УДК681.326:519.713

ИНФРАСТРУКТУРА ПРОЕКТИРОВАНИЯ SoC ДЛЯ МЕТОДА МУЛЬТИВЕРСНОГО СИНТЕЗА

ОБРИЗАН В.И.

Предлагается программно-аппаратная реализация моделей, методов и структур данных для проектирования цифровых систем на кристаллах, которая включает процедуры создания спецификации, синтеза, тестирования, моделирования и верификации на основе инфраструктуры, учитывающей промышленные средства компаний Aldec и Xilinx. Рассматриваются вопросы тестирования программных продуктов на реальных цифровых проектах создания IP-Core как примитивов для реализации цифровых систем на кристаллах.

Введение. Общая характеристика исследования

Цель — разработка и тестирование инфраструктуры проектирования цифровых систем на кристаллах, которая характеризуется параллельным выполнением мультиверсного синтеза функциональности, обеспечивающей существенное уменьшение времени создания проекта в условиях ограничения на аппаратные затраты.

Задачи:

1. Разработка метода мультиверсного синтеза управляющих и операционных автоматов в заданной инфраструктуре проектирования, ориентированных на архитектурные решения в метрике, минимизирующего время выполнения функциональности за счет распараллеливания операций при ограничении на аппаратные затраты.

2. Программная реализация моделей и методов мультиверсной разработки операционных устройств в рамках интегрированной системы проектирования функциональных и архитектурных решений SoC на основе использования продуктов верификации и синтеза компаний Aldec и Xilinx.

3. Тестирование и верификация программных модулей инфрастурктуры проектирования цифровых систем на кристаллах, а также определение эффективности предложенных моделей, методов и структур данных при создании реальных компонентов цифровых изделий.

1. Организация системы автоматизированного проектирования

По своей сути программа синтеза C++ в VHDL является машиной по трансформации исходного описания в результирующее. Таких трансформаций происходит несколько. Входная модель представлена на языке C++. Это алгоритмическое описание, которое решает поставленную перед инженером задачу.

Первый этап преобразования – *синтаксический анализ*. На этом этапе во входном описании из потока символов выделяются лексические элементы: ключевые слова, операторы и лексемы. В результате этого этапа получается синтаксическая модель исходного алгоритмического описания.

Второй этап преобразования – *трансформации на уровне синтаксической модели*. Они применяются для получения моделей с меньшим количеством состояний и занимающих меньше аппаратных ресурсов. К таким трансформациям относятся: вычисление констант, удаление недостижимого кода, встраивание функций, операции над циклами (развертки, свертки, распараллеливания).

Третий этап преобразования – *построение граф-схемы алгоритма*. В этой модели алгоритм представлен в виде отношений вершин двух типов: операций (арифметических, логических или ввода/вывода) и ветвлений.

Четвертый этап преобразования – *построение автоматной модели*. Она представлена в виде вершин с операциями и переходов между ними.

Пятый этап преобразования — *синтез операционного и управляющего автоматов*. В результате этого этапа получается структурная модель, определенная на множестве логических элементов, регистров, арифметико-логических устройств и элементов памяти.

Шестой этап преобразования – *сохранение модели операционного и управляющего автомата в VHDL-код*. Результатом этого этапа является набор исходных файлов на языке VHDL, которые описывают устройство на уровне регистровых передач. Эта VHDL-модель реализует исходный алгоритм, изначально составленный на языке C++.

2. Тестирование системы

Тестирование системы (рис.1)состоит из нескольких этапов:

– разработка схемы тестирования компилятора;

– подготовка тестов;

– подготовка эталонов;

– исполнение тестов и анализ результатов.

Были подготовлены тесты для различных частей компилятора: синтаксический анализатор; построитель граф-схемы алгоритма; построитель цифрового автомата; построитель VHDL-модели синтезируемого устройства.

При подготовке тестов к синтаксическому анализатору учитывались следующие особенности компилятора:

– работа препроцессора;

– поддержка базовых типов языка C++;

– поддержка базовых конструкций языка C++.

Тест для компилятора — это исходный текст на языке C++. Примитивный тест показан в листинге 1. Например, этот тест проверяет успешную декларацию и инициализацию объектов типа int, декларацию функции, арифметическую операцию «сложение».

Листинг 1. Примитивный тест для компилятора

```
1. int f1()
2. {
3. int a = 2, b = 3;
4. return a + b;
5. }
```

При выполнении теста сохраняется возвращаемое значение функции в файл для последующего сравнения с эталоном. Эталоны были подготовлены с использованием компилятора Microsoft Visual Studio. Было предположено, что он не содержит ошибок.

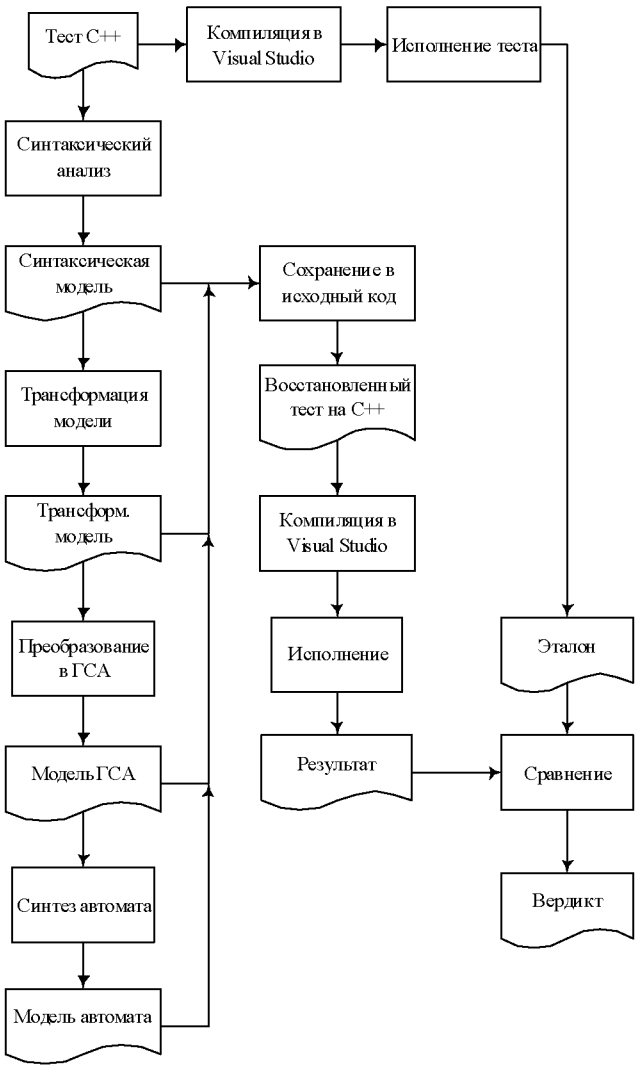

Рис. 1. Схема тестирования различных внутренних моделей компилятора

3. Сравнение с аналогами

В американском патенте № 6 226 776 «System for Converting Hardware Designs in High-Level Programming Language to Hardware Implementations» [1] предлагается система автоматизированного проектирования, которая конвертирует алгоритмическое описание на языке ANSI C в аппаратные реализации на ПЛИС или на кристаллах жесткой логики. В патенте предложены многочисленные примеры преобразований алгоритмов из языка C++ в синтезируемое подмножество языка Верилог. Рассмотрим пример, показанный в листинге 2.

Программа высокоуровневого синтеза, предложенная в этой работе, произвела оптимизированную ГСА, как показано на рис. 2.

При компиляции примера были сделаны следующие оптимизации: встраивание функций, выявление параллелизма на уровне микроопераций. Таким образом, после синтеза мы получили модель автомата Мили с 7-ю состояниями. В патенте показано, что в результате конвертации этого примера получается 16 состояний автомата.

