -- Function max(a,b:integer)
    end;
```

Имеются несколько очень полезных перезагружаемых функций в библиотеке ieee. Функции «+», «-», «*», «=», «<», «>», « / = », « < = » и « > = », например, перезагружаются для типов данных std_logic_vector или целого числа в пакете ieee.std_logic_unsigned. Некоторые расширенные инструментальные средства синтеза поддерживают эти перезагружаемые функции, что означает, что код может быть написан очень читабельно и эффективно. Функции определены не только с входными параметрами std_logic_vector, но также и с одиним входным параметром std_logic_vector и с другим входным параметром — целым числом.

Пример:

```
function "=" (L:std_logic_vector; R-.integer) return boolean;
function ">" (L:std_logic_vector; R:integer) return boolean;
```

Пример:

```
        Architecture rtl of ex is
        signal a,b,c: std_logic_vector(15 downto 0);
        begin
```

```
process(a,b,c,dl,d2,d3)
begin
if a=12 or c=11 then
        q<=dl;
   elsif b>5
   then q<=d2;
   else q<=d3;
   end if;
   end process;
   end;
```

Библиотека ieee содержит два пакета, которые имеют std_logic_vector как перезагружаемый тип данных: std_logic_unsigned и std_logic_signed. В зависимости от того какой из этих пакетов был объявлен в коде VHDL, std_logic-vectors будет интерпретироваться без знака или со знаком.

Пример:

```
   Library ieee;
   Use ieee.std_logic_1164.ALL;
   Use ieee.std_logic_unsigned.ALL;
   Architecture rtl of ex is
   begin
q<=a + b;
   end;
   Library ieee;
   Use ieee.std_logic_1164.ALL;
   Use ieee.std_logic_signed.ALL;
   Architecture rtl of ex is
      Begin
   q<=a + b; -- signed add
   end;
```

Если оба вектора и знаковый и без знака требуются в той же самой архитектуре, код VHDL может быть написан следующим образом:

```
   Library ieee;
   Use ieee. std_logic_l 164. ALL;
   Use ieee.std_logic_arith.ALL;
Architecture rtl of ex is
begin
   ql<=unsigned(a) + unsigned(b);
   q2<=signed(a) + signed(b);
      end;
```

Пакет std_logic_unsigned использовался в некоторых примерах в этой книге. Если инструмент синтеза, используемый читателем, не поддерживает этот пакет, строка Use, объявляющая пакет, просто должна быть замене-

на на пакет, поддерживаемый инструментом синтеза. Если используется пакет std_logic_arith, важно знать длину вектора, возвращаемого функцией.

Пример:

```
signal a,b: std_logic_vector(3 downto 0);
q1<=unsigned(a) + unsigned(b);
q2<=unsigned(a) + signed(b);
q3<=signed(a) + signed(b);
```

Ниже - таблица для функции «+».

	Без знака	*Знаковое*
Без знака	4 бита	5 битов
Знаковое	5 битов	4 бита

Если, например, используется пакет ieee.std_logic_unsigned, чтобы сложить два вектора, и в результате требуется перенос, один из операндов должен быть расширен на один бит ('0'). Функция «+» дает вектор, длина которого равна длине самого длинного входного параметра.

Пример:

```
Use ieee.std_logic_unsigned.ALL;
        ...
signal a,b,q2: std_logic_vector(7 downto 0);
signal q1: std_logic_vector(8 downto 0);
        ...
        q1<=('0' & a) + b;          -- с переносом (9 бит)
        q2<=a + b;                  -- без переноса (8 бит)
```

5.5 Преобразование типов

VHDL - язык со строгим контролем типов, это означает, что не допустимо присваивать сигналу значение другого сигнала, если они имеют различные типы данных. Эта проблема может обычно минимизироваться, используя тот же самый тип данных последовательно для большинства сигналов в проекте. Многие из самых обычных операторов также перезагружены (переопределены), что, например, означает, что функция «=» может использоваться, чтобы сравнить std_logic_vector с целым числом без преобразования типов. Поскольку многие из инструментальных средств синтеза не поддерживают такую перезагрузку (такое переоп-ределение), а потребности в преобразовании типов нельзя всегда избежать, имеются несколько готовых функций для этой цели. Имеются различия между различными инструментальными средствами синтеза, в терминах какого пакета инструмент поддерживает преобразования, но принцип - всегда один, то есть при смене инструмента синтеза отличаются только имена функции и пакета. Имеются несколько пакетов в библиотеке ieee. Кроме всего прочего, пакет std_logic_1164, содержит функции преобразования типов между следующими типами данных:

std_logic <----> bit
std_logic_vector <----> bit_vector
std_ulogic <----> bit
std_ulogic_vector <----> bit_vector

Пример: в пакете ieee.std_logic_1164:

function to_stdlogicvector(s: bit_vector) **return** std_logic_vector;

Пример того, как использовать функцию преобразования типов:

```
Library ieee;
Use ieee.std_logic_l164.ALL;
Entity ex is port( a,b: in bit_vector(3 downto 0);
q: out std_logic_vector(3 downto 0));
        end;
        Architecture rtl of ex is
        begin
q<=to_stdlogicvector(a and b);
        end;
```

Можно преобразовывать из std_logic_vector в целое число и наоборот. Эта функция преобразования содержится в пакете ieee.std_logic_unsigned, например:

function conv_integer(arg: std_logic_vector) **return** integer;

function conv_std_logic_vector(arg: integer; size integer) **return** std_logic_vector;

Пример:

```
Entity ex is
        port( a,b,c: in      integer      range 0 to 15;
                q: out      std_logic_vector(3 downto 0));
        end;
        Architecture rtl of ex is
        begin
        q<= conv_std_logic_vector(a, 4) when conv_integer© = 8 else
              conv_std_logic_vector(b, 4);
        end;
```

Код был бы более читаем, если все сигналы были типа std_logic. Тогда код мог быть написан следующим образом:

```
Entity ex is
        port( a,b,c:    in std_logic_vector(3 downto 0);
              q: out    std_logic_vector(3 downto 0));
        end;
        Architecture rtl of ex is
        begin
           q<= a when conv_integer© = 8 else b;
        end;
```

Если инструмент синтеза также поддерживает функцию «=», перезагруженную для типа данных std_logic_vector и integer, код может быть сделан еще более читаемым:

```
Entity ex is
    port( a,b,c:  in    std_logic_vector(3 downto 0);
          q:      out   std_logic_vector(3 downto 0));
end;
Architecture rtl of ex is
begin
   q<= a when c = 8 else b;
end;
```

С точки зрения синтеза, не имеет значения, используется ли функция преобразования или нет. **Функция преобразования не занимает никаких логических элементов**. Функция преобразования используется только по двум причинам: сделать код проще для записи, а документацию для понимания. Хороший проект должен, однако, содержать минимум функций преобразования, поскольку рекомендация состоит в том, чтобы использовать тип данных std_logic_vector в максимально возможной степени. С хорошим инструментом синтеза, который также поддерживает сравнение std_logic_vector с целыми числами, в принципе, возможно проектировать очень сложные специализированные интегральные схемы (ASIC) в VHDL без единой функции преобразования.

5.6 Операторы сдвига

Операторы сдвига были добавлены в стандарт VHDL-93. Для типа данных bit_vector определены шесть различный операторов сдвига:

sll - сдвиг влево
sri - сдвиг право
rol - циклический сдвиг влево
ror - циклический сдвиг право
sla - сдвиг влево с сохранением значения справа
sra - сдвиг право с сохранением значения слева

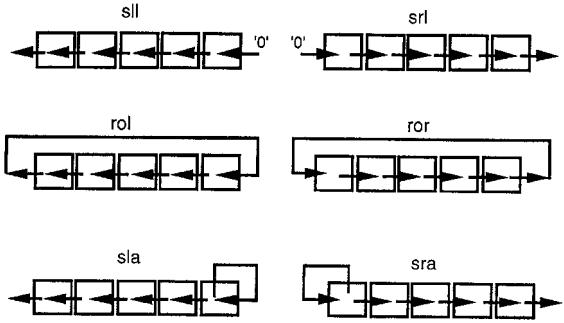


Рисунок 5.2 - Операторы сдвига

Пример:

```
Architecture behv of ex is
    begin
    a<=”01101”;
        q1<= a sll 1;          -- q1 = “11010”
        q2<= a srl 3;          -- q2 = “00001”
        q3<= a rol 2;          -- q3 = “10101”
        q4<= a ror 1;          -- q4 = “10110”
        q5<= a sla 2;          -- q5 = “10111”
        q6<= a sra 1;          -- q6 = “00110”
 end;
```

Если инструментальные средства синтеза не поддерживают указанные операторы сдвига, должны будут использоваться функции сдвига из пакета. Например, две функции сдвига определены в пакете ieee.std_logic_unsigned.. Эти функции вызываются:

```
function shl(arg: std_logic_vector; count: std_logic_vector) return std_logic_vector;
function shr(arg: std_logic_vector; count: std_logic_vector) return std_logic_vector;
        shl     -- сдвиг влево
        shr     -- сдвиг вправо
```

Пример:

```
        ql<=shl(data,»l»);          -- сдвиг влево на 1
        q2<=shr(data,»101»);        -- сдвиг право на 5
        q3<=shr(data, count);       -- сдвиг право на count
```

Возможно, конечно, записать VHDL код без использования любых операторов сдвига. Например:

```
Architecture rtl of ex is
signal data: std_logic_vector(7 downto 0);
begin
    process(clk,resetn)
```

```
    begin
  if resetn='0' then
        ql<=(others=>T);
        q2<=(others=>T);
  elsif clk'event and clk='1' then
— сдвиг вправо на 1
        ql(6 downto 0)<=ql(7 downto 1);
        ql(7)<=d_in;
— сдвиг влево на 1
        q2(7 downto 1)<=q2(6 downto 0);
        q2(0)<=d_in;
  end if;
  end process;
  end;
```

5.7 Упражнения

1. В какой библиотеке и пакете определен битовый тип данных?
2. Какие библиотека(и) и пакет(ы) являются всегда видимыми?
3. Определите четыре составляющих, которые могут храниться в пакете.
4. Определите три места, где может быть определена функция.
5. Какое из этих трех мест облегчает многократное использование функции?
6. Что Вы должны делать, чтобы использовать пакеты других проектировщиков?
7. Что понимается под тем, что тип данных std_logic разрешаемый?
8. В чем различие между функцией и процедурой?
9. Почему длина векторов в функции/процедуре не должна быть определена в объявлении функции?
10. Что такое перезагрузка?
11. Спроектируйте пакет, который содержит две функции: *average* и *sum.* Функция *average* должна возвращать среднее двух чисел (округленное вниз), в то время как функция *sum* должна возвратить сумму. Обе функции должны быть определены для типов данных integer и std_logic_vector.
12. Спроектируйте компонент (c1), который использует функции в (a). Компонент должен иметь четыре входа *a, b, c* и *d.* Входы *a* и *b* имеют тип integer (от 0 до 127), а входные сигналы *c* и *d* имеют тип std_logic_vector (7 **downto** 0). Компонент должен иметь четыре выхода *(average1, average2, sum1, sum2),* то есть один от каждой функции. *average1* и *sum1*должны быть типа integer(0 to 127), а *average2* и *sum2* типа std_logic_vector (7 **downto 0).**
13. Что является преобразованием типа?

14. Содержит ли VHDL стандарт готовые функции преобразования?
15. Какие из следующих функций правильны для VHDL кода?
    1) function f(a,b: in std_logic) return std_logic is
       begin

       if a'event then return b;
       else return a;
       end if;
       end;
    2) function f(signal a,b: std_logic) return std_logic is
       begin

       if a'event then return b;

       else return a;
       end if;
       end;
    3) function f(constant a,b: in std_logic) return std_logic is
       begin
       if a'event then return b;
       else return a;
       end if;
       end;
    4) function f(a: in std_logic) return std_logic is
       begin

       wait on a;
       return a;
       end;
    5) function f(signal a: in std_logic) return std_logic is
       begin

       wait on a;
       return a;
       end;
16. Какие операторы сдвига определены в стандарте VHDL-93?
17. Можно ли спроектировать регистры сдвига без использования этих операторов?

6 ОРГАНИЗАЦИЯ ПАМЯТИ

6.1 ПЗУ

Имеются два способа определения ПЗУ в коде VHDL:

- Использование константы типа массив;
- Технологически специфическое исполнение ПЗУ.

Использование константы типа массив

Описание ПЗУ с помощью константы-массива эффективно в отношении площади, только если ПЗУ относительно маленькое. Если ПЗУ большое, то избыточная площадь будет намного больше, чем для альтернативы 2. Рекомендуется размещение ПЗУ, объявляемого с помощью константы-массива, в пакете. Это облегчает многократное использование ПЗУ. Пакет должен определять размер ПЗУ, используя константы. Ниже рассматривается пример ПЗУ 4 x 8 битов (4 байта);

```
Library ieee;
Use ieee. std_logic_l 164. ALL;
 Package rom is
            constant  rom_width:integer:=8;
             constant  rom_length:integer:=4;
            subtype  rom_word is std_logic_vector
             (rom_width-l downto 0);
type rom_table is array (0 to rom_length-l) of rom_word;
constant rom: rom_table:=rom_table'(
«00101111»,
«11010000»,
«01101010»,
«11101101»);
end;
```

ПЗУ может использоваться в проекте VHDL следующим образом:

