СЕРИЯ СИСТЕМЫ ПРОЕКТИРОВАНИЯ

А.К. Поляков

ЯЗЫКИ VHDL и VERILOG В ПРОЕКТИРОВАНИИ ЦИФРОВОЙ АППАРАТУРЫ

Базовые понятия HDL

Описание HDL

VHDL-реализация автомата RC4

Справочные данные

ISBN 5-98003-016-6

9 785980 030162

«СОЛОН»

Серия «Системы проектирования»

А. К. Поляков

Языки VHDL и VERILOG в проектировании цифровой аппаратуры

Москва
СОЛОН-Пресс
2003

УДК 681.3
ББК 32.973.26-018.2
П 54

Поляков А. К.
П 54 Языки VHDL и VERILOG в проектировании цифровой аппаратуры. — М.: СОЛОН-Пресс, 2003. — 320 с.: ил. — (Серия «Системы проектирования»).

ISBN 5-98003-016-6

Книга посвящена проектированию цифровых систем с помощью высокоуровневых языков описания аппаратуры (Hardware Description Language — HDL) — Verilog и VHDL. Эти языки являются международным стандартом и используются как системами анализа (моделирование), так и системами синтеза цифровой аппаратуры. С единых позиций изложены основные концепции этих языков. Даны рекомендации по стилю кодирования, синтезабельности и верификации HDL-описаний проектируемых систем.

Приведены примеры синтезабельных описаний узлов и устройств и организации функциональных тестов.

В приложение вынесены справочные данные по языкам VHDL и VERILOG.

Автор предполагает, что читатель знаком с основами программирования и основами проектирования цифровых устройств.

Эту книгу можно заказать по почте (наложенным платежом — стоимость 212 руб.) двумя способами:

1) выслать почтовую открытку или письмо по адресу: 123242, Москва, а/я 20;
2) передать заказ по электронной почте (e-mail) по адресу:
magazin@solon-r.ru.

Необходимо написать полный адрес, по которому выслать книги. Обязательно указывать индекс и Ф. И. О. получателя!

При наличии — указать телефон, по которому с вами можно связаться, и адрес электронной почты (E-mail).

Цены действительны до 15 апреля 2003 г.

Вы можете в любое время получить свежий каталог издательства «СОЛОН-Пресс» по **Интернету**, послав пустое письмо на робот-автоответчик по адресу katalog@solon-r.ru, а также подписаться на **рассылку новостей** о новых книгах издательства, послав письмо по адресу news@solon-r.ru с текстом «SUBSCRIBE» (без кавычек) в теле письма.

ISBN 5-98003-016-6

© Макет и обложка «СОЛОН-Пресс», 2003
© А. К. Поляков, 2003

Предисловие
Языки VHDL и VERILOG

С начала 70-х годов стала актуальной проблема создания стандартного средства документации схем и алгоритмов дискретных систем переработки информации и цифровой аппаратуры (ЦА), одинаково пригодного как для восприятия человеком, так и для обработки в ЭВМ.

Этим средством явились языки VHDL и VERILOG, обобщенно называемые HDL (Hardware Description Language — язык описания аппаратуры).

К основным достоинствам языков VHDL и VERILOG следует отнести следующие:

1. *Стандартность.* Лучше иметь плохой стандарт, чем не иметь никакого. Это подтверждает библейский опыт создания вавилонской башни, разноязычие строителей которой, по преданиям, привело к печальному результату.

VHDL и VERILOG официально признаны стандартом описания цифровой аппаратуры, который поддерживается военно-промышленным комплексом и радиоэлектронной промышленностью западных стран. Этот стандарт облегчает обмен документацией между отдельными группами разработчиков и эксплуатационщиков аппаратуры, различными системами автоматизации проектирования (САПР).

2. *Многоаспектность и многоуровневость.* Универсальное средство заменяет несколько специализированных. Языки VHDL и VERILOG пригодны для описания как схем аппаратуры, так и функциональных тестов и алгоритмов функционирования. Они покрывают широкий диапазон уровней структурной детализации описаний ЦА: от описаний архитектуры ЭВМ на уровне устройств типа процессор-память до описаний узлов типа триггер на уровне вентилей и МОП-ключей, от описаний алгоритмов ЭВМ на уровне команд до описаний алгоритмов устройств на уровне межрегистровых передач и булевских функций, от описаний функциональных тестов до тестов проверки схем. На высших уровнях абстракции VHDL- и VERILOG-описания можно рассматривать как средство спецификации требований к проекту.

3. *Человеко-машинность.* Документация пишется один раз, а читается многократно. Создатели VHDL и VERILOG нашли довольно удачный компромисс между требованиями к языку как к средству документирования, удобному для восприятия человеком, и как формальному средству, удобному для ввода и обработки описаний систем в ЭВМ. VHDL- и VERILOG-описания аппаратуры пригодны для обработки такими компонентами САПР, как подсистемы моделирования, подсистемы формальной верификации, reuse-checker'ы (программы проверки стиля HDL-описаний на синтезабельность и пригодность к повторному использованию в различных проектах), подсистемы логического синтеза, подсистемы синтеза с учетом тестопригодности, системы синтеза и анализа контрольных тестов, временные анализаторы, кремниевые компиляторы, подсистемы автоматизации конструкторского проектирования и т. п.

Известно большое число отечественных и зарубежных предшественников современных HDL.

Российские — «МОДИС», «Автокод-М», «МОДИС-В78», MPL, ОСС-2, «Форос», «Алгоритм», «Пульс», «Симпатия» и др.;

Зарубежные — CDL, DDL, ISPS, CONLAN, HILO и др.

Языки VHDL и VERILOG, помимо отмеченных выше свойств, отличаются от них более мощной общеалгоритмической базой, гибкостью, развитыми средствами отображений временных соотношений и используемых алфавитов моделирования, большим диапазоном охватываемых систем и уровней их описаний, а главное — стандартностью, т. е. признаваемостью мировым сообществом и мощной системной поддержкой.

Данное краткое руководство знакомит читателей с общими концепциями языков описания дискретных систем VHDL и VERILOG и их подмножествами, реализованными в САПР логического синтеза на примерах описаний простых узлов и устройств ЭВМ и цифровой аппаратуры. В приложения вынесены справочные сведения по синтаксису и семантике основных конструкций языков, а также пакетам прикладных программ, расширяющих их возможности.

Предполагается, что читатель знаком с основами проектирования цифровой аппаратуры и моделирования и одним из современных языков программирования (ПАСКАЛЬ — желательно для VHDL или Си — желательно для VERILOG).

Пособие явилось следствием переработки и модернизации предыдущей работы автора [8] — «Моделирование ЭВМ на языке VHDL» (МЭИ, 1994 г.)

Косвенным подтверждением полезности предыдущей работы автора может служить воспроизведение главы 2 пособия 1994 г. в главе «VHDL» книги Стешенко В. Б. «ПЛИС фирмы ALTERA: проектирование устройств обработки сигналов» (М.: ДОДЕКА, 2000), естественно, без каких- либо ссылок на первоисточник.

Помимо материала по языку VERILOG, в данное, существенно переработанное и расширенное издание добавлены материалы по верификации HDL-описаний и стилю разработки синтезабельных и повторно пригодных описаний проектов.

Книга не претендует на полное описание языков и методик их применения — для этого не хватило бы и 2000 страниц (одно руководство по VERILOG содержит более 400 страниц). Даны только основы этих HDL, поясненные на простых примерах, и только в последней главе представлены реальные проекты.

Автор стремился выделить общность элементов и концепций языков и проиллюстрировать это следующим образом: в левой части страницы — VHDL-текст, в правой — VERILOG-текст примеров. Поэтому читатели, желающие опустить один из языков, могут сделать это относительно безболезненно.

В главе 1 HDL представлен глазами инженера-схемотехника (как средство кодирования схем) и инженера-программиста (как средство записи алгоритмов).

Глава 2 знакомит со специфическими средствами HDL — параллельные процессы, задержки, многозначный алфавит и т. п.

В главе 3 разобраны примеры, иллюстрирующие применение средств HDL в процессе проектирования простейшей комбинационной схемы и схем с памятью.

Глава 4 дает материал по функциональной верификации, структурам тестирующих программ и их отладке.

В главе 5 приведены рекомендации по созданию синтезабельных HDL- описаний устройств, примеры моделей комбинационных схем и автоматов.

Главы 6 и 7 содержат примеры разработки двух типов проектов — схемной реализации шифроалгоритма RC4 на базе FPGA серии VIRTEX фирмы Xilinx и функциональной модели микросхемы двухпортовой памяти.

В приложение вынесены краткие справочники по VHDL и VERILOG.

В свое время на кафедре вычислительной техники МЭИ работали выдающиеся представители отечественной науки и техники в области автоматизации проек-

тирования, моделирования и программирования: профессора Н. Я. Матюхин, А. И. Китов, И. М. Тетельбаум, А. Г. Шигин, оказавшие поддержку автору в его первых шагах на этом поприще.

Автор благодарен коллегам — профессорам МЭИ Потемкину И. С., Дерюгину А. А., Лодыгину И. И, Кутепову В. П. за повседневную помощь и сотрудничество.

Реализация одних из первых отечественных систем моделирования — Автокод-М (1968 г.) и МПЛ (1975) — была бы невозможна без аспирантов и студентов МЭИ Д. Д. Горбатенко, М. М. Ляшко, М. Кава, Б. Кацарова, З. Ковача, К. Пруналя и других, которым автор выражает глубокую признательность.

В работе над данным пособием большую помощь автору оказали сотрудники и студенты МЭИ: С. Ю. Наконечный, Шанкер Ума, А. Ю. Дронова, сотрудник «Дизайн-центра» при МЭСИ А. В. Пеженков, профессор Вирджинского политехнического института Дж. Армстронг, чья книга [3] была одной из первых публикаций по VHDL в России, а также сотрудники компании Seva (затем Intrinsix), США, в которой автору удалось поработать в 1998—2001 гг.: профессор Ю. А. Татарников, доцент А. А. Сохацкий, инженер В. В. Ковалев, руководитель северо-западного подразделения фирмы Yatin Trivedi — автор книги [17] по VERILOG, технический менеджер James Lee — автор учебника [16] по VERILOG, и Larry F. Saunders — один из создателей стандарта VHDL.

Автор особенно благодарен П. Н. Бибило и Ю. В. Соколовскому за ценные замечания и советы по улучшению книги.

Неоценимый труд проделан моей дочерью Машей, помогавшей в наборе рукописи на компьютере.

Введение
HDL — исторический экскурс и перспективы

В настоящее время известны два стандартных языка описания аппаратуры (HDL) — VHDL и VERILOG-HDL, в дальнейшем для краткости VERILOG.

Язык VHDL (Very high Speed integrated circuit Hardware Description Language — язык описания сверхскоростных БИС) был разработан международной группой по заданию Министерства обороны США в начале 80-х годов. Стандарт IEEE 1076 на версию этого языка был утвержден в 1987 г. [6]. В приложении 1 дано краткое изложение его синтаксиса. Работу над усовершенствованием стандарта ведет группа VASG (VHDL Analysis and Standartisation Group). Срок регулярного пересмотра стандарта — пять лет, и очередной вариант языка появился в 1993 г. [1]. Ведутся также работы по расширению VHDL в области описания аналоговой аппаратуры — VHDL-A, стандартизации внутренней формы представления VHDL-описаний в ЭВМ (группа VI-FASG), формы задания тестов для VHDL-моделей (группа WAVES), задания параметров задержек компонент (VITAL), алфавита представления значений сигналов в моделях и операции в этом алфавите (стандарт IEEE std_logic_1164) и т. д.

Язык VERILOG был разработан в 1985 г. одной из корпораций США и позднее утвержден институтом электроинженеров США (IEEE) — стандарт IEEE 1364 в 1994 г. [2]. В приложении 2 дано краткое описание его подмножества. В отличие от VHDL, который строго типизирован и синтаксически напоминает языки ADA и PASCAL, VERILOG базируется на С, имеет меньше встроенных возможностей саморасширения, но зато более прост в реализации (более простой и быстрый компилятор), имеет более развитый интерфейс с языком Си и лаконичен, что позволяет уменьшить объем описаний схем примерно в полтора раза по сравнению с VHDL. В настоящее время принят вариант нового стандарта IEEE 1364—2001 (VERILOG-2001), который, в частности, включил в себя ряд стилистических средств, сближающих его с VHDL [15] (см. приложение 2). Тем не менее остается впечатление елки, на которую все время добавляют новые украшения, не имея возможности убрать ненужные старые. Ниже VERILOG-2000 означает то же, что и VERILOG-2001.

В целом по поводу этих языков можно сказать, что у них скорее больше общего, чем различий, как, например, у брюнетки и блондинки — обе дамы приятной наружности, только одна из них (VHDL) посолиднее и пофигуристей, а другая (VERILOG) в этом смысле не очень, но полегче, быстрее отслеживает моду и имеет лучшую связь с заграницей (в нашем случае с языком С).

Примеры возможного использования HDL

Проектировщик БИС может составить функциональное HDL-описание проектируемого кристалла и, используя систему моделирования САПР, проверить его соответствие спецификации (функциональная верификация). После этого с помощью системы логического синтеза автоматически синтезировать схему (получить ее структурное HDL-описание) в заданном элементном базисе. Затем моделированием полученной логической схемы оценить корректность результатов синтеза. После чего с помощью системы автоматизированного конструкторского проектирования провести трассировку соединений, а моделированием проверить правиль-

ность работы схемы с учетом задержек и наводок в проводах. Возможен автоматический синтез схем с учетом контролепригодности, синтез контролирующих тестов, а также анализ тестов на полноту и корректность.

HDL используется не только для представления проектируемых схем, но и для описания тестирующих программ (testbench) и тестов.

Имея в своем распоряжении выполненные с учетом требований многократного использования HDL-описания ранее спроектированных устройств, с помощью САПР несложно включать эти описания в состав новых проектов, повторно реализовать их на более современной технологии и т. д.

Эксплуатационщик радиоэлектронной аппаратуры, найдя в комплекте документации HDL-описание устройства и тестирующей программы, на их базе может осуществлять модернизацию схем, может использовать HDL-модели при поиске неисправностей в схеме и доработке контрольных тестов.

С помощью HDL-приложений к учебникам и автоматизированным обучающим курсам более эффективно решаются задачи тренинга в сфере проектирования и эксплуатации радиоэлектронной аппаратуры.

Стандартизация входных языков и внутренних интерфейсов подсистем САПР, в том числе и на базе HDL, создает общую коммуникационную среду для САПР, позволяет упростить стыковку продуктов различных фирм, обмен библиотеками моделей компонент и проектов, модернизацию отдельных подсистем САПР.

Являются ли VHDL и VERILOG идеальными языками?

Как и все универсальные средства, по отдельным аспектам уступающие специализированным, VHDL и VERILOG представляются весьма громоздкими и избыточными, что затрудняет их реализацию и изучение в полном объеме. Естественно выделение подмножеств языка для отдельных компонент САПР и групп пользователей. Например, известны подмножества HDL для синтеза, для построения тестопригодных схем, верификабельные подмножества и т. д.

Также как развитие языков программирования привело к созданию более выразительных и эффективных языков программирования (от ФОРТРАН к ПЛ/1 и далее к языку АДА, С++), так и VHDL-версии IEEE 1076—1987, 93, и VERILOG-версии 1995, 2000 удалось дополнить средствами описания аналоговой аппаратуры, интерфейсом с другими языками.

Возможно, в ближайшем будущем удастся ввести в HDL развитые средства объектно-ориентированного программирования. По крайней мере, это наблюдается в группе языков, развивающих возможности HDL в области функционального тестирования (VERA фирмы Synopsys, SPECMAN фирмы Vericity, TEST BUILDER — фирмы Cadence и др.).

Основы метода имитационного моделирования

Проектирование можно представить как чередующийся процесс порождения новых вариантов системы — синтез, и оценки, проверки этих вариантов — анализ.

Одним из эффективных методов анализа сложных систем является метод имитационного моделирования. Этот метод предполагает построение математической модели системы в виде алгоритма, отображающего существенные ее свойства и реализацию модельного эксперимента на ЭВМ. Большая емкость памяти и быстродействие современных ЭВМ позволяют имитировать поведение систем, состоящих из десятков и сотен тысяч элементов, воспроизводить процессы, включающие миллионы элементарных актов.

В процессе проектирования аппаратуры и математического обеспечения (МО) цифровых систем (ЦС) этим методом решается большое число задач.

1. На этапе структурного проектирования: анализ и оценка вариантов структур ЦС и МО, рассматриваемых как система массового обслуживания с точки зрения производительности, быстродействия, времени обслуживания заявок (задач потребителей), максимальной длины очередей на обслуживание, выявления узких мест системы и т. п. (языки GPSS, GASP, SIMULA).

2. На этапе алгоритмического проектирования на имитационных моделях проверяются алгоритмы отдельных устройств. Для ЦВМ с микропрограммным управлением отлаживаются микропрограммы. На интерпретационных комплексах отлаживаются элементы МО проектируемой ЭВМ-программы технического обслуживания (тесты), компоненты операционной системы и т. п. (наряду с языками программирования C, C++, PASCAL, здесь часто используются VHDL и VERILOG).

3. На этапе функционально-логического проектирования моделирование используется при проверке функциональных описаний устройств (так называемых описаний на уровне регистровых передач (register transfer level -RTL) и отработке функциональных тестов. Кроме того, оно применяется при верификации автоматически синтезируемых логических схем (HDL-языки — VHDL, VERILOG, их подмножества и их модификации типа AHDL).

4. На этапе конструкторского проектирования имитационное моделирование помогает проверять схемы с учетом задержек в проводах, анализировать и синтезировать тесты контроля и т. п. (HDL-языки — VHDL,VERILOG, стандартные форматы EDIF, SDF и др.).

5. На стадии проектирования элементной базы этот метод используется при расчете электрических схем элементов, анализе переходных процессов и исследовании влияния разброса параметров электронных компонент (pSpice и др.).

В процессе исследования системы методом имитационного моделирования можно выделить ряд этапов:

1. Постановка задачи — определение границ исследуемой системы, целей исследования и путей их достижения.

2. Планирование исследования, подготовка исходных данных.

3. Формализация моделируемого процесса и построение его математической модели.

4. Построение моделирующего алгоритма.

5. Создание теста, отладка и верификация модели.

6. Реализация экспериментов на модели.

7. Оптимизация модели и при необходимости постановка новых экспериментов на ней.

8. Анализ конечных результатов и их документирование.

Проведение машинного эксперимента с моделями сложных систем существенно упрощается при использовании проблемно-ориентированных языков моделирования и соответствующих пакетов прикладных программ (ППП). Для этих языков характерно присутствие средств, автоматизирующих выполнение отдельных этапов модельного эксперимента. Например, наличие в языке моделирования встроенных объектов-аналогов соответствующих элементов моделируемых структур и процессов позволяет упростить и совместить этапы 3 и 4, наличие в ППП развитых средств построения тестов и отладки упрощает этап 5, наличие в ППП подсистем планирования и оптимизации помогает автоматизировать этапы 6 и 7 модельного эксперимента.

Рассматриваемые в книге языки VHDL и VERILOG являются классическим примером таких языков моделирования.

Модельное время и имитация параллельных процессов

Особенности инструмента, с помощью которого реализуется модельный эксперимент (в нашем случае это ЭВМ), определенным образом сказываются на структуре описания модели и характере погрешностей.

В структурном аспекте моделируемая система (объект проекта — object, entity) представляется как композиция из элементов-компонент (component) и связей между ними (net). Компонентом принято считать объект, внутренняя структура которого не представляет интерес на принятом уровне рассмотрения.

В поведенческом аспекте моделируемая система представляется множеством процессов (process), взаимодействующих путем обмена сигналами (signal). Изменение сигнала — событие (event) в одном процессе может запускать другие процессы, которые, в свою очередь, порождают следующие события.

В реальной системе элементы могут работать одновременно, процессы протекать параллельно. При использовании обычных однопроцессорных ЭВМ для имитации таких систем приходится последовательно воспроизводить функционирование их элементов. Это приводит к необходимости введения *дискретизации модельного времени* и применения специальных мер для уменьшения погрешностей, обусловленных последовательной организацией моделирующего алгоритма. *Масштаб модельного времени* определяется *интервалом* ΔT реального времени, принятого в модели за единицу. В языке VERILOG пользователь сам задает *единицу модельного времени* и точность ее представления директивой `timescale, например, `timescale 1ns/100 ps, что означает — одна наносекунда с точностью 100 пикосекунд. В языке VHDL это делается неявно.

Следует отличать tm — *модельное время в условных единицах* (число интервалов ΔT) от tmr — *модельного времени в физических единицах времени* ($tmr = tm \cdot \Delta T$) и tc — *машинного времени* (продолжительности выполнения моделирующей программы на компьютере), затраченного на воспроизведение tm или tr единиц модельного времени.