Листинг 2. Исходный алгоритм на языке C++

```
6.  int sum1 (int n)
7.  {
8.   int i, sum = 0;
9.   for (i = 0; i < n; i++)
10.  sum += i;
11.  return sum;
12. }
13.
14. int sum2 (int array[], int size)
15. {
16.  int i, sum = 0;
17.  for (i = 0; i < size; i++)
18.  sum += array[i];
19.  return sum;
20. }
21.
22. int f1()
23. {
24.  int i;
25.  int array[10];
26.  int size = sizeof(array)/sizeof(*array);
27.  for (i = 0; i < size; i++)
28.  array[i] = i * 2;
29.  return sum1(size) + sum2(array, size);
30. }
```

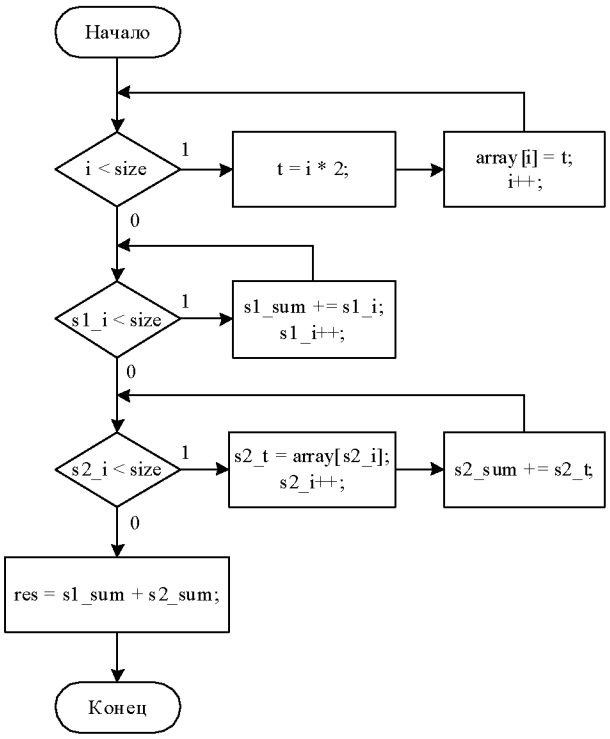

Рис. 2. СГСА, полученная в результате синтеза

4. Проектирование системы на кристалле

Рассмотрим систему на кристалле, показанную на рис. 3.

Рис. 3. Архитектура тестовой системы на кристалле

В данной тестовой системе на кристалле используется микропроцессор OpenRISC 1200 [2]. OpenRISC — это 32-разрядый RISC-микропроцессор с открытым исходным кодом. Модель процессора распространяется на правах свободной лицензии, что позволяет его изучать, моделировать, развивать и использовать в коммерческих системах без уплаты авторских отчислений. OpenRISC сделан на основе гарвардской архитектуры, где предполагается раздельное хранение инструкций и данных. Микропроцессор имеет пятиступенчатый конвейер, тем не менее, большинство инструкций могут быть выполнены за один такт.

В этой системе используется шина Wishbone на тех же правах.

Для компиляции исходных текстов программ на языке C применяется компилятор gcc.

При запуске системы микропроцессор начинает читать инструкции по адресу 256.

5. Общая организация системы проектирования

На рис. 4 показана структура процесса проектирования с использованием пространства проектных решений. На начальных этапах осуществляется ввод спецификации проекта на системном уровне с применением языков C++ и SystemC. Ввод моделей осуществляется как в текстовом виде, так и в графическом с использованием блочных диаграмм и содержательных граф-схем алгоритмов. При определении спецификации указываются функциональные и нефункциональные требования. Первые описывают закон функционирования модели, а также входные воздействия на систему и ожидаемые ответные реакции. Среди нефункциональных требований можно выделить следующие:

– ограничения на получаемые проектные решения: стоимость реализации, быстродействие, производительность, аппаратные затраты, затраты оперативной памяти, энергопотребление;

– доступные архитектуры: типы интегральных схем, элементная база, а также их характеристики; типы встроенных микропроцессоров.

Исходные тексты входных спецификаций должны соответствовать международным стандартам C++ [3] и SystemC [4].

На следующем этапе осуществляется синтаксический анализ исходной модели. В случае обнаружения ошибок создается отчет, на основании которого инженер может исправить ошибки и повторить трансляцию модели. Если исходные тексты модели не содержат ошибок, то строится синтаксическая модель проектируемой системы.

Синтаксическая модель пригодна для широкого класса задач автоматизированного проектирования. Среди них: декомпозиция проекта на программную и аппаратную части, высокоуровневая оптимизация, логический синтез и компиляция, построение проверяющих тестов.

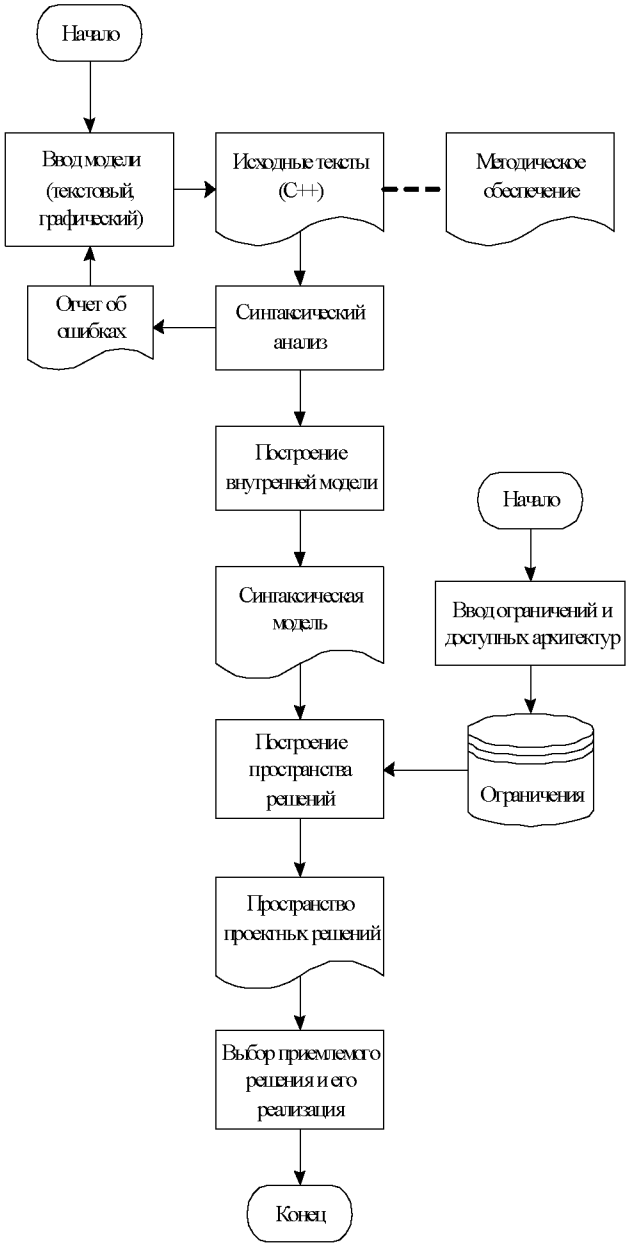

Рис. 4. Структура процесса проектирования [5] с использованием пространства проектных решений