```
Library ieee;
Use ieee.std_logic_l 164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;
Entity waveform is
      port( addr: in  std_logic_vector(l downto 0);
q:  out  rom_word);
      end;
      Architecture testbench of waveform is
      begin
      q<=rom(conv_integer(addr));
      end;
```

Инструмент синтеза создаст ПЗУ, используя комбинационную логику. Рис. 6.1 показывает результат синтеза вышеупомянутого примера.

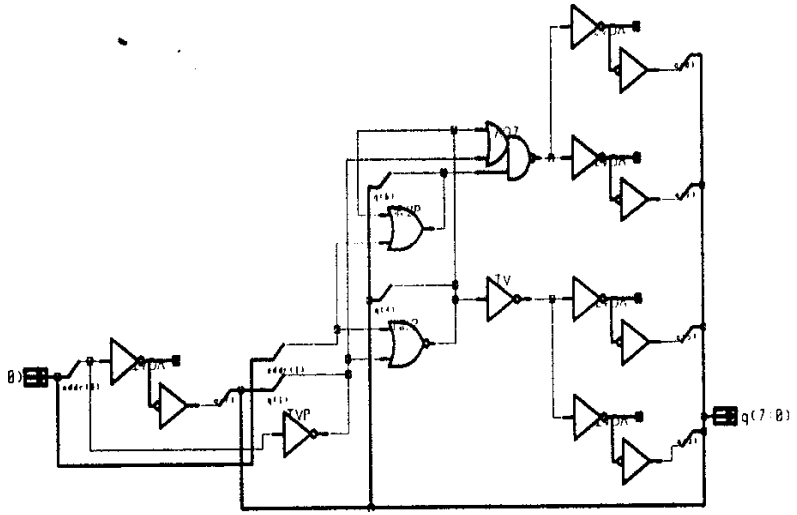


Рисунок 6.1 - Результат синтеза ПЗУ

В вышеупомянутом примере адрес ПЗУ (addr) не мог быть рассчитан заранее. Если бы адрес мог бы быть рассчитан, выходной сигнал (*f)* был бы постоянен, что означает, что инструмент синтеза не генерировал бы никакую логику для такой конструкции. Примером рассчитываемого заранее адреса ПЗУ мог бы быть:

Architecture testbench **of** waveform **is**
begin
q<=rom(2); -- чтение данных с ROM
end;

Поскольку адрес в ПЗУ может быть рассчитан, никакая логика не была сгенерирована. VHDL код только назначит выходному сигналу *q* значение rom(2), который определен как константа со значением «01101010». Различные размеры ПЗУ могут быть получены путем изменения констант в пакете.

Исполнение технологически специфичного ПЗУ

Большие схемы ПЗУ должны быть копируемы, что означает, что библиотека выбранной технологии должна содержать схему ПЗУ. Код VHDL, при этом становится зависимым от используемой технологии, так как имеются ссылки к специфичным ROM-макрокомандам в библиотеке. Другой недостаток в том, что могут возникать проблемы с VHDL симуляцией, если не имеется никакой VHDL модели для ПЗУ. В качестве альтернативы может использоваться имитатор смешанного уровня (уровни

VHDL и вентилей). ПЗУ исполняется точно так же, как любой другой компонент. Эти две альтернативы для представления ПЗУ приведены в табл.6.1.

Таблица 6.1- ПЗУ

	Эффективная площадь	**Простота моделирования**	**Технологическая независимость**
Alt l	НЕТ	ДА	ДА
Alt 2	ДА	НЕТ	НЕТ

6.2 ОЗУ

Так же, как и для ПЗУ, имеются две альтернативных процедуры, когда дело доходит до представления ОП в коде VHDL:

- Использование регистров;
- Копируемая ОП.

Использование регистров

Регистры могут использоваться для очень маленькой ОП. С увеличением ОП излишняя площадь будет намного большей при использовании регистров, чем при копируемой ОП. Так как регистры описываются, используя тактируемый процесс:

```
process(clk,resetn)
begin
if  resetn='0' then
q<=(others=>'0');
elsif clk'event and clk='1' then
if wr='1' then
q<=data;
      end if;
end if;
                                        end process;
```

Исполнение оперативной памяти

Невозможно получить хороший результат синтеза для ОЗУ без использования библиотек. Как и в случае с ПЗУ, недостатком является зависимость VHDL кода от используемой технологии и проблемы, возникающие во время VHDL эмуляции. Предложения ОЗУ различаются соответственно выбору производителя и технологии.

Несколько схем ОЗУ может быть использовано для достижения требуемого размера. Для такого исполнения может быть полезна команда

generate. Например, если используются четыре схемы ОЗУ 4х1 бит, будет получено ОЗУ 4х4 бит:

```
 Architecture rtl of RAM4 is
component RAM4xl
port( d,a0,al,we: in  std_logic;
 q:  out std_logic);
 end component;
 -    for Ul:RAM4xl use Entity work.RAM4xl(rtl);
    begin
    for i in 0 to 3 generate
 RAM_b: RAM4xl port map (d_in(i),a0,al, write.d out(i')1:
end generate;
end;
```

ОЗУ может быть введено в схему автоматически некоторыми инструментами синтеза поведенческого уровня. Такой метод использования ОЗУ – простейший и лучший, так как он эффективен в отношении площади схемы, легок для эмуляции и технологически независим. VHDL код использует массив, который читается и записывается подобно любому другому сигналу. Инструмент поведенческого уровня может заменить этот массив любыми схемами ОЗУ, доступными в данной технологии.

6.3 Упражнения

1. Какие преимущества и недостатки двух методов исполнения ПЗУ в VHDL коде?
2. Какие преимущества и недостатки трех методов исполнения ОЗУ в VHDL коде?
3. Спроектируйте компонент *cl*, реализующий ОЗУ 4х8 бит. Библиотека содержит только ОЗУ 4х1:

```
 Entity RAM4xl is

        port( d,a0,al,we:  in  std_logic;
      q:  out std_logic);
end;
```

7 ИСПЫТАТЕЛЬНАЯ ПАНЕЛЬ

Как только проектировщик описал проект, необходимо выполнить проверку, чтобы убедиться правильно ли функционирует проект. Самый общий метод проверки – использование сигналов возбуждения входов во время имитации с последующим «чтением» выходных сигналов схемы. Главный недостаток использования языка возбуждения входов в том, что он варьируется от одного имитатора к другому.

Еще один метод проверки это написание тестовой модели для генерации и проверки выходных сигналов в VHDL. Это означает, что вы проектируете испытательную панель, которая генерирует входные сигналы и проверяет выходные сигналы схемы. Так как панель сама может генерировать ошибки, она должна быть также протестирована. Рекомендуется использовать и испытательную панель VHDL, и провести проверку с использованием моделей окружения, известную как системное моделирование.

Преимуществом системного моделирования является то, что возбуждающие сигналы для компоненты генерируются моделями, которые должны представлять реальное окружение. Недостаток системного моделирования в том, что оно отнимает много времени. Преимущества испытательных панелей VHDL в их скорости и том, что они платформо-независимы, так как описаны на VHDL. Недостатком испытательных панелей является возможность содержания в ней тех же ошибок, что и в тестируемой схеме. Т.е. проектировщик думает, что тест верен, в то время когда это не так.

В некоторых случаях, тем не менее, может быть выгодно, построить испытательное окружение полностью или частично. Это можно сделать с помощью испытательной панели. Модель схемы получает сигналы возбуждения входов (входные сигналы) и выдает реакцию (выходные сигналы) на панель.

Испытательная панель состоит из двух частей. Одна часть генерирует входные сигналы для модели, которая будет проверена, в то время как другая часть проверяет выходные сигналы от модели.

Ниже - пример того, как могут быть сгенерированы входные сигналы для модели:

```
in_model <= ‘1’ after 5 ns,
‘0’ after 100 ns,
‘1’ after 200 ns,
‘0’ after 300 ns + 4 ns;
clk <= not clk after 50 ns;
```

Сигнал *in_model* обеспечивает прямоугольный сигнал, который инвертируется в моменты времени 5, 100, 200 и 304 нс. Сигнал *clk* будет изменяться с периодом 100 нс.

Инструкция назначения тактирующего сигнала **clk<=not clk after 50 ns**, может приводить к проблемам, если *clk* не объявлен как данные типа бит. Поскольку рекомендуется для схем VHDL использовать тип данных std_logic в максимально возможной степени, *clk* в испытательной панели должен также иметь тот же самый тип, чтобы избежать преобразования типов. Проблема состоит в том, что все сигналы в VHDL инициализируются signal'LEFT. В случае std_logic это означает значение 'U'(неизвестное). Инверсия 'U' – тоже 'U'. Эта проблема может легко быть решена инициализацией *clk* значением '0', например, в объявлении *clk:*

signal clk:std_logic:=’0’;

Если компонент, который будет проверен, содержит автоматы с памятью, может быть удобно, использовать команду **wait**, чтобы назначить входным сигналам значения в соответствующих состояниях:

```
process
begin
in_signall<=’1’;
in_signal2<=’0’;
wait until resetn=’1’;
wait until clk=’1’;
in_signall<=’1’
in_signal2<=’0’
wait until clk=’1’;
in_signall<=’l’ in_signal2<=’0’
wait until clk=’1’;
in_signall<=’l’
in_signal2<=’0’
end process;
```

Как альтернатива к ожиданию фронта тактового импульса, возможно объявить период для *clk* как постоянную, чтобы ее можно было использовать в команде **wait**. Например:

```
constant period:time:=50 ns;
wait for period;
```

Преимущество этого метода состоит в том, что поведенческая модель может не содержать тактирующий генератор. Также возможно ожидание нескольких тактовых циклов в одной строке:

```
wait for 2 * period;
```

Если команда **wait for** используется в нескольких тактах, рекомендуется, чтобы время было объявлено как постоянная:

```
constant period:time:=50 ns;
constant period2:time:=2 * period;
                . . .
wait for period2;
```

Это рекомендуется потому, что имитатор мог бы быть вынужден вычислять значение 2 * period несколько раз в течение одного цикла моделирования, если строка выполняется несколько раз. Это, конечно, отнимает лишнее время имитации.

Вторая часть испытательной панели проверяет выходные сигналы от модели. Оператор **assert** часто используется на этой стадии. Команда **assert** выполняется, если выражение ложно. Пример команды **assert**:

```
process(clk)
begin
assert now < 900 ns
report “stopping simulator (max simulation time 900 ns)”
severity Failure;
end process;
```

Имитатор останавливается после 900 нс.

```
process(out_model, in_model)
begin
if out_model = 1 and in_model = 1 then
assert false
report “The signals out and in are ‘1’ at the same time”
severity Error;
 end process;
```

Если выходные и входные сигналы совпадают, имитатор останавливается и выводит код ошибки. Вышеупомянутый пример содержит **if** оператор, который определяет, должен ли оператор **assert** быть выполнен или нет. Выражение для **assert** в примере - всегда ложно («**assert false**»). Также возможно использовать выходной сигнал, который деактивирован, если испытание проходит неудачно:

```
process(out_model, in_model)
begin
if out^model = 1 and in_model = 1 then
        test_ok<=’0’;
        end if;
 end process;
```

Техника использования команды **wait until clk** может также применяться для проверки выходных сигналов конечного автомата. Испытательная панель записывается таким образом, что она может использоваться и на уровне вентилей. В уровне вентилей выходной сигнал не может изменяться немедленно по фронту синхросигнала, что является нормой в VHDL имитации. Допуск на это должен быть сделан в испытательной панели. Простой метод - использование команды **wait until clk = '0'**;. Выходные сигналы должны, однако, также быть устойчивы на уровне вентилей после половины такта. Аль-

тернативно, проверка выходных сигналов может иметь место как раз перед или одновременно с фронтом синхроимпульса.

Модель схемы может быть в виде VHDL кода, netlist или в другом формате. Это означает, что одна и та же панель может использоваться в течение различных фаз проектных работ. Модель схемы проверяется с использованием панели. Только в поведенческой модели схемы и выше, модель усовершенствуется до модели RTL, после чего проверяется netlist модель. В конце концов проверяется модель с задержками маршрутизации. Когда модель изменяется, в некоторых случаях интерфейс ввода-вывода с испытательной панелью также должен быть изменен.

На поведенческом уровне возможно работать с большим количеством абстрактных типов данный чем на netlist уровне. Это приводит к различиям между интерфейсами панели и схемы. Например, при использовании чисел с плавающей запятой на поведенческом уровне, трансляция чисел с плавающей запятой в число соединений (вектор) производится на более низком уровне абстракции.

Обычно проще иметь однородный интерфейс во всех моделях. Это может быть выполнено путем использования векторов в объекте на поведенческом уровне, а затем использовать более абстрактные типы данных в архитектуре. Одно из преимуществ этого состоит в том, что становится очень просто производить смену различных моделей. Единственная вещь, которая должна быть сделана, состоит в том, чтобы изменить спецификацию компоненты на испытательной панели на спецификацию компоненты, которую нужно имитировать.

Пример:

Поведенческий уровень:

For Ul: **Use entity** work.my_comp(behv);

Уровень меж регистровых передач:

For Ul: **Use entity** work.my_comp(rtl);

Вентельный уровень:

For Ul: **Use entity** work.my_comp(gate);

7.1 Различные уровни испытательной панели

Панели могут быть разделены на три различных класса, от первой категорией, являющейся наименее сложной до третьей наиболее сложной (см. табл. 7.1). Разница между различными испытательными панелями состоит в том, проверяют ли они время и значение выходных сигналов. Все варианты генерируют входные сигналы для модели, которая будет проверена.

Таблица 7.1- Типы испытательных панелей.

	Возбуждение входов	Проверка выходных сигналов	Проверка по времени
Класс 1	Панель	Ручная	Ручная
Класс 2	Панель	Панель	Ручная
Класс 3	Панель	Панель	Панель

В нормальной испытательной панели компонент, который будет проверен, реализуется с помощью структурного VHDL. Поскольку именно испытательная панель генерирует сигналы возбуждения входов для компоненты, испытательная панель не должна содержать никакого входного сигнала объекта. Предположим, что мы имеем следующий VHDL компонент для испытания на панели.