Коэффициент замедления моделирования определяет отношение машинного времени к воспроизводимому (модельному) времени. Например, имитация 100 микросекунд работы логической схемы может потребовать трех секунд машинного времени (коэффициент замедления 30 000). Если интервал модельного времени равен 0,5 микросекунды, то 100 микросекунд реального времени соответствуют 200 единицам модельного времени.

Ошибка, связанная с дискретизацией модельного времени, может быть сделана сколь угодно малой за счет выбора более мелкого шага времени ΔT. Однако во многих случаях это связано с увеличением коэффициента замедления и дополнительными затратами ресурсов памяти и времени инструментальной ЭВМ. Поэтому при выборе масштаба модельного времени приходится идти на компромисс. Для действий, имеющих продолжительность меньшую, чем единица модельного времени, мы как бы допускаем в модели возможность существования причинно-следственных отношений вне времени. Это приводит к необходимости включения в HDL специальных понятий и средств, таких, как неявная бесконечно малая дельта-задержка в операторах присваивания и дельта-цикл итерации в работе моделирующей программы.

Специфика задач, возникающих в связи с дискретизацией модельного времени и имитацией параллельно протекающих процессов последовательно выполняемой

на ЭВМ программой, может быть пояснена следующим примером. Пусть модель состоит из упорядоченного подмножества связанных между собой компонентов-моделей элементов Эк (к = 1 : n), не обладающих задержками. Состояние выхода некоторого К-го элемента есть функция состояний его соседей, часть из которых имеет больший, часть — меньший индекс К. Если вычисление значений новых состояний элементов, соответствующих новому Т + delta-моменту модельного времени осуществляется в порядке возрастания индексов элементов, то при вычислении нового состояния Эi предшествующие (1: (i − 1)) элементы уже приняли новое состояние, в то время как оставшиеся ((i + 1) : n) еще находятся в старом. Поэтому для устранения возможной ошибки часто используются итерации (дельта-цикл). **Только после того как вычислится новое состояние всех элементов, система моделирования произведет смену старых состояний на новые, увеличит модельное время на бесконечно малое дельта и снова повторит дельта-цикл.** И так до тех пор, пока состояния элементов перестанут меняться или не будет превышен предел допустимого количества дельта-циклов (итераций). Пример системы с несходящимися итерациями — наличие среди ее элементов генератора типа Y = not Y.

Моделирование по интервалам времени и по событиям

Как уже отмечалось, для имитационного моделирования характерно воспроизведение моделируемого процесса во времени. Существуют два способа воспроизведения модельного времени: моделирование по интервалам времени и по событиям.

Для метода моделирования *по интервалам времени* (time driven simulation, cycle based simulation) характерно, что на каждом интервале модельного времени заново вычисляются новые состояния всех элементов модели, даже в том случае, если они не должны изменяться, т. е. в них не должно происходить событий.

Для моделирования *по событиям* (event driven simulation) характерно рассмотрение системы только в те моменты времени, когда в ней возможны события, и только тех ее элементов, в которых эти события могут произойти.

Наглядным аналогом способа моделирования по интервалам может служить процесс воспроизведения электронным лучом изображения в телевизоре — на каждом интервале времени (0,04 с) изображение (картинка) воспроизводится заново. Интервал времени ΔТ в телевидении выбран таким, чтобы глаз не замечал смены «картинок», а развертка реализована чересстрочно (сначала все нечетные, а затем четные строки) так, чтобы глаз не замечал искажений из-за того, что изображение картинки воспроизводится последовательно, сверху вниз, строка за строкой.

Аналогом процедуры моделирования по событиям могут служить телевизионные системы, в которых осуществляются передачи только изменяемой части изображения, только тех элементов «картинки», в которых происходят события (изменения). Если изображение меняется медленно, то некоторое усложнение аппаратуры, связанное с событийной передачей изображения, окупается существенным уменьшением полосы частот видеосигнала. Аналогичная ситуация имеет место и при выборе способа моделирования — по интервалам или по событиям.

В большинстве HDL-систем моделирования общего пользования реализован событийный метод, однако во многих специализированных системах моделирования схем на вентильном уровне, когда число событий в схеме на одном такте работы схемы велико, часто используется интервальный метод.

Рассмотрим простой пример схемы из трех последовательно включенных вентилей-повторителей P1, P2, P3, по которым проходит входной сигнал A, достигая выхода D.

Вариант, когда все элементы обладают задержкой, отличной от нуля

Пусть каждый из элементов имеет задержку 10 ns, а внешний сигнал A меняется каждые 100 ns. Описание может быть ранжировано — случай, когда порядок уравнений соответствует порядку прохождения сигнала по схеме (левый столбец уравнений) или нет (правый столбец).

P1:B=A del 10;	P3: D=C del 10;
P2: C=B del 10;	P2: C=B del 10;
P3: D=C del 10;	P1:B=A del 10;

Пусть требуется воспроизвести tmr = 1000 наносекунд времени поведения схемы.

Моделирование по интервалам времени

Если все элементы имеют задержки, отличные от нуля, как в нашем примере, ранжирование не влияет на процесс моделирования. Интервал модельного времени ΔT можно взять равным 10 ns (в общем случае это наибольший общий делитель задержек элементов). Тогда предел модельного времени будет tm = 100 интервалов и, так как воспроизведение одного интервала требует решения n = 3 уравнений, всего потребуется решить n * tm = 300 уравнений (tm — модельное время в условных единицах).

Моделирование по событиям

При каждом изменении сигнала A (всего будет 10 таких событий (внешних), т. к. сигнал A меняется каждые 100 ns) начинаются события внутри схемы — их 3 в цепочке событий. Каждое событие связано с решением одного уравнения — того, с которым связан изменившийся выходной сигнал ранее решенного. Всего за 100 единиц модельного времени будет 30 событий, и придется решить 30 уравнений (в 10 раз меньше, чем при интервальном методе). Если учесть дополнительные затраты машинного времени и памяти на прослеживание цепочки событий (включение — исключение события в календарь, поиск события с минимальным временем в календаре событий и т. п.), то экономия машинного времени, получаемая при событийном моделировании, будет не в 10 раз, но тем не менее существенная (в схемах с коэффициентом разветвления по выходам, например, равном 2, каждое внутреннее событие порождает необходимость решения 2 уравнений и число решаемых уравнений в 2 раза больше числа событий в схеме).

Вариант, когда все элементы не обладают задержкой

Рассмотрим тот же пример для случая моделирования без задержек:

P1: B=A;	P3: D=C;
P2: C=B;	P2: C=B;
P3: D=C;	P1: B=A;

Интервал модельного времени ΔT можно взять равным 100 ns т. е. в 10 раз большим, и модельное время уменьшится на порядок — tm = 10.

Моделирование схемы требует n * tm * (r +1) = 120 решений уравнений (где r — ранг схемы — наибольший путь от входов к выходам схемы — определяет максимальное число итераций уравнений на каждом интервале модельного времени). И только если мы заранее знаем, что в схеме нет обратных связей без задержек и можно отказаться от дельта-задержек в операторах присваивания (пример схемы, ранжирование которой невозможно — RS-триггер на вентилях И-НЕ.

$g1 = \neg(R \wedge g2)$; $g2 = \neg(S \wedge g1)$;). Моделирование по интервалам ранжированной схемы без задержек потребует $n * tm * 2 = 3 * 10 = 30$ решений уравнений. В обоих вариантах это меньше, чем в случае моделирования схемы с задержками за счет уменьшения числа интервалов модельного времени на порядок.

При событийном методе оценка (по сравнению со случаем моделирования схемы с задержками) не изменится, так как ни ранжирование, ни отсутствие задержек в элементах не повлияют на число событий в схеме, равное 30. Для чего нужна дельта-задержка? Она нужна для того, чтобы отследить порядок срабатывания элементов при прохождении сигнала, мы как бы вводим бесконечно малую задержку на каждом элементе. Сначала срабатывает элемент P1, через бесконечно малое время дельта изменяется сигнал B, потом срабатывает элемент P2, через время дельта изменяется сигнал C и т. д.

Как это выглядит на HDL

Следующие фрагменты HDL-описаний соответствуют рассмотренной последовательности из трех повторителей для случая моделирования с задержками и без них (некоторая разница в реализации моделей задержек в приведенных примерах VHDL-VERILOG пока не обсуждается).

Модель с задержками:

VHDL	**VERILOG**
	`timescale 1 ns/1 ns
D<=C after 10 ns;	assign #10 D=C;
C<=B after 10 ns;	assign #10 C=B;
B<=A after 10 ns;	assign #10 B=A;

Модель без задержек (бесконечно малая дельта-задержка подразумевается):

VHDL	**VERILOG**
D<=C;	assign D=C;
C<=B;	assign C=B;
B<=A;	assign B=A;

Методы трансляции HDL-описаний

Различают *компилирующие (compiler)* и *интерпретирующие системы* моделирования. В компилирующих сначала осуществляется перевод текста модели в машинный код, а затем полученная программа может исполняться. Для рассмотренного примера в случае компиляции каждому уравнению типа B = A соответствуют примерно две команды машинного кода IBM-PC.

В системах интерпретирующего типа текст модели преобразуется во внутреннее представление — некий псевдокод, который затем в ходе моделирования пооператорно расшифровывается и исполняется программой-интерпретатором.

Компилирующие системы обычно имеют на порядок большую скорость моделирования (их примерами могут служить для HDL системы NC-VERILOG, NC-VHDL фирмы Cadence). Интерпретирующие системы затрачивают меньшее время на трансляцию и лучше приспособлены для отладки моделей (пример: VERILOG-XL фирмы Cadence и др.), но медленнее моделируют.

Компилятивное интервальное моделирование эффективно и применяется только в специализированных HDL-системах моделирования (cycle based, time driven simula-

tor) схем, описываемых на вентильном уровне без задержек с автоматическим ранжированием описаний при компиляции. В HDL-системах моделирования универсального типа, как уже отмечалось, применяется событийное.

Средства автоматизации проектирования

Редакторы

Для ввода HDL-описаний используют как обычные текстовые редакторы (типа NOTEPAD), так и графические (вводится рисунок схемы), а также и синтаксически ориентированные редакторы, реализующие подсказку, проверяющие HDL-код, выделяющие цветом ключевые слова, и т. п. (пример — настроенный на HDL редактор EMAX, встроенные редакторы систем ACTIVE-HDL и др.).

Отдельно можно выделить системы, помогающие строить тестирующие программы (testbench builder), например, встроенный в ACTIVE-HDL редактор построения теста объекта проекта по HDL-описанию объекта).

HDL-анализаторы и reuse-checker'ы

Программы, предназначенные только для проверки (анализа) HDL-описаний, называются анализаторами (analyzer), а проверяющие также стиль описаний и соблюдение правил синтезабельности — чекерами (lint tool).

Схемные компиляторы (синтезаторы)

Системы синтеза, преобразующие высокоуровневые HDL-описания в схемы, называются синтезаторами, схемными компиляторами (design compiler).

Временные анализаторы

Программы, помогающие проследить задержки и критические пути в синтезированных схемах, называются статическими временными анализаторами (static timing analyzer), пример — Prime Time фирмы Synopsys.

Свободные интернет-ресурсы

Читателям, желающим использовать при изучении HDL свободно распространяемые через Интернет системы или их демоверсии, можно рекомендовать следующие, апробированные автором в лабораторном практикуме МЭИ (часть других см. в разделе «Интернет-ресурсы» списка литературы).

Моделирование

VHDL

Direct VHDL demo (www.gmvhdl.com). Простая система, но с большими ограничениями — в модели допускается только одна компонента, слабый редактор текстов и диагностика ошибок.

Active-HDL (www.aldec.com). Фирма Aldec периодически разрешает скачивать новые (с ограничениями) версии своей двуязычной системы моделирования.

VERILOG

Silos demo (www.simucad.com). Отличная система — до 200 компонент, Verilog-справочник, оценка полноты покрытия кода тестом.

Active-HDL (www.aldec.com). Фирма Aldec периодически разрешает скачивать новые (с ограничениями) версии своей двуязычной системы моделирования.

Программы проверки стиля описаний (Reuse checker)

VERILOG

Reuse Checker — демоверсия (http://midc.miem.edu.ru). Реализует проверку развитого набора правил и стиля синтезабельного описания проекта. Разработка А. Сохацкого и студентов МИЭМ.

Синтезаторы

E-MAX, MAX+ PLUS, QUARTUS (www.altera.com) — синтез ПЛИС фирмы Altera.

Xilinx WebPack (www.xilinx.com) — синтез ПЛИС фирмы Xilinx.

Глава 1
HDL — взгляд схемотехника и взгляд программиста

1.1. HDL — взгляд разработчика аппаратуры

1.1.1. Отображаемые аспекты

В HDL-описании, как и в любой модели, отражаются только некоторые аспекты (характеристики) реальной системы.

Цифровую аппаратуру характеризуют, например, такие аспекты, как функциональный (реализуемая функция, алгоритм); временной (задержки, производительность, время отклика); структурный (типы и связи компонент); ресурсный (число вентилей, площадь кристалла); надежностный (время наработки на отказ); конструктивный (вес, габариты); стоимостной и т. д.

HDL содержит средства, позволяющие отобразить в основном первые три аспекта: функциональный, временной и структурный (рис. 1.1).

Как уже отмечалось, функция (поведение) аппаратуры может детализироваться от уровня системы команд и алгоритмов устройств до булевских функций; структура — от уровня устройств типа процессор — память до уровня вентилей и переключающих элементов; время — от задержек фронтов сигналов (нано — ns и фемтосекунды — fs) до тактов и задержек электромеханических устройств (секунды и часы).

Степень детализации аспектов, отображаемых в описаниях аппаратуры, определяется конкретными задачами. Например, описание некоторой микропроцессорной системы может строиться как описание структуры, состоящей из микросхем БИС и СИС, а описание самих микросхем строится как поведенческое, если их описание на вентильном уровне либо отсутствует, либо слишком громоздко.

В HDL встроен ряд понятий, обычно используемых проектировщиками аппаратуры. Ниже при изложении этих понятий в скобках приводятся английские термины, используемые в HDL как ключевые, зарезервированные слова.

Рис. 1.1. Возможности HDL в области отображения основных характеристик (структурный, временной, функциональный аспект) аппаратуры

1.1.2. Интерфейс объекта проекта

Полное HDL-описание каждого объекта проекта (в дальнейшем для краткости — объекта — entity, модуля — module) состоит из двух частей: *описания интерфейса объекта* и *описания тела объекта* (описание архитектуры — architecture в терминологии VHDL).

Интерфейс объекта проекта (module, entity) определяет его имя, входы-выходы и параметры. Входы-выходы — это состав портов (port), их имена, направленность (входы — in, input, выходы — out, output, двунаправленные — inout), разрядность (один бит — bit или вектор — bit_vector), алфавит кодирования (0, 1 — один из стандартов для VHDL или 0, 1, Z, X — стандарт для VERILOG) сигналов, способ представления их значений (целый — integer, вещественный — real) и т. д.

Например, у объекта проекта по имени SM (рис. 1.2) три входных (input, in) порта: A, B, C и один выход (output, out): S, на которые могут поступать сигналы, имеющие двоичные (bit) значения 0 или 1.

Рис. 1.2. Интерфейс объекта проекта SM

Однотипные объекты проекта могут различаться параметрами настройки (generic, parameter), наример разрядностью или задержкой. TZ — параметр, характеризующий задержку SM.

Ниже приведены HDL-описания интерфейса объекта проекта SM.

VHDL

```
entity SM is
generic(TZ: time:=0 ns);
  port (A, B, C: in bit;
        S: out bit);
end SM;
```

VERILOG

```
`timescale 1 ns/100 ps
module SM (A, B, C, S);
  parameter TZ=0;
  input  A, B, C;
  output  S;
```

Последняя версия языка VERILOG (VERILOG-2000) позволяет описывать порты объекта проекта в форме, более близкой к принятой в языке VHDL.

```
module SM (input  wire A,B,C, output  wire S);
```

Порты объекта проекта, как уже отмечалось, характеризуются направлением потока информации.

Они могут быть входными (в VHDL зарезервированное слово in, в VERILOG — input), выходными (VHDL — out, VERILOG — output), двунаправленными (inout). Кроме того, в VHDL могут быть однонаправленными буферными (buffer) (с порта типа buffer выходной сигнал объекта можно считывать в том же объекте, что нельзя делать с портом типа out) и связными — linkage.

Порты также имеют тип, характеризующий тип значения поступающих на них сигналов.

В VHDL — это, например, стандартный (встроенный) тип bit или n-битовый вектор — bit_vector, — каждый разряд которого двузначный (принимает значение 0 или 1).

В VERILOG — это обычно одноразрядный или n-битовый вектор, разряд которого имеет четырехзначный тип (по умолчанию) со значениями 0, 1, z — высокий импеданс, x — неопределенно. Подробнее об алфавите представления сигналов смотрите ниже и в главе 2.

1.1.3. Описание структуры объекта проекта

Тело объекта проекта специфицирует его структуру и функцию (поведение, алгоритм). Его описание в HDL VERILOG следует за описанием интерфейса модуля, а в языке VHDL выделяется в отдельную часть и содержится в *описании архитектуры объекта* (architecture) (рис. 1.3).

Рис. 1.3. Пример соответствия множества архитектур одному интерфейсу в VHDL

Если проектировщик представил функцию устройства как схему (структуру) из элементов низшего уровня сложности, то HDL для него в первом приближении это форма кодирования схем.

Средства HDL для отображения структур цифровых систем базируются на представлении о том, что *описываемый объект проекта* (entity, module) представляет собой структуру из более простых объектов-*компонентов* (component), соединяемых друг с другом *линиями связи* (проводами — wire (VERILOG) или сигналами — signal (VHDL). Каждый компонент, в свою очередь, является объектом и может состоять из компонент низшего уровня (иерархия объектов). Взаимодействуют объекты путем передачи *сигналов* (signal) по линиям связи. Линии связи (wire — в VERILOG) или отождествляемые с ними сигналы (signal — в VHDL) подключаются к входным (in, input) и выходным (out, output) портам (port) связываемых компонентов.

Например, компонент M2 (рис. 1.4) объекта SM имеет два входных порта — X1, X2 и один выходной — Y.

Экземпляры однотипных компонент, как уже отмечалось, могут различаться параметрами настройки (generic в VHDL, parameter в VERILOG), например задержкой сигналов.

На рис. 1.4 представлена структура объекта проекта SM, состоящего из двух экземпляров компонентов типа M2, с именами экземпляров K1 и K2 и задержками 15 ns и 10 ns.

Рис. 1.4. Структурный аспект объекта проекта SM, состоящего из двух экземпляров компонентов типа M2 (схема объекта SM)

В HDL-описании схемы имена входных сигналов (портов) объекта проекта SM и имена идущих от них линий связи совпадают (они отождествляются). Для внутренних сигналов (линий связи, цепей соединений), связывающих компоненты друг с другом, необходимо вводить индивидуальные имена. Например, линия S_TMP (сигнал S_TMP на рис. 1.4) объединяет компоненты K1 и K2.

Структурное описание объекта проекта SM строится как описание связей экземпляров (конкретизаций — instance) компонентов, каждый из которых имеет персональное имя (имя конкретизации), тип, значение параметров и связи с входными и выходными сигналами (список конкретизации).

В простых случаях используется позиционный принцип сопоставления портов компонентов поступающим на них сигналам (позиционная карта портов — port map).

То есть первый сигнал в карте портов (списке конкретизации) сопоставляется первому порту компонента, второй — второму и т. п. (смотри пример конкретизации K1).

В сложных случаях при большом количестве портов компонентов и произвольном порядке их перечисления в карте портов используется ключевой принцип (явный — explicit): список конкретизации образуют пары имя порта компонента — имя подсоединенного к нему сигнала (смотри пример конкретизации K2).

Например, описание связей компонентов объекта проекта SM, представленного на рис. 1.4, выглядит следующим образом (после символов -- в VHDL и // в VERILOG и до конца строки следует комментарий):

--VHDL

```
architecture STRUCTURA of SM is

  signal S_TMP: bit;-- промежуточный

  begin -- для K1 позиционное
     K1: entity M2 generic map (15 ns)
          port map (A, B, S_TMP);
    -- для K2 - ключевое
    -- соответствие
    K2: entity M2  port  map
      (X1=>S_TMP,
            Y=> S,
           X2=> C);
end STRUCTURA;
```

VERILOG

```
// описание интерфейса SM было
// выше
  wire S_TMP;//промежуточный

//для K1 позиционное соответствие
      M2  #(15)
          K1 (A,B,S_TMP);
//для K2 - ключевое
//соответствие и порядок пар
//порт-сигнал произвольный
    M2  K2 (.X1(S_TMP),
            .Y(S),
             .X2(C));
endmodule //SM
```

Как уже отмечалось, в описании связей компонентов объекта проекта SM использованы следующие обозначения:

— K1, K2 — имена экземпляров компонента M2;

— M2 — тип компонента;

— 15 ns задержка для конкретизации K1; конкретизация K2 имеет задержку 10 ns по умолчанию, VERILOG — описание SM предполагает масштаб времени 1 ns (timescale — см. ниже также описание объекта проекта M2) с точностью 100 ps;

— A, B, C, S — имена внешних сигналов, связанных с портами;

— S_TMP — имя промежуточного сигнала.