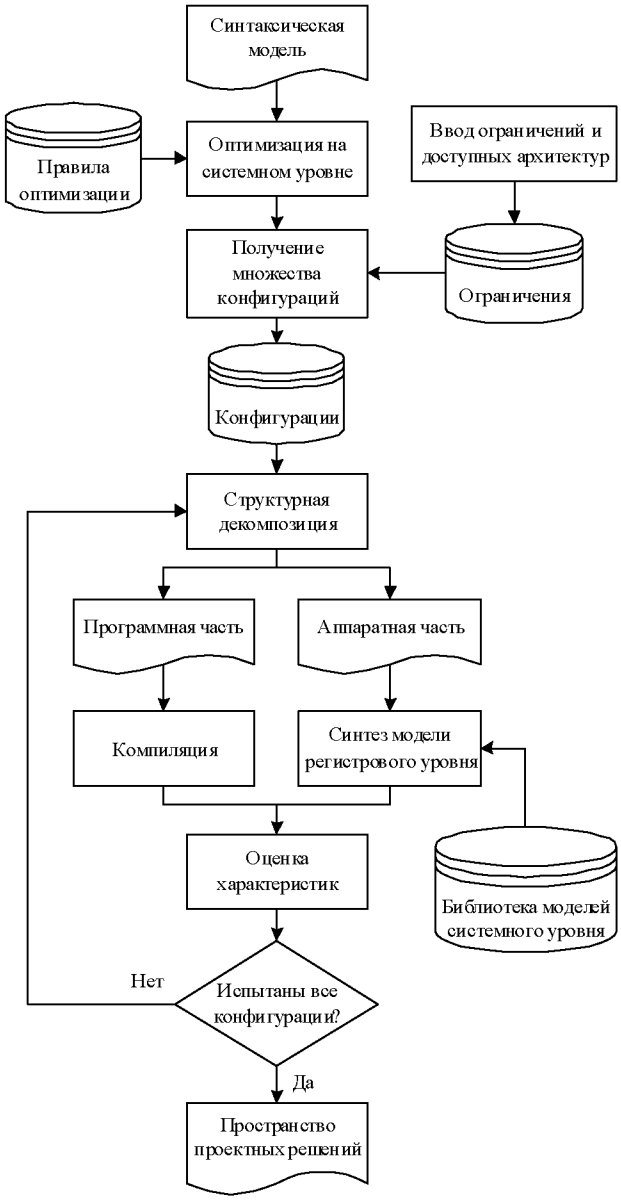

Рис. 5. Процесс построения пространства проектных решений

6. Структурная декомпозиция проекта

Для успешной реализации проекта необходимо решить задачу структурной декомпозиции проекта на программную и аппаратную части. Введем несколько определений, которые широко используются в языке SystemC.

Определение 1. Модуль — класс C++, унаследованный от класса *sc_module*, объявленного в библиотеке SystemC. В языке VHDL понятию модуля соответствует пара *entity* и *architecture*. Модуль имеет определенный интерфейс — множество входных и выходных сигналов определенного типа, а также множество параллельно взаимодействующих процессов с соответствующими списками чувствительности. Процесс в SystemC аналогичен конструкции *process* в языке VHDL или *always* в языке Verilog.

Определение 2. Иерархия модулей — множество экземпляров модулей, созданных при компиляции SystemC модели. На этом множестве определены

отношения «А есть дочерний модуль Б», «А есть родительский модуль Б».

Определение 3. Модуль верхнего уровня — экземпляр модуля, который не установлен ни в один другой модуль. В случае несвязанной иерархии модулей может быть несколько модулей верхнего уровня.

Рассмотрим блочную диаграмму проекта *sc_main*, показанного на рис. 6 (исходный текст модели на языке SystemC может быть найден в [6]). Блочная диаграмма отражает иерархию модулей, а также отношения модулей между собой. Здесь модули *m_producer* и *m_consumer* являются дочерними по отношению к модулю *testbench*, который в свою очередь является дочерним к модулю *sc_main*.

Проект цифровой системы, описанный на языке SystemC, удобно представить в виде корневого дерева, определенного на множестве иерархии модулей проекта. Это дерево отображает отношение «быть установленным в». Корень дерева — модуль верхнего уровня, в котором установлены все другие экземпляры модулей низших уровней. На рис. 7 показано корневое дерево для проекта sc_main.

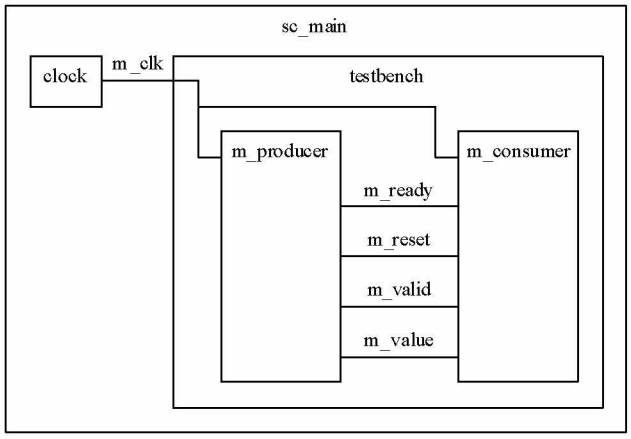

Рис. 6. Блочная диаграмма проекта sc_main

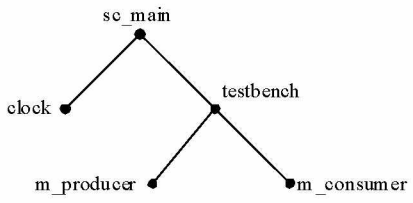

Рис. 7. Представление иерархии модулей в виде корневого дерева

Для реализации каждого модуля есть две альтернативы: аппаратная реализация и программная. Количество конфигураций при таком условии равно $k = 2^n$, где n — количество модулей в иерархии. Это количество может быть сокращено за счет того, что не все модули могут быть логически синтезированы в аппаратурные структурные модели. Таким образом, количество конфигураций может быть определено $k = 2^m, m \le n$, где m — количество модулей, для которых возможно построить корректную аппаратную модель из SystemC описания. Например, для модели на рис. 4 k = 32.

7. Методы структурных и алгоритмических трансформаций

Существует компромисс между объемом аппаратных ресурсов и временем выполнения той или иной задачи. Аппаратура обладает громадными возможностями в плане выполнения функций с большим уровнем параллелизма. При распараллеливании функция может быть выполнена быстрее, но для этого требуются дополнительные аппаратурные затраты.

Методы структурной оптимизации заключаются в применении различных трансформаций моделей или описаний, которые приводят к улучшению временных или аппаратных характеристик устройства.

Методы могут быть применены на различных уровнях абстракции. На алгоритмическом уровне объектом оптимизации становится алгоритм работы устройства. На микроархитектурном уровне объектом оптимизации выступает структурная модель операционного автомата, а также графа управляющего автомата. Базовые сведения о минимизации условных и операторных вершин автоматов могут быть найдены в работах С.И. Баранова [7].

Векторизация – такая трансформация цикла, при которой циклические действия выполняются сразу над множеством операндов, а не над одним. В классификации Флинна такой организации вычислений соответствует модель SIMD – Single Instruction, Multiple Data –одна операция, множество операндов.

Циклы – типичная горячая точка в приложении. Даже быстрая операция или функция, выполненная миллион раз, замедлит работу системы. Кроме полезной нагрузки: тела цикла, имеются и накладные расходы на организацию цикла: счетчик итераций (регистр), операция инкремент/декремент, ветвление. При векторизации цикла используется операция «развертка цикла». Операнды объединяются в векторы с определенным числом элементов, называемым *степень векторизации цикла*. Как правило, операция выполняется за один такт над всеми элементами вектора.

Рассмотрим пример, показанный в листинге 3. Здесь идет последовательное суммирование элементов двух векторов, результат записывается в третий вектор.

Листинг 3. Последовательная обработка элементов массива в цикле

```
31. for (int i = 0; i < N; i++)
32.     c[i] = a[i] + b[i];
```

На рис. 8 схематически показано выполнение арифметических операций в оригинальном цикле (а) и в векторизованных циклах (б, в). В первом случае за одну итерацию выполняется действие над одной парой операндов, а во втором и третьем случаях – над двумя и четырьмя парами операндов.