```
    Library ieee;
    Use ieee.std_logic_1164.ALL;
    Use ieee.std_logic_unsigned.ALL;
    Entity my_comp is
    port(clk,resetn,d_in:  in std_logic;
a,b: in std_logic_vector(2 downto 0);
    d out,en,overflow: out std_logic;
    q:  out std_logic_vector(2 downto 0));
    end;
    Architecture rtl of my_comp is
    begin
Controller:Block
type state_type is (s0,sl,s2);
    signal state:state_type;
    begin
    process(clk,resetn)
    begin
    if resetn='0' then  state<=s0;
    d_out<=T;
    en<='0';
elsif clk'event and clk='1' then
case state is
when s0=> if d_in='1' then  state<=sl;end if;
    d_out<='1';
    en<='0';
    when sl=> if d_in=='0' then state<=s2; end if;
    d_out<='0';
    en<='l';
```

```
    when s2 => state<=s0;
    d_out<='0';
    en<='0';
    when others => state<=s0;
    d_out<='l';
    en<='0';
    end case;
    end if;
    end process;
    end block;
    check:Block
    signal q_b: std_logic_vector(2 downto 0);
    begin
    q<=q_b;
    q_b<=a + b;
    overflow<='l' when q_b>2 else '0';
    end block;
    end;
```

Класс 1

Испытательные панели первого класса только генерируют входные сигналы для модели, выходные сигналы должны быть проверены вручную.

```
    Library ieee;
    Use ieee.std_logic_1164.ALL;
Entity test_my_comp is
          port( d_out,en,overflow:out std_logic;
q:  out std_logic_vector(2 downto 0));
end;
    Architecture testbench of test_my_comp is
    component my_comp
    port( clk,resetn,d_in: in std_logic;
    a,b: in std_logic_vector(2 downto 0);
    d_out,en:  out std_logic;
                overflow:  out std_logic;
                q:         out std_logic_vector(2 downto 0));
    end component;
                signal clk:  std_logic: ='0';
                signal resetn: std_logic:='0';
                signal d_in:  std_logic;
                signal a,b:  std_logic_vector(2 downto 0);
                For Ul:my_comp Use Entity work.my_comp(rtl);
    begin
    Ul:my_comp port map
```

```
(clk,resetn,dJn,a,b, d_out,en, overflow,q);
clk<=not elk after 50 ns;
resetn<=’l’ after 125 ns;
a<= “000”,
“010” after 125 ns,
“100” after 175 ns;
b<= “000”,
“100” after 125 ns,
“01l” after 175 ns;
    process
    begin
 d_in<=’0’;
 wait until resetn=’1’;
 d_in<=’1’;
 wait until clk=’1’;
    d_in<=’0’ after 10 ns;
    wait;
    end process;
    end;
```

Испытательная панель первого класса, представленная выше генерирует входные сигналы для компонента, в то время как выходные сигналы должны быть проверены вручную. В примере выходные сигналы были включены в объект испытательной панели. Это, тем не менее, не необходимо, поскольку в имитации возможно брать выходные сигналы от компоненты my_comp на иерархическом уровне ниже верификации. С точки зрения документации, однако, преимуществом является наличие сигналов, которые нуждаются в ручной проверке, в объекте испытательной панели. Если это правило всегда выполняется, легко определить, которые из выходных сигналов должны быть проверены вручную.

Класс 2

Испытательные панели этого класса не только генерируют входные сигналы для модели, но и также проверяют корректность выходных сигналов модели. Однако синхронизация должна быть проверена вручную. Пример испытательной панели второго класса:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity test2_my_comp is
port(test_ok:  out std_logic:=’H’);
end;
Architecture testbench of test2_my_comp is
component my_comp
 port( clk,resetn,d_in: in std_logic;
```

```
    a,b:  in std_logic_vector(2 downto 0);
    d_out,en,overflow: out std_logic;
    q: out std_togic_vector(2 downto 0));
        end component;
        signal elk: std_logic:='0";
        signal resetn: std_logic:='0';
        signal d_in: std_logic;
        signal a,b: std_logic_vector(2 downto 0);
        signal d_out,en,overflow: std_logic;
        signal q: std_logic_vector(2 downto 0);
        For Ul:my_comp Use Entity work.my_comp(rtl);
        begin
        Ul:my_comp port map
        ( clk,resetn,d_in,a,b,d_out,en,overflow,q);
        clk<=not elk after 50 ns;
                resetn<='l'  after 125 ns;
                a<= "000",
                "010" after 125 ns,
                "100" after 175 ns;
                b<= "000",
                "100" after 125 ns,
                "011" after 175 ns;
                p0:process begin
                wait for 125 ns;
                if q/="000" or overflow/='0' then test_ok<='0';
                end if;
                wait for 50 ns;
                if q/="110" or overflow/='1' then  test_ok<='0';
        end if;
        wait for 50 ns;
        if q/="lll" or overflow/='l' then  test_ok<='0';
        end if;
        wait;
        end process;
        process
        begin
d_in<='0';
wait until resetn='1';
d_in<='1';
wait until clk='1';
            d_in<='0' after 10 ns;
            wait;
```

```
end process;
pl:process
begin
        wait for 30 ns;
    if en/='0' or d_out/='1' then  test_ok<='0'; end if;
    wait until resetn='l';
    wait until clk='1';
    if en/='0[1] or d_out/='l' then  test_ok<='0'; end if;
    wait until clk=1;
    if en/=T or d_out/='0'   then
    test_ok<='0';
    end if;
    wait until clk='1';
    if en/='0' or d_out/='0'  then
    test_ok<='0';
    end if;
    wait;
    end process;
    end;
```

Испытательная панель второго Класса в примере выше проверила выходные сигналы, используя сигнал *Test_ok*. *Test_ok* установлен в значение 'H' в объявлении объекта. Сигнал *Test_ok* объявлен как тип данных std_logic. Std_logic - разрешенный тип данных. Это означает, что, если несколько источников выдают сигнал, будет вызываться функция разрешения конфликтов. Эта функция решает, какая оценка выходному сигналу будет даваться при моделировании. Вышеупомянутая испытатель-ная панель непрерывно дает сигналу *Test_ok* значение 'H' и, в случае ошибки, также значение '0'.

Так как *Test_ok* задается несколькими устройствами одновременно, вызывается функция разрешения конфликтов со входными параметрами 'H' и '0', с предпочтением в пользу '0'. Если, с другой стороны, сигнал *Test_ok* все еще будет иметь значение 'H' после того, как прошла имитация, испытательная панель не найдет никаких ошибок, и компонент пройдет испытание.

В вышеупомянутом примере *Test_ok* должен быть инициализирован в объекте к 'H' а не '1'. Это потому что *Test_ok* имеет два задающих устройства, связанных с ним. Процессы p0 и p1 оба задают значение сигнала *Test_ok*. Это означает, что функция разрешения будет вызываться со входными параметрами 'H' и 'H' с самого начала. Если один из процессов находит ошибку, *Test_ok* устанавливается в '0' задающим устройством этого процесса, то есть функция разрешения вызывается со входными параметрами '0' и 'H', с решением в пользу '0'.

Если бы *Test_ok* был инициализирован к '1' вместо 'H', функция разрешения конфликтов вызывалась бы со входными параметрами '0' и '1', в результате в случае ошибки давая ' X '. Если, с другой стороны, испытательная панель написана таким способом, что имеется только одна параллельная команда, задающая значение *Test_ok,* то *Test_ok* мог быть инициализирован значением '1' без проблем.

Проверка выходных сигналов и сигнала d_out выполняется одновременно с фронтом синхроимпульса. Ни модели VHDL, ни уровень вентилей не сумели обновить свои выходные сигналы, то есть выходные сигналы все еще имеют старое значение. Выходные сигналы VHDL модели будут модифицированы 1 дельта промежуток времени после того, как выполнен тест. На уровне вентилей изменение значений на выходах вероятно запоздает на несколько наносекунд (в зависимости от выбора технологии). Как альтернатива сигналу *Test_ok,* команда **assert** может использоваться для сообщения об ошибке. В команде **assert** возможно установить уровень серьезности, так, чтобы имитация останавливалась или продолжалась, когда встречается ошибка. По умолчанию, большинство VHDL имитаторов останавливается на ошибке серьезного уровня, хотя обычно возможно установить уровень, на котором эмулирующее устройство должно остановиться. Конечно, возможно комбинировать выходной сигнал *Test_ok* с командой **assert**. Преимущество этого метода состоит в том, что, используя сигнал *Test_ok* просто понять, пройден ли тест, но команда **assert** также сообщает открытым текстом, который из тестов не пройден.

Пример:

```
    if en/=’0’ or d_out/=’0’ then
    test_ok<=’0’;
    assert false
    report “en/=’0’ or d_out/=’0’ “
severity warning;
        end if;
```

Класс 3

Испытательные панели этого класса генерируют входной сигналы для модели и проверяют значения и синхронизацию выходных сигналов. Пример испытательной панели третьего класса:

```
    Library ieee;
    Use ieee.std_logic_1164.ALL;
    Entity test3_my_comp is
port(test_ok:  out std_logic:=’H’);
    end;
    Architecture testbench of test3_my_comp is
component my_comp
port( clk,resetn,d_in: in std_logic;
```

```
    a,b: in std_logic_vector(2 downto 0);
    d_out, en, overflow: out std_logic;
    q: out std_logic_vector(2 downto 0));
end component;
    signal clk: std_logic:='0';
    signal resetn: std_logic:='0';
    signal d_in: std_logic;
    signal a,b: std_logic_vector(2 downto 0);
    signal d_out,en,overflow: std_logic;
    signal q: std_logic_vector(2 downto 0);
    For Ul:my_comp Use Entity work.my_comp(rtl);
    begin
    Ul:my_comp port map
    (clk,resetn,d_in,a,b, d_out,en,overflow,q);
    clk<=not clk after 50 ns;
    resetn<='l' after 125 ns;
    a<= "000", "010" after 125 ns, "100" after 175 ns;
    b<= "000", "100" after 125 ns, "011" after 175 ns;
    process
    begin
        wait for 125 ns;
        if q/="000" or not q'stable(15 ns) then test_ok<='0';
     elsif overflow/='0' or not overflow'stable(15 ns) then
     test_ok<='0';
        end if;
        wait for 50 ns;
        if q/="110" or not q'stable(15 ns) then test_ok<='0';
        elsif overflow/='l' or not overflow'stable(15 ns) then
     test_ok<='0';
        end if;
        wait for 50 ns;
        if q/="lll" or not q'stable(15 ns) then test_ok<='0';
        elsif overflow/='1' or not overflow'stable(15 ns) then
    test_ok<='0';
    end if;
    wait;
    end process;
    process
    begin
    d_in<='0';
    wait until resetn='1';
    d_in<='1';
```

```
    wait until clk='l';
    d_in<='0' after 10 ns;
    wait;
    end process;
    process
    begin
wait for 30 ns;
if en/='0' or d_out/='1' then test_ok<='0'; end if;
    wait until resetn='1';
    wait until .clk='1';
    if en/=0" or d_out/='1' then test_ok<='0';
    elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
    test_ok<='0';
    end if;
    wait until clk='1';
    wait until clk='1';
    if en/='l' or d_out/='0' then test_ok<='0';
    elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
                    test_ok<='0';
    end if;
    wait until clk='1';
    if en/='0' or d_out/='0' then test_ok<='0';
    elsif not en'stable(80 ns) or not d_out'stable(80 ns) then
    test_ok<='0';
    end if;
    wait;
    end process;
    end;
```

Испытательная панель третьего класса в примере, приведенном выше, также проверяет, что выходные сигналы устойчивы в течение 15 нс и 80 нс соответственно. Сложение **q <= a + b** проверяется командой **q'stable (15) after 50 ns**, то есть сложение должно занимать максимум 50-15 = 35 нс для успешной проверки.

При положительном фронте синхроимпульса выходные сигналы en и d_out проверяются, были ли они устойчивы в течение 80 нс, то есть у них есть 100-80 = 10 нс от внутреннего триггера до выходного сигнала схемы. Такое ограничение синхронизации без труда удовлетворяется в VHDL эмулировании. Только в эмулировании вентильного уровня может происходить нарушение синхронизации.