Если какой-то порт не соединен с сигналом, в VHDL это обозначается ключевым словом open, в VERILOG — при позиционном способе это место обозначается запятой.

Ключевой способ может применяться и при указании фактических значений параметров настройки.

В нашем случае для конкретизации K1 при незадействованном порте X2 и ключевом способе указания значения параметра настройки TDEL мы имели бы:

VHDL	VERILOG
K1: entity M2 generic map(TDEL=>15 ns); port map (A, open, S_TMP);	M2 K1 (A, ,S_TMP); defparam K1.TDEL= 15;

VERILOG-2000 допускает более похожую на VHDL форму передачи параметров настройки : M2 #(.TDEL(15)) K1(A, ,S_TMP);

1.1.4. Связь имен компонентов и объекта проекта

Остается вопрос о связывании имени компонента с именем соответствующего объекта проекта. VERILOG предполагает полное соответствие этих имен. VHDL более гибок, допуская различие имени компонента и имени соответствующего объекта проекта. При совпадении имен компонента и объекта проекта можно применять прямое создание экземпляра компонента, как это было сделано выше. Можно также объявить VHDL-компонент и использовать связывание по умолчанию, когда система ищет в рабочей библиотеке проекта последнее из откомпилированных описаний объектов с таким же именем. Связывание (конфигурирование — configuration) этих имен осуществляется либо так называемым объявлением конфигурации либо спецификацией конфигурации. В нашем VHDL-примере использование описания компонента и спецификация конфигурации выглядели бы так:

```
architecture STRUCTURA_COMP of SM is
signal S_TMP;--промежуточный
component M2      -- компонент M2
                                        -- в VERILOG раздел описаний
   generic TDEL=10 ns;-- задержка       -- компонент отсутствует, но
   port(X1,X2:in bit;  -- входы         -- где-то вовне в библиотеке
                                        -- проекта должно быть
        Y :out bit);                    -- описание соответствующих
end component M2;                       -- объектов с такими же именами
for all:M2 use entity work.M2(BEH);     -- спецификация конфигурации
  begin                                 -- для нашего VHDL примера излишняя
              -- можно было  использовать  связывание по умолчанию
```

```
  K1:  M2 generic map (15 ns)  port map (A, B, S_TMP);
  K2:  M2  port  map (S_TMP, C,S);
end STRUCTURA_COMP;
```

В версию VERILOG-2000 включен подобный механизм, так называемый блок конфигурации. Использование этой конструкции позволяет зафиксировать в коде VERILOG-описания имена и размещение библиотек с описаниями объектов проекта, сопоставленных компонентам.

VERILOG

```
config  CONF_SM //имя конфигурации для примера CONF_SM
design ADDER_LIB.TOP //Устанавливает порядок поиска объектов проекта,
                //сопоставленных компонентам - в примере
                // в виртуальной библиотеке ADDER_LIB
library ADDER_LIB "./*.v"; //определяет место виртуальной библиотеки -
                // в примере текущая директория и  файлы с расширением .v
endconfig
```

1.1.5. Поведение объекта проекта

Из приведенных примеров видно, что, несмотря на существенную разницу в синтаксисе языков, общность структурных описаний схем очевидна. Структурное представление отображает типы и связи компонентов, но их поведение в явном виде в нем не описывается, и что делает, например, компонент M2 с входными сигналами, из него не ясно.

Компонент представляет объект проекта низшего ранга, функция которого может быть раскрыта, или он может быть, в свою очередь, описан структурно. Так, описание функции объекта проекта M2 в случае, если он реализует операцию сложения по модулю 2 (xor — исключающее ИЛИ), может быть таким:

VHDL

```
entity M2 is
generic (TDEL: time:=10 ns);
port (X1,X2 : in bit;
      Y : out bit);
end;

architecture BEH of M2 is
begin
   Y<=  (X1 xor X2) after TDEL;
end;
```

VERILOG

```
`timescale 1 ns /100 ps
module M2 (X1,X2,Y);
   parameter TDEL=10;
   input X1,X2;
   output Y;
  // конец интерфейса

  //ниже тело модуля M2

  assign #(TDEL) Y= (X1 ^ X2);
endmodule
```

Если в текстовый файл с описанием объекта проекта SM включить описание объекта M2, например вставив его в начало файла, то мы получим полное описание проекта SM, в котором раскрыто поведение компонента M2.

Когда функция объекта M2 реализована в операциях классического базиса И, ИЛИ, НЕ и пользователь хочет это отразить, то функция M2 в этих классических операциях могла бы выглядеть так:

VHDL

```
Y<=  ((X1 and not X2) or
     (not X1 and X2)) after TDEL;
```

VERILOG

```
assign #(TDEL) Y= (X1 & ~X2)|
                  (~X1 & X2);
```

Читателю в качестве упражнения предлагается самостоятельно разработать структурное описание объекта проекта M2, состоящего из компонентов AND2, OR2, NOT1 и составить описания объектов проекта AND2, OR2, NOT1.

Язык VERILOG, в отличие от VHDL, включает встроенные примитивы n-разрядных вентилей на произвольное число входов: and, or, nand, xor, xnor, одновходовые примитивы not, buf и др. В нашем простейшем случае структурное VERILOG-описание объекта проекта SM могло бы быть таким:

//VERILOG

```
`timescale 1 ns/100 ps
module SM_FROM_PRIMITIVES (A, B, C, S);
input  A, B, C; output  S;wire S_TMP;
        xor #15 K1 (S_TMP ,A,B); //в примитиве xor
                                 //- выход - первый в списке параметров
        xor #10 K2 (S, S_TMP,C);
endmodule
```

1.1.6. Разнообразие стилей описаний архитектур

HDL допускает большое разнообразие стилей описаний объектов проекта, и, например, объект проекта SM может быть представлен чисто поведенческим способом (как «черный ящик»).

VHDL

```
entity SM_BEH is
   generic (TDEL: time:=25 ns);
   port (A, B, C: in bit;
         S: out bit);
end SM_BEH ;--конец интерфейса
arhitecture BEHAVIOUR of SM_BEH is
begin
  S<=  (A xor B xor C) after TDEL ;
end;
```

VERILOG

```
`timescale 1 ns/100 ps
module SM_BEH (A, B, C ,S);

  input  A, B, C;
  output  S;
  parameter TDEL=25;

  assign #(TDEL) S= (A ^ B ^ C);
endmodule // SM_BEH
```

Остаток этого параграфа содержит примеры, иллюстрирующие другие, более близкие к возможностям обычных языков программирования средства HDL в сфере описания функционирования объектов. Детально эти средства будут рассмотрены позже, и читатель, чувствующий затруднения, может без ущерба пропустить этот материал до начала следующего раздела.

Одноразрядный сумматор. Дадим пример использования объекта проекта SM-полусумматора как компонента одноразрядного сумматора adder, который помимо суммы sum формирует перенос cout. Иллюстрируется смешанный структурный — для суммы — sum и потоковый (вход регистра, преобразуясь, проходит на его выход) для переноса — cout стиль описания объекта adder.

VHDL

```
entity adder is
  generic(T_SM: time:=25 ns);
  port (a : in bit;
     b : in bit;
```

VERILOG

```
`timescale 1ns/10 ps
module adder

(a,b,cin,sum,cout);
input a,b,cin;
```

```
    cin  : in bit;
    sum  : out bit;
    cout : out bit);
end adder;
architecture mix of adder is
begin
  summa: entity SM port map
       (a,b,cin,sum);
  cout <= (a and b) or
        (cin and a) or
       (cin and b)after T_SM;
end;
```

```
output sum,cout;
parameter T_SM= 25;

SM summa
      (a,b,cin,sum);
assign #(T_SM)cout = (a & b)|
                 ( cin & a) |
                (cin & b);
endmodule
```

n-разрядный сумматор с последовательным переносом. Приведем пример поведенческого описания n-разрядного сумматора (adderN) без задержек (сумматор с последовательным переносом).

VHDL

```
entity adderN is
  generic (n:integer:=10);
  port (a,b: in bit_vector
        (n downto 1);
     sin : in bit; sum : out
     bit_vector(n downto 1);
     cout :out bit);
end;
architecture beh of adderN is
 begin
   p1: process(a, b, cin)
       variable vsum :
     bit_vector(N downto 1);
       variable carry : bit;
 begin
   carry := cin;
   for i in 1 to n loop
   vsum(i) := (a(i)
          xor b(i)) xor carry;
   carry := (a(i) and
           b(i)) or (carry and
          (a(i) or b(i)));
    end loop;
    sum <= vsum;
    cout <= carry;
    end process p1;
end beh;
```

VERILOG

```
module adderN
   (a,b,cin,sum,cout);
    parameter n=10;

  input [n:1]a,b; input cin;
  output[n:1]sum;
  output cout;

   integer i;
   reg [n :1] vsum;
   reg carry;
   always @ (a or b or cin)
  begin
    carry = cin;
   for(i=1;i<=n;i=i+1) begin
    vsum[i] =(a[i]
            ^ b[i]) ^ carry;
    carry = (a[i] & b[i]) |
                (carry &
                (a[i] | b[i]));
    end //for
   end
   assign sum = vsum;
  assign cout = carry;
endmodule //adderN
```

n-разрядный сумматор. Далее приведен пример еще более абстрактного уровня описания n-разрядного сумматора adder N_B (не ясно, какой перенос — параллельный или последовательный).

VHDL-модель использует подключение пакетов, в которых определен тип STD_LOGIC_VECTOR и арифметические операции над ним (для BIT_VECTOR они не определены). В обоих описаниях применена операция конкатенации (сцепления) — символ & в VHDL и символы{,} в VERILOG.

В VHDL сигнал vsum имеет дополнительный разряд слева, чтобы хранить перенос, который затем присваивается в cout, в VERILOG это делается в составе конкатенации {cout,sum} в левой части присваивания.

VHDL

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.std_logic_unsigned.all;
entity adderN_B is
  generic (n:integer:=10);
  port (a,b:
    in std_logic_vector
    (n downto 1);
  cin : in std_logic;
  sum : out
  std_logic_vector(n downto 1);
    cout :out std_logic);
end adderN;
architecture beh of adderN_B is
  begin process
 variable    vsum,a1:
   std_logic_vector(n+1 downto 1);
   begin wait on a,b,cin
     a1:= '0'& a;
     vsum := a1+b+ cin;
    sum <=vsum (n downto 1);
    cout <=vsum(n+1);
   end process;
end beh;
```

VERILOG

```
`timescale 1 ns/100 ps

module adderN_B
 (a,b,cin,sum,cout);

 parameter n=10;

 input [n:1]a,b;input cin;
 output[n:1]sum;;
 output cout;

 reg [n :1] sum;
 reg cout;
 always @ (a or b or cin)
 begin
 {cout,sum}= a+b+cin;

 end

endmodule //adderN_B
```

Тестирующая программа. Здесь приведен пример программы, тестирующей одноразрядный сумматор путем подачи двух входных наборов: a, b, c = 000,111.

VHDL

```
entity adder_tb is end;
architecture BEH of adder_tb is
      component adder
      port(
            a, b, cin : in bit;
            sum : out bit;
            cout : out bit);
  end component;
  -- Stimulus signals
  signal a : bit;
  signal b : bit;
  signal c : bit;
  -- Observed signals
  signal sum : bit;
 signal cout : bit;
begin
-- ниже тестовые векторы
```

VERILOG

```
`timescale 1 ns/100 ps
module adder_tb;

reg a;
reg b;
reg c;

wire sum;
wire cout;
```

```
    process begin
    a<='0';b<='0';c<='0';
    wait for 100 ns;
    a<='1';b<='1';c<='1';
    wait for 100 ns;
  end process;
  -- Unit Under Test port map
  UUT : adder

        port map (
              a => a,
              b => b,
              cin => c,
              sum => sum,
              cout => cout
        );
end BEH;
```

```
initial begin
 a<=0;b<=0;c<=0;
    #100;
 a<=1;b<=1;c<=1;
 #100; $finish;
end
//тестируется adder
    adder  UUT

        (
         .a(a),
         .b(b),
         .cin(c),
         .sum(sum),
         .cout(cout)
        );
endmodule //adder_tb
```

Из приведенных примеров видно, что при описании поведения сложных объектов приходится использовать разнообразные общеалгоритмические и специальные средства HDL, частично рассмотренные ниже.

1.2. HDL — взгляд программиста

Если посмотреть на HDL глазами программиста, то можно сказать, что он состоит как бы из двух компонентов — общеалгоритмического и проблемно-ориентированного (рис. 1.5).

Рис. 1.5. Два компонента HDL: общеалгоритмический и проблемно-ориентированный

Общеалгоритмический компонент VHDL — это язык, близкий по синтаксису и семантике к языкам программирования типа ПАСКАЛЬ, АДА (см. приложение 1, где приведено неполное краткое изложение синтаксиса и семантики языка VHDL).

Общеалгоритмический компонент VERILOG — это язык, близкий по синтаксису и семантике к языкам программирования типа С, С++ (см. приложение 2).

Ниже приведены некоторые конструкции, общие для HDL и стандартных языков программирования общеалгоритмическая компонента HDL.

В качестве первого простого примера рассмотрим программу печати приветствия (например, Hello, HDL!), обычно используемую в пособиях для начинающих изучать языки программирования.

VHDL

```
entity Privet1  is
end Privet1;
architecture Simple of Privet1 is
begin
  process
   begin

   report "Hello, HDL!";
   wait;--останов
 end prosess;
end Simple;
```

VERILOG

```
module Privet1 ();

  always
  begin
    $display
    ("Hello, HDL!");
    $finish;//останов
  end
endmodule
```

1.2.1. Лексические элементы HDL

Комментарий

Комментарий в VHDL — это текст, начинающийся с символов «--» и до конца строки, в VERILOG — это текст, начинающийся с символов «//» и до конца строки.

В VERILOG, как и в С++, допускается многострочный комментарий, начинающийся с символов «/*» и кончающийся символами «*/».

Имена

Имена (идентификаторы) — это последовательность букв и цифр, начинающаяся с буквы. В число разрешенных внутри идентификатора включен символ «_». В VERILOG в число разрешенных включен также символ «$», и имя может начинаться с символа «_» и заканчиваться им.

В VHDL, в отличие от VERILOG, строчные и прописные буквы не различаются.

Примеры правильных имен:

VHDL	VERILOG	Примечания
Abba_11 ABBA_11	Abba_11 ABBA_11	В VHDL первое и второе имя — это одинаковые имена В VERILOG — это разные имена
—	$A	Только в VERILOG в имени допускается символ «$»

Примеры запрещенных идентификаторов:

VHDL	VERILOG	Примечания
2a _e1 У2__У; P_	2a 73M +ab E2++E4	Имя начинается с запрещенного в начале символа VHDL — два __ подряд запрещены
if	if	Имя if зарезервировано (ключевое слово)

Некоторые имена зарезервированы — это ключевые слова языка. Многие из них общеприняты в языках программирования.

Примеры: begin, end, if, else, for, while и т. п.

Список ключевых слов VHDL приведен в приложении 1, VERILOG — в приложении 2.

Часто встречающаяся ошибка в именах-при подготовке текстов программ перепутываются внешне одинаковые символы латиницы и кириллицы и близкие по изображению символы букв и цифр O и 0, l и 1 и т. п. Например имя AC1, набитое в латинском алфавите, отличается от имени AC1, набитого в русском.

Расширенные имена

Расширенные имена могут включать любые печатные символы кода ASCII.

Они начинаются в VERILOG с символа «\» и заканчиваются пробелом.

В VHDL они ограничены символом «\».

Расширенные имена позволяют увеличить диапазон допустимых имен.

Пример:

VHDL	VERILOG
\f+bxo_xo\	\f+bxo_xo

Составные имена

Составные имена — это последовательность имен, разделенных точками (с их помощью осуществляется доступ к элементам структур и другим объектам — например, пакетам VHDL, внутренним переменным компонент (экземпляра модуля) VERILOG и т. п.).

Пример: VS.DD.E1

Константные значения (литералы)

Константные числовые значения (литералы) могут быть представлены в двоичной, восьмеричной, десятичной форме. Символ «_» внутри константы повышает наглядность записи значений.

Примеры чисел:

VHDL	VERILOG	Примечания
127	127	Целое число 127 в десятичной системе
16#7F#	7'h7F	Число 127 в шестнадцатеричной системе
2#111_1111#	7'b111_1111	Число 127 в двоичной системе
0.5	0.5	Вещественное число 0.5
—	1'b1	Одноразрядное число 1

Символьные и строковые литералы

Примеры:

VHDL	VERILOG	Примечания
'0'	"0"	Символ 0
'A'	"A"	Символ A
"10N1"	"10N1"	Четырехсимвольная строка

Битово-строковые литералы

Примеры:

VHDL	VERILOG	Примечания
B"1110"	4'b1110	Битовая строка 1110
X"E"	4'he	Шестнадцатеричный код 1110
O"16"	6'o16	Восьмеричный код 001110

Поименованные константы

Описание проекта должно легко модифицироваться и вместо константных значений в тексте лучше использовать их имена.

Примеры объявлений констант (первый пример допускает изменение значения константы S1 из внешнего модуля, так как она объявлена как параметр, второй пример для константы ONE — не допускает):

VHDL	VERILOG	Примечания
generic (S1:integer:=20);	parameter S1= 20;	
constant ONE :bit:='1';	`define ONE 1'b1;	

Версия VERILOG-2000 расширяет возможности явного описания констант, объявляя их в блоке specify локальными параметрами, не изменяемыми из внешнего модуля. Пример: localparam BB= 33; // VERILOG-2000.

Ниже приведен пример программы печати приветствия Hello, Vasia с использованием констант.

VHDL	VERILOG
entity Privet2 is end Privet2; architecture Simple of Privet2 is constant s1:string :="Hello, Vasia!"; begin process begin report s1; wait; end process; end Simple;	module Privet2 (), parameter s1= "Hello, Vasia!"; always begin $display ("%s",s1); $finish; end endmodule

1.2.2. Данные (объекты): типы и виды

Имена данных (объектов по терминологии VHDL), используемых в модуле, должны быть объявлены. При описании указываются:

— **тип объекта**, задающий диапазон возможных значений и операций (например, данные целого типа — integer, вещественного — real);

— **вид или класс объекта**, задающий дополнительные свойства, например состав применимых к объекту операторов присваивания (например, в VHDL — это виды: **константа**, **сигнал** и **переменная** — constant, signal, variable; в VERILOG — это **параметр**, **переменная** и **соединение** — parameter, reg, wire). Здесь перечислены типы и виды объектов (данных) VHDL и VERILOG.

VHDL		VERILOG	
Типы	Виды	Типы	Виды
Скалярные типы:			
перечислимый (enumerated)	переменная (variable)	reg	переменная
целый (integer)	сигналі (signal)	nteger	
		time	
физический	константа (constant)	real	
плавающий (float)		realtime	
файл (file)			
ссылочный (access)		*Подвиды*	
		wire, wand	соединение
Агрегатные типы:		wor, triand	(нет)
Индексируемый (array)		trior	
		tri0,tri1	
структурный (record)		supply0, supply	
		событие (event)	событие
		integer,real	параметр
		ASCII строка reg,time	(parameter)

Из вышеприведенного следует что если, например, в VHDL целые (integer) могут быть как переменными (variable), так и сигналами (signal), то в VERILOG целыми могут быть только переменные. Зато соединения (цепи) VERILOG могут быть нескольких подвидов — обычные (wire), схемное И (Wand), схемное ИЛИ (Wor) и др., но не могут быть вещественного типа (real).

VHDL имеет три класса типов данных: **скалярные типы** (числовые (целые, плавающие), физические, перечислимые); **составные (агрегатные) типы** (индексируемые и структурные); **ссылочные типы** и **файловые типы**. Примеры объявлений типов VHDL:

TYPE BYTE is range 0 to 255; — целый тип с диапазоном значений от 0 до 255
TYPE A is ('0', '1'); -- перечислимый тип со значениями '0' и '1'

VHDL включает предопределенные (объявленные в пакете STANDARD (см. приложение 1), подключенном по умолчанию) перечислимые типы: bit, character, boolean, severyty_level, file_open_kind, file_open_status; целый тип integer, плавающий тип real и физический тип — time, индексируемые — string и bit_vector; подтипы целого типа: positive (>0) и natural (>=0). При подключении пакета

STD_LOGIC_1164 пользователю становятся доступными многозначные типы std_logic и std_logic_vector, являющиеся расширениями двузначного bit-(«0»,«1») в девятизначный алфавит, рассматриваемый в следующей главе.

VERILOG данные вида переменные и соединения, в частности, допускают скаляры и векторы (одномерные массивы) многозначного типа (значения разряда 0, 1, x, z). Над видом переменные (типы reg, integer и т. п.) определен, в частности, оператор процедурного присваивания (Procedural Assignment): блокирующий (blocking assignment) имеет символ = и неблокирующий (Non blocking assignment) символ <= .

Над объектом вида соединение (связь — wire, wor и т. д.) определен оператор непрерывного присваивания (Continuous Assignment). Подвиды соединений, как уже отмечалось, соответствуют типичным схемотехническим элементам: монтажное И (wand), ИЛИ (wor), источник питания (supply1, supply0) и т. д. Кроме того, цепям может задаваться различная сила сигнала (см. приложение 2).

Пример wire A,B; assign A=B;

Примеры объявлений данны:

VHDL	VERILOG	Примечания
signal E1,E2: bit;	wire E1,E2;	сигналы E1,E2
variable M2:bit;	reg M2;	переменная M2
signal E13,E23:bit_vector(1 to 3);	reg [1:3] E13,E23;	трехбитовые E13, E23
signal AII_2: integer;	integer AII_2;	целый AII_2

Данные (объекты) вида переменная — variable в VHDL аналогичны обычным переменным языков программирования. Они описываются и локализованы в пределах описаний подпрограмм и блоков процессов (понятие, рассматриваемое ниже). Им можно сопоставить данные соответствующего вида языка VERILOG (данные типов reg, integer, real, time могут быть в нем только переменными).

Данные вида соединение — (VERILOG) и вида сигнал — signal (VHDL), как уже отмечалось, можно сопоставить проводам-сигналам в модели.

Массивы данных:

VHDL	VERILOG	Примечания
type MA is array (0 to 10) of bit; signal A: MA;	reg A[0:10];	одноразрядный массив A (вектор)
type MB is array (1 to 5) of bit_vector (2 to 6); variable B : MB;	reg [2:6] B [1:5];	массив пяти битовых векторов B

Verilog-1995 допускал только одномерные массивы и только для переменных (reg, integer, real, time) и исключал прямое обращение к разрядам и полям отдельных слов массивов — приходилось копировать все слово массива во вспомогательную переменную, а затем из нее извлекать нужный бит или поле. Verilog-2000 не имеет этого ограничения.

Пример Verilog-2000:

```
reg [31:0] arr2 [0:25][0:15];// объявлен двумерный массив arr2
wire [7:0] out2 = arr2[100][7][31:24]; // из массива arr2 извлекается поле 31:24
```

VERILOG позволяет использовать векторные данные со знаком, объявляя их с дополнением signed. Арифметические операции учитывают это.

Пример Verilog-2000:

```
reg signed [6:0] F1;  wire signed [8:7] B2 [0:10];
input signed [1:0] a;function signed [12:0] XX;
```

VHDL допускает арифметические операции над двоичными и булевскими векторами только при подключении специальных пакетов (см. приложение 1).

Объявления данных — область расположения

Как правило, объявления данных должны предшествовать их использованию.

В VHDL объявления располагаются между заголовком блока (architecture, process, block, function, procedure, package) и ключевым словом begin.

В VERILOG местоположение описаний контролируется не так строго, но описания должны предшествовать использованию данных.

Начальные значения данных

Перечислимые типы данных VHDL по умолчанию принимают первое из указанных в описании типа значений.

VHDL	**VERILOG**
boolean- FALSE,bit-'0', std_logic-'U'	reg и wire- 1'bx

Однако можно явно указать начальное значение в описании данных:

signal A: bit :='1';	reg A=1'b1;// VERILOG-2000

Преобразование типов данных

В VHDL для стандартных числовых типов integer, real это достигается указанием типа перед аргументом в скобках, для строкового типа — указанием типа и атрибута «image» или «value», при подключении пакетов — указанием включенных в них функций преобразования (см. главу 2 и приложение 1). Для остальных преобразований необходимо использовать специальные функции, см., например, описание памяти в главе 3.

Пример: преобразование строки в целый — integer'value("314") — дает 314;
преобразование целого в строку — integer'image(314) — дает "314";
преобразование вещественного в целое — integer(3.14) — дает 3;

В VERILOG преобразования осуществляются функциями §realtobits(x) — вещественное в вектор и §bitstoreal(x) — наоборот.

Данные, объявляемые по умолчанию

VHDL, как строго типизированный язык, подразумевает, что все данные должны быть объявлены. Лишь необъявленный параметр цикла воспринимается как integer.

VERILOG не столь строг. Например, в VERILOG необъявленный идентификатор в левой части оператора непрерывного присваивания в ряде случаев подразумевается объявленным как wire с длиной, равной длине правой части. Это может приводить к неожиданным ошибкам.

Типы и подтипы данных, определяемых пользователем (VHDL)

Язык VHDL позволяет вводить новые типы данных.

Пример объявления и использования нового типа:

TYPE mem IS ARRAY (0 TO 1023) OF BIT_VECTOR(31 DOWNTO 0);
(массив 1024 32-битовых векторов)

Пример его использования: variable RAM: mem;

Пример объявления типа BIT_VECTOR — вектор неопределенной длины, взятый из стандартного пакета STANDARD (см. приложение 1).

type bit_vector is array(NATURAL range <>) of BIT; -- индексы от 0 и выше

Подтип (subtype) позволяет вводить объекты, область значений которых является подмножеством базового типа и которые могут иметь ассоциированную с ними разрешающую функцию (см. гл. 2 и приложение 1).

Пример:

subtype byte is bit_vector(7 downto 0);signal A:byte;

1.2.3. Операции и выражения

1.2.3.1. Операции

Помимо общепринятых арифметических операций (+, -, *,/) и операций отношения, в HDL введено большое число логических операций. Ниже приведены примеры только ряда из них.

Особенности операций в многозначном алфавите поясняются в следующей главе. Результат VERILOG-операций иногда приводится в той же строке таблицы в скобках.

Примеры:

Операция	Пример		
	VHDL	VERILOG	Результат
Арифметические операции (над целыми и вещественными — VHDL, всеми типами — VERILOG)			
Сложение	2+3	2+3	5
Вычитание	2-3	2-3	-1
Умножение	2*3	2*3	6
Деление цел.	2/3	2/3	0
Модуль	2 mod 3	2 % 3	2
Остаток	-2 rem 3	—	-2
Абсолютное	abs (-1)		1
Степень (VERILOG-2000)	2 ** 3	—	8
Унарные арифметические			
Плюс	+1	+1	1
Минус	-1	-1	-1
Минус		- 4'b1011	4'b0101

При работе с VERILOG следует обратить внимание на способ представления отрицательных чисел: используется дополнительный код (дополнение до 2). Например, в integer i;initial i=-12; значение -12 представлено как ffffffff4 в шестнадцатеричном коде (дополнительный код), и если присвоить i, например, в 16-разрядную переменную reg [15:0] D; D=i; то в D будет шестнадцатеричный код fff4.

Операция	Пример		
	VHDL	VERILOG	Результат
Логические операции над булевскими (Boolean) (VHDL)			
И	TRUE AND FALSE	—	FALSE
ИЛИ	TRUE OR FALSE	—	TRUE
НЕ	NOT TRUE	—	FALSE
Поразрядные логические операции над двоичными (bit) и (bit_vector) векторами VHDL и различными типами переменных и цепей VERILOG			
Поразрядное И	B"1011" AND B"0010"	4'b1011 & 4'b0010	B"0010"(4'b0010)
Поразрядное ИЛИ	B"1011" OR B"0010"	4'b1011 | 4'b0010	B"1011"(4'b1011)
Отрицание	not B"1011"	~ 4'b1011	B"0100"(4'b0100)
Исключающее ИЛИ	B"1011" XOR B"0010"	4'b1011 ^ 4'b0010	B"1001"(4'b1001)
Поразрядная инверсия	NOT B"0010"	~4b0010	B"1101"(4'b1101)
Логические операции (VERILOG) дают 1-битовый результат			
Логическое И	—	4'b1011 && 4'b0010 1'b1 && 1'b0	1'b1
Логическое ИЛИ	—	4'b1011 || 4'b0010 1'b1 || 1'b0	1'b1
Логическое отрицание	—	! 4'b1011	1'b0
Некоторые из унарных операций редукции над n-разрядными данными (VERILOG)			
И всех разрядов	—	&4'b1101	1'b0
ИЛИ всех разрядов	—	| 4'b1101	1'b1
Операции отношения (над числами и строками)			
Равенство	B"1011" = "0010"	4'b1011 == 4'b0010	FALSE (1'b0)
Равенство чисел	5 = 6	5 == 6	FALSE
Неравенство	B"1011" /= B"0010"	4'b1011 != 4'b0010	TRUE (1'b1)
Больше	B"1011" > B"0010"	4'b1011 > 4'b0010	TRUE (1'b1)
Меньше	B"1011" < B"0010"	4'b1011 < 4'b0010	FALSE (1'b0)
Больше-равно	B"1011" >= B"0010"	4'b1011 >= 4'b0010	TRUE (1'b1)
Меньше-равно	5 <= 5	5 <= 5	TRUE (1'b1)
Идентичность (VERILOG)			
Идентичность	—	4'b1011 === 4'b0010	1'b0 (FALSE)
Неидентичность	—	4'b1011 !== 4'b0010	1'b1 (TRUE)

Различие VERILOG-операций отношения и идентичности заметно в многозначном алфавите. Пример:

Операция 4'd1xx0 != 4'd1011 дает результат 1'bx, что, например, означает FALSE для условия в операторе if, а операция 4'd1xx0 !== 4'd1011 дает результат 1'b1, т. е. TRUE.

Операция	Пример		
	VHDL	VERILOG	Результат
Логические сдвиги (в примерах на 2)			
Сдвиг влево	B"1011" SLL 2	4'b1011 << 2	"0010" (4'b1100)
Сдвиг вправо	B"1011" SRL 2	4'b1011 >> 2	"0010" (4'b0010)
Арифметические сдвиги (VHDL)			
Сдвиг влево	B"1011" SLA 2	—	"1111"
Сдвиг вправо	B"1011" SRA 2	—	"1110"
Циклические сдвиги (VHDL)			
Сдвиг влево	B"1011" ROL 2	—	"1110"
Сдвиг вправо	B"1011" ROR 2	—	"1110"
Разные операции			
Конкатенация (сцепление)	"100" &"11"	{3'b100,2'b11}	"10011" (5'b10011)
Реплика-повтор	—	{2{2'b01}}	6'b010101

Приоритет операций — как и в обычных языках программирования — наибольший у унарных, но во избежание ошибок лучше пользоваться скобками для указания порядка вычислений.

*Пример: (((a+b)*c)> (m-n)).*

1.2.3.2. Выражения

Обычные выражения строятся из констант, имен, операций и скобок.

Пример:

VHDL	VERILOG
(F+H)* (C-D)	(F+H)* (C-D)

VHDL строго контролирует типы данных, операций и длины операндов (векторов) в выражениях, например запрещено сложение 2+'1', если не подключен соответствующий пакет (см. приложение 1). VERILOG не столь строг, допуская выражения вроде 2+1'b1+"A".

Выражения с полями векторов и массивов.

VERILOG не допускает работу с полями переменной длины. Однако его версия VERILOG-2000 позволяет использовать переменные индексы полей фиксированной длины.

Пример:

VHDL	VERILOG
variable A:bit_vector(32 downto 0);	reg [32:0] A;
variable B,k,nI,j:integer	reg [3:0] B;
variable C bit_vector(7 downto 0)	wire [7:0] C;
C<=A((B+1)*8 downto B*8);	assign C=A[B*8 +: 8];//VER-2000

C(K downto N)<=A(I downto J); -- это возможно в VHDL, а VERILOG подобное поле переменной длины не допускает.

Помимо обычных выражений допускаются задержанные.

Задержанное выражение используется в операторах присваивания (в VHDL — это оператор назначения сигналу, VERILOG — присваивания переменной) и является его частью. Сначала оно вычисляется и через время задержки присваивается).

Пример:

VHDL	VERILOG
S<= A+B after TD;	S<=#(TD) A+B;

Подробнее вопросы задержек рассмотрены в главе 2.

VHDL позволяет назначать временные диаграммы сигналов.

Пример: S<= 10 after 2 ns, 40 after 10 ns, 2 after 20 ns;

VERILOG допускает также **условные выражения** — их значение определяется условием, например, выражение (A>B) ? 3'b100:3'b111 дает результат 3'b100, если A > B истинно, т. е. равно 1'b1.

Пример вложенных условных выражений: x=(a>b)? c: (v && L)?d-1: 0;

1.2.4. Операторы

HDL обеспечивает возможность описания обычных последовательных алгоритмов.

Последовательно выполняемые (последовательные) операторы HDL могут использоваться в описаниях процессов, процедур и функций. Их состав включает (см. приложения 1, 2) следующие операторы:

Оператор	Пример VHDL	Пример VERILOG
1. **Ожидания**		
условия	wait until A1=B1;	wait A1==B1;
задержки	wait for 10 ns;	#10;
события	wait on A,B;	@(A or B);

HDL содержит средства описания ожидания процессами определенного типа событий — фронтов сигналов:

ожидание фронта C:	wait until C'event and C='1';	@posedge(C) ;
ожидание среза C:	wait until C'event and C='0';	@negedge(C);

Их использование будет рассмотрено в других главах.

VHDL-оператор ожидания позволяет объединять все три типа условий.

Пример: wait on a until d>f for 19 ns;--после события в а ждать пока станет истинно d>f и потом еще 19 ns.

VERILOG допускает отсутствие символа «;» после оператора ожидания — тогда он становится условием запуска следующего оператора.

Пример: @(a) wait(d>f); #19;//так выглядит предыдущий пример

2. **Оператор присваивания** переменной (VHDL), процедурного блокирующего присваивания (VERILOG) переменной	V := V1+V2; --variable V	V = V1+V2; // reg V
3. **Последовательный оператор** назначения сигнала (VHDL) процедурного неблокирующего присваивания (VERILOG) переменной	--signal S S1<=S2;	//reg S S1<=S2;

4. **Условный оператор**	if A=B then
	S1<=S2;
	end if;
		if (A==B)
		S1<=S2;

5. **Оператор выбора**	case S is
	when A => B:=Y;
	when D=>B:=notY;
	when others=>B:='1';
	end case;
		case (S)
		A: B=Y;
		D: B=~Y
		default:B=1;
		endcase

6. **Оператор цикла**	for i in 1 to 4 loop M(i):=0; end loop; while A<B loop A:=A+1; end loop;	for (i=1;i<=4;i=i+1) M[i]=0; while A<B A=A+1;
7. **Оператор возврата**	return F;	//нет аналога
8. **Оператор вызова процедуры**	P(X1,X2);	P(X1,X2);
9. **Оператор выхода** из цикла (VHDL), из группы операторов (VERILOG)	exit;	disable GG;
10. **Оператор перехода к следующей итерации** в цикле	next;	//нет аналога
11. **Оператор вывода**	report "A=B";	$display ("A=B");
12. **Пустой оператор**	null;	//нет аналога
13. **Последовательный оператор утверждения**	assert A=B report "XOXO";	//нет аналога

Ниже детальнее рассмотрены только некоторые из них (подробнее см. приложения 1, 2).

1. Обычное для языков программирования присваивание

VHDL	VERILOG
оператор присваивания переменной :=	оператор блокирующего процедурного присваивания переменной =

Примеры:

VHDL	VERILOG	Примечания
variable X1,X2: bit; X1 := '1'; X2 := X1;	reg X1,X2; X1 = 1'b1; X2 = X1;	X1 присваивается 1 после этого в X2 присваивается 1;

Обратите внимание на вид данных в левой части и символ оператора.
В VHDL — присваивание переменной — вид variable и символ := ,
В VERILOG — блокирующее присваивание переменной и символ =. ,
Не следует путать этот оператор с оператором, который обозначается символом <= и называется последовательным оператором назначения сигнала в VHDL или неблокирующего процедурного присваивания в VERILOG.

VERILOG — дополнительные виды операторов присваивания переменным:

А) Усиленное присваивание force.

Устанавливает значение и блокирует возможность изменения переменной или связи другими операторами до снятия запрета оператором release.

Пример: reg a; force a=1; // установка a в 1 с блокировкой
release a; // снятие блокировки с a

В) Процедурное непрерывное присваивание переменной assign, перекрывающее другие возможные присваивания переменной до отмены его оператором deassign (некоторая аналогия с оператором assign для соединений)

```
reg a,b,c; assign a=b & c;
           deassign a;
```

2. Оператор <= назначения сигнала (VHDL) — неблокирующего процедурного присваивания переменной (VERILOG)

Примеры:

VHDL	VERILOG	Примечания
SIGNAL X1,X2 :bit;	reg X1,X2;	
X1<= '1';	X1<=1'b1;	X1 будет присвоено новое значение 1 через ΔT
X2<=X1;	X2<=X1;	X2 будет присвоено старое значение X1 через бесконечно малое ΔT

Оператор <= обладает задержкой присваивания как минимум равной бесконечно малой величине T_DELTA и не блокирует выполнение следующих за ним операторов. Подробнее он будет рассмотрен в следующей главе.

VERILOG, в отличие от VHDL, не контролирует соответствие разрядности правой и левой частей оператора присваивания и, если правая часть больше, обрезает ее слева, а если меньше — дополняет правую слева по особым правилам.

Пример:

VERILOG

reg [1:3] A; A=4'b1100;	A примет значение 3'b100
A=2'11;	A примет значение 3'b011
A=2'x1;	A примет значение 3'bxx1.

3. Условный оператор (if)

VHDL	VERILOG
if A>B then X1:=Y;	if (A>B) X1=Y;
else C:=D;	else begin
X1:=not Y;	C=D; X1=~Y;
end if ;	end

Если условие истинно (TRUE в VHDL или равно 1 в VERILOG), то выполняется первый оператор, в противном случае выполняется оператор, стоящий за else. Часть else может отсутствовать.

Для вложенных операторов if в VHDL предусмотрено сокращение elsif.

4. Оператор выбора (case)

Оператор выбора (case) выбирает одну из альтернатив, а если их список не исчерпывает всех возможностей, то выполняет умалчиваемую ветвь:

others (VHDL), default(VERILOG).

VERILOG также допускает еще две разновидности case:

casez-endcase, которая позволяет указывать z на месте безразличного значения разряда в условии выбора и

casex-endcase, которая позволяет указывать на этом месте z и x.

VERILOG

```
`define state1 3'b0x1
`define state2 3'b1xx
reg [0:2]state;
casex (state)
        `state1: B=Y;
        `state2: B=1;
        default:B=0;
endcase
```

5. Оператор цикла (for, while) и дополнительные VERILOG циклы — repeate, forever

VHDL

```
variable i:integer;
type MEM is array (1 to 11)of integer;
variable M,N :MEM;
for i in 1 to 10 loop
M(i):= M(i+1);
N(i+1):= N(i);
end loop;
```

VERILOG

```
integer i;
integer M[1:11];
integer N[1:11];
for (i=1;i<=10; i=i+1) begin
  M[i]= M[i+1];
  N[i+1]=N[i];
end
```

VERILOG выделяет также частные случаи циклов — бесконечный цикл forever (навсегда) и с заданным числом повторений — repeat.

Пример генератора сигналов: initial forever begin x=0;#10;x=1;#10;end

Пример генератора 20 тактов: initial repeat(20) begin x=0;#10;x=1;#10;end

6. Операторные скобки (группа операторов)

VERILOG использует пары ключевых слов begin- end, fork- joint для группировки операторов. Группа begin- end включает последовательно исполняемые операторы, fork-joint — параллельные ветви.

В отличие от VERILOG, язык VHDL поддерживает концепции структурного программирования. Сложные операторы формируют пары ключнвых слов:

if — end if; process — end process; case — end case, loop — end loop и т. д.

Поэтому отпадает необходимость в специальных операторных скобках для выражения составных операторов, и, например, запись if X>Y then A:=B; C:=D; if означает последовательное выполнение двух присваиваний при истинности условия X>Y.

7. Функции и процедуры

Как и в обычных языках программирования, в HDL для описаний общих фрагментов моделей можно использовать функции и процедуры.

Пример объявления функции и процедуры и их вызова:

VHDL

```
function F1(X,Y:bit)return bit is
variable B:bit;
begin
  B:= not X and Y;
  return B;
 end;
 procedure P1(X: in bit;
      variable Y: out bit) is
end;
```

VERILOG