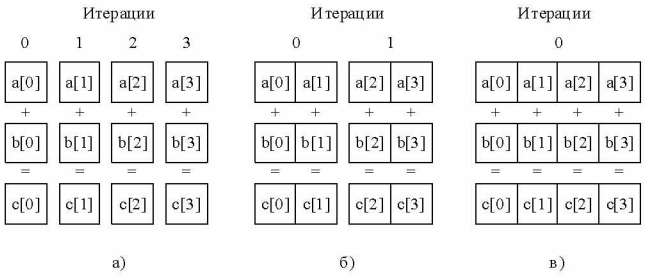

Рис. 8. Выполнение операций: а – за четыре итерации в оригинальном цикле; б – за две итерации в цикле со степенью векторизации 2; в – за одну итерацию в цикле со степенью векторизации 4

На рис. 9 приведена содержательная ГСА для этого примера. Таким образом, цикл включает в себя четыре состояния автомата. Для выполнения программы суммирования векторов из N элементов потребуется $k = 4 \times N$ тактов работы устройства. Аппаратные затраты: 4 регистра (a1, b1, c1, i), 2 сумматора, функция сравнения «меньше».

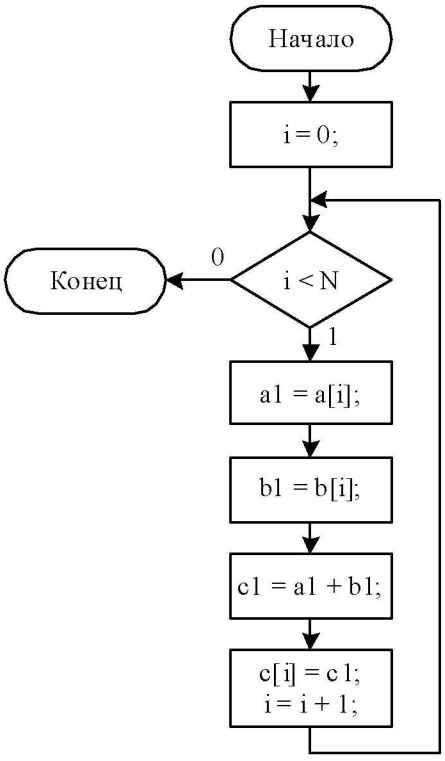

Рис. 9. СГСА для примера в листинге 3

Рассмотрим пример, показанный в листинге 4. Осуществлена трансформация «векторизация цикла». Здесь на одном витке цикла выполняется две операции сложения. Важное условие — N должно быть кратно степени векторизации.

Листинг 4. Векторизованный цикл из листинга 1

```
33. for (int i = 0; i < N; i += 2) {
34.     c[i] = a[i] + b[i];
35.     c[i+1] = a[i+1] + b[i+1];
36. }
```

На рис. 10 показана СГСА для примера из листинга 4. Цикл включает в себя 7 состояний автомата. Для выполнения программы суммирования векторов из N элементов потребуется k=(7xN)/2 тактов работы устройства. Аппаратные затраты: 7 регистров (a1, a2, b1, b2, c1, c3, i), 3 сумматора, функция сравнения «меньше».

В этом примере видно, что в теле цикла присутствует четыре операции чтения из памяти и две операции записи в память, которые должны выполняться последовательно. Две операции сложения могут быть выполнены параллельно. Можно подсчитать, что цикл со степенью векторизации l, в котором содержится s операций (выполняются последовательно) и p операций (выполняются параллельно), может быть выполнен за k тактов, что определяется формулой:

$$k = \frac{(s+1) \times N}{l}. \qquad (1)$$

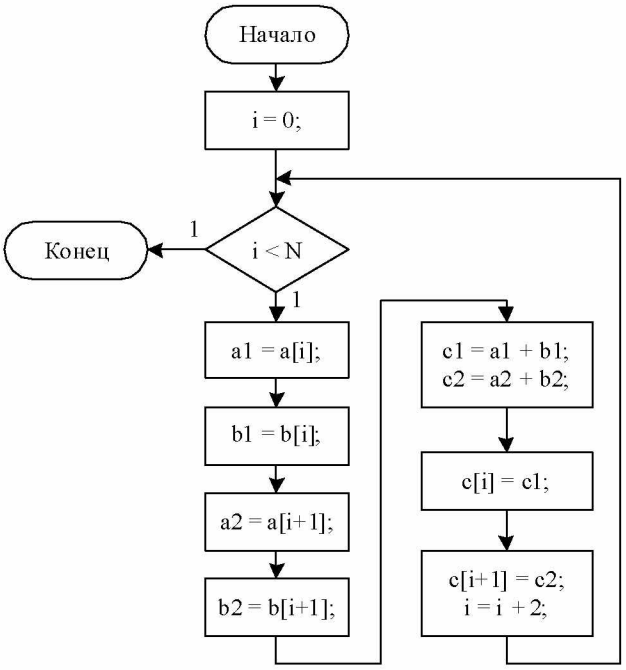

Рис. 10. СГСА для примера из листинга 4

Рассмотрим различные оптимизации на микроархитектурном уровне. *Слияние (объединение) микроопераций* – такая трансформация ГСА, при которой микрооперации в различных операторных блоках (рис. 11, а) объединяются в один операторный блок (рис. 11, б). Такая трансформация возможна, если микрооперации y_1 и y_2 могут выполняться параллельно и их одновременное выполнение приводит к тому же результату, что и последовательное. Допускается объединение произвольного количества микроопераций на линейном участке ГСА. При такой трансформации сокращается количество операторных вершин, что приводит к сокращению количества состояний управляющего автомата Мура и сокращению количества тактов в цикле. Эта трансформация не влияет на аппаратные затраты.

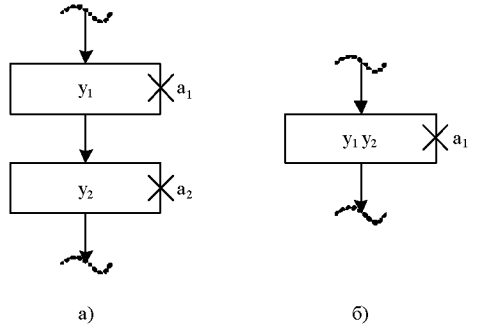

Рис. 11. Слияние микроопераций:
а – исходный подграф; б – трансформированный подграф

В том случае, если между двумя операторными вершинами есть входящая дуга (рис. 12, а), то слияние операций происходит по следующим правилам. Необходимо объединить операции y_1 и y_2 в состояние a_1 по правилам, указанным выше. Микрооперацию y_2 следует внести в цепь до схождения дуг (рис. 12, б). Такое преобразование не приводит к изменению количества состояний в автомате и не влияет на аппаратурные затраты. Выигрыш заключается в сокращении количества тактов в цикле.

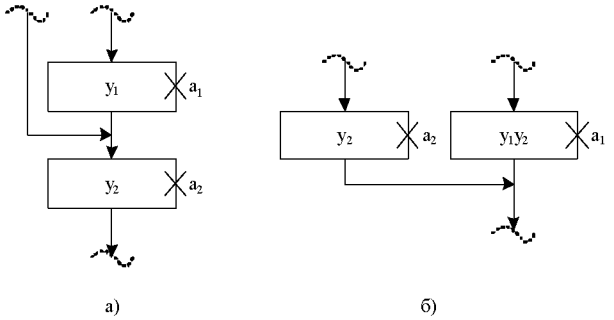

Рис. 12. Слияние микроопераций:
а – исходный подграф; б – трансформированный подграф

Рассмотрим более сложный случай объединения операторов. В исходном подграфе на рис. 13, а микрооперации y_1 и y_2 могут быть выполнены параллельно. В цепи между состояниями a_1 и a_2 присутствует условный оператор и разветвление. В том случае, если y_1 и y_2 будут объединены в состояние a_1, то в операторе альтернативной ветви необходимо поместить микрооперацию y_2' (читается «игрек два штрих»), которая имеет обратное действие микрооперации y_2.