7.2 Смещение сигналов

Если входной или выходной сигнал компонента, который будет разработан, должен иметь смещение вверх или вниз в ячейке ввода-вывода, смещение вверх/вниз может быть описано двумя способами:

- В VHDL компоненте;
- В испытательной панели VHDL.

Предположим, что мы имеем следующий компонент, в котором необходимо придать смещение двунаправленному выходу io.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity my_comp is
port( a,b: in  std_logic;
io: inout std_logic;
                q:  out  std_logic);
               end;
               Architecture rtl of my_comp is
               begin
               q<=a and io;
               io<='1' when a='1' and b='0' else 'Z';
               end;
```

Способ моделировать это - назначение двунаправленному сигналу значения 'H'. Сигнал должен иметь тип данных std_logic. Как было ранее сказано, std_logic - тип данных с разрешением конфликтов. Так как сигналу io непрерывно присваивается 'H' а иногда также другие значения ('0' или '1') в то же самое время, тогда будет вызываться функция разрешения конфликтов.

Если, например, двунаправленный сигнал приводится испытательной панелью или внутренней схемой к значению '0', функция разрешения будет вызвана с входными аргументами 'H' и '0', и победит значение '0'. Это соответствует действительности. Если поместим описание смещения в компоненте, будет получена следующая архитектура:

```
Architecture rtl of my_comp        is
begin
io<='H':
q<=a and io;
io<='1' when a='1' and b='0' else 'Z';
end;
```

Если мы хотим только инициализировать сигнал в компоненте, будет получена следующая архитектура:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity my_comp is
port( a,b: in  std_logic;
```

```
io: in out std_logic:='H';
                q: out  std_logic);
               end;
```

Обратите внимание, что, как только в архитектуре сигналу присвоено значение, инициализированное значение «отсоединяется» от сигнала. Инструмент синтеза проигнорирует факт, что порт *io* смещен к/инициализирован 'H'. Это смещение должно быть введено при соединении и выборе, какой тип ввода-вывода должна иметь ячейка компоненты. В VHDL эмулировании, однако, порту *io* будет назначено правильное значение ('H') если не имеется никакого задающего устройства (внутреннего или внешнего) связанного с портом. Как альтернатива вышеупомянутому, смещение может быть учтено в испытательной панели. Ниже – простая панель первого класса, в которой учтено смещение.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
    Entity tb is
port( io: in out std_logic;
              q:  out  std_logic);
             end;
             Architecture testbench of tb is
begin
io<='H';
a<= '0', '1' after 100 ns,  '0' after 50 ns;
b<=  '0', '1' after 150 ns,  '0' after 50 ns;
 io<= '0', 'Z' after      100 ns,      '0' after 50 ns;
 end;
```

7.3 Несколько компонентов в одной испытательной панели

Если несколько компонентов, которые должны быть связаны между собой, разрабатываются одновременно, все они могут быть проверены в одной испытательной панели. В этом случае некоторые из входных сигналов управляются компонентами и некоторые - панелью. В этой испытательной конфигурации проверяется смесь системного моделирования и моделирования на испытательной панели. Предположим, что мы имеем логическое соединение двух компонентов, показанных на рис.7.1.

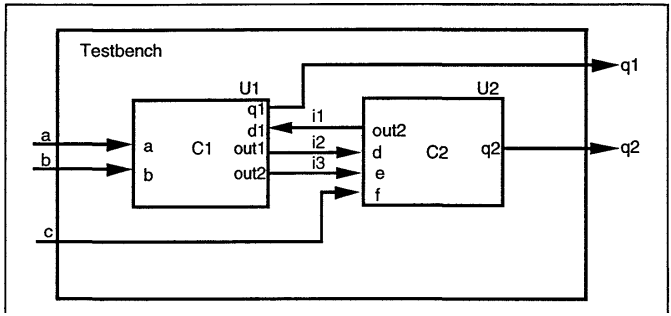


Рисунок 7.1 - Два компонента на испытательной панели

Испытательная панель может тогда быть описана следующим образом:

```
    Library ieee;
    Use ieee.std_logic_1164.ALL;
    Entity tb2comp is
    port( ql,q2: out std_logic);
    end;
    Architecture testbench of tb2comp is
component cl
          port( a,b,dl: in std_logic;
ql,outl,out2: out std_logic);
    end component;
    component c2
    port( d,e,f: in std_logic;
    q2,out2: out std_logic);
    end component;
    For Ul:cl Use entity work.cl(rtl);
    For U2:c2 Use entity work.c2(rtl);
    signal a,b,c,il,i2,i3: std_logic;
    begin
    Ul:cl port map(a,b,il,ql,i2,i3);
    U2:c2 port map(i2,i3,c,q2,il);
    a<= ‘1’, ‘0’ after 50 ns, ‘1’ after 125 ns;
    b<= ‘0”, ‘1’ after 70 ns, ‘0’ after 125 ns,
‘1’ after 225 ns;
                    c<= ‘1’, ‘0’ after 50 ns, ‘1’ after 150 ns,
                          ‘0’ after      250 ns;
     end;
```

7.4 Генератор формы волны

Другой способ генерации сигналов возбуждения входов на испытательной панели состоит в том, чтобы использовать устройство, известное как генератор форм волн. Генератор форм волн использует ПЗУ, которое содержит информацию о значениях сигналов возбуждения. Предположим, что мы имеем испытательную панель, показанную на рис. 7.2.

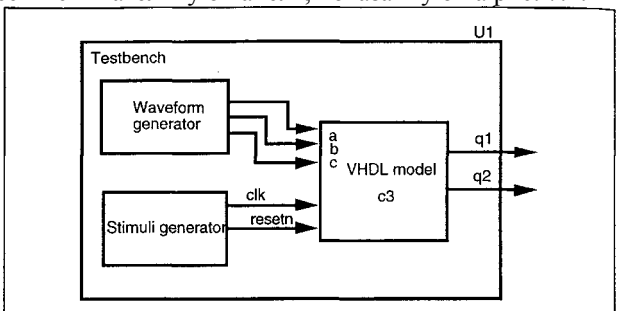


Рисунок 7.2 - Традиционный генератор форм волн.

Ниже представлен пример испытательной панели, которая использует генератор форм волн, чтобы генерировать сигналы возбуждения входов для компонента c3.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Package rom is
      constant  rom_width:integer:=3;
      constant  rom_length:integer:=10;
 subtype rom_word is std_logic_vector(rom_width-l downto 0);
 type rom_table is array (0 to rom_length-l) of rom_word;
 constant rom: rom_table:= rom_table'("001", "110", "011",
"111", "101", "000", "101", "001", "101", "010");
end;
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;
Entity waveform is
port( ql,q2: out std_logic);
end;
Architecture testbench of waveform is
component c3
port( clk,resetn,a,b,c: in std_logic;
```

```
    ql,q2: out std_logic);
    end component;
    For Ul:c3 Use entity work.c3(rtl);
    signal clk: std_logic:='0';
    signal resetn: std_logic;
    signal step:std_logic_vector(3 downto 0);
    signal waves:rom_word;
    begin
    resetn<= '0', '1' after 25 ns;
    clk<=not clk after 50 ns;
    process(clk,resetn)
begin
          if resetn='0' then
          step<=(others=>'0'
elsif clk'event and clk='1' then
if step/=rom_length-l then
          step<=step+l
          else
step<=(others=>'0'
          end if;
     end if;
end process;
waves<=rom(conv integer(step)):
Ul:c3 port map( clk,resetn,waves(2),waves(l),waves(0),
                                                    ql,q2);
end;
```

Испытательная последовательность, показанная на рис. 7.3 будет получена от генератора форм волн в течение имитации (первые четыре тактовых цикла).

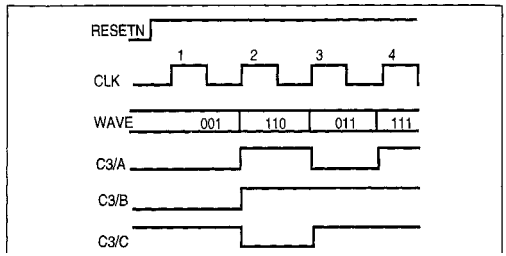


Рисунок 7.3 - Формы волн.

Поскольку генератор форм волн основан на синтезируемом ПЗУ, эта методика может также использоваться в схеме. Метод может быть намного

более эффективен в плане занимаемой площади схемы в некоторых приложениях, чем использование абстрактного автомата для генерации последовательности выходных сигналов.

Если входные сигналы от генератора форм волн не часто изменяют значение, вентили тратятся впустую при использовании традиционного метода описания генератора форм волн. В случае схемы, и также в случае испытательных панелей, эффективнее только определить значение в ПЗУ при изменении сигнала и число тактовых циклов, во время которых значение должно быть устойчиво.

Для этого также используется модель ПЗУ. Предположим, что два выходных сигнала должны иметь форму волны как на Рисунке 7.4, которая должна повторяться каждые 16 тактовых циклов.

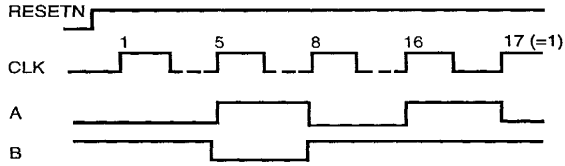


Рисунок 7.4 - Формы волны.

Если это описать, используя традиционный метод форм волн, будет получен следующий код VHDL:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Package rom is
constant  rom_width:integer:=2;
        constant  rom_length:integer:=16;
        subtype rom_word is std_logic_vector(rom_width-l downto 0);
        type rom_table is array (0 to rom_length-l) of rom_word;
        constant rom: rom_table:= rom_table'("01", "01", "01", "01", "10",
"10", "10", "01", "01", "01", "01", "01", "01", "01", "01", "11");
end;
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Use work.rom.ALL;
Entity waveform is
        port( clk,resetn: in  std_logic;
          waves:  out rom_word);
end;
Architecture testbench of waveform is
signal start_up:std_logic;
signal step:std_logic_vector(3 downto 0);
```

```
    begin
        process (clk, resetn)
        begin
if resetn='0' then
step<=(others=>'0'
start_up<='1';
    elsif clk'event and clk='1' then start_up<='0";
    if start_up='0' then  -- Удаление одного тактового цикла после сброса
if step/=rom_length-l then step<=step+l;
else step<=(others=>'0');
          end if;
end if;
    end if;
    end process;
    waves<=rom(conv_integer(step));
    end;
```

Если мы используем более сложный генератор формы волны, необходима декларация следующего:

- значения выходных сигналов;
- числа тактовых циклов, для которых выходные сигналы должны быть устойчивы

Пример:

```
    Library ieee;
    Use ieee.std_logic_l 164.ALL;
    Package rom is
constant rom_width:integer:=2;
          constant rom_length:integer:=4;
     subtype rom_word is std_logic_vector(rom_width-l downto 0);
     type rom_table is array (0 to rom_length-l) of rom_word;
     constant rom: rom_table:= rom_table'( "01", "10", "01", "11");
     subtype max_cycle is integer range 0 to 15;
     type cycle_length_table is array (0 to rom_length-l) of max_cycle;
     constant cycle_length: cycle_length_table :=cycle_length_table'(4,3,8,l);
    end;
    Library ieee;
    Use ieee.std_logic_1164.ALL;
    Use ieee.std_logic_unsigned.ALL;
    Use work.rom.ALL;
    Entity wave_adv is
        port( clk,resetn:  in  std_logic;
      waves:  out rom_word);
    end;
```

```
    Architecture testbench of wave_adv is
signal step:std_logic_vector(3 downto 0);
    begin
    process(clk,resetn)
    variable count:max_cycle;
    begin
    if resetn='0' then
    step<=(0=>'l',others=>'0');  - step=l
    count:=cycle_length(0);
    elsif clk'event and clk='1' then
    if count=0 then
    count:=cycle_length(conv_integer(step));
if step/=rom_length-l then
step<==step+l
else
step<=(others=>'0');
          end if;
    end if;
    count:=count-l;
    end if;
    end process;
    waves<=rom(conv_integer(step));
end;
```

В вышеупомянутом пакете ПЗУ, которое содержит данные, должно быть только 4 x 2 бита в размере вместо 16 x 2 в предыдущем примере. Число тактовых циклов, для которых выходные сигналы должны быть декларированы в отдельном ПЗУ:

```
constant rom: rom_table:= rom_table'("01", "10", "01", "11");
cycle_length:cycle_length_table:=cycle_length_table'(4,3,8,l);
```

Вышеупомянутые константы должны интерпретироваться следующим образом:

1. Значение «01» должно посылаться в течение 4-х тактовых циклов.
2. Значение «10» должно посылаться в течение 3-х тактовых циклов.
3. Значение «01» должно посылаться в течение 8-ми тактовых циклов.
4. Значение «11» должно посылаться в течение 1-го тактового цикла.

Если мы сравниваем площадь схем после синтеза для этих двух маленьких примеров, традиционный генератор форм волн будет в этом случае компактней. Если же, с другой стороны, имелись больше чем два сигнала или если бы последовательность была дольше, результат был бы противоположным. Выбор генератора форм волн должен быть основан на формах волн, которые Вы хотите генерировать.

7.5 Текстовый ввод-вывод

Подпрограммы текстового ввода-вывода встроены в стандарт VHDL. С их помощью можно читать файлы и писать в них. Используемые подпрограммы следующие:

read(..)
readline(..)
write(..)
writeline(..)

Чтобы быть способным использовать эти подпрограммы, следующая строка должна быть включена в начале кода VHDL:

Use std.textio.**ALL**;

Одна область приложения для текстового ввода-вывода находится в испытательных панелях. Входные сигналы для модели могут быть помещены во внешний файл, а значение ожидаемых выходных сигналов может быть сохранено в другом файле.

Предположим, что мы определили значение выходных сигналов в файле (my_inputs.vec). Испытательная панель должна подключить эти входные сигналы ко входным сигналам *dl, d2* и *d3* в компоненте c4:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Use ieee.std_logic_arith.ALL;
Use std.textio.ALL;
Entity text_io is
port(ql,q2:  out  std_logic_vector(2 downto 0));
end;
Architecture testbench of text_io is
component c4
            port(clk,resetn:  in  std_logic;
           dl,d2,d3:  in  std_logic_vector(3 downto      0);
           ql,q2:  out std_logic_vector(2 downto 0));
end component;
        signal clk:  std_logic:='0';
       signal resetn:  std_logic:='0';
 signal dl,d2,d3:  std_logic_vector(3 downto 0);
      For Ul:c4 Use Entity work.c4(rtl);
begin
Ul:c4 port map(clk,resetn,dl,d2,d3,ql,q2);
clk<=not elk after 50 ns;
resetn<='1' after 125 ns;
process
variable text_line:line;
variable il:integer;
```

```
    variable i2:integer;
    variable i3:integer;
    file my_file: text is in  “my_inputs.vec”;  -- файл входных векторов
begin
while not endfile(my_file) loop
          readline(my_file,text_line); -- чтение строки из файла
     read(text_line,il);  --чтение первого символа в переменную i1
    d 1 <=conv_std_logic_vector(i 1 ,d 1 “length);
    read(text_line,i2);  -- чтение второго символа в переменную i2
        d2<=conv_stdJogic_vector(i2,d2’length);
    read(text_line,i3);
    d3<=conv_std_logic_vector(i3,d3’length);
    wait until clk=’l’;
    end loop;
    wait;
    end process;
    end;
```

Предположим, что файл my_inputs.vec имеет следующее содержание:

my_inputs.vec

3	**6**	**8**
5	**7**	**2**
12	**1**	**9**

Испытательная панель тогда присвоит сигналам d1, d2 и d3 следующие значения:

Time	Signal	Value
0	d1	0011
0	d2	0110
0	d3	1000
50	d1	0101
50	d2	0111
50	d3	0010
150	d1	1100
150	d2	0001
150	d3	1001

К сожалению, VHDL-93 не совместим с VHDL-87 стандартом, когда речь идет о текстовом вводе-выводе. Вышеупомянутый пример может эмулироваться в VHDL-87 стандарте, но если должен использоваться VHDL-93, следующая строка в коде должна быть заменена:

```
    file my_file: text is  in “my_inputs.vec”;  -- VHDL-87
    file my_file: text open read mode is “my_inputs.vec” - VHDL-93
```

В VHDL-93, read_mode задается по умолчанию и может не определяться в вышеупомянутом примере, то есть строка могла быть написана следующим образом:

file my_file: text **is** “my_inputs.vec” - VHDL-93

Другое отличие между VHDL стандартами 1987 и 1993 - то, что в один и тот же файл можно и писать и читать в одной эмуляции в VHDL-93, хотя и не одновременно. Этот метод использования текстового ввода-вывода в испытательных панелях очень гибок и довольно прост. Недостаток в том, что запросы текстового ввода-вывода являются обычно весьма медленными при эмулировании по сравнению с обычным кодом VHDL.

Также важно указать, что запрос на текстовый ввод-вывод - не синтезируемый, и может использоваться только для имитации.

7.6 Упражнения

1. Каковы различные части испытательной панели?
2. Каковы основные различия для трех классов испытательных панелей?
3. Что является генератором форм волн в VHDL?
4. Какова разница между простым и сложным генератороми форм волн?
5. Что такое текстовый ввод-вывод (TextIO)?
6. Как может текстовый ввод-вывод (TextIO) использоваться в испытательной панели?
7. Спроектируйте испытательную панель, чтобы проверить следующий компонент:

```
Library ieee;
Use ieee.std_logic1164.ALL;
Use ieee.std_logic_unsigned.ALL;
Entity my_comp is
 port( clk,resetn,a,b: in std_logic; c,d: in std_logic_vector(2      downto   0);
 ql,q2: out std_logic; q3: out stdJogic_vector(5 downto 0));
end;
Architecture rtl of my_comp is
type state_type is (s0,sl,s2);
signal state:state_type;
signal q3_b): std_logic_vector(5 downto 0)
begin
q3<=q_b); q3_b<=c * d;
end rtl
```

8 АВТОМАТЫ С ПАМЯТЬЮ

Использование автоматов с памятью - это эффективное средство реализации функций управления. Например, в фон Неймановской структуре (CPU) были бы нужны фазы выборки и исполнения, шина данных, регистры АЛУ, и т.д. Производительность автоматов с памятью гораздо выше, чем у CPU, так как поведенческая модель закодирована в автомате с помощью комбинационной схемы, которая сравнима с булевыми функциями; состояние хранится в некотором количестве триггеров. Следующий код может исполняться ЦПУ или автоматом с памятью, описанном на VHDL:

```
if a>37 and c<7 then
state <= alarm;
out_a <= '0';
out_b <= '0';
out_analog <= a+b;
else
state <= running;
end if;
```

Если код реализуется в CPU, он транслируется примерно в 10-20 машинных инструкций. Исполнение программы требует различного числа ассемблерных инструкций, в зависимости от выбора ветки программы в операторе if. Это значит, что определить точное время исполнения невозможно, и приходится использовать вместо него минимальное и максимальное время исполнения. Если тот же код исполняется триггерами, он будет исполнен за один такт. **Вывод: производительность и детерминизм автоматов с памятью исполненных на комбинационных схемах и триггерах значительно лучше, чем у CPU.**

Автомат с памятью работает в две фазы. Во время первой фазы вычисляется новое состояние, и во второй фазе новое состояние записывается в регистры. Единственное что определяет максимальную тактовую частоту, с которой может работать автомат с памятью это время, необходимое для «вычисления» следующего состояния. В предыдущем примере было показано что условие «a>37 and c<7» определяет, будет ли в следующий раз состояние «alarm» или «running» На рис.8.1 дано графическое объяснение этого примера. На каждой строке новое значение записывается в регистр. Это значит, что внутренние сигналы могут быть нестабильны между этими операциями записи в регистр. Они нестабильны, потому что входные сигналы логической схемы меняются на каждом такте, и изменение сигнала требует определенное время, прежде чем все установятся все уровни сигналов. Сигналы для автомата с памятью должны быть стабильны ко времени начала следующего такта.

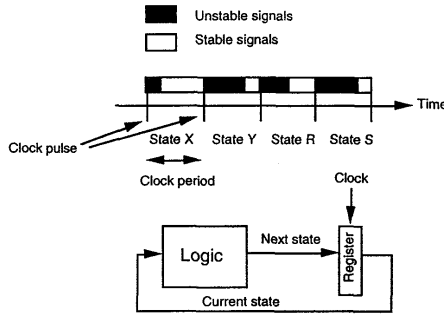


Рисунок 8.1 - Автомат с памятью.

Есть два базовых типа автоматов с памятью: автоматы Мура и Мили. Разница между автоматами Мура и Мили лежит в функции формирования выходных сигналов.

- Автомат Мили: Функция формирования выходных сигналов автомата Мили является функцией текущего состояния и всех входных сигналов.
- Автомат Мура: Функция формирования выходных сигналов автомата Мура является функцией, зависящей только от текущего состояния.

Кроме того, автомат Мили всегда работает на один такт впереди автомата Мура, так как автомат Мили изменяет выходные сигналы непосредственно после изменения входных сигналов. Автомату Мура необходимо сначала изменить состояние, то есть подождать один такт, прежде чем изменять выходные сигналы.

Следующий пример объясняет построение автомата с памятью. В качестве примера взят индикатор направления вращения для определения направления вращения (POS and NEG), например в двигателе. Направление вращения определяется двумя пульсирующими сенсорами, которые дают сигнал '1' для белого фона и '0' для черного. Схема использует автомат с памятью и карты Карно для трансляции в комбинационную схему (NAND) и минимизации. Измерительное оборудование представлено на рис.8.2.

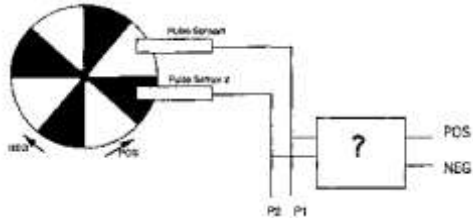


Рис. 8.2 - Схема архитектуры

Пульсирующий сенсор производит два импульса, смещенных по фазе на 90 градусов, по линиям P1 и P2. Направление движения определяется как положительное (POS-1, NEG-0) или отрицательное (POS=0, NEG=1) в зависимости от того, какой импульс придет первым. Возможно также изменять чувство направления в любой момент времени (рис.8.3). Направление вращения определено только когда входные сигналы P1 и P2 изменяются с «00» к «01» или «10».

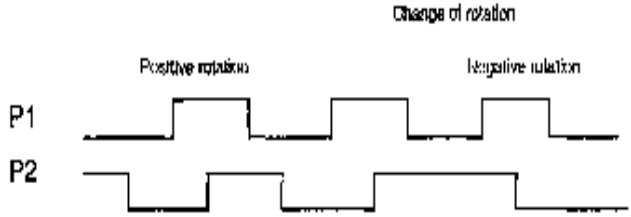


Рисунок 8.3 - Сигналы P1 и P2.

Описание проекта

Автомат Мили с четырьмя состояниями будет достаточен для описания управляющего блока (рис. 8.4). Для описания четырех состояний достаточно двух бит.

Состояние

Состояние закодировано следующим образом:

(00) + «Положительное» направление

(11) - «Отрицательное» направление

(01) Pos «Положительное» направление

(10) Neg «Отрицательное» направление

Состояние «+» (см. рис. 8.4) может измениться на состояние «Pos» если входной сигнал примет значение «01» (P2, P1) или «Neg» если входной сигнал примет значение «10». Одновременно с изменением состояния, выходной сигнал принимает значение «0» (Pos Neg). Это характерно для автомата Мили.

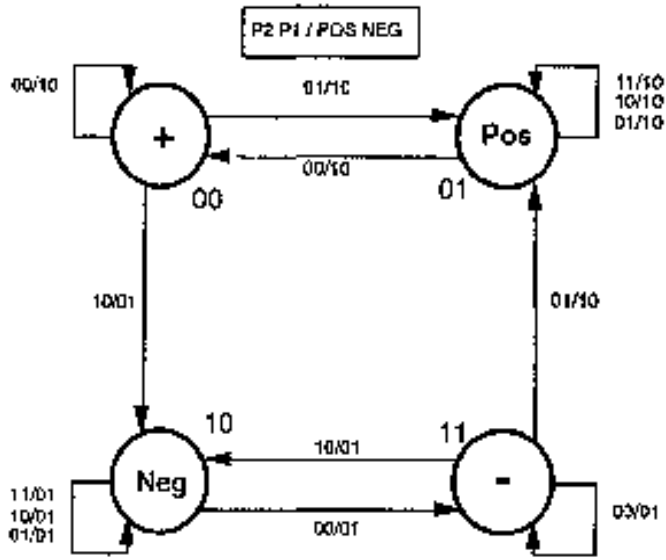


Рисунок 8.4 - Автомат с памятью

Карта Карно используется для минимизации и описания логических функций. Карты Карно описаны во всех книгах по основам электроники. Автомату соответствуют следующие булевы функции:

$D2=q_2p_1+q_2q_1+q_1p_2= ((q_2p_1)'\cdot(q_2q_1)'\cdot(q_1p_2)')'$

$D1=q_2p_1+q_1p_1+q_2p_2p_1+q_2q_1p_2= ((q_2p_1)'\cdot(q_1p_1)'\cdot(q_2p_2p_1)')'\cdot q_2q_1p_2=$
$=((((q_2p_1)'\cdot(q_1p_1)'\cdot(q_2p_2p_1)')')'\cdot q_2q_1p_2)')'$

$POS=q_2p_2+q_1p_1+q_2q_1= ((q_2p_2)'\cdot(q_1p_1)'\cdot(q_2q_1)')'$

$NEG= q_2p_1+q_2q_1+q_1p_2=((q_2p_1)'\cdot(q_2q_1)'\cdot(q_1p_2)')'$

D1 и D2 описывают следующее состояние и являются входами для двух триггеров, q1 и q2 описывают текущее состояние и являются выходными сигналами триггеров. В булевых функциях D2 и NEG совпадают, так что для них используется один и тот же участок схемы. POS является инверсией NEG, и для него тоже не понадобится собственный участок схемы. Вместо этого делается ответвление от сигнала NEG и инвертируется. В результате из булевых функций получена схема, показанная на рис. 8.5.