```
function F1;
input X,Y;
 //по умолчанию
 //F1-однобитовое
   F1=~X & Y;
endfunction
task P1 (X,Y);
  input X;output Y;
endtask
```

Здесь X и Y — формальные параметры, а тело процедуры опущено.

Язык VHDL позволяет специально указывать функции с побочным эффектом (impure). VERILOG не контролирует это свойство функций.

Например, impure function pop (A, V) — это заголовок функции, которая не только возвращает значение из вершины стека V, но и меняет указатель A.

Пример обращений к функции и процедуре:

VHDL

```
M<=F1(A,B);
P1(A, B);
```

VERILOG

```
M<=F1(A,B);
P1(A, B);
```

VERILOG-2000 допускает рекурсивные функции и процедуры — ключевое слово automatic (старый VERILOG не позволял этого, и, например, нельзя было параллельно вызывать одну и ту же VERILOG-процедуру).

Пример рекурсивного вычисления факториала:

```
function automatic [63:0] factorial;// VERILOG 2000
    input [31:0] n;
    if (n == 1)  factorial = 1;
   else          factorial = n * factorial(n-1);
endfunction
```

Ниже приведен пример программы вывода приветствия «Hello, Vasia!» с использованием процедуры.

VHDL

```
entity Privet3  is
end Privet3;
architecture PROC of Privet3 is
procedure PRINT(s1:in string)) is

   begin
    report  s1;
end ;
begin
process begin
PRINT("Hello, Vasia!");
  wait;
end process;
end;
```

VERILOG

```
module Privet3 ();


task PRINT;
input [20 *8:0]s1;
begin
$display  ("%s",s1);
endtask

always begin
PRINT("Hello, Vasia!");
$finish;
 end
endmodule
```

8. Операторы ввода-вывода

Эти операторы HDL обычно используются при отладке и тестировании моделей.

В VHDL ввод-вывод (помимо report) поддерживается стандартным окружением языка (пакетом STD.TEXTIO), в VERILOG — системными процедурами и функциями (их имена начинаются с символа «$»).

Ниже вариант печати приветствия с использованием переменных и (VHDL) пакета ввода-вывода textio. Символ занимает 8 битов, и это отражено в VERILOG-варианте. Для битовой переменной C на следующей строке печатается имя и значение.

VHDL

```
use STD.TEXTIO.all;
entity Privet4  is
end Privet4;
architecture Simple of Privet4 is
begin
  process
  variable s3:string(1 to 11);
  variable B,C,D:bit;
  variable  A1:line;
  begin
   B:='1';D:='0';C:=D and B;
   s3 :="Hello" & "Vasia!";
  write (A1,s3);
  writeline(output,A1);
  write(A1,"C=");
  write(A1,C);
  writeline(output,A1);
  wait;
 end prosess;
end Simple;
```

VERILOG

```
module Privet4 ();
  reg[13*8:1] s3;
  reg B,C,D;

 always


 begin
   B=1;D=0;C=D & B;
   s3={ "Hello", "Vasia!"};
   $display
   ("%s", s3);

   $display("C=%b", C);

   $finish;
 end //always
endmodule
```

Другие последовательные операторы HDL смотрите в приложениях 1, 2.

1.2.5. Механизм расширения языка

VHDL унаследовал от языка ADA механизм пакетов. Пакет является средством расширения языка. В **пакет** можно включать нужные типы данных, переопределять операции над ними, включать в пакет нужные процедуры. Пакет компилируется, включается в библиотеку и при необходимости подключается к описанию объекта проекта. Стандартный пакет STANDARD, содержащий описания стандартных типов данных и операций, считается подключенным по умолчанию (библиотека STD). Подробнее см. приложение 1.

VERILOG не имеет пакетов, не допускает введения новых типов данных и переопределения операций, но имеет мощный интерфейс с языком C (PLI интерфейс) и, как и язык C, имеет препроцессор. **Препроцессор** позволяет, например, включать заранее подготовленные файлы с описаниями данных, процедур и функций в тексты описаний объектов проектов. Типичные директивы препроцессора (компилятора): 'define, 'include, 'ifdef (подробнее см. приложение 2).

Приведем примеры оформления пакета по имени PAC1 и файла FILE1.v с вышеприведенной процедурой P1:

VHDL

```
package PAC1 is
procedure P1(X: in integer;
 variable Y: out integer);
end;-- окончено описание пакета PAC1
package body PAC1 is
    procedure P1(X: in integer;
    signal Y: out integer) is
  -- X и Y - формальные параметры
     variable Z1: integer:=0; --локальная Z1
     begin -- далее исполняемые операторы
     if X=1 then Y<=10;
       else Y<=X+1;
            Z1:=X;
      end if;
    end P1;
end ;-- окончено тело пакета PAC1
```

VERILOG

```
//пусть этот текст с task P1

//содержится в файле FILE1.v

task P1 (X,Y);
 input [1:32] X; output [1:32]Y;
 integer Z1;
 begin
      if (X==1)Y=10;
       else begin Y=X+1;
            Z1=X;
      end //if
endtask //P1
```

Примеры: VHDL-подключение пакета PAC1, предварительно откомпилированного в рабочую библиотеку WORK, и VERILOG-включения файла FILE1.v из текущей директории в описание объекта XX, не имеющего портов.

VHDL

```
Use work.PAC1.all ;
--весь пакет
entity XX is
end;--обьект XX не имеет портов
architecture primer of xx is
signal A,B: integer;
begin
--ниже обращение к процедуре P1
   process begin
   A<=1; wait for 1 ns;
   P1(A, B);--
   wait on B;
   end process;
end ;
```

VERILOG

```
module xx ();
  // модуль XX не имеет портов
  // включение файла FILE1.v
     `include "FILE1.v"
     integer A,B;

  always begin
     A<=1;#1;
     P1(A, B);
     @(B);
  end
endmodule //xx
```

Более детально с особенностями описаний VHDL-пакетов можно ознакомиться в следующей главе на примере фрагментов пакета std_logic_1164 и в приложении 1.

Выход на другие языки в VHDL обеспечивают подпрограммы с описателем foreign, например foreign function F1(X,Y:bit)return bit;

VERILOG имеет встроенный выход на язык Си — интерфейс PLI, включающий более 100 стандартных подпрограмм, а также механизм доступа ко внутренним форматам представления данных и к компонентам системы моделирования.

1.2.6. Область видимости данных

Видимость в VHDL

Язык VHDL, как и все современные алгоритмические языки, предусматривает блочную структуру программ, которые могут строиться из вложенных друг в друга блоков (block).

Блок — это ограниченный фрагмент VHDL-текста, содержащий раздел описаний и раздел исполняемых операторов.

Пример архитектурного тела (являющегося тоже блоком), содержащего внутренние блоки:

VHDL

```
entity M is  port (B: in real);  end M;
   library IEEE; use IEEE.std_logic_1164.all;
  -- подключена библиотека IEEE и ее пакет std_logic_1164
   architecture EE of M is      -- архитектура - это особый вид блока
   signal C: real;              -- раздел описаний  EE
  shared variable XOXO:integer:=1;-- разделяемая переменная видна везде
   signal Z: std_logic;          -- тип std_logic из пакета std_logic_1164
     begin
     B1: block                   -- вложенный в EE блок B1
        signal D: real;          -- раздел описаний блока B1
          begin
           -- D<=E;              -- это было бы неверно,
           --      т.к. обьект E в B1, EE, M и IEEE.std_logic_1164 не описан!
            C<=B+C;              -- B1 и B2 взаимодействуют через сигнал C
     end block B1;
     B2: block                   -- блок B2
        signal E: real;          -- раздел описаний блока B2
          begin
        P1: process
             variable J: positive;  -- локальная переменная  процесса P1
               begin                -- виден сигнал C
                 E<=C; wait for 10 ns; J:=J+1;XOXO:=XOXO +1;
            end process;
       P2: process    begin        -- J не видна, но видна
              wait for 20 ns; XOXO:=XOXO -1;--разделяемая переменная XOXO
            end process;

   end block B2;
end EE;
```

В данном примере показано, что из блока B1 «видимы»: сигнал C, описанный во внешнем блоке EE, и порт B, но «невидим» сигнал E из блока B2 и тем более J — локальная переменная процесса P1.

Кроме того, в архитектуре EE «видим» тип std_logic, объявленный в пакете std_logic_1164, указание использования которого (use) стоит перед описанием архитектуры EE.

Особенностью локальных переменных (variable), описанных в блоках-процессах, являются их статичность, т. е. сохранение их значений в периоды пассивности и возможность изменения в периоды активности процессов только операторами своего процесса (переменная J в данном примере).

Однако разделяемые (shared) переменные, декларированные с описателем shared в заголовке архитектуры, видимы внутри всех ее процессов (переменная XOXO в данном примере).

В основном все структурные элементы VHDL-описаний, содержащие исполняемые операторы, строятся по такой схеме:

начальное ключевое слово — architecture, process, block, package, package body, procedure, function, entity;

раздел объявлений;

begin

раздел исполняемых операторов;

конечное ключевое слово — end, end process, end block, end package и т. п. с возможностью добавления имени.

Видимость в VERILOG

VERILOG имеет четыре вида пространств видимости данных.

Глобальные имена — видимы везде. Это имена модулей, примитивов, конфигураций (VERILOG-2000) и макросов (`define).

Локальные имена — видимы в пределах своей области определения (если не использовать иерархических имен).

Области: модуль, объявление функции, объявление процедуры, поименованная группа (begin-end fork-join).

Имена, обьявленные в блоке specify, действуют в нем.

Атрибуты (VERILOG-2000) действуют в сопоставленном им обьекте.

Итак, область видимости данных локализована в таких конструкциях, как модуль (module), описание функции (function), процедура (task), поименованная группа операторов (begin-end, fork-joint), блок спецификации (specify). Система программирования ищет описание данных сначала в локальном блоке, а потом по иерархии во внешних, и так до границ модуля.

Однако при использовании иерархических имен могут становиться видимыми извне имена данных внутренних блоков и из одного модуля внутренние имена данных другого.

Пример (внутреннее имя tmp модуля M1, видимо, из внешнего модуля tb, в котором модуль M1 конкретизируется):

VERILOG

```
module M1(x,y);input x; output y;reg y;integer tmp;
      initial tmp=0; always @ (posedge x) begin tmp=tmp+x; y=tmp+x; end
endmodule//M1
module tb();
  reg a;initial a=0;wire b; // переменная a и цепь b видимы в модуле tb везде
  initial repeate (20) #(20)a=~a;
  M1 UU1 (a,b);
  // доступ к tmp ниже при печати его значения
  always @(b)
     $display ("tmp=% b",UU1.tmp);//UU1 - имя конкретизации модуля M1
endmodule //tb
```

1.2.7. Модули и библиотеки проекта

Разработка больших проектов, как аппаратных, так и программных, включает создание и сопровождение **проектных библиотек** или **библиотек проектов** (design library).

VHDL предполагает, что программная система, называемая **анализатором** (подсистема компилятора), осуществляет проверку фрагментов VHDL-описаний систем и, если они не содержат ошибок, помещает их в **рабочую библиотеку проекта** (work).

Фрагменты описаний, которые могут независимо анализироваться VHDL-системой и при отсутствии ошибок помещаться в библиотеку проекта, называются **проектными модулями** (design unit). Такими модулями могут быть объявление объекта проекта (entity), объявление архитектуры (architecture), объявление конфигурации (configuration), объявление пакета (package) и объявление тела пакета (package body).

Модули проекта, в свою очередь, можно разбить на две категории: **первичные** (primary) и **вторичные**. К первичным модулям относятся объявления пакета (package), объекта проекта (entity), конфигурации (configuration). К вторичным — объявление архитектуры и тела пакета (package body). Это разделение связано с тем

что анализ вторичного модуля невозможен без предварительного анализа соответствующего первичного. Например, используемые в описании архитектуры порты объявляются в описании объекта. Соответственно в ходе анализа вторичного архитектурного модуля системе необходимо знать, в какой библиотеке искать первичный — описание интерфейса объекта.

Модуль, помещенный в библиотеку, называется библиотечным (library unit). По отношению к сеансу работы с VHDL-системой существует два класса библиотек проекта: **рабочие библиотеки** и **библиотеки ресурсов**.

Рабочая библиотека — это библиотека WORK, с которой в данном сеансе работает пользователь и в которую помещается модуль, полученный в результате анализа модуля проекта.

Библиотека ресурсов — это библиотека, содержащая библиотечные модули, ссылка на которые имеется в анализируемом модуле проекта.

В каждый конкретный момент времени пользователь может работать с одной рабочей библиотекой и произвольным количеством библиотек ресурсов.

Перед описанием каждого модуля должно стоять указание об используемых библиотеках. При этом подразумевается, что каждый модуль проекта содержит следующее указание по умолчанию:

```
library STD,WORK; use STD.STANDARD.ALL;
```

Логическое имя STD обозначает библиотеку проекта, в которой содержатся пакеты STANDARD и TEXTIO. Описание использования use в примере делает все объявления, содержащиеся внутри пакета STANDARD, «видимыми» в соответствующем модуле проекта. Логическое имя WORK обозначает текущую рабочую библиотеку.

Вопросы к главе 1

1. Перечислите преимущества и недостатки языка VHDL и VERILOG.
2. Сколько архитектурных тел может соответствовать одному объявлению объекта VHDL?
3. Чем отличается поведенческое описание архитектуры от структурного описания?
4. Внутри каких конструкций HDL могут использоваться последовательные операторы?
5. Почему не будут различаться результаты выполнения фрагментов а) и в) процессов, содержащие VHDL-назначения сигналам, VERILOG — неблокирующие присваивания переменным при начальном Y1 = 0:

VHDL	VERILOG
a) Y1<=1; Y2<=Y1+1;	a) Y1<=1; Y2<=Y1+1;
в) Y2<=Y1+1; Y1<=1;	b) Y2<=Y1+1; Y1<=1;

и будут различаться для фрагментов c) и d), содержащих присваивания: переменным (variable — VHDL), процедурные блокирующие присваивания в переменные — VERILOG:

c) Y1:=1; Y2:=Y1+1;	c) Y1=1; Y2=Y1+1;
d) Y2:=Y1+1; Y1:=1;	d) Y2=Y1+1; Y1=1;

Глава 2
Базовые понятия HDL — процессы, задержки, алфавит

Проблемно-ориентированная компонента HDL — это та его составляющая, ко торая позволяет описывать цифровые системы в привычных разработчику аппа ратуры понятиях и терминах. Помимо отмеченных в первой главе понятий: объ ект проекта (интерфейс, архитектура), компонента и т. п. — сюда можно отнести (рис. 2.1):

— **понятие параллельных взаимодействующих процессов** и параллельных опера торов;

— **понятие автоматически наращиваемого модельного времени** (функци NOW — в VHDL и $time — в VERILOG) и задержек;

— **понятие присваивания с задержкой** (механизм отображения задержки рас пространения сигналов — значения сигналов изменяются не мгновенно, как обычных переменных, а обязательно с некоторой задержкой);

— **понятие многозначного** алфавита представления сигналов.

Рис. 2.1. Специфика HDL

Мы начнем с рассмотрения процесса (частный вид блока) как более широко используемого на практике и имеющего очень похожие средства отображения в VHDL и VERILOG.

2.1. Параллельные процессы

Средства HDL для отображения поведения систем базируются на представле нии их как *совокупности параллельных взаимодействующих процессов (process, always)*. **Все параллельные операторы HDL могут быть представлены эквивалентными по следовательными операторами, вложенными в оператор процесса.**

2.1.1. Параллельные операторы HDL

Название	Пример VHDL	Пример VERILOG
	signal SA,SB:bit;	reg RA,RB;
1. Оператор процесса	process (SB)begin SA<=SB; end process;	always @(RB) RA<=RB,

2. Параллельный оператор назначения сигнала (VHDL) (непрерывного присваивания (VERILOG))	SA<=SB;	wire WA assign WA=RB;
3. Оператор конкретизации компонента	M:N port map(SA,SB);	N M(WA,RB);

Далее перечень параллельных операторов, присущих только VHDL.

1. Оператор блока

```
V1:block --          - позволяет вводить
                     -- иерархию в описание проекта
begin                -- внутри блок содержит
 X<= SB;             -- параллельные операторы
 SM<= X;
end block;
```

2. Параллельный оператор утверждения (контроля)

```
assert SA=SB          -- каждое событие в SA или SB
report "BAD SA";      -- вызывает проверку условия
                      -- и при ложном значении
                      -- условия происходит выдача сообщения
```

3. Параллельный оператор вызова процедуры

```
P(SA,SB);             -- если SA - входной параметр, то
                      -- каждое
                      -- событие в нем параллельно со
                      -- следующим оператором
                      -- запускает выполнение P
```

4. Оператор генерации

```
for i in 1 to 10 generate          -- пример заменяет
                                      строки
L1:M2 port map (A(I),B(I),S(I));   -- текста с 10-ю
  end generate;                    -- конкретизациями
                                   -- компонента M2
```

Перечень параллельных операторов, присущих только VERILOG.

1. Оператор инициализации

```
initial begin #3 RA=0; #5 RB=1; end
/*однократно исполняемый процесс
 запускается в начальный момент времени
*/
```

2. Оператор внутри параллельной группы fork

```
fork    //одновременно запускаются
  RA=1;     // оба оператора, и когда
 #3 RB=0; // оба завершатся, завершится
 join       // выполнение группы
```

2.1.2. Оператор процесса

Оператор процесса включает заголовок процесса (process — VHDL, always — VERILOG) и последовательность операторов, отображающих действия процесса

по переработке информации. Хотя бы один из этих операторов должен быть оператором ожидания (задержки на определенное время или до выполнения условия продолжения). Все операторы внутри процесса выполняются последовательно. Процесс может находиться в одном из двух состояний:

— *пассивном*, когда процесс ожидает прихода сигналов его запуска (продолжения) или наступления соответствующего момента модельного времени;

— *активном* — когда исполняется.

Параллельные процессы взаимодействуют через общие сигналы (соединения).

Пример 1 — автономный непрерывный процесс.

Ниже приведен пример простого автономного процесса, описывающего генератор сигналов с периодом 10 ns: NOW и §time — функции, возвращающие текущее значение модельного времени.

VHDL

```
signal Y: bit;begin
GEN1:process
begin
   if NOW=0 ns then Y<="0";
   end if;
   wait for 5 ns;
   Y<= NOT  Y;
end process;
```

VERILOG

```
`timescale 1 ns/1 ns
      reg Y;
  always
  begin: GEN1

    if($time==0)Y<=1'b0;
     #5;
    Y<= ~Y;
  end //GEN1
```

VERILOG имеет (как уже отмечалось) специальный вид однократно протекающего процесса, запускаемого в начальный момент времени. Ниже пример его использования с оператором бесконечного цикла при описании генератора.