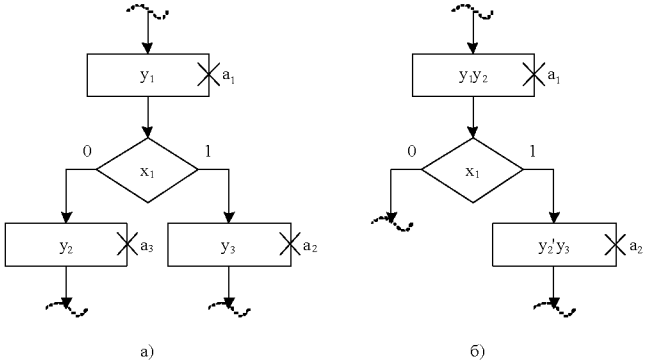

Рис. 13. Сложный случай объединения операторов:
а – исходный подграф; б – трансформированный подграф

Вычисление констант — такая трансформация операционного автомата, при которой вносится аппаратная избыточность, вычисляющая константную функцию от переменной (рис.14). При такой трансформации сокращается количество состояний автомата, а также длина цикла в тактах. При этом возрастает время распространения сигнала от регистра к выходам.

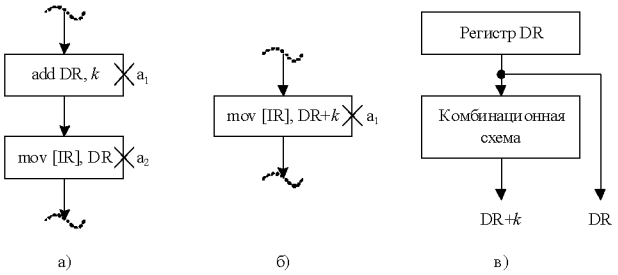

Рис. 14. Вычисление констант: а – исходный подграф; б – трансформированный подграф; в – структурная реализация

На рис. 15 показана оптимизированная ГСА, в которой объединены микрооперации и вычислено константное выражение. Количество состояний сокращено с 5 до 3. Самый короткий цикл a_2 составляет один такт, самый длинный a_2a_3 – два такта.

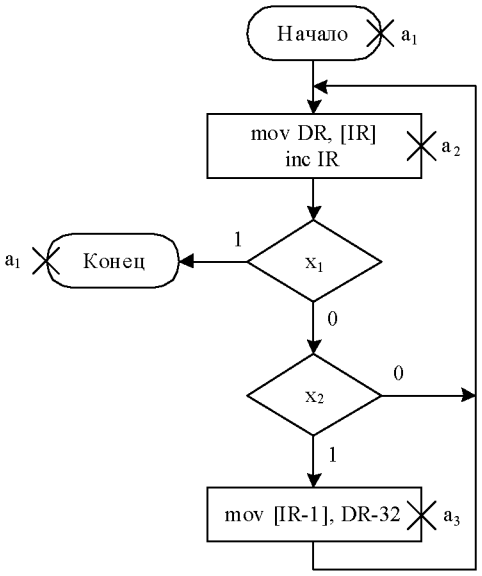

Рис. 15. Оптимизированная ГСА

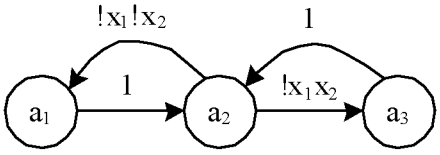

Рис. 16. Оптимизированный граф переходов

8. Преобразование несинтезируемых конструкций

Синтаксис языка C++ чрезвычайно многообразен. Существует большое множество библиотек и при-

ёмов программирования на C++, которые не могут быть преобразованы в схему на уровне регистровых передач на текущий момент. Например, популярными являются функции и классы ввода-вывода информации: scanf, printf, std::cin, std::cout, а также операции с файлами, которые не имеют прямого отображения в логическую схему. В настоящий момент программы синтеза обнаруживают такие конструкции на ранних стадиях компиляции исходных файлов, а затем либо игнорируют эти функции, либо сообщают о присутствии несинтезируемых конструкций и останавливают процесс синтеза.

В некоторых случаях целесообразно подготовить замену такой несинтезируемой конструкции, чтобы продолжить верификацию модели, пока не будет готова синтезируемая замена. На рис. 17 показан график проекта, в котором верификация не может быть начата до тех пор, пока не будут подготовлены синтезируемые замены несинтезируемым конструкциям.

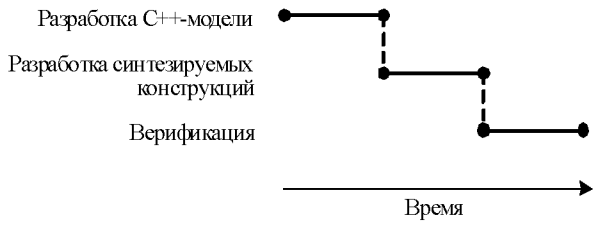

Рис. 17. График проектирования устройства

Существует решение этой проблемы, которое заключается в использовании виртуальных прототипов на языке SystemC. Виртуальный SystemC-прототип — это модель на языке SystemC, интерфейс которой строго соответствует проектируемому устройству, а архитектура выполнена на высоком уровне описания. Обычно SystemC-прототипы не синтезируются, а используются для верификации системы в целом на ранних этапах проектирования.

На рис. 18 показан график проекта, в котором несинтезируемые конструкции заменены SystemC-прототипами. Из-за быстрого перехода от C++ к SystemC модели верификация всей цифровой системы может быть начата раньше, чем будет закончена разработка синтезируемых замен.

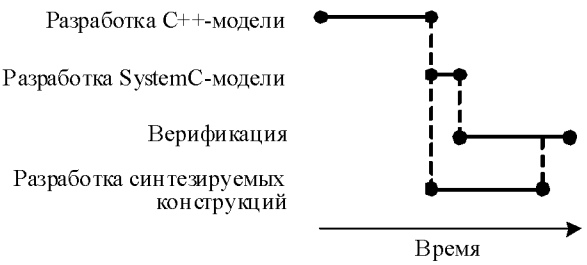

Рис. 18. График проектирования устройства с использованием SystemC-прототипов

Основные этапы преобразования следующие.

1. Определить фрагмент кода, который не может быть синтезирован.

2. Определить информационные зависимости этого фрагмента от смежного кода.

3. Выделить этот фрагмент кода в отдельную C++-функцию.

4. Разработать SystemC-прототип на основе этой C++-функции.

5. Выполнить синтез C++ модели в структурную VHDL модель.

6. Интегрировать SystemC модель в структурную VHDL модель.

В результате преобразования получается система, показанная на рис. 19. На этапе верификации будет использована SystemC-модель (рис. 19, а). После того, как будет готов структурный эквивалент, будет использована VHDL-модель компонента (рис. 19,б).

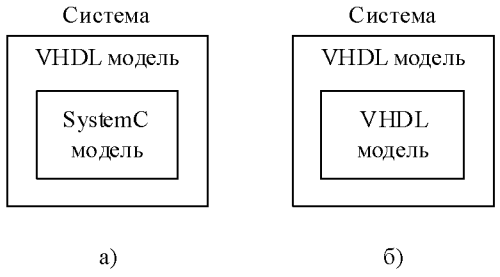

Рис. 19. Иерархия структурной модели: а – с использованием SystemC-модели компонента; б – с использованием VHDL-модели компонента

Рассмотрим пример, показанный в листинге 5. В этом примере создается объект статической памяти, объемом 1024 ячейки, которые инициализируются псевдослучайными значениями. Предположим, что функция *rand()* не имеет структурного эквивалента на этот момент.