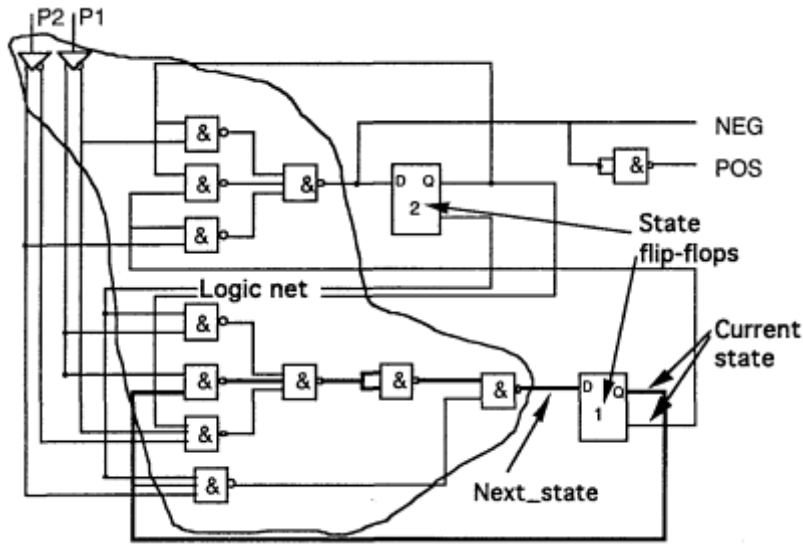


Рисунок 8.5 - Схема уровней ключей

Триггеры 1 и 2 хранят состояние автомата. Синхросигнал и сброс не нарисованы на схеме, но они подключены к триггерам. Часть схемы, реализующая логику выбора следующего состояния, обведена линией. Выходные сигналы NEG и POS соединены с «логической» частью схемы. Это означает, что в данных сигналах могут возникать гонки. Самый длинный путь в схеме – 4 логических элемента (к триггеру D1, обозначено жирной линией), так что, считая, что задержка равна 2.5 нс на каждом элементе, получаем максимальную тактовую частоту 100 МГц (10 нс).

Схема на рис.8.5 могла бы быть принята как упражнение при трансляции с уровня RTL на уровень комбинационных схем, но была бы отвергнута (забракована) как схема RT уровня. Комментарии по поводу решения данной проблемы следующие:

- Изменение направления вращения может быть определено только если предыдущий входной сигнал был «00».
- Выходы не свободны от всплесков

Эти замечания настолько серьезны, что требуется перепроектирование. Современным проектировщикам не обязательно уметь пользоваться картами Карно, минимизировать функции, транслировать в зависимые от используемой технологии комбинационные схемы или чертить схемы. Все это исполняется автоматически с помощью инструментов синтеза RTL уровня.

8.1 Автомат Мура

Как можно видеть на диаграмме (рисунок 8.6), выходные сигналы являются функцией вектора состояния, то есть реализуются как комбинационная логическая схема между вектором состояния и выходами. Автомат Мура также может быть представлен в виде диаграммы состояний (рис. 8.7).

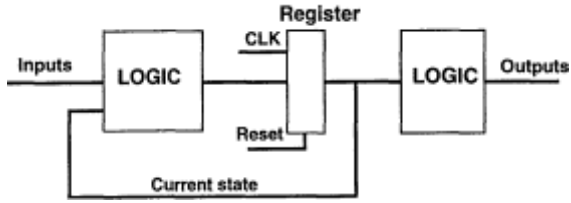


Рисунок 8.6 - Диаграмма автомата Мура

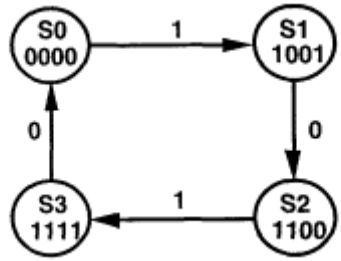


Рисунок 8.7 - Диаграмма состояний автомата Мура

Как показано на рис. 8.6, выходные сигналы зависят только от текущего состояния автомата, из-за чего выходные сигналы обычно изображаются внутри вершин состояний. Нули и единицы внутри «кружков» на рис. 8.7 являются значениями выходных сигналов, то есть выходной сигнал состояния S0 – «0000», а для состояния S1 – «1001» и т.д. Не следует путать эти значения с кодами, используемыми для обозначения состояний S0, S1, S2, S3. На рисунке 8.7 ничего не сказано о том, как были закодированы состояния. Кодирование состояний не влияет на поведение автомата, и поэтому не нуждается в описании на VHDL. Развитые инструменты синтеза часто имеют в своем составе оптимизатор автоматов с памятью. В таком случае, возможно выбрать кодирование состояний в инструменте синтеза для достижения желаемой производительности и площади схемы.

Пример автомата Мура:

```
        Entity demo is
port( clk, in1, reset:  in      std_logic;
```

```
                                                    out1:  out std_logic);
    end demo;
    Architecture moore of demo is
    type state_type is (s0,s1,s2,s3);  -- декларация состояний
    signal state: state_type;
    begin
    demo_process: process(clk. reset)
    begin
    if reset = '1' then
  state<=s0;          -- начальное состояние
    elsif clk'event and clk='1' then
    case state is
    when s0=> if in1='1' then
    state<=s1;
    end if;
    when s1=> if in1='0' then
    state<=s2;
    end if;
    when s2=> if in1='1' then
    state<=s3;
    end if;
    when s3=> if in1='0' then
    state<=s0;
    end if;
    end case;
    end if;
    end process;
    output_p: process(state)
        begin              case state is
        when s0=> out1<="0000";
        when s1=> out1<="1001";
        when s2=> out1<="1100";
        when s3=> out1<="1111";
        end case;
        end process;
        end moore;
```

Автомат с памятью состоит из трех частей: декларации, тактируемого процесса и комбинационного процесса.

Декларация

Декларация объявляет используемые состояния. Состоянию может быть дано любое имя с помощью перечислимого типа данных. С точки

зрения документации, выгодно использовать понятные имена для обозначения состояний.

Пример:

type state_type **is** (start_state, run_state, error_state);
signal state: state_type;

Использование имен также облегчает эмуляцию. Вектор состояния будет использовать значения, определенные для его типа данных, то есть start, idle, wait и error. Эти значения будут напечатаны в окне, отображающем значения сигналов, как для любых других сигналов.

Тактируемый процесс

Тактируемый процесс решает, когда автомат должен изменить состояние. Этот процесс активизируется синхросигналом автомата. В зависимости от текущего состояния и значения входных сигналов, автомат с памятью может изменять состояние на каждом активном фронте синхросигнала. После проверки на наличие активного фронта синхросигнала, соответствующая команда case используется для определения текущего состояния автомата. Команда if-then-else очень удобна для проверки значений входных сигналов. Использование команд case и if-then-else обычно дает удобочитаемый и хорошо структурированный VHDL код. Вектор состояния в предыдущем примере хранится в сигнале state, который был объявлен в декларации проекта.

Комбинационный процесс

Комбинационный процесс присваивает выходным сигналам значения в зависимости от текущего состояния. В примере, только векторы состояния перечислены в списке чувствительности для процесса output_p, что совпадает с определением выходных сигналов автомата Мура.

Главная модель – лишь один из многих способов описания автомата с памятью на VHDL. Другие модели используют три различных процесса: один для декодирования следующего состояния, один для присваивания вектору состояния current_state и один для выходных сигналов. Следующий код на VHDL немного более похож на диаграмму для автомата Мура (рис.8.8). Недостаток метода в немного большем количестве VHDL кода и несколько более медленной эмуляции. Нет большой разницы, какую модель использовать. Тем не менее, с точки зрения документирования, может быть выгодно использовать одну модель всем проектировщикам фирмы. Разницы в результатах синтеза для модели с тремя процессами, рассматриваемой ниже, и предыдущей с двумя процессами для описания автомата Мура, не будет.

Пример автомата Мура (три процесса):

```
Entity demo is
port( clk, in1, reset:   in     std_logic;
out1:  out  std_logic);
```

```
        end demo;
        Architecture moore of demo is
        type state_type is (s0,s1,s2,s3);  -- Объявление состояния
        signal current_state, next_state: state_type;
        begin
        P0:process(state.in1)
        begin
        case state is              -- Комбинационный процесс
        when s0=> if inl='1' then
        next_state<=s1;
        end if;
        when s1=> if inl='0' then
        next_state<=s2;
        end if;
        when s2=> if inl='1' then
        next_state<=s3;
        end if;
        when s3=> if inl='0' then
        next_state<=s0;
        end if;
        end case;
        end if;
        end process;
        P1: process(clk, reset)       -- Тактируемый процесс
        begin
        if reset = '1' then
state<=s0;  -- Сброс состояния
elsif clk'event and clk='1'then
        current_state<=next_state;
        end if;
        end process;
        P2: process(current state)  -- Комбинационный процесс
        begin
        case state is
        when s0=> out1<="0000";
        when s1=> out1<="1001";
        when s2=> out1<="1100";
        when s3=> out1<="1111";
        end case;
        end process;
        end moore;
```

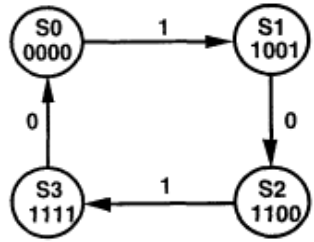


Рисунок 8.8 - Автомат Мура с тремя процессами

Некоторые несовершенные инструменты синтеза не поддерживают определение вектора состояния с помощью перечислимого типа данных. Для них необходимо выполнять кодирование состояний в VHDL коде. В некоторых таких инструментах также встречается ограниченная поддержка команды case. В таком случае VHDL код для автомата Мура может быть записан следующим образом:

```
        Entity demo is
port(clk, in1, reset:    in     vibit;
out1-:       out   vlbit_vector(3 downto 0);
end demo;
Architecture moore of demo is
type state_type is array (1 downto      0) of vibit;
constant s0: state_type =”00”  --Кодирование состояний.
        constant s1: state_type =”01”
        constant s2: state_type =”10”
        constant s3: state_type =”11”
        signal state: state_type;
        begin
        demo_process: process
        begin
        wait until prising(clk);
        if reset = ‘1’ then
        state<=s0;
        else
        if state=s0 then
        if ml=’1’ then
        state<=s1;
        end if;
        elsif state=s1 then
        if inl=’0’ then
        state<=s2;
        end if;
```

```
elsif state=s2 then
if inl='1' then
state<=s3;
end if;
else
if inl='0' then
state<=s0;
end if;
end if;
end if;
end process;
begin
if reset = '1' then
state<=s0;
elsif clk'event and clk='1' then
case state is
when s0=> if inl='1' then
state<=s1;
end if;
when s1=> if inl='0' then
state<=s2;
end if;
when s2=> if inl='l' then
state<=s3;
end if;
when s3=> if inl='0' then
state<=s0;
end if;
end case;
end if;
end process;
output_p: process(state,inl)
begin
case state is
when s0 => if inl='1' then
out1<="1001";
else
out1<="0000";
end if;
when s1=> if inl='0' then
out1<="1100";
else
```

```
out1<=”1001”;
end if;
when s2=> if inl=‘1’ then
out1<=”1111”;
else
out1<=”1001”;
end if;
when s3=> if inl=’0’ then
out1<=”0000”;
else
out1<=”1111”;
end if;
end case;
end process;
end mealy;
```

Как уже упомянуто в примерах машины Мура, имеются более простые инструментальные средства синтеза, которые не поддерживают общую Мили модель, показанную выше. Самая общая проблема состоит в том, что кодирование состояния должно быть включено в VHDL код. В этом случае VHDL код может быть написан следующим образом:

```
Architecture mealy of demo is
type state_type is array (1 downto 0) of vlbit;
constant   s0:   state_type =”00”;
constant   s1:   state_type =”01”;
constant   s2:   state_type =”10”;
constant   s3:   state_type =”11”;
signal     state: state_type;
begin
demo_process: process
begin
wait until prising(clk);
if reset = ‘1’ then state<=s0;
else
if state=s0 then if ml=‘1’ then state<=s1;
end if;
elsif state=s1 then if inl=’0’ then state<=s2;
end if;
elsif state=s2 then if inl=‘1’ then state<=s3;
end if;
else
if inl=’0’ then state<=s0;
end if;
```

```
end if;
end if;
end process;
output_p:process(state, inl)
begin
if state=s0 then if inl='l' then out1<="1001";
else
out1<="0000";
end if;
elsif state=s1 then if inl='0' then out1<="1100";
else
out1<="1001";
end if;
elsif state=s2 then if inl='1' then out1<="1111";
else
out1<="1100";
end if;
else
if inl='0' then out1<="0000";
else
out1<=="1111";
end if;
end if;
end process;
end;
```

8.2 Варианты автоматов

Машины Мили и Мура могут иметь выбросы в выводах. Это потому, что сигналы вывода исходят из комбинационной логики. Автомат может иметь свободные от выбросов выводы в некотором диапазоне температур и питающего напряжения, но иметь выбросы в другом диапазоне температур или питающего напряжения. Обычно такие выбросы - незначительны. В синхронном проекте только на активном фронте синхроимпульса данные сигнала должны быть устойчивы. Сигналы данных могут иметь выбросы вскоре после фронта синхроимпульса без наличия какого либо эффекта. Если, с другой стороны, сигналы вывода из автомата должны использоваться для схем с тремя состояниями, или как синхроимпульсы, например, выводы Мили или машины Мура не могут использоваться, поскольку они, не будут свободными от выброса.