```
initial begin
  Y<=1'b; forever  begin #5; Y<= ~Y;end
end //GEN2
```

Пример 2 — два взаимодействующих процесса.

Для того чтобы иметь возможность остановить генератор, надо включить в его описание условие остановки по сигналу из другого процесса.

VHDL

```
signal Y,DONE:bit;
 constant TPULS:TIME:=5ns;
constant TSTOP:time :=1000 ns;
GEN2:process
begin
   If NOW=0 ns then Y<='0'; end if;
   Wait for TPULS;
   Y<= NOT Y;
   if DONE ='1' then
   WAIT; end if;
end process;-- GEN2
STOP_PROC:process
begin
   DONE<='0';
   wait for TSTOP;
```

VERILOG

```
`timescale 1 ns/1 ns
   reg Y,DONE;
   parameter TPULS= 5;
   parameter TSTOP=1000;
  always
   begin: GEN2
    if($time==0)Y<=1'b0;
     #(TPULS);
     Y<= ~Y;
    if (DONE ==1)
     wait (DONE==0);
  end  // GEN2
 always
   begin: STOP_PROC:
    DONE<=1'b0;
     #TSTOP;
```

```
  DONE<= '1';                          DONE= 1'b1;
  WAIT;                               wait (DONE==1'b0);
end process STOP_PROC;              end// STOP_PROC
```

Процесс STOP_PROC после установки DONE в 1 зависает. Процесс GEN2 каждые TPULS проверяет значение DONE и, если оно становится равным 1, зависает в бесконечном ожидании противного.

Пример 3 — взаимодействующие процессы с условием ожидания событий.

Процессы могут реагировать не только на определенные значения сигналов, но и на их изменения (события, фронты).

Например, описание функции модуля M2 (см. главу 1, раздел 1.2), использующее общую форму представления процессов, ожидающих событий во внешних сигналах X1 и X2, может быть таким:

VHDL

```
entity M2 is
generic (TDEL: time:=10 ns);
port (X1,X2 : in bit;
      Y : out bit);
end M2;
acrhitecture BEHAVIOUR of M2 is
begin
variable TMP:bit;
 A: process
   begin
     TMP:=  X1  xor X2 ;
     wait for TDEL; Y<= TMP;
   wait on (X1,X2);
  end process;
end BEHAVIOR;
```

VERILOG

```
`timescale 1 ns/ 1ns
module M2 (X1,X2,Y);

 input X1,X2;
 output Y;reg Y;
 // конец интерфейса
 // ниже тело модуля
 parameter TDEL= 10;
 reg TMP;
 always
 begin :A
    TMP=  X1  ^ X2 ;
    #(TDEL); Y<= TMP;
    @(X1 or X2);
 end
endmodule //M2
```

В данном примере в VHDL используется вариант оператора wait с условием ожидания события (изменения) или в сигнале X1, или в сигнале X2. Это же условие в VERILOG-процессе (@ — символ ожидания события, or в списке означает — любое из событий).

Пример 4 — процессы со списком чувствительности.

HDL позволяет вынести условие запуска процесса в список чувствительности процесса. Однако если в VHDL-процессе вынести в список сигналы, к которым процесс чувствителен (в смысле ожидания в них событий), то в нем нельзя использовать оператор wait. Поэтому соответствующий VHDL вариант процесса A (см. пример 3) ниже использует оператор назначения сигнала Y с задержкой:

VHDL

```
entity M2 is
generic (TDEL: time:=10 ns);
port (X1,X2 : in bit;
      Y : out bit);
end;
architecture BEHAVIOUR1 of M2 is
```

VERILOG

```
`timescale 1 ns/ 1 ns
module M2 (X1,X2,Y);

input X1,X2;
output Y;reg Y;
// конец интерфейса
```

```
 begin
A:  process (X1,X2)
   begin
     Y<=  (not X1 and X2) or
          (X1 and not X2)
        after TDEL;
  end process;
end;
```

```
parameter TDEL= 10;
always @(X1 or X2)
begin :A
 Y<=#(TDEL)(~X1 & X2) |
        (X1 & ~ X2);

end
endmodule
```

VERILOG-2000 допускает форму записи списка чувствительности процесса подобную VHDL (вместо or — запятая), а также можно использовать обобщенный символ @* для указания о том, что в списке должны присутствовать все сигналы к которым процесс чувствителен.

В нашем примере always @(X1, X2) или always @*.

Пример 5 — процессы, сопоставленные компонентам.

Как уже отмечалось выше, объект проекта модуль M2, рассмотренный в разделе 1.2 главы 1, реализует операцию сложения по модулю 2 или xor. То есть Y будет равен 1 только в том случае, если X1 = 0 и X2 = 1 или X1 = 1 и X2 = 0. При каждом изменении значения входных сигналов X1 или X2 (событии на входе) запускается процесс вычисления нового значения выходного сигнала, которое присваивается в Y с задержкой 10 ns. Если Y изменяется, происходит событие на выходе модуля M2. Как уже отмечалось, процессы взаимодействуют путем обмена сигналами. На рис. 2.2 приведен возможный вариант поведенческого представления архитектуры, рассмотренного в главе 1 объекта SM, как состоящего из двух процессов: P1 и P2.

Рис. 2.2. Представление поведения объекта SM как совокупности двух взаимодействующих процессов P1 и P2, сопоставленных компонентам K1 и K2 структурного представления (см. гл. 1)

Если сопоставить его со структурным представлением (см. рис. 1.4), то в нашем примере P1 соответствует компоненту K1, P2 — компоненту K2. Процессы P1 и P2 осуществляют суммирование по модулю 2 входных сигналов и передают результата на выход с задержкой. В общем случае в поведенческом описании состав процессов не обязательно соответствует составу компонентов, как это имеет место в структурном описании, и объект SM может быть представлен не двумя одним, тремя и т. д. процессами.

Далее дан пример записи процессов в объекте SM средствами HDL с использованием списка чувствительности. Каждый из процессов ожидает, пока не изменится хотя бы один из указанных в списке чувствительности входных сигналов. Когда это *изменение (событие)* произойдет, процесс становится активным, пос

довательно выполняет свои операторы до тех пор, пока не достигнет конца своего описания. Процессы действуют циклически, оператор end process в конце процесса возвращает управление в его начало.

Процесс P1 вырабатывает новое значение сигнала S_TMP с задержкой 15 ns, после чего снова переходит в режим ожидания. Аналогично P2 суммирует S_TMP и C, вырабатывая с задержкой 10 ns новое значение суммы S.

VHDL

```
entity SM is
    port (A, B, C: in bit;
          S: out bit);
end ;--конец интерфейса
architecture BEHAVIOR of SM is
    signal S_TMP: bit;
      begin
---------процесс P1
        P1: process(A,B)
         begin
            S_TMP<=A xor B after 15 ns;
          end process P1;
        -- процесс P2
        P2: process(S_TMP,C)
             begin
             S<=S_TMP xor C after 10 ns;
           end process P2;
end behavior;
```

VERILOG

```
timescale 1 ns/ 1 ns
module SM (A, B, C ,S);
    input  A, B, C;
    output  S;reg S;
//далее следует тело модуля

reg  S_TMP;

//процесс P1
always @(A or B)
begin: P1
  S_TMP<=#15 A ^ B ;
end
//процесс P2
always @(S_TMP or C)
begin:P2
  S<= #10 S_TMP ^ C;
end
endmodule//SM
```

2.1.3. Краткие формы записи процессов

Для определенных, наиболее часто встречающихся типов процессов в HDL предусмотрены краткие (неявные) формы записи.

1. Параллельное (неблокирующее) присваивание

Это процессы с одним изменяемым ими сигналом. Показанный на рис. 2.2 состав процессов можно представить двумя параллельными операторами назначения сигнала (терминология VHDL) или двумя операторами непрерывного непроцедурного присваивания (терминология VERILOG), являющимися краткими формами записи процессов. Порядок их записи, как и процессов, безразличен.

Каждый из этих операторов срабатывает при изменении хотя бы одного из сигналов в своей правой части и каждый имеет как минимум дельта-задержку:

VHDL

```
architecture SM_SHORT of SM is
    signal S_TMP: bit;  -- сигнал S
    begin----процесс P1 ниже
      S_TMP<=A xor B after 15 ns;
          -- процесс P2 ниже
      S<=S_TMP xor C after 10 ns;
end SM_SHORT;
```

VERILOG

```
`timescale 1 ns/ 1ns
 module SM_SHORT (A, B, C, S);
    input  A, B, C;  output  S;
    //по умолчанию S вида wire
    wire S_TMP;
     //процесс P1 ниже
      assign #15  S_TMP=A ^ B ;
     //процесс P2 ниже
     assign  #10 S=S_TMP ^ C;
endmodule// SM_SHORT
```

Отличить параллельное VHDL-назначение сигнала от обычного последовательного можно контекстно: внутри процесса — последовательное, вне — параллельное.

Помимо простой формы существуют **две разновидности оператора параллельного VHDL-назначения сигнала: условное** (when) и **выборочное** (select).

Примеры:

```
S1<=S2 when A=1 else S3;                    -- условное назначение
with A select  S1<=S2 when 1, S1<=S3 when others;-- выборочное
```

В случае, когда нежелательны лишние транзакции в левую часть оператора условного назначения, используется ключевое слово unaffected.

Пример: S1<=S2 when A=1 else unaffected;

2. Параллельные VHDL-операторы: назначения, утверждения, процедуры выражаются через оператор процесса

Пример:

```
параллельное утверждение
assert x=y report "не равны x и y" severity level error;--параллельный
эквивалентно процессу с последовательным утверждением
process (x,y)begin
assert x=y report "не равны x и y" severity level error;--последовательный
end process;
```

На рис. 2.3 представлена временная диаграмма сигналов в объекте SM в предположении, что все они в момент T = 100 ns были равны 0. Изменение A и C в 1 в момент T = 120 ns приводит к запуску процессов P1, P2 и установке S в 1 в T = 130 ns и S_TMP в 1 в T = 135 ns. После этого запускается ожидавший S_TMP процесс P2. S становится равным 0, принимая свое установившееся значение в момент T = 145 ns.

Рис. 2.3. Временная диаграмма сигналов в объекте SM

2.1.4. Присваивание с дельта-задержкой

Определенные особенности связаны с моделированием процессов, не имеющих явных задержек и включающих оператор <=. Предполагается, что **операция <= выполняется с бесконечно малой задержкой — дельта-задержкой** (архитектура SM_NODEL).

VHDL

```
architecture SM_NODEL of SM is
    signal S_TMP: bit;
    begin
        P1:  S_TMP<=A xor B;
        P2:  S<=S_TMP xor C;
    end;
```

VERILOG

```
module SM_NODEL (A, B, C ,S)
 input  A, B, C;
 output  S;reg S,S_TMP;
 always @(A or B or C or S_TMP)
    begin
       S_TMP<=A ^ B;
       S<=S_TMP ^ C;
    end
 endmodule
```

Дельта-задержка бесконечно мала и вводится только для правильного воспроизведения причинно-следственных отношений в модели. При той же временной диаграмме внешних сигналов в модели SM без задержек (рис. 2.4) в момент T = = 120 ns запустятся оба оператора — P1 и P2. В T = 120 ns + Tdelta сигнал S будет равен 1, S_TMP тоже равен 1. В момент T=120 ns+2*Tdelta S примет установившееся значение 0. Если за очень большое число приращений Tdelta, равное, например, 10 000, сигналы в модели не примут установившихся значений, то это означает, что в модели допущена ошибка и моделирование можно прекратить.

Рис. 2.4. Временная диаграмма сигналов в объекте проекта SM для случая моделирования

Отложенные процессы — VHDL

Процессы, описанные с описателем postponed, запускаются соответствующими событиями только после того, как после серии дельта-циклов сигналы в модели принимают установившиеся значения. Это свойство полезно, например, для процессов контроля запрещенных состояний модели (если они отложены, то не срабатывают во время, когда сигналы имеют переходные значения).

2.1.5. Механизм воспроизведения модельного времени

Поведение HDL-объектов, в которых одновременно развивается много процессов, воспроизводится на ЭВМ, и приходится учитывать особенности воспроизведения параллельных процессов на однопроцессорной ЭВМ. Особая роль в син-

хронизации процессов отводится механизму *событийного воспроизведения модельного времени (NOW, $time)*. При этом в памяти моделирующей ЭВМ строится календарь событий, в котором для каждого планируемого в будущем события (event) хранится его время, место и другие параметры. Система моделирования каждый раз выбирает из календаря ближайшее по времени событие и запускает процесс его выполнения.

Выполнение модели состоит из фазы инициализации (в момент модельного времени = 0), за которой следует циклическое выполнение операторов процессов. В начале каждого цикла модельное время становится равным времени ближайшего запланированного события. Процессы, в которых на это время запланированы события, становятся активными.

Когда активные процессы исполняются, их операторы выполняются последовательно, один за другим, и планируют новые события до тех пор, пока каждый из процессов не попадает в свой оператор ожидания (wait) и не становится пассивным. События, как уже отмечалось, связаны с изменениями значений сигналов. Сигналы в VHDL изменяются операторами назначения (параллельного или последовательного — символ «<=»). Ниже рассмотрен механизм реализации событий, используемый в VHDL.

Когда процесс вырабатывает новое (будущее) значение сигнала, в терминологии VHDL это называется *выработкой сообщения (transaction), он формирует пару: время будущего события и будущее значение сигнала*. С одним и тем же сигналом может быть связано множество сообщений. Это множество называется *драйвером сигнала (driver)*. При записи нового сообщения в драйвер возможны конфликты с предыдущими. Эти конфликты будут рассмотрены ниже.

Таким образом, драйвер сигнала — это множество пар: будущее значение сигнала и время (множество планируемых событий в сигнале).

В VHDL, как уже отмечалось, реализуется двухстадийный механизм циклического событийного воспроизведения модельного времени.

На первой стадии событийно наращивается модельное время, изменяются значения всех сигналов, события в которых запланированы на данный момент.

На второй стадии все процессы, которые оказываются чувствительными к этим изменениям, активизируются (запускаются), планируют новые события и исполняются до тех пор, пока не попадают в свои операторы ожидания wait, после чего цикл повторяется.

Другая особенность VHDL-процессов связана с так называемыми разрешенными (resolved) сигналами. Если несколько процессов изменяют один и тот же сигнал (сигнал имеет несколько драйверов), в описании сигнала должна указываться функция разрешения. Функция разрешения объединяет значения из разных драйверов и вырабатывает одно. Это позволяет учитывать, например, особенности работы нескольких элементов на общую шину, что будет рассмотрено ниже. VHDL-переменные никаких драйверов и функций разрешения не имеют.

Механизм воспроизведения модельного времени в VERILOG аналогичен VHDL. Присваивания в переменные и соединения могут быть с задержкой.

Функция разрешения жестко определена видом соединения (например, wor-монтажное ИЛИ). Кроме того, определен порядок проверки возможности начала исполнения процессов — после продвижения модельного времени и изменения сигналов, сначала исполняются процессы, описанные операторами assign, а потом остальные.

Вопросы и упражнения

Почему неверны приведенные ниже примеры генераторов сигналов?

VHDL

```
signal Y: bit;
GEN_WRONG:process (Y)
 begin
  If NOW=0 ns then Y<='0'; end if;
    Wait for 5 ns;
  Y<= NOT Y;
end process GEN_WRONG;
```

VERILOG

```
`timescale 1 ns/1 ns
 reg Y;
 always @(Y)
 begin: GEN_WRONG
   if($time==0)Y<=1'b0;

      Y<= ~Y; #5;
end //GEN_WRONG
```

Ответ:
нельзя использовать оператор wait в процессе со списком чувствительности

В момент, когда Y меняется, процесс еще не достиг своего конца и список чувствительности не реагирует на изменение У, а потом процесс зависает

```
GEN_WRONG2:process
begin
Y<= '1' after 2 ns,'0' after 4 ns;
end
end process
```

```
always
   begin: GEN_WRONG2
  Y<=#2 1; Y<= #4 0;
end
```

Оба процесса не имеют операторов задержки и зациклятся.

2.2. Задержки сигналов

Объект с задержкой можно представить как бы состоящим из двух компонентов — идеального элемента и элемента задержки. Например, модель элемента 2И на рис. 2.5 состоит из идеального вентиля и блока задержки.

Рис. 2.5. Модель элемента 2И, состоящая из идеального элемента и элемента задержки. Вспомогательная переменная Z — идеальный сигнал без задержек

2.2.1. Инерционная и транспортная задержка

В языке VHDL в оператор назначения сигнала <= встроены две модели задержек — **инерционная** (inertial — подразумевается по умолчанию) и **транспортная** (transport).

VERILOG-оператор <= подразумевает инерционную, а транспортную приходится моделировать более сложным способом.

Инерционная модель задержки предполагает, что элемент не реагирует на сигналы, длительность которых меньше порога, равного времени задержки элемента. **Транспортная модель** лишена этого ограничения. Модель транспортной задержки с резекцией (reject) обрезает сигналы, длительность которых меньше времени резекции.

На рис. 2.6 представлены временные диаграммы работы элемента 2И для инерционной (Y) и транспортной (YT) моделей в предположении, что задержка фронта и среза выходного сигнала равны и составляют 10 ns.

Инерционная модель, как отмечалось, по умолчанию встроена в оператор назначения сигнала языка VHDL (можно опускать ключевое слово inertial). Ниже, в примере, оператор назначения описывает работу вентиля 2И и соответствует инерционной модели задержки рис. 2.6.

Рис. 2.6. Временная диаграмма инерционной и транспортной моделей задержки сигнала в вентиле 2И

VHDL	VERILOG
Y<= X1 and X2 after 10 ns;	Y<=#10 X1 & X2;

Следует отметить что в приведенном примере сначала вычисляется значение функции И, потом через задержку оно присваивается!

Если требуется другое — сначала задержать, а потом вычислить и присвоить, — то следовало бы записать так:

wait 10 ns; Y<= X1 and X2;	#10 Y<= X1 & X2;

Указание на использование транспортной модели в VHDL обеспечивается ключевым словом transport в правой части оператора назначения. Например, оператор YT<=transport X1 and X2 after 10 ns; отображает транспортную модель задержки вентиля 2И и иллюстрируется нижней временной диаграммой на рис. 2.6.

В качестве примера приведем варианты VHDL описаний архитектур объекта проекта I2 с инерционной (A1_INERT) и транспортной (A1_TRANSPORT) задержкой, величина которой T.

VHDL

```
entity I2 is
-- параметр настройки T по умолчанию = 10 ns
   generic (T: time:=10 ns);
```

```
    port (X1, X2:in bit; Y:buffer bit);
end I2;
architecture A1_inert of I2 is   -- инерционная
begin
  Y<= X1 and X2 after T;
end A1_inert;
architecture A1_transport of I2 is  -- транспортная
begin
 Y<=transport X1 and X2 after T;
end;
```

Более сложной представляется ситуация, когда необходимо отобразить тот факт, что задержки фронта и среза сигналов не совпадают или зависят от путей прохождения сигналов в схеме и ее предыдущего состояния. Например, вентилю 2И с задержкой фронта 10 ns и задержкой среза 3 ns соответствует диаграммы инерционной (Y) и транспортной (YT) моделей, показанные на рис. 2.7.

Рис. 2.7. Диаграммы сигналов в инерционной и транспортной моделях вентиля 2И при задержках фронта Тф = 10 ns и среза Тс = 3 ns

Один из вариантов описания инерционного поведения вентиля 2И с разными задержками фронта и среза может быть следующим:

VHDL

```
architecture  INERT of I2 is
 begin

 process (X1, X2)
  variable Z:bit;
   begin
   -- выход идеального вентиля - Z
   Z:=X1 and X2;
   if Z='1'and Y='0' then
       Y<='1' after 10 ns;  -- фронт
   elsif Z='0' and Y='1' then
       Y<='0' after 3  ns; -- срез
   end if;
 end process;
end;
```

VERILOG

```
module I2_INERT(X1,X2,Y);
 input X1,X2; output Y;
 reg Y;reg Z;
 always @( X1 or X2)

  begin

    Z=X1  & X2;
   if(( Z==1) &&(Y==0 ))
    Y<=#10  1'b1;
   else if(( Z==0) && (Y==1))
    Y<=#3  1'b0;

  end
endmodule
```

Описание транспортной VHDL-модели аналогично приведенному, за исключением слова TRANSPORT, добавляемого в правую часть операторов назначения сигнала Y. Обратите внимание, что диаграммы инерционной (Y) и транспортной (YT) моделей (см. рис. 2.7) в данном случае совпадают.

Объяснить этот факт можно особенностями реализации механизма учета задержек сигналов в VHDL. Как уже отмечалось, с сигналом ассоциируется драйвер — множество сообщений о планируемых событиях в форме пар: время — значение сигнала.

В случае транспортной задержки, если новое сообщение имеет время большее, чем все ранее запланированные, оно просто включается в драйвер последним. В противном случае предварительно уничтожаются все сообщения, запланированные на большее время.

В случае инерциальной задержки также происходит уничтожение всех сообщений, запланированных на большее время. Однако разница в том, что происходит анализ событий, запланированных на меньшее время, и если значение сигнала в них отличается от нового, то они уничтожаются. Возвратимся к рис. 2.5.

Для инерционной (архитектура A1_inert) и транспортной (архитектура A1_transport) моделей вентиля I2 и диаграммы входных сигналов (см. рис. 2.5) имеем такие состояния драйверов сигналов Y и YT, которые представлены на рис. 2.8.

Время	Драйвер Y		Драйвер YT			
15	1	25	1	25		
20	1	25	1	25		
30	0	40	0	40		
35	1	45	0	40	1	45
40	0	50	1	45	0	50

Рис. 2.8. Состояние драйверов сигналов Y и YT

2.2.2. Резекция и неопределенность коротких сигналов

Определенные проблемы возникают при моделировании ситуаций, когда длительность входных сигналов меньше задержки элемента.

Можно игнорировать сигналы, длительность которых меньше определенного порога (резекция), можно распространять неопределенное значение на выход на это время.

VHDL

Инерционная модель не пропускает на выход импульсы, длительность которых меньше времени задержки. Обойти это ограничение можно с помощью опции reject и транспортной модели задержки.

Пример резекции сигнала длительностью менее 1 ns (более длительные импульсы проходят на выход)

```
YT<=reject 1 ns transport X1 and X2 after 10 ns;
```

Этот оператор эквивалентен двум:

```
TMP<=X1 and X2 after 1 ns;-- фильтр-резектор (инерциальная задержка)
YT<= transport TMP after 9 ns;-- задержка (транспортная)
```

VERILOG

VERILOG имеет более развитые встроенные средства, позволяющие осуществлять не резекцию, а передачу на выход неопределенного значения X в случае коротких входных импульсов. Это обеспечивается специальными средствами описания задержек на путях сигналов, используемыми в блоке specify.

Пример описания задержки фронта =5 и среза =8 в повторителе, реализующем функцию a<=b:

```
module p(b,a); input[3:0]b; output[3:0]a; reg[3:0]a;
specify
    (b=>a)=(5,8)
endspecify
```

При подаче в момент T на такой элемент сигнала длительностью 2 единицы модельного времени, на выходе a будет неопределенное значение X в период времени с T + 5 до T + 8.

VERILOG-2000 имеет средства описания еще более пессимистического короткоимпульсного поведения с помощью оператора установки режима pulsestyle_ondetect_out.