Листинг 5. Пример программы с несинтезируемой конструкцией

```
37. int main()
38. {
39.     const int size = 1024;
40.     int mem[size];
41.     for (int i = 0; i < size; i++)
42.                 mem[i] = rand();
43.     return 0;
44. }
```

Введем несколько определений.

Определение 4. Транзакция — модельное событие высокого уровня, которое включает в себя множество событий низшего уровня. В транзакции соблюдается строгая последовательность и временные интервалы между событиями. Обычно транзакция — это целостная операция, например, чтения или записи

данных. Синтаксически транзакция выглядит как вызов функции или метода класса.

Определение 5. Транзакционная модель – такая модель системы, где все события между элементами происходят на уровне транзакций.

Определение 6. Транзактор – элемент системы, который преобразовывает транзакцию к множеству событий модели низшего уровня, например, регистрового или вентильного. Транзактор также выполняет обратное преобразование – с регистрового и вентильного уровней на уровень транзакций. Транзактор реализуется с помощью класса, имеющего два интерфейса: высокого уровня и уровня регистровых передач или даже вентильного.

Здесь и далее будет использоваться нотация для моделей системного уровня, как показано на рис. 20. Нотация взята из [8]. Интерфейс системного уровня соответствует понятию интерфейса языка C++ — совокупность абстрактных методов, которые реализует модуль.

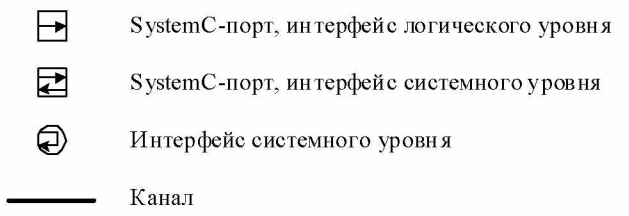

Рис. 20. Нотация моделей системного уровня

Представим эту программу в виде транзакционной модели [9] на языке SystemC, как показано на рис. 21. Необходимо сделать декомпозицию на два модуля: а) Main — основной, где происходят все вычисления; б) Rand — модуль, где будет производиться вычисление функции rand(). Модуль Main будет иметь один порт абстрактного типа rand_if (см. листинг 5).

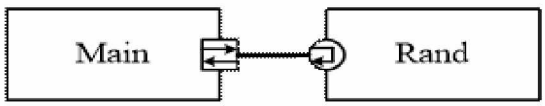

Рис. 21. Система из двух модулей

Создание абстрактного интерфейса необходимо для того, чтобы можно было создавать различные модули, которые наследуют один и тот же интерфейс.

Листинг 6. Абстрактный интерфейс для модуля rand

```
45. struct rand_if : public sc_interface {
46.     virtual int rand_f() = 0;
47. };
```

Модифицированный модуль Main показан в листинге 7. Здесь создан класс-обёртка на языке SystemC.

Листинг 7. Модуль Main

```
48. SC_MODULE(Main) {
49.     sc_port<rand_if> rand_m;
50.     SC_CTOR(Main) {
51.         SC_THREAD(process);
52.     }
53.     void process() {
54.         for (int i = 0; i < size; i++)
55.             // подстановка вместо функции rand()
56.             mem[i] = rand_m->rand_f();
57.         sc_stop();
58.     }
59. private:
60.     const int size = 1024;
61.     int mem[size];
62. };
```

Реализация модуля rand показана в листинге 8.

Листинг 8. Реализация модуля Rand.

```
63. SC_MODULE(Rand), public rand_if {
64.     virtual int rand_f() {
65.         return rand();
66.     }
67. };
```

Для успешной интеграции виртуального прототипа в систему на вентильном уровне необходимо детализировать интерфейсы системы. Для этого между модулями Main и Rand нужно поместить два транзактора: а) H2L (High Level To Low Level), который преобразует высокоуровневый интерфейс rand_if к протоколу на уровне логических сигналов; б) L2H (Low Level to High Level) — транзактор, обратный H2L. По своей сути Rand — это генератор случайных чисел, который генерирует очередное число по запросу. Для этого в систему введен модуль Clock — генератор синхроимпульсов. Таким образом, при детализации системы в ней появилось понятие времени (рис.22). Мы получили временную транзакционную модель из безвременной транзакционной модели.

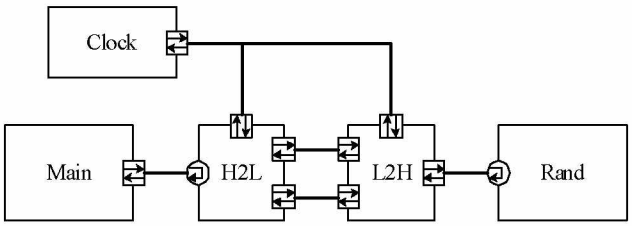

Рис. 22. Детализированная система

Иерархия классов на языке UML показана на рис. 23. Из диаграммы классов видно, что модули Rand и H2L имеют идентичный интерфейс rand_if. Это дает возможность свободно подключать любой из них к порту модуля Main.

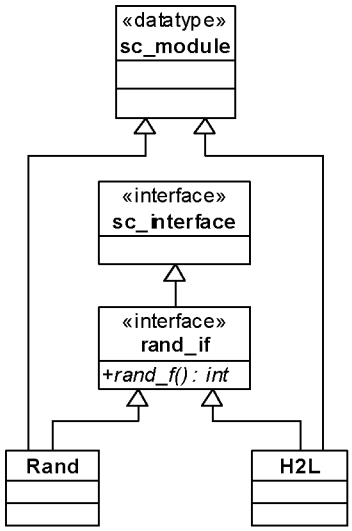

Рис. 23. Иерархия классов: множественное наследование от класса sc_module и интерфейса rand_if

Построим содержательную граф-схему алгоритма для основного процесса модуля Main (рис. 24).

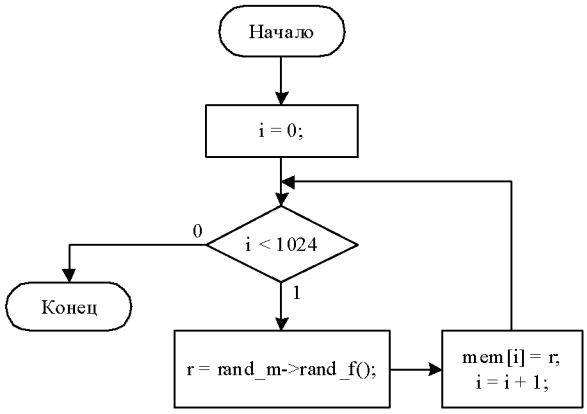

Рис. 24. СГСА для процесса process модуля Main

В листинге 9 показана реализация транзактора [10] L2H. Логика работы транзактора следующая: он реагирует на передний фронт синхронизации clk, и если сигнал разрешения en установлен в единицу, то выставить на выход output значение функции rand_f() модуля, подключенного к порту rand_m. Важно отметить то, что интерфейс транзактора (сигналы clk, enable, output) уже меняться не будут. Эти же сигналы будут использованы при установке реального синтезируемого модуля Rand, когда такой модуль будет доступен. То же самое касается и протокола работы этого транзактора.

Листинг 9. Реализация транзактора L2H

```
68. SC_MODULE(L2H) {
69.     sc_port<rand_if> rand_m;
70.     sc_in<bool> clk;
71.     sc_in<en> enable;
72.     sc_out<int>  output;
73.     SC_CTOR(L2H) {
74.                 SC_METHOD(process);
75.                 sensitive << clk.posedge();
76.     }
77.     void process() {
78.                 if (en)
79.                         output = rand_m->rand_f();
80.     }
81. };
```

На рис. 25 показана временная диаграмма работы транзактора L2H.