Если свободные от выброса выводы требуются, имеются три различных варианта аввтоматов, которые могут использоваться: Машина Мура с

синхронизированными выводами; Мили машина с синхронизированными выводами.

8.3 Машина Мура с синхронизированными выводами

Автомат Мура с тактируемыми выходными сигналами

Разница между автоматом Мура с тактируемыми выходными сигналами и обычным автоматом Мура в том что все выходы синхронизируются дополнительным D-триггерами для устранения гонок. Это значит что сигналы на выходе отстают на один такт от обычного автомата Мура. Нет действительно хорошего способа представления такого автомата в виде диаграммы состояний: она совпадет с диаграммой обычного автомата (см. рис.8.9). На блочной диаграмме (рис.8.10), с другой стороны, ясно видно, что сигналы синхронизированы дополнительными D-триггерами, что значит, что выходные сигналы свободны от гонок. В VHDL коде это значит, что назначение выходных сигналов должно быть выполнено в тактируемом процессе. Это добавит к каждому выходному сигналу дополнительный D-триггер, так как все присвоения сигналам значений внутри тактируемого процесса дают в результате триггер.

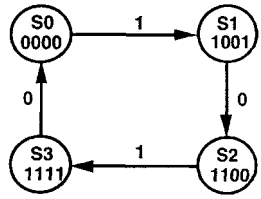


Рисунок 8.9 - Диаграмма состояний автомата Мура
с тактируемыми выходами

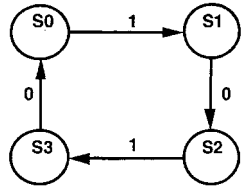


Рисунок 8.10 - Диаграмма состояний

Следующее кодирование состояний получено для различных вариантов:

Последовательное	*Серое*	*One-hot*
s0=”00”;	s0=”00”	s0=”0001”
s1=”01”	s1=”01”	s1=”0010”
s2=”10”	s2=”ll”	s2=”0100”
s3=”11”	s3=”10”	s3=”1000”

Последовательное	*Серое*	*One-hot*
s0=”00”	s0=”00”	s0=”0001”
s1=”01”	s1=”01”	s1=”0010”
s2=”10”	s2=”11”	s2=”0100”
s3=”11”	s3=”10”	s3=”1000”

Если автомат с памятью оптимизируется без особого оптимизатора состояний, получается последовательное кодирование (sequential state coding). Серое кодирование (gray coding) значит, что при изменении состояния автомата изменяется только один бит вектора состояний. При кодировании One-hot вводится столько триггеров, сколько имеется состояний. Это значит, что количество схем увеличивается, но в некоторых архитектурах FPGA (типа Xilinx) используется большое количество триггеров. Это значит, что кодирование One-hot может быть эффективно при синтезе FPGA.

8.4 Неиспользуемые состояния

Если не используется One-hot кодирование, максимальное количество состояний равно 2**N, где N равно длине вектора состояний. В автоматах, которые не используют все состояния, есть три варианта:

1. Позволить «случаю» решать, что произойдет при переходе в неопределенное допустимое состояние.

2. Определить реакцию на переход в неопределенное допустимое состояние путем добавления в конце оператора case ключевых слов when others.

Определит все возможные состояния в VHDL коде.

В первом варианте используется минимальный объем логических схем, так как не требуется декодирование неопределенных состояний. Это, однако, может привести к зацикливанию автомата в неопределенном состоянии, с единственным способом выхода из него – сбросом. Второй вариант использует больше схем, но он безопаснее. К сожалению, не все инструменты синтеза поддерживают эту альтернативу, что вынуждает пользоваться третьим вариантом, то есть определяя все возможные состояния в VHDL коде.

Пример автомата дан на рис. 8.11.

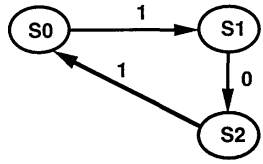


Рисунок 8.11 - Пример диаграммы состояний

На рисунке автомат с памятью имеет только три состояния. Если автомат оптимизирован как обычная схема (без оптимизатора состояний), будет получено последовательное кодирование состояний, то есть:

S0=”00”, S1=”01” S2=”10”

Состояние «11» не определено. Далее следует рассмотрение всех трех вариантов решения.

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity ex is
port( clk,resetn,a:      in    std_logic;
q:    out    std_logic);
end;
```

Вариант 1:

```
Architecture alt1 of ex is
type state_type is (s0,s1,s2)
signal state: state_type;
begin
process(clk,resetn)
begin
if resetn=’0’ then
state<=s0;
q<=0”;
elsif clk=’1’ and clk’event then
case state is
when s0 => state<=s1;
q<=’0’;
when s1 => if a=‘1’ then
state<=s2;
end if;
q<=‘1’;
 when s2 => state<=s0;
q<=0”;
end case;
end if;
end process;
end;
```

Вариант 2:

```
Architecture alt2 of ex is
type state_type is (s0,s1,s2);
signal state: state_type;
begin
process(clk.resetn)
```

```
    begin
    if resetn='0' then
    state<=s0;
    q<='0';
    elsif clk='l' and clk'event then
    case state is
    when s1 => state<=s1;
    q<='0';
    when s1 => if a='1' then state<=s2;
    end if;
  q<='1';
  when s2 => state<=s0;
  q<=0';
  when others = > state<=s0;
  q<='0';
  end case;
  end if;
  end process;
  end;
```

Вариант 3

```
  Architecture alt3 of ex is
  type state_type is (s0,s1,s2,s3)
  signal state: state_type;
  begin
  process(clk,resetn)
  begin
  if resetn='0' then state<=s0;
  q<='0';
  elsif clk='1' and clk'event then
  case state is
  when s0 => state<=s1;
  q<=0';
  when s1 => if a='1' then
  state<=s2;
  end if;
  q<='1';
  when s2 => state<=s0;
  q<=0';
  when s3. => state<=s0;
  q<='0';
  end case;
  end if;
```

```
end process;
end;
```

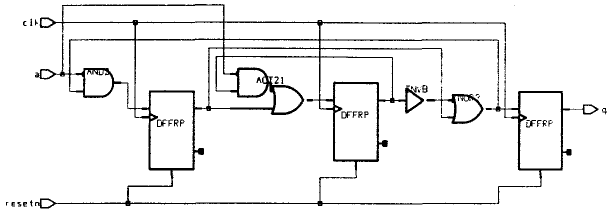


Рисунок 8.12 - Результат синтеза по второму варианту

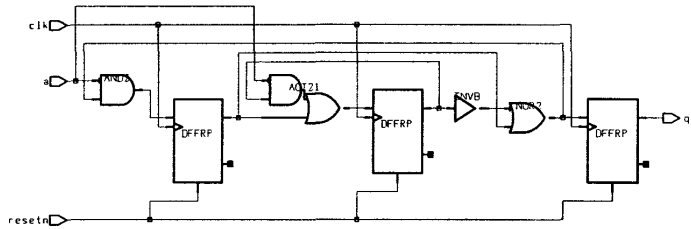


Рисунок 8.13 - Результат синтеза по третьему варианту

Как показывают рисунки 8.12 - 8.13, варианты 2 и 3 совпадают. Автомат построенный по первому варианту может «зациклиться» в неопределенном состоянии «11», например, если неисправность приведет к переходу туда. Тогда автомат будет оставаться в этом состоянии пока входной сигнал a равен ‘1’. Это поведение может быть получено только после синтеза. Результат в большей или меньшей степени случаен, в зависимости от того как инструмент синтеза оптимизирует логику. Этот вариант конечно неприемлем, если недопустимо «зацикливание» автомата в случае неисправности. Рекомендуется обработка неиспользуемых состояний, так чтобы «зацикливание» было невозможно. Первый вариант должен рассматриваться только в том случае, когда варианты 2 и 3 дают слишком большое увеличение количества схем.

Выбор варианта зависит как от схемы, так и от инструмента синтеза. Если требуется, чтобы автомат не «зацикливался» в неопределенном состоянии (в случае неполадки или рассинхронизации), должны быть выбраны варианты 2 и 3, в зависимости от используемого инструмента синтеза. Synopsys является одним из инструментов, поддерживающих альтернативу 2. Но если вы используете Synopsys и второй вариант, вы должны избегать использовать оптимизатор состояний, так как он может проигнорировать строку others. Пользуйтесь обычной оптимизацией VHDL кода.

8.5 Как написать оптимальный автомат с памятью на VHDL

Мы уже рассмотрели различные автоматы с памятью и различные варианты кодирования состояний. Выбор автомата с памятью полностью зависит от проекта (отсутствие гонок, площадь, синхронизация и т.д.). Иногда для оптимального решения требуется комбинация автоматов Мили и Мура (с тактированием и без него); то есть вы тактируете только те выходные сигналы, в которых не должно быть гонок с регистром (автоматы Мура/Мили с тактируемыми выходами), и делаете другие выходные сигналы сигналами «чистых» автоматов Мили и Мура.

Пример комбинации автоматов Мили и Мура, с тактированием и без него:

```
    Library ieee;
    Uыse ieee.std_logic_1164.ALL;
    Entity mix is
port( clk, resetn, a, b: in     std_logic;
ql:   out   std_logic; --Выход Мили
q2:   out   std_logic; --Выход Мура
q3:   out   std_logic; --S. Выход Мили
q4:   out   std_logic); --S. Выход Мура
    end;
    Architecture rtl of mix is
    type state_type is (s0,s1)
    signal state: state_type;
    begin
    process(clk,resetn)
    begin
    if resetn='0' then
    state<=s0;
    q3<=0';
    q4<=0';
    elsif clk'event and clk='1' then
    case state is
    when s0 =>   if a='1' then
    state<=s1;
    q3<='0';
    else
    q3<='1';
    end if;
q4<='1';
when s1 => if b='1' then
state<=s2;
    q3<='1';
```

```
    else
    q3<='0';
    end if;
    q4<='0';
    end case;
    end if;
    end process;
    process(state,a,b)
    begin
    case state is
    when s0 =>   if a='1' then
    ql<='0';
    else
    ql<='1';
    end if;
q2<=0-;
when s1 =>        if b='1' then
ql<='1';
elsif a='0' then
    ql<='0';
    else
    ql<='1';
    end if;.
    q2<='1';
    end case;
    end process;
    end;
```

Если возможно выбрать тип автомата «output=state», это всегда следует делать, так как это самый быстрый и компактный вариант. Единственное исключение – автомат Мили быстрее изменяет выходные сигналы. Выходные сигналы автомата Мили изменяются сразу же по изменении значений входных сигналов, в то время как автомат «**output=state»** ждет один тактовый цикл прежде чем изменит свои выходные сигналы.

Чаще всего вариант «**output=state»** возможно использовать только для относительно небольших автоматов. «**Output=state»** требует, чтобы не было двух или более состояний с одинаковыми выходными сигналами. Если выходные сигналы будут совпадать, кодирование даст одинаковые результаты для этих состояний, что, конечно, недопустимо.

Следующий пример показывает автомат, в котором выходные сигналы совпадают в состояниях S1 и S2. Иногда оправдано добавление дополнительного D-триггера в вектор состояния для введения различия между

состояниями, чтобы было возможно использование автомата типа «output=state».

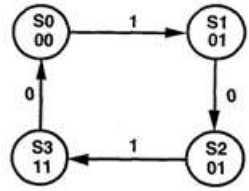


Рисунок 8.14 - Диаграмма состояний, в которой выходы двух состояний идентичны

Как видно из рис.8.14, выходные сигналы имеют одинаковое значение в состояниях S1 и S2.

В следующем примере в вектор состояния был добавлен дополнительный D-триггер, чтобы можно было использовать вариант «output=state» для автомата с рис.8.14.

```
    Architecture rtl of ex is
    constant s0: std_logic_vector(2. downto 0):=”000”;
    constant s1: std:logic_vector(2. downto 0):=”001”;
    constant s2: std_logic_vector(2. downto 0):=”101”;
    constant s3: std:logic_vector(2. downto 0):=”011”;
    signal state: std_logic_vector(2. downto 0);
    begin
    process(clk,resetn)
    begin
    if resetn=’0’ then
    state<=s0;
    elsif clk’event and clk=‘1’ then
    case state is
    when s0 =>  if a=‘1’ then
            state<=s1;
            end if;
when s1 =>      if b=‘1’ then
            state<=s2;
            end if;
when s2 =>      if a=’0’ then
state<=s3;
    end if;
    when others => state<=s0;
    end case;
    end if;
    end process;
```

```
q_out<=state(1 downto 0);
end;
```

При проектировании автомата с памятью важно определить значения выходных сигналов на каждом такте времени. Пример дан на рис. 8.15

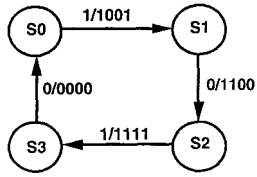


Рисунок 8.15 - Диаграмма состояний

С функциональной точки зрения такое описание корректно. Однако присвоение выходным сигналам значений происходит не на каждом активном фронте синхроимпульса. К примеру, если автомат находится в состоянии S1 и входной сигнал *a* имеет значение ‘1’, присвоение выходному сигналу значения не происходит. Это трактуется инструментом синтеза (вполне справедливо) как сохранение выходным сигналом старого значения.