```
specify  pulsestyle_ondetect_out;
    (b=>a)=(5,8)
endspecify
```

В этом случае неопределенность в a наступает в момент времени T и длится до T + 8.

Блок specify позволяет описывать еще более сложные ситуации с задержками и проверками временных соотношений сигналов — они частично рассмотрены в других главах. Аналогичные средства VHDL реализованы в IEEE-пакетах Vital_timing и Vital_primitives.

2.3. Векторные операции и компактность описаний систем

2.3.1. Векторы

Одним из средств повышения компактности описаний и скорости моделирования цифровых устройств является использование векторных представлений сигналов и операций над ними. Например, пусть некоторый объект проекта M2_V выполняет ту же функцию, что и объект проекта M2 (см. глава 1), но над двадцатиразрядными двоичными векторами X1 и X2 (рис. 2.9).

Описание интерфейса определяет порты как двоичные векторы, а в потоковой форме описания архитектуры используются операции над векторами:

VHDL

```
entity M2_V  is
port (X1, X2:in bit_vector (1 to 20);
  Y:out bit_vector (1 to 20));
```

VERILOG

```
module M2_V(X1,X2,Y);
  input [1:20] X1,X2;
  output [1:20] Y;
```

Рис. 2.9. Многоразрядная версия SM

```
end M2_V;
architecture BEH of M2_V is
  begin
  Y<=  (X1 xor X2);                    assign Y=  (X1 ^ X2);
end BEH;                            endmodule //M2_V
```

Как VHDL, так и VERILOG содержит средства работы с полями векторов. Пример:

VHDL	VERILOG
Signal A,B: bit_vector(100 downto 0);	reg [100:0]A,B;
B(30 downto 10)<=A(40 downto 20);	B[30:10]=A[40:20];
B(I downto J)<= A (I+2 downto J+2);	

VERILOG в векторных операциях не допускает переменных границ полей, в данном примере необходимо организовать поразрядный цикл:

```
for (k=I;k<=j;k=k-1) B[k]=A[k+2];
```

Зато он допускает конкатенацию в левой части оператора присваивания:
{C,SM}=A+B;//перенос идет в C

2.3.2. Оператор генерации

Использование оператора генерации позволяет повысить компактность HDL-описаний.

Структурное описание архитектуры для варианта реализации объекта проекта M2_V как совокупности однобитовых объектов проекта M2 (рис. 2.10), представленное ниже, выполнено с его помощью.

VHDL	VERILOG//2000
architecture STRUCT_M2_V of M2_V is	module STRUCT_M2_V (X1,X2,Y);
component M2 port (X1, X2:in bit;Y:out bit);	input [1:20]X1,X2; output [1:20]Y;
end component;	genvar I;//I для генерации
begin	
K1: M2 port map (X1(1),X2(1),Y(1));	M2 K1 (X1[1],X2[1],Y[1]);

Рис. 2.10. Многоразрядная версия M2

```
K2:M2 port map(X1=>X1(2),
               Y=>Y(2),
               X2=>X2(2)
               );

GEN:for I in 3 to 20 generate
    K:M2 port map (X1(I),X2(I),Y(I));
end generate;
end STRUCT_M2_V;
```

```
M2 K2  (.X1(X1[2]),
  .Y(Y[2]),
  .X2(X2[2])
  );
generate
 for (I=3; I<=20; I=I+1)begin:M2_V
 M2 K (X1[I],X2[I],Y[I]);
 endgenerate
endmodule // STRUCT_M2_V
```

Первый компонент конкретизирован обычным способом с использованием позиционного соответствия сигналов портам.

Второй компонент конкретизирован с использованием ключевого указания соответствия сигналов ее портам (сопоставление).

Компоненты K3—K20 конкретизированы с использованием оператора генерации, позволяющего компактно описывать регулярные фрагменты схем.

После генерации каждая конкретизация компонента M2 будет иметь индивидуальное имя — например, для нашего VERILOG-примера это будет имя, включающее имя блока генерации и значение переменной генерации — M2_V[3].K, M2_V[4].K и т. д.

2.4. Алфавит моделирования

Приведенные описания объекта M2 базировались на стандартных средствах языка HDL — сигналы представлялись в алфавите «1», «0», логические операции И, ИЛИ, НЕ также определялись в этом алфавите и т. д.

Во многих случаях приходится описывать поведение объектов в других алфавитах. Например, в реальных схемах сигнал, кроме значений «1» и «0», может принимать значение высокого импеданса — «Z» (на выходе буферных элементов). Можно использовать неопределенное состояние — «X», например отражая неизвестное начальное состояние триггеров.

2.4.1. Четырехзначный алфавит

Наиболее простое представление об уровне сигнала дает двоичной (булевский) алфавит, в котором сигнал может иметь истинное (TRUE, 1) или ложное (FALSE, 0) значение. В VHDL — это данные типа булевский (BOOLEAN) и битовый (BIT).

Однако для практики это слишком грубое представление, и в VERILOG, как уже отмечалось, используется четырехзначный алфавит с дополнительными значениями:

z — высокий импеданс (например входной порт «висит» в воздухе), обозначается как 1'bz;

x — неопределенное значение (например, выход неустановленного триггера), обозначается как 1'bx.

В VHDL пакет STD_LOGIC_1164 позволяет использовать девятизначный алфавит и, в частности, включает подтип X01Z с четырехзначным алфавитом, аналогичным четырехзначному алфавиту VERILOG.

Таблица истинности логических функций И, ИЛИ, Исключающее ИЛИ, НЕ в четырехзначном алфавите представлены ниже (в функции И доминирует 0, а при его отсутствии в значениях аргументов — X, в ИЛИ — 1 и X). Значение Z аргумента воспринимается как X.

Таблица истинности логических функций в четырехзначном алфавите

Аргументы	Значение функции			
	И (AND, &)	ИЛИ (OR, ||)	XOR	НЕ (NOT, ~)
0 0	0	0	0	—
0 1	0	1	1	—
1 0	0	1	1	—
1 1	1	1	0	—
0	—	—	—	1
1	—	—	—	0
0 Z	0	X	X	—
Z 0	0	X	X	—
1 Z	X	1	X	—
Z 1	X	1	X	—
Z Z	X	X	X	—
0 X	0	X	X	—
X 0	0	X	X	—
1 X	X	1	X	—
X 1	X	1	X	—
X X	X	X	X	—
Z X	X	X	X	—
X Z	X	X	X	—
Z				X
X				X

Нет необходимости помнить громоздкие таблицы многозначных алфавитов. Можно использовать принцип мажорирования: в функции И мажорирует 0 (если один из аргументов = 0, то и результат = 0), в функции ИЛИ — 1. Если в аргументах функции И нет 0 (а в ИЛИ нет 1), но есть X или Z, то они мажорируют и результат = X.

Пример:

VHDL(std_logic)	VERILOG	Результат
"X01Z" AND "XXZZ"	4'bX01Z & 4'bXXZ1	"X0XX" (4'bX0XX)

2.4.2. Девятизначный алфавит VHDL

Для еще более точного представления сигналов можно либо вводить понятие силы сигнала, как это делается в VERILOG (см. приложение 2), либо расширять значность алфавита: как уже отмечалось, стандарт расширения VHDL (IEEE пакет STD_LOGIC_1164) использует девятизначный алфавит со следующими значениями:

— «U» — неинициализированное;
— «X» — сильная неопределенность;
— «W» — слабая неопределенность;
— «0» — сильный 0;
— «L» — слабый 0;
— «1» — сильная 1;
— «H» — слабая 1;
— «Z» — высокий импеданс;
— «~» — безразлично.

Операции AND и OR в этом алфавите нетрудно представить, используя идею мажорирования в алфавите 0, Z, X, 1 и объединения подмножеств:

«~», «X», «W» в «X»;
«L», «0» в «0»;
«H», «1» в «1».

Пример:
"U01XWLHZ" AND "U~0UXLHZ" равно "U00XX01X"

Ниже дан фрагмент текста пакета STD_LOGIC_1164 (введены типы: std_ulogic, его подтип с функцией разрешения STD_LOGIC и другие подтипы). Переопределены операции И, ИЛИ и т. п. для новых типов. Представлены тела трех функций: resolved, and, not. Текст пакета состоит из двух частей — описание пакета (описание интерфейса пакета) и описание тела пакета.

```
use STD.textio.all;
PACKAGE std_logic_1164 IS
--новые типы и подтипы-------------------------
TYPE std_ulogic IS ('U', 'X','0','1','Z','W','L','H' ,'-');
TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <> ) OF std_ulogic;
FUNCTION resolved (s : std_ulogic_vector ) RETURN std_ulogic;
SUBTYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;
SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1';
SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z';
SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1';
```

```
SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z';
-- переопределенные операции-----------------------
FUNCTION "and" (l : std_ulogic; r : std_ulogic) RETURN UX01;
FUNCTION "not" (l : std_ulogic ) RETURN UX01;
FUNCTION "and" (l, r : std_logic_vector) RETURN std_logic_vector;
FUNCTION "and" (l, r : std_ulogic_vector) RETURN std_ulogic_vector;
FUNCTION "not" (l : std_logic_vector) RETURN std_logic_vector;
FUNCTION "not" (l : std_ulogic_vector) RETURN std_ulogic_vector;

-- функции преобразования типа-----------------------
FUNCTION To_bit (s : std_ulogic; xmap : BIT := '0') RETURN BIT;
FUNCTION To_bitvector (s : std_logic_vector ; xmap : BIT := '0')
RETURN BIT_VECTOR;
FUNCTION To_bitvector (s : std_ulogic_vector; xmap : BIT := '0')
RETURN BIT_VECTOR;
FUNCTION To_X01Z (s : std_logic_vector) RETURN std_logic_vector;
FUNCTION To_X01Z (s : std_ulogic_vector) RETURN std_ulogic_vector;
 FUNCTION To_X01Z (s : std_ulogic) RETURN X01Z;
FUNCTION To_X01Z (b : BIT_VECTOR) RETURN std_logic_vector;
FUNCTION To_X01Z (b : BIT_VECTOR) RETURN std_ulogic_vector;
 FUNCTION To_X01Z (b : BIT) RETURN X01Z;
END std_logic_1164;
--НИЖЕ ФРАГМЕНТ ТЕЛА ПАКЕТА----------------------------
PACKAGE BODY std_logic_1164 IS
TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

--ниже в комментариях значения аргументов, а в таблице - функции
 CONSTANT resolution_table : stdlogic_table :=
--  'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
(('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), --U
 ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), --X
 ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), --0
 ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), --1
 ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), --Z
 ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), --W
 ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), --L
 ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'),--H
 ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'));--~
FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
  VARIABLE result : std_ulogic := 'Z';
   BEGIN IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
       ELSE FOR i IN s'RANGE
           LOOP
            result := resolution_table(result, s(i));
           END LOOP;
       END IF;
   RETURN result;
 END resolved;

--ниже в комментариях значения аргументов, а в таблице - функции
 CONSTANT and_table : stdlogic_table :=
  --  'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
   (('U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U'),
    ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),
```

```
    ('0', '0', '0', '0', '0', '0', '0', '0', '0'),
    ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'),
    ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),
    ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'),
    ('0', '0', '0', '0', '0', '0', '0', '0', '0'),
    ('U', 'X', '0', '1', 'X', 'X', '0', '1', 'X'),
    ('U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X'));
  CONSTANT not_table: stdlogic_1d :=
   -- 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
    ('U', 'X', '1', '0', 'X', 'X', '1', '0', 'X');
FUNCTION "and" (l : std_ulogic; r : std_ulogic) RETURN UX01 IS
BEGIN RETURN (and_table(l, r)); END "and";
```

Пример использования пакета STD_LOGIC_1164 при описании объекта F_P4:

```
Library IEEE;
use IEEE.STD_LOGIC_1164.ALL; -- все его функции (ALL)
entity F_P4 is
port (A1, A2:in std_logic; B1, B2:out  std_logic);
end F_P4;

architecture F_flow of F_P4_ is
    begin    -- функции not и and можно употреблять как операции
      B2<=not (A1 and A2);          -- но можно и как функции
      B1<=A1 and A2;                -- например, B1<=and (A1, A2);
end F_flow;
```

Начальное значение переменных перечислимого типа равно по умолчанию первому элементу списка. В данном примере — «U».

2.4.3. X-пессимизм и оптимизм

Использование многозначного алфавита — необходимое, но недостаточное средство повышения точности описаний объекта проекта. Сравните два варианта фрагментов описаний поведения вентиля И.

VHDL

```
    Library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
 signal Y,X1,X2: std_logic;
        Y<=X1 and X2;   --a)

        if(X1 and X2)='1' then  --б)
        Y<='1';
         else Y<='0';
        end if;
```

VERILOG

```
reg Y,X1,X2;
 Y<=X1 & X2; // a)

if(X1 && X2) // б)
Y<=1;
else    Y<=0;
```

Если первый вариант (а) пропускает неопределенное значение ('X' или 1'bx) на выход Y, то второй (б) устанавливает Y либо в 1, либо в 0 и например (VHDL) при X1='1' и X2= 'U' выход Y будет равен '0'. В данном случае имеем дело с X-оптимизмом [20]. Оптимизм заключается в иногда неоправданном превращении неопределенных значений сигналов (X) в определенные.

X-пессимизм [20] арифметических операции иллюстрируется ниже:

В языке VERILOG возможностью разрешения конфликтов обладают данные вида соединение. Например, связь, объявленная как wire или tri, принимает значение одного из источников, только если другие имеют значение Z. В противном случае ее значение X.

Ниже пример описания общей шины:

VHDL

```
signal D:std_logic;
 D<=not(X1 and X2) when
       E1='1' else 'Z';
 D<=not(X3 and X4) when
       E2='1' else 'Z';
```

VERILOG

```
wire D;
assign D=( E1==1)?~( X1 & X2):
                          1'bz;
assign D=( E2==1)?~( X3 & X4):
                          1'bz;
```

2.5.2. Монтажное И, ИЛИ

Ниже рассмотрен пример монтажного ИЛИ.

Описание выполнено в стиле DATA FLOW с применением кратких форм записи процессов.

VERILOG — монтажное ИЛИ реализуется соединением, объявленным как соединение типа wor, монтажное И — как wand.

VHDL — объявление сигнала должно включать имя соответствующей функции разрешения или соответствующего подтипа.

VHDL

```
subtype WO is Wired_OR bit;
signal Y:WO;
Y<=X1 and X2;
Y<=X3 and X4;
```

VERILOG

```
wor Y;
 assign Y=X1 & X2;
 assign  Y=X3  & X4;
```

Тело VHDL-функции разрешения WIRED_OR (монтажное ИЛИ, рис. 2.12), например, может быть таким:

```
function WIRED_OR (INPUTS:bit_vector) return bit is
  variable X:bit:='0';
    begin
      for I in INPUTS'RANGE loop
          if INPUTS(I)='1' then X:='1'; exit; end if;
    end loop;
          return X;
end;
```

Драйверы сигнала INPUTS неявно рассматриваются как битовый массив, границы которого определяются атрибутом 'RANGE.

Функция WIRED_OR сканирует драйверы сигнала и, если хоть один из них равен '1', возвращает значение '1', иначе — '0'.

VHDL	VERILOG
Library IEEE;	
use IEEE.STD_LOGIC_1164.ALL;	
use IEEE.STD_LOGIC_arith.ALL;	
use IEEE.STD_LOGIC_UNSIGNED.ALL;	
----------	
signal Y,X1,X2: std_logic_vector(0 to 3);	reg [0:3]Y,X1,X2;
X1<="0001"; X2<="000X";	X1<=4'b0001; X2<=4'b000X;
Y<=X1 + X2;	Y<=X1 + X2;

-- ("0001" + "000X" дает "XXXX", а не "00XX" - т.е. имеет место X-пессимизм).

Пессимизм заключается в изменении предосторожности — лишних X значений сигналов.

2.5. Описание монтажных И (ИЛИ) и общей шины

В цифровой аппаратуре используются общие шины на элементах с тремя состояниями выхода (рис. 2.11) и монтажные ИЛИ (И) (рис. 2.12).

Если каждому компоненту (K1, K2) схемы сопоставить процесс, то для схем, представленных на рис. 2.11 и 2.12, имеем два параллельных процесса, каждый из которых вырабатывает свой выходной сигнал (D — на рис. 2.11, Y — на рис. 2.12) и создает для него соответствующий драйвер.

При этом возможен конфликт, когда общий сигнал принимает значение X.

Рис. 2.11. Общая шина на элементах с трехстабильными выходами

Рис. 2.12. Монтажное ИЛИ

2.5.1. Общая шина

В языке VHDL предусмотрен механизм разрешения конфликтов, возможных в подобных ситуациях, когда сигнал имеет несколько драйверов. Этим механизмом является функция разрешения сигнала. Функция разрешения обычно описывается в пакете (например, в пакете std_logic_1164 ее имя resolved). Имя функции разрешения указывается при описании типа соответствующего сигнала (см. в std_logic_1164 описание типа std_logic) или непосредственно в описании данных, тип которых не имеет этой функции.

Представление о предопределенных атрибутах сигналов можно получить из временной диаграммы для атрибутов и функций битового сигнала S, представленной на рис. 2.14. На рис. 2.14 иллюстрируется оператор назначения сигнала S:

S<='1' after 0 ns, '0' after 30 ns, '1' after 50 ns; в предположении, что до этого он был равен 0.

В сигнале S оператором назначения запланированы активности в моменты NOW, NOW + 30ns, NOW + 50ns. Сигнал становится активным, когда изменяет свое значение или, когда его значение не изменяется, но было запланировано подтверждение, например, в моменты 30 и 50:

S<='1' after 0 ns, '1' after 30 ns, '1' after 50 ns.

При описании объектов, срабатывающих от фронтов сигналов, как уже отмечалось, в VHDL часто используется атрибут 'event, а в VERILOG — условия событий posedge и negedge

	VHDL	VERILOG
условие фронта сигнала S	S'event and S='1' not S'stable and S='1'	@(posedge S)
условие среза	S'event and S='0' not S'stable and S='0'	@(negedge S)

Рис. 2.14. Временная диаграмма, поясняющая атрибуты сигнала S

2.6. Атрибуты объектов и контроль запрещенных ситуаций

2.6.1. Контроль запрещенных ситуаций

Описания систем могут содержать информацию о запрещенных ситуациях, например, недопустимых комбинациях сигналов на входах объектов, рекомендуемых длительностях или частотах импульсов и т. п. Например, в вентиле 2И возникает **риск сбоя** в ситуациях, представленных на рис. 2.13, когда фронт одного сигнала перекрывает срез другого. Термин риск сбоя относится не к самому вентилю, а к схеме, где он используется — короткий импульс может появиться и изменить значение триггера в одних сочетаниях задержек и не изменить в других.

Рис. 2.13. Риск сбоя в единицу

Входные сигналы X1 и X2 изменяются в противоположном направлении и время их перекрытия меньше необходимого, допустим, равного 10 ns.

Средством отображения информации о запрещенных ситуациях в языке VHDL является оператор утверждения assert. В нем, помимо контролируемого условия, которое не должно быть нарушено, т. е. должно быть истинным, записывается сообщение (report) о нарушении и уровень серьезности ошибки (severity).

Для приведенного на рис. 2.13 примера в описание архитектуры вентиля 2И может быть вставлено утверждение о том, что все будет нормально, если инерционный сигнал Zi совпадает с транспортным (Zt).

VHDL