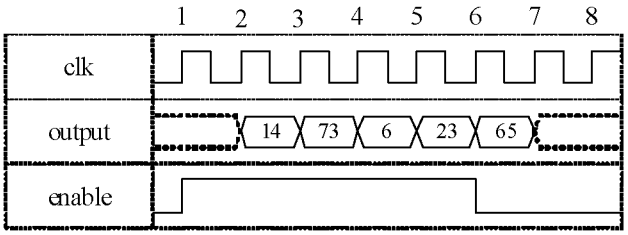

Рис. 25. Временная диаграмма работы транзактора H2L

Следующим этапом интеграции виртуального прототипа является аппаратный синтез модуля Main и его слияние с транзактором H2L. После этого шага система примет вид, как показано на рис. 26. Модуль Main-HW представляет собой аппаратную реализацию модуля Main (см. листинг 7).

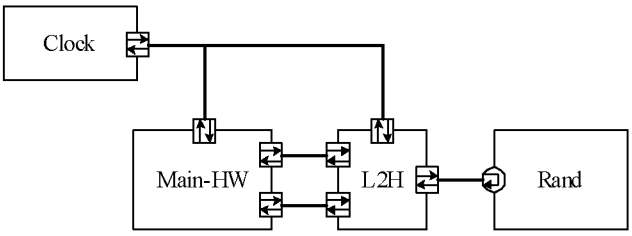

Рис. 26. Слияние транзактора H2L и модуля Main в модуль Main-HW

СГСА для этого модуля, с условием подключения к транзактору L2H, показана на рис. 27. Подразумевается, что сигналы en и output — это соответствующие сигналы транзактора L2H.

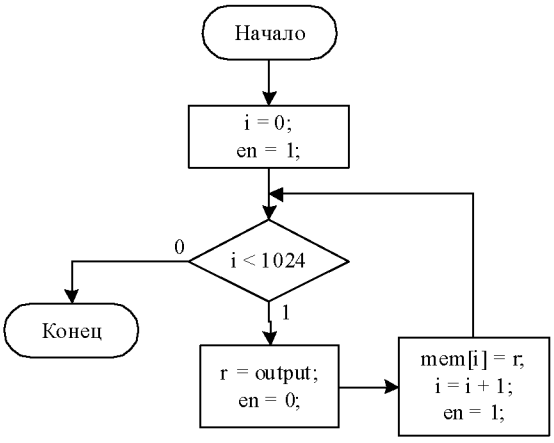

Рис. 27. Модифицированная СГСА для модуля Main-HW

9. Результаты практического синтеза мультиверсного метода проектирования

Было проведено сравнение производительности разработанной аппаратной модели и программной реализации на микропроцессоре экспериментальным путем. Оценивались следующие характеристики: временные затраты на проектирование, быстродействие, энергопотребление, площадь на кристалле.

Было выбрано три самые распространённые реализации: последовательностная, параллельная, конвейерная [11].

Последовательностной называется реализация, в которой все микрооперации упорядочены во времени. Преимущество применения такой реализации в том, что используется меньшая площадь на кристалле, меньшее энергопотребление, недостаток – низкая производительность.

Параллельной называется реализация, в которой параллельно выполняются две и более микрооперации за один такт работы. Особенности: высокое энергопотребление, высокая производительность и площадь на кристалле.

Конвейерной называется реализация, в которой за один такт работы выполняется две и более микрооперации на разном этапе выполнения. Особенности: высокая производительность, занимает больше места на кристалле и имеет высокое энергопотребление.

Для аддитивной (мультипликативной) оценки эффективности методов все значения были интегрально оценены. При интегральной оценке самое эффективное значение принималось за единицу, а остальные высчитывались по следующей формуле:

$$y_i = x_i / x_{max}, \quad (2)$$

где x_{max} – самое эффективное значение; x_i – значение, которое было получено в ходе эксперимента.

Эта оценка хорошо показала, в какой реализации какой метод более эффективен. В последовательностной реализации по всем критериям эффективным является автоматический метод. В параллельной реализации по площади на кристалле эффективнее работает ручной метод, а по всем остальным критериям выигрывает автоматический метод. В конвейерной реализации ручной метод выигрывает по энергопотреблению, во всем остальном эффективнее автоматический метод.

После этого была сделана аддитивная (мультипликативная) оценка эффективности каждой по отдельности реализации и общая аддитивная (мультипликативная) оценка эффективности метода.

Общий вид аддитивной модели следующий:

$$y_i = \sum_{i=1}^{n} x_i, \quad (3)$$

где y_i – коэффициент каждого инженера для аддитивной оценки, x_i – коэффициент по каждому критерию, n – количество критериев.

Общий вид мультипликативной модели имеет вид:

$$y_i = \prod_{i=1}^{n} x_i, \quad (4)$$

здесь y_i – коэффициент каждого инженера для мультипликативной оценки, x_i – коэффициент по каждому критерию, n – количество критериев.

На рис. 28-30 представлены результаты аддитивной оценки каждой реализации.

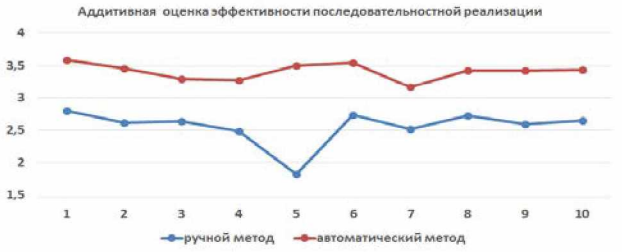

Рис 28. Аддитивная оценка эффективности последовательностной реализации

Рис 29. Аддитивная оценка эффективности параллельной реализации

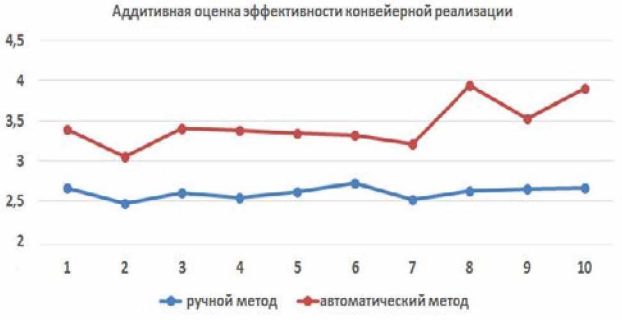

Рис 30. Аддитивная оценка эффективности конвейерной реализации

На рис. 31-33 представлены результаты мультипликативной оценки каждой реализации.