```
Architecture bad of ex is
type state_type is (s0,s1,s2,s3)
signal state: state_type;
begin
process(clk,resetn)
begin if resetn=’0’ then
state<=s0;
q<=(others=>’0’);
elsif clk’event and clk=’l’ then
case state is
        when s0 =>      if a=‘1’ then
        state<=s1;
        q<=”1001”;
        end if;
        when s1 =>      if a=’0’ then
        state<=s2;
        q<=”1100”;
        end if;
        when s2 =>      if a=’0’ then
        state<=s3;
        q<=”1111”;
        end if;
        when s3 =>      if a=’0’ then
        state<=s0;
```

q<=”0000”;
end if;
end case;
end if;
end process;
end;

В предыдущем примере VHDL кода не происходит присвоения сигналу нового значения если автомат остался в том же состоянии. Диаграмма состояний на рисунке 8.16 иллюстрирует такое поведение. Если изменить диаграмму так, чтобы присвоение сигналу происходило в любом состоянии, без изменения функциональности, код будет выглядеть так:

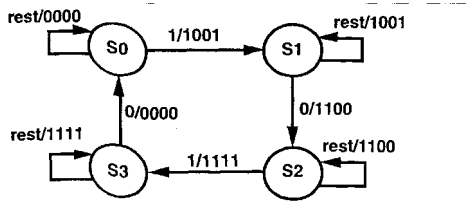


Рисунок 8.16 - Диаграмма состояний

Architecture good of ex is
type state_type **is** (s0,s1,s2,s3)
signal state: state_type;
begin
process(clk, resetn)
begin
if resetn =’0’ **then**
state<=s0;
q<=(others=>’0’);
elsif clk’evcnt **and** clk=‘1’ **then**
case state **is**
when s0 => **if** a=’1’ **then**
state<=s1;
q<=”1001”;
else
q<=”0000”;
end if;
when **s1 =>** if a=’0’ **then**
state<=s2;
q<=”1100”;
else
q<=”1001”:

```
end if;
when s2 => if a='0' then
state<=s3;
q<="1111";
else
q<="1100":
end if;
when s3 => if a='0' then
state<=s0;
q<="0000";
else
q<="1111":
end if;
end case;
end if;
end process;
end;
```

Как было ранее отмечено, поведение будет идентичным как во время VHDL эмуляции, так и после синтеза. Недостаток первого описания в том, что потребуется на 12 логических элементов больше для того, чтобы выполнять присваивания не на каждом такте. Это неизбежно, так как требуются дополнительные аппаратные средства для сохранения старого значения сигнала. Инструменты синтеза решают эту задачу, вводя обратную связь выходного сигнала с мультиплексором (рис.8.17).

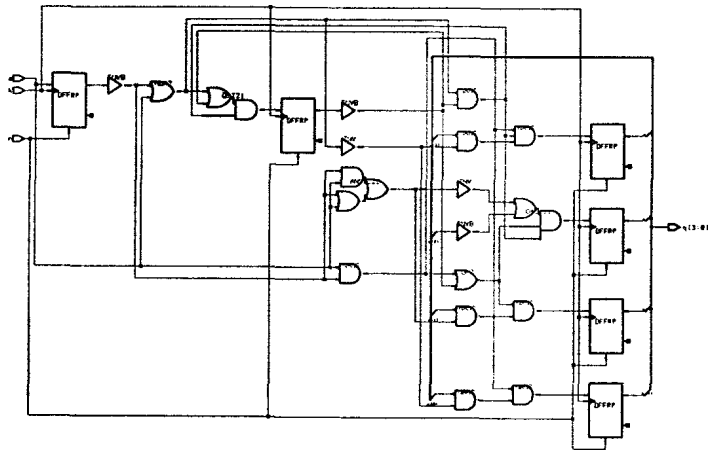


Рисунок 8.17 - Результат синтеза (без присваивания на каждом такте)

Мультиплексор посылает значение сигнала обратно на тех тактах, когда значение сигнала не изменяется. Таким образом, сохраняется значение сигнала. Если же сигналу присваивается значение на каждом такте, мультиплексор и обратная связь выходного сигнала не нужны (рис.8.18), что экономит примерно 3 элемента на выходной сигнал.

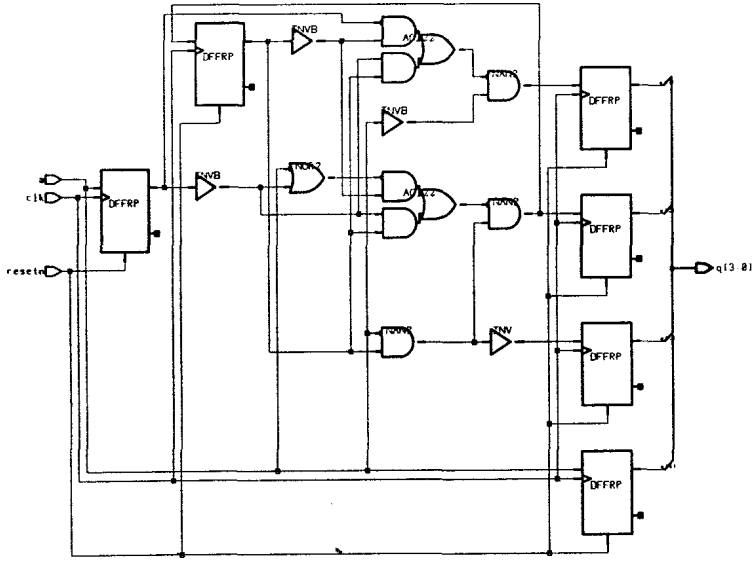


Рисунок 8.18 - Результат синтеза
(с присвоением значений на каждом такте)

Отметим, что вектор состояния не нуждается в присваивании значения на каждом такте, так как он содержит линии обратной связи, и не требует лишних логических элементов. Три логических элемента на выходной сигнал может показаться не очень большим излишком, но в небольших схемах или при использовании нескольких автоматов с памятью, этот излишек может потребовать большую площадь для реализации логики. Несколько элементов также могут оказаться «перышком, ломающим спину верблюда» в случае больших проектов. Более того, синхронизация в автомате ухудшится, если в шину будет введен излишний мультиплексор. Кроме того, возрастет емкость выходов, если у них будет обратная связь с мультиплексором. Дополнительная емкость означает, что выходы станут несколько медленнее в плане синхронизации.

8.6 Асинхронные автоматы

Все рассматриваемые до сих пор автоматы были тактируемыми, то есть синхронными. Если же, напротив, в обратную связь с вектором состояния не вводятся логические элементы (рис.8.19), автомат будет асинхронным. Преимущество асинхронных автоматов в том, что они обычно быстрее чем тактируемые. С методологической точки зрения на проектирование, асинхронные автоматы не являются оптимальным решением. Их стоит использовать, только если требования к производительности так высоки, что только использование асинхронных автоматов позволяет найти решение.

При проектировании асинхронных автоматов с памятью, проектировщик должен убедиться в отсутствии такого явления, как «гонки». Гонки – это переход автомата между несколькими состояниями в неуправляемом состоянии. В асинхронной машине очень важно кодирование состояний. Вектор состояния может изменяться только в одном бите при переходе из состояния в состояние, что означает, что кодирование должно быть выполнено в VHDL коде. Многие инструменты синтеза вообще не поддерживают асинхронные автоматы. Те же которые поддерживают, не могут эффективно оптимизировать их площадь и синхронизацию.

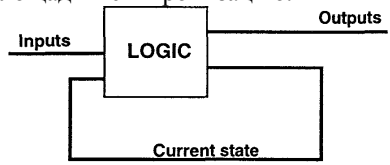


Рисунок 8.19 - Асинхронный автомат

Диаграмма состояний для асинхронного автомата (рис. 8.19) представлена ниже:

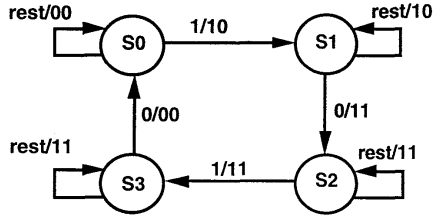


Рисунок 8.22 - Диаграмма состояний

Можно записать следующий VHDL код:

```
Library ieee;
Use ieee.std_logic_1164.ALL;
Entity test is
```

```
port( a, resetn:   in       std_logic;
q:     out     std_logic_vector(1 downto 0));
end;
Architecture rtl of test is
type state_type is array (1 downto 0) of std_logic;
constant s0: state_type:=”00”;
constant s1: state_type:=”01”;
constant s2: state_type:=”11”;
constant s3: state_type:=”10”;
signal state: state_type;
begin
process(resetn,state,a)
begin
if resetn=’0’ then
state<=s1;
q<=”10”;
else
case state is
when s0 =>if a=’1’ then
state<=s1;
q<=”10”;
else
state<=s0;
q<=”00”;
end if;
when s1 =>if a=’0’ then
state<=s2;
q<=”11”;
else
state<=s1;
q<=”10”;
end if;
when s2 =>if a=‘1’ then
state<=s3;
q<=”10”;
else
state<=s2;
q<=”11”;
end if;
when others => if a=’0’ then
state<=s0;
q<=”11”;
```

```
else
state<=s3;
q<=”11”;
end if;
end case;
end if;
end process;
end;
```

Результат синтеза показан на рис.8.23.

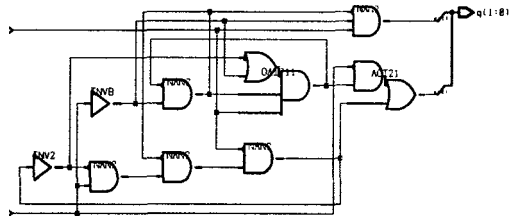


Рисунок 8.23 - Результат синтеза

8.7 Упражнения

1. В чем разница между автоматами Мили и Мура?
2. Являются ли выходы автоматов Мили и Мура свободными от гонок?
3. Назовите две ситуации, в которых необходимы выходы автомата без гонок.
4. Возможно ли сочетание различных автоматов в одной схеме?
5. Что такое неиспользуемые состояния в автоматах с памятью?
6. Какие есть методы описания по отношению к неиспользуемым состояниям?
7. Влияет ли кодирование на функционирование автомата Мили/Мура?
8. Что значит серое, последовательное и one-hot кодирование?
9. Могут ли различные виды кодирования быть одинаково подходящими для различных технологий?
10. Если вектор состояния определен следующим образом:

type state_type is (s0, s1, s2, s3, s4);

signal state: state_type;

11. Сколько триггеров потребуется для последовательного / one-hot кодирования?
12. В каком автомате выходы изменяются быстрее всего после изменения входов?

13. Какое требование налагается на выходные сигналы, если используется автомат типа «output=state»?

14. Что происходит, если в автомате Мили/Мура с синхронизацией выходов не происходит назначение значений выходам на каждом такте?

15. Рис. 8.24 задает состояние и блочную диаграмму автомата. Напишите VHDL код для объекта и архитектуры. При reset_n='0' автомат должен инициализируется состоянием S0.

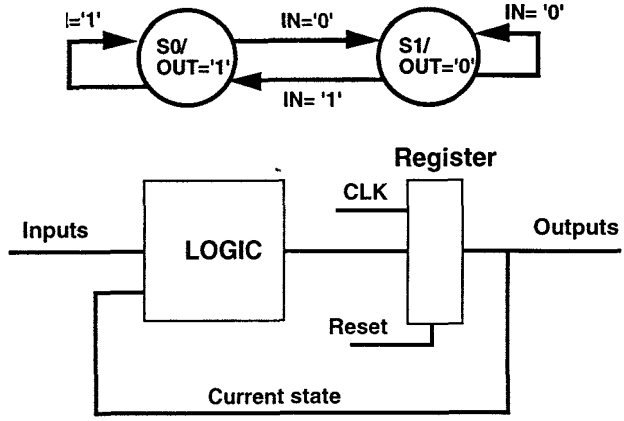


Рисунок 8.24 - Граф состояний и блочная диаграмма автомата

Нарисуйте законченную блочную диаграмму и запишите код для следующего автомата Мура (со всплесками) (рис. 8.25).

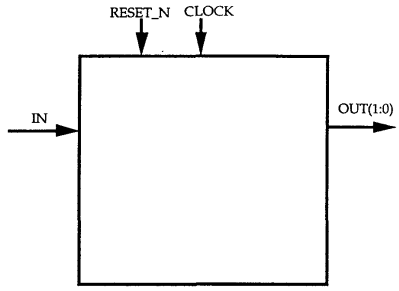


Рисунок 8.25 - Интерфейс проектируемого автомата

СПИСОК ЛИТЕРАТУРЫ

1. П.Н. Бибило. Основы VHDL языка. ”Солон-Р”, М.:2000, 200 с.
2. Прихожий А.А. Язык описания аппаратуры VHDL. http://ns.kv.minsk.by.
3. Active-VHDLTM Series book 1 VHDL Reference Guide
4. Active-HDL™Series Version 3.5 BOOK # 3 GETTING STARTED GUIDE
5. ACTIVE-HDL™SERIES APPLICATION NOTES Version 3.5