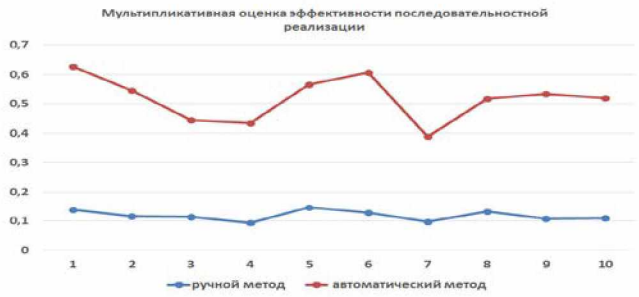

Рис. 31. Мультипликативная оценка эффективности последовательностной реализации

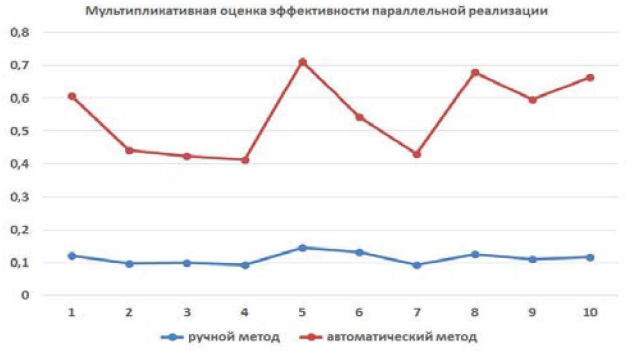

Рис. 32. Мультипликативная оценка эффективности параллельной реализации

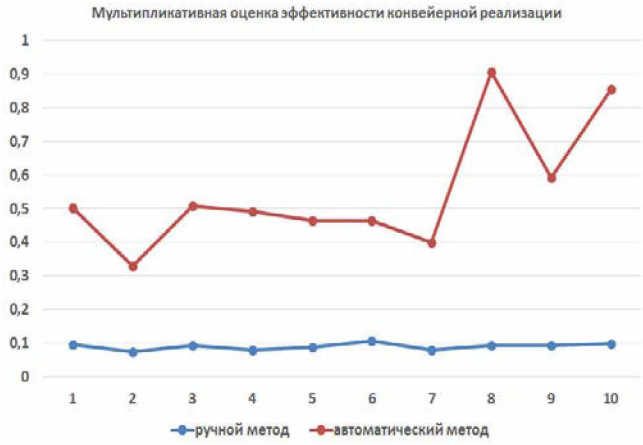

Рис. 33. Мультипликативная оценка эффективности конвейерной реализации

В результате исследования получили, что в случае аддитивной оценки автоматический метод лучше в 1,301292012 раз (рис. 34).

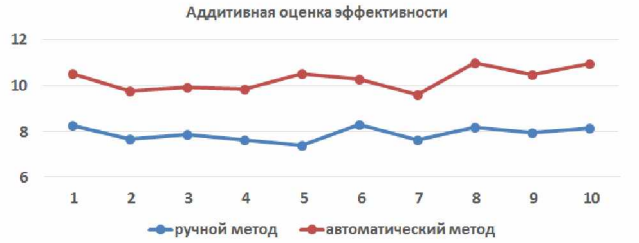

Рис. 34. Аддитивная оценка эффективности

Рис. 35. Мультипликативная оценка эффективности

При мультипликативной оценке автоматический метод лучше в 133,2182966 раз. На рис. 35 видно, что значения ручного метода практически стремятся к нулю, в то время как в автоматическом методе значения стремятся к 1. Это доказывает, что автоматический метод действительно эффективнее.

Выводы

Разработаны программно-аппаратные реализации моделей, методов и структур данных для проектирования цифровых систем на кристаллах, которые включают процедуры создания спецификации, синтеза, тестирования, моделирования и верификации на основе предложенной инфраструктуры, учитывающей промышленные средства компаний Aldec и Xilinx. Рассмотрены вопросы тестирования разработанных программных продуктов на реальных цифровых проектах создания IP-Core как примитивов для реализации цифровых систем на кристаллах.

При этом достигнута ***цель*** – разработаны и верифицированы инфраструктурные модули проектирования цифровых систем на кристаллах, которые характеризуются параллельным выполнением мультиверсного синтеза функциональности, обеспечивающей существенное уменьшение времени создания проекта в условиях ограничения на аппаратные затраты.

Решены следующие *задачи*:

1. Разработан и описан метод мультиверсного синтеза управляющих и операционных автоматов в заданной инфраструктуре проектирования, ориентированных на архитектурные решения в метрике, минимизирующий время выполнения функциональности за счет распараллеливания операций при ограничении на аппаратные затраты.

2. Выполнена программная реализация моделей и методов мультиверсной разработки операционных устройств в рамках интегрированной системы проектирования функциональных и архитектурных решений SoC на основе использования продуктов верификации и синтеза компаний Aldec и Xilinx.

3. Выполнено тестирование и верификация программных модулей инфрастурктуры проектирования цифровых систем на кристаллах, а также определена

эффективность предложенных моделей, методов и структур данных при создании реальных компонентов цифровых изделий.

Литература: 1. *Патент* 6 226 776 США, G 06 F 17/50. System for Converting Hardware Designs in High-Level Programming Language to Hardware Implementations / Yuri V. Panchul, Donald A. Soderman, Denis R. Coleman; заявитель и патентообладатель Synetry Corporation (США); заявл. 16.09.1997, опубл. 01.05.2001. http://www.google.com/patents/about?id=BM4GAAAAEBAJ&dq=6226776 **2.** *OR1200* OpenRISC Processor. http://opencores.org/ wiki1/index.php?title=OR1K_CPU_Cores&oldid=1404 **3.** *Язык* программирования C++. Международный стандарт ISO/IEC 14882. Второе издание, 15 октября 2003 г. American National Standards Institute, New York. **4.** *Jayaram Bhasker.* A SystemC Primer. 2002. **5.** *Лисяк В.В., Лисяк Н.К.* Обзор европейских производителей программного обеспечения САПР РЭА // Известия ЮФУ. Технические науки. 2008. №9 С.81-86. **6**. https://github.com/systemc/systemc-2.2.0. **7.** *Баранов С.И.* Синтез микропрограммных автоматов (граф-схемы и автоматы). Ленинград: Издательство 'Энергия'. Ленинградское отделение, 1979. **8.** *SystemC*: From the Ground Up, Second Edition. Black, D.C., Donovan, J., Bunton, B., Keist, A. Springer. 2010. P. 279. **9.** *Weiwei Chen, Rainer Dömer, Weiwei Chen, Rainer Dömer.* Transaction Level Models with Relaxed Timing. 2011. **10.** *Tareq Hasan Khan.* Automatic Generation of Transactors in SystemC. Concordia University. 2007. **11.** *Ed Tam.* A Microarchitectural Survey of Next Generation Microprocessors. 1997.

Translitereted bibliography:

1. *Patent* 6 226 776 SShA, G 06 F 17/50. System for Converting Hardware Designs in High-Level Programming Language to Hardware Implementations / Yuri V. Panchul, Donald A. Soderman, Denis R. Coleman; zayavitel i patentoobladatel Synetry Corporation (SShA); zayavl. 16.09.1997, opubl. 01.05.2001. http://www.google.com/patents/about?id=BM4GAAAAEBAJ&dq=6226776 **2.** *OR1200* OpenRISC Processor. http://opencores.org/ wiki1/index.php?title=OR1K_CPU_Cores&oldid=1404

3. *Yazyik* programmirovaniya S . Mezhdunarodnyiy standart ISO/IEC 14882. Vtoroe izdanie, 15 oktyabrya 2003 g. American National Standards Institute, New York.

4. *Jayaram* Bhasker. A SystemC Primer. 2002.

5. *Lisyak V.V., Lisyak N.K.* Obzor evropeyskih proizvoditeley programmnogo obespecheniya SAPR REA // Izvestiya YuFU. Tehnicheskie nauki. 2008. #9 S.81-86.

6. https://github.com/systemc/systemc-2.2.0.

7. *Baranov S.I.* Sintez mikroprogrammnyih avtomatov (graf-shemyi i avtomatyi). Leningrad: Izdatelstvo 'Energiya'. Leningradskoe otdelenie, 1979.

8. *SystemC*: From the Ground Up, Second Edition. Black, D.C., Donovan, J., Bunton, B., Keist, A. Springer. 2010. P. 279.

9. *Weiwei Chen, Rainer Dtsmer, Weiwei Chen, Rainer Dtsmer.* Transaction Level Models with Relaxed Timing. 2011.

10. T*areq Hasan Khan.* Automatic Generation of Transactors in SystemC. Concordia University. 2007.

11. *Ed Tam.* A Microarchitectural Survey of Next Generation Microprocessors. 1997.

Поступила в редколлегию 11.02.2016

Рецензент: д-р техн. наук, проф. Кривуля Г.Ф.

Обризан Владимир Игоревич (Obrizan Vladimir), старший преподаватель кафедры АПВТ ХНУРЭ. Научные интересы: компьютеная инженерия. Адрес: Украина, 61166, Харьков, пр. Науки, 14.