```
architecture C1 of I2 is
   signal Zi, Zt:bit;
    begin
      Zi<=X1 and X2 after 10 ns;
      Zt<=transport X1 and X2 after 10 ns;
        assert Zi=Zt report "риск сбоя в 1 в вентиле I2" severity warning;
      Y<=Zi;
   end C1;
```

2.6.2. Атрибуты VHDL-сигналов

Атрибуты, ассоциируемые с объектами VHDL, позволяют более полно отображать их свойства. Полный перечень атрибутов объектов приведен в приложении 1, ниже даны атрибуты сигналов. Атрибуты могут объявляться.

Например, attribute pin_no: positive;

Кроме того, существуют предопределенные атрибуты.

Пример процесса реагирующего на фронт сигнала X печатью слова FRONT·

VHDL	VERILOG
process (X) begin if X'event and x='1' then report "FRONT", end if, end process;	always @(posedge X) $display ("FRONT"),

Вопросы к главе 2

1. Перечислите параллельные операторы VHDL и VERILOG
2. Перечислите краткие формы оператора процесса VHDL и VERILOG.
3. Почему ошибочно нижеследующее описание вентиля И-2 (and2)?

VHDL	VERILOG
process begin Y<= X1 and X2 after 10 ns; end process;	always Y <= #10 X1 & X2,

Ответ — процессы без операторов задержек и списка чувствительности.

4. Почему не будут различаться результаты выполнения а) и в) двух параллельных процессов при Y1 = 0.

VHDL	VERILOG
a) Y1<=1; Y2<=Y1+1; в) Y2<=Y1+1, Y1<=1;	a) assign Y1=1, assign Y2=Y1+1, b) assign Y2=Y1+1, assign Y1=1,

и будут различаться для внутренних фрагментов процессов с) и d), содержащих последовательные операторы присваивания (переменным (variable — VHDL), процедурные блокирующие в reg — VERILOG)

c) Y1:=1;
Y2:=Y1+1;
d) Y2.=Y1+1;
Y1:=1;
	c) Y1=1;
	Y2=Y1+1;
	d) Y2=Y1+1,
	Y1=1,

5. Почему будут различаться результаты выполнения а) и в) двух фрагментов процессов при начальном Y1=0?

VHDL	VERILOG

a) Y1<=1;
wait 10 ns;
Y2<=Y1+1;
b) Y1<=1 after 10 ns;
Y2<=Y1+1;
	a) Y1=1,
	#10,
	Y2=Y1+1,
	b) Y1<=#10 1,
	Y2<=Y1+1;

Ответ — в случае а) Y1 успеет измениться к моменту присваивания в Y2, в случае б) — нет.

6. Вопрос «на засыпку» — так чем же отличаются переменные от сигналов (соединений) и зачем все это нужно?

7. Зачем нужна дельта-задержка в операторах присваивания?

8. Как отличить оператор последовательного назначения (VHDL) сигнала от параллельного?

Глава 3
Способы HDL-описаний простых узлов

3.1. Комбинационная схема F

Для иллюстрации возможностей HDL рассмотрим пример проектирования простой комбинационной схемы, назовем ее объектом проекта F. Объект проекта F имеет два входа — A1 и A2 и два выхода — B1 и B2. Его условное графическое изображение дано на рис. 3.1,*а*. Объект проекта должен реализовать логическую функцию, представленную на рис. 3.1,*б*. Отметьте, что логическая функция отражает работу проектируемого объекта в двоичном (булевском) алфавите значений 0,1.

3.1.1. Описание интерфейса

Описание интерфейса объекта проекта F (объявление объекта проекта) на языках VHDL и VERILOG может быть таким:

VHDL

```
entity F is
  port (A1, A2:in bit; B1, B2: out bit);
end F;
```

VERILOG

```
module F (A1, A2,B1, B2);
input A1,A2;
output B1,B2; reg B1,B2;
```

VERILOG 2000 позволяет объявлять интерфейс более компактно:

```
module F( input A1,A2,output reg B1,B2);
```

Помимо имени объекта проекта — F в описании перечисляются его входные порты A1, A2 и выходные порты B1, B2. Для портов указывается направление и тип сигналов, которые могут поступать на них — bit, т. е. сигналы могут принимать значение '0' или '1' в VHDL и 1'b0, 1'b1, 1'bx, 1'bz в VERILOG. Если VHDL подразумевает для всех портов только вид «сигнал», то VERILOG допускает для выходных портов виды: соединение (wire) и переменная (reg).

а)

Входы		Выходы	
A1	A2	B1	B2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

б)

Рис. 3.1. Объект проекта F: а — условное графическое изображение объекта F; б — реализуемая им функция

3.1.2. Процессная форма описания поведения

В начале процесса проектирования объект проекта F можно представить как «черный ящик», реализующий заданную таблицей (рис. 3.1, *б*) функцию: B1, B2 = F(A1, A2).

Приведенный ниже вариант описания архитектуры BEHAVIOR объекта проекта F учитывает, что только при обоих входах A1 и A2, равных 1, выходы принимают значение B1 = 1 и B2 = 0 (последняя строка таблицы, рис. 3.1,*б*). В остальных случаях, наоборот (первые три строки таблицы, рис. 3.1,*б*), B1 = 0, B2 = 1.

VHDL

```
architecture F_BEHAVIOR of F is
  begin
process
   begin
      if (A1='1') and (A2='1')
         then B1<='1'; B2<='0';
         else B1<='0'; B2<='1';
      end if;
    wait on A1, A2;
end process;
 end F_BEHAVIOR;
```

VERILOG

```
//заголовок был выше

always
  begin
 if ((A1==1) && (A2==1)) begin
     B1<=1; B2<=0; end
 else begin B1<=0; B2<=1; end

   @(A1 or A2);
  end //always
endmodule //F
```

Для иллюстрации большого разнообразия форм записи процессов в HDL выше приведено описание процесса без списка чувствительности. Этот HDL-текст надо понимать следующим образом. После заголовка, содержащего имя архитектуры (BEHAVIOR), следует ее тело, являющееся процессом (process, always). Процесс, находясь в операторе ожидания wait on — VHDL @()-VERILOG), ожидает, пока не изменится хотя бы один из сигналов (A1 или A2), указанных в списке оператора ожидания. Как только это произойдет, он, дойдя до оператора конца описания процесса (end process в VHDL), вернется к своему началу. Выполнится условный оператор if, который в зависимости от истинности условия передает управление той или другой группе операторов назначения (<=) выходным сигналам B1 и B2. В итоге процесс снова будет ожидать в операторе ожидания. Вместо однобитовых констант 1'b1 и 1'b0 для краткости в VERILOG-примере использованы эквивалентные 1 и 0.

Второй вариант поведенческого описания архитектуры объекта проекта F, назовем его F_CASE, использует оператор выбора (CASE) HDL и учитывает то свойство функции F, что для первых трех строк ее значение не меняется. В HDL-описании в заголовке процесса дан список чувствительности процесса (A1, A2). Это указание эквивалентно оператору ожидания событий в A1, A2 (см. предыдущий пример) в конце описания процесса. VHDL-процесс со списком чувствительности не может содержать оператор wait. Ниже приведен текст второго варианта (для VERILOG — синтаксис VERILOG-2000):

VHDL

```
architecture F_CASE of F is
 begin
   process (A1, A2)
   variable EE:bit_vector(0 to 1);
    begin -- & - конкатенация
      EE:=A1& A2;
     case EE is
```

VERILOG-2000

```
module F_CASE(input wire A1,A2,
                 output reg B1,B2);
  always @(A1 , A2) // V-2000

    begin //{} конкатенация

      case ({A1, A2} )
```

```
        when "11" => B1<='1'; B2<='0';
        when others => B1<='0'; B2<='1';

    end case;
  end process;
end F_CASE;
```

```
    2'b11: begin B1=1; B2=0;end
    default: begin B1=0;
                   B2=1;end
  endcase
end
endmodule //F_CASE
```

Переменная EE введена в VHDL-пример, так как оператор case требует статически определенной длины выражения выбора.

Для проверки результатов этого этапа проектирования требуется произвести модельный эксперимент. Как это делается, см. в следующей главе.

3.1.3. Потоковое описание поведения

Наряду с вышеприведенной процессной формой для описания поведения объекта проекта F может использоваться потоковая форма, являющаяся краткой формой записи процессов с одним изменяемым сигналом.

VHDL

```
architecture F_DATAFLOW of F is
  begin
      B1<='1' when (A1 & A2)="11"
              else '0';
     B2<='0' when (A1 & A2)="11"
              else '1';
end F_DATAFLOW;
```

VERILOG

```
module F_DATA(A1,A2,B1,B2);
input A1,A2;output B1,B2;
  assign   B1=(({A1, A2} ) ==2'b11)? 1:0;

  assign   B2=(({A1, A2} ) ==2'b11)? 0:1;

endmodule //F_DATAFLOW
```

Обратите внимание на вид портов (wire по умолчанию) в этом варианте VERILOG-модели.

В процессе проектирования объекта проекта F могут быть предложены различные варианты его функциональных схем (на базе программируемых логических матриц (ПЛМ); в булевском базисе И, ИЛИ, НЕ; на элементах ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR), И-НЕ и т. п.) с учетом различных критериев (минимизации оборудования, времени срабатывания, регулярности структуры, потребляемой мощности и т. п.). Некоторые из вариантов схем даны на рис. 3.2. Например, схема, представленная на рис. 3.2,*а*, при прочих равных условиях быстрее схемы, представленной на рис. 3.2,*с*. (Подробнее о синтезе см. главу 5).

Потоковый стиль описания аппаратуры предполагает представление процесса работы объекта в виде последовательности параллельных операторов преобразования информации на регистрах.

Описание архитектуры объекта проекта F, соответствующее варианту схемы, представленной на рис. 3.2,*а* (назовем его F_A), может быть таким:

VHDL

```
architecture F_A of F is
  begin
    B1<= A1 and A2;
    B2<= not(A1 and A2);
  end  F_A;
```

VERILOG

```
module F_A(A1,A2,B1,B2);
      input A1,A2;output B1,B2;
      assign  B1= A1 & A2;
      assign  B2= !(A1 & A2);
endmodule //F_A
```

Рис. 3.2. Варианты аппаратной реализации объекта проекта F

Другой вариант описания архитектуры (F_C) использует вариант оператора условного назначения сигналу (VHDL) или оператор assign и условное выражение (VERILOG).

VHDL

```
architecture F_C of F is
   begin
   B1<='1' when (A1 and A2)='1' else '0';
   B2<='1' when not(A1 and A2)='1' else '0';
end F_C;
```

VERILOG

```
module F_C(A1,A2,B1,B2);
input A1,A2;output B1,B2;
assign  B1=(A1 && A2)?1:0;
assign  B2=!(A1 && A2)?1:0;
endmodule  //F_C
```

В этом варианте описания каждому элементу схемы сопоставлен процесс, отображающий последовательность преобразования входной информации и передачи ее на выход. Процесс представлен в форме оператора параллельного назначения сигнала (VHDL) или непрерывного присваивания — continuous assignement (VERILOG). Кроме того, можно заметить, что все VERILOG-версии потоковых описаний имеют одинаковый список портов, и можно выделить этот фрагмент в файл (допустим, под именем F_INTERFACE.v) и включать его директивой `include.

В этом файле текст

```
(A1,A2,B1,B2);  input A1,A2;output B1,B2;
```

Параллельные HDL-операторы назначения сигнала (в VHDL это <= вне process, в VERILOG это присваивание (assign и =) в данные вида соединение) срабатывают при изменении хотя бы одного из сигналов в своих правых частях.

Варианту, представленному на рис. 3.2,*b*, соответствует другая архитектура объекта проекта F, назовем ее F_B.

В схеме, представленной на рис. 3.2,*b*, элементы включены последовательно. Это отражено и в описании архитектуры F_B.

VHDL

```
architecture F_B of F is
  signal X:bit; -- внутренний сигнал X
   begin
     B2<= not(X); -- параллельные
      X<= A1 and A2; -- назначения
      B1<=X;
end F_B;
```

VERILOG

```
module F_B(A1,A2,B1,B2);
   input A1,A2;output B1,B2;
   wire X;
   assign B2= !(X);
   assign X= A1 & A2;
   assign B1=X;
endmodule //F_B
```

Сигнал B2 вырабатывается только после изменения сигнала X. Оператор B2<=not(X); сработает только тогда, когда изменится сигнал X, т. е. после оператора X<=A1 and A2, так как он реагирует только на изменение сигнала в своей правой части. В данном случае это произойдет через бесконечно малую дельта-задержку, точнее, через две дельта-задержки: сначала изменится X, потом B1 и B2.

Промежуточный сигнал X введен в VHDL-описание архитектуры F_B объекта F в связи с тем, что в описании интерфейса объекта проекта F порт B1 объявлен выходным (out), т. е. с него нельзя считывать сигнал и запись B2<=not(B1) была бы некорректной (если бы его объявить как buffer или inout, промежуточныйй сигнал X был бы не нужен). Избавиться от него позволяет также применение VHDL-атрибута 'driving_value в процессе (B1' driving_value).

VHDL

```
entity F_BUF is
  port (A1, A2:in bit; B1, B2: buffer bit);
end F_BUF;
architecture F_B1 of F is
  begin
    B2<= not(B1);
     B1<= A1 and A2;
end F_B1;
```

VERILOG

```
module F_B1(A1,A2,B1,B2);
  input A1,A2;output B1,B2;
     assign B2= !(B1);
     assign B1= A1 & A2;
endmodule //F_B1
```

Как отмечалось, HDL позволяет в явной форме указывать задержку сигналов. Например, если в схеме, представленной на рис. 3.2,*b*, задержка в элементе E1 равна 1.0 ns, а в E2 равна 0.5 ns, описание архитектуры F может быть таким (назовем его F_B_TIME):

VHDL

```
architecture F_B_TIME of F is
 signal X:bit;
  begin
    -- задержка на E1 - 1.0 ns,
    -- на E2 - 0.5 ns
    E1:B1<=X;
    E2:B2<= not(X) after 0.5 ns;
    X<= A1 and  A2  after  1.0 ns;
end F_B_TIME;
```

VERILOG

```
`timescale 1 ns/ 100 ps
module F_B_TIME (A1,A2,B1,B2);
    input A1,A2;output B1,B2;
    wire X;
   // задержка на E1 - 1.0 ns,
   // на E2 - 0.5 ns
    assign  B1=X;
    assign #0.5 B2=!(X);
    assign #1.0 X= A1 & A2;
endmodule   // F_B_TIME
```

Через 1.0 ns после изменения одного из входных сигналов (A1 или A2) может измениться выходной сигнал B1 и с задержкой 0.5 ns после него — измениться B2.

3.1.4. Структурное описание

Структурное описание архитектуры представляет структуру объекта как композицию из компонентов, соединенных между собой и обменивающихся сигналами. Функции, реализуемые компонентами, в явном виде, в отличие от предыдущих примеров, в структурном описании не указываются. Структурное описание включает имена и типы компонентов, из которых состоит схема, и их связи. Полные (интерфейс + архитектура) описания объектов, сопоставленных компонентам, должны быть ранее помещены в проектную библиотеку, подключенную к структурному описанию архитектуры.

Например, структурное описание архитектуры объекта проекта F как варианта схемы, представленного на рис. 3.2, *c*, может быть таким:

VHDL

```
architecture STRUCT_F_C of F is
    component INE2   -- интерфейс
        port  (X1, X2:in bit; Y:out bit);
    end component;
    signal X: bit;   -- вспомогательный
begin -- ниже описание связей  компонент
  E1:INE2 port map (A1, A2, X);  - - E1
  E2:INE2 port map (X, X, B1);  --  E2
  B2<=X;  -- передача X на порт B2
end STRUCT_F_C;
```

VERILOG

```
module STRUCT_F_C
`include "F_INTERFACE.v"
// включено описание портов

  wire X;

  INE2 E1 (A1, A2, X);  //  E1
  INE2 E2 (X, X, B1);  //  E2
 assign B2=X;
endmodule // STRUCT_F_C
```

В VHDL-описании архитектуры STRUCT_F_C объекта проекта F сначала указан интерфейс компоненты, из которой строится схема. Это компоненты типа INE2 с двумя входными и одним выходным портом. Затем, после begin, идут операторы конкретизации компонентов. Для каждого экземпляра компонента следуют ее имя, тип и карта портов, указывающая соответствие портов компоненты поступающим на них сигналам. Например, для компонента по имени E1 типа INE2 на порт X1 подан сигнал A1, на порт X2 — сигнал A2. Порядок конкретизации компонента безразличен, так как это параллельные операторы.

Для того чтобы описание объекта F было полным, в библиотеке проекта необходимо иметь описание интерфейса и архитектуры некоторого модуля, сопоставляемого компоненте INE2. В VHDL допускается отличие его имени от имени компоненты, в VERILOG они должны быть одинаковы.

Допустим, этот объект в VHDL-библиотеке обозначен как LA3 и имеет задержку 1 ns. Описание объекта LA3 состоит из интерфейса и тела:

VHDL

```
entity LA3 is
   port (X1, X2 :in bit; Y:out bit);
 end LA3;
architecture DF_ LA3 of LA3 is
  begin
     Y<= not(X1 and X2) after 1 ns;
end;
```

VERILOG

```
`timescale 1 ns /100 ps
 module INE2 (X1, X2, Y);
  input X1,X2;
  output Y;

  assign #1 Y=! (X1 & X2);
endmodule // INE2
```

У VHDL-объекта LA3 может быть несколько архитектур. В примере дан вариант потокового описания архитектуры DF_ LA3 объекта LA3, который содержит оператор назначения сигналу Y инверсного значения конъюнкции сигналов X1 и X2 с задержкой 1 ns.

VERILOG-описание STRUCT_F_C предполагает одинаковые имена компонентов и соответствующих объектов (INE2). Использование промежуточной переменной X в этом описании излишне.

3.1.5. Объявление конфигурации

VHDL

Обозначение типа компонента (INE2) и его портов VHDL-описании архитектуры STRUCT_F_C объекта F и обозначение соответствующего объекта проекта (LA3) в библиотеке проекта могут не соответствовать друг другу, чем достигается большая независимость отдельных групп проектировщиков. Связывание обозначений осуществляется в форме объявления конфигурации (configuration). Для того чтобы задать информацию о том, что использованная при описании архитектуры STRUCT_F_C объекта F компонент INE2 соответствует библиотечному объекту LA3 и варианту его архитектуры под названием DF_LA3, надо объявить конфигурацию (configuration). Конфигурация V1 указывает, что из рабочей библиотеки проекта (library work) для архитектуры STRUCT_F_C объекта F и компонента с именами E1 и E2 типа INE2 следует использовать архитектуру DF_LA3 объекта LA3. Если бы имена компонент и модуля совпадали, связывание этих имен производилось бы автоматически и необходимость описания конфигурации отпала бы.

```
library WORK;              -- подключается рабочая библиотека
  use WORK.all;            -- проекта, используются все (all)
                           -- компонента библиотеки WORK
configuration V1 of F is     -- конфигурация по имени V1 объекта F
   for STRUCT_F_C          -- для архитектуры STRUCT_F_C
  -- конкретизации E1, E2 компоненты INE2 соответствуют объекту LA3  с
  -- архитектурой DF_LA3 из библиотеки WORK
        for E1, E2: INE2
           use entity LA3 (DF_LA3) port map (X1=>X1, X2=>X2, Y=>Y);
        end for;
     end for;
end V1;
```

Библиотека WORK подключена по умолчанию, и первые два оператора примера избыточны. Связывание имен компонентов проекта с библиотечными объектами можно реализовать и в архитектурах с помощью оператора спецификации компонент и прямым созданием экземпляра объекта проекта (это было показано в начале главы 1, § 1.1.4).

VERILOG-2000

В версию VERILOG-2000, как уже упоминалось в главе 1, включен так называемый блок конфигурации. Использование этой конструкции позволяет зафиксировать в VERILOG-описании имена библиотек и размещение библиотек с описаниями объектов проекта, сопоставленных компонентам.

```
config V1                  //имя конфигурации V1
  design F_LIB.TOP         //Устанавливает порядок поиска объектов проекта,
                           //сопоставленных компонентам, - в примере
                           // в виртуальной библиотеке F_LIB
 library F_LIB "./*.v"; //определяет место виртуальной библиотеки -
                        // в примере текущий каталог и файлы с расширением .v
endconfig
```

3.1.6. Контроль временных соотношений

Описание схем может учитывать не только задержки сигналов, но и контролировать их временные соотношения. Например, частота тактовых сигналов не должна превышать предельной частоты работы схемы, должно соблюдаться время предустановки и удержания и т. п.

Допустим, для нашего объекта проекта F предельная частота работы 500 мегагерц (период =2 ns и длительность состояний 1 и 0 одинакова и не менее 1 ns). Сам объект объявлен как F_T_CTL, а его временные параметры объявлены как параметры настройки T_PERIOD,T_AND,T_NOT;

HDL-описание включает процесс Proverka_A1, в котором фиксируется время последнего изменения сигнала A1 и контролируется разница текущего времени (now, $time) и последнего изменения сигнала A1.

Аналогичный процесс для A2 изображен иными средствами:

VERILOG-описание для A2 использует системную функцию $period, которая включена в блок specify, VHDL — параллельный оператор контроля (утверждения) и атрибуты сигнала A2.

VHDL

```
entity F_T_CTL is
 generic(T_PERIOD: time :=2 ns;
        T_AND, T_NOT:time:=1 ns);
  port (A1, A2:in bit; B1, B2: out bit); ;
end;

architecture F_B_TIME_CTRL
                    of F_T_CTL is
signal X:bit;
begin
Proverka_A1:process (A1)
 variable T_E:time:=0 ns;
begin
   if now>0 ns then
     assert (( now-T_E)>T_PERIOD/2)
     report "A1 period violation";
   end if;
 T_E:=now;
end process;
 assert (A2 /=
     A2'delayed(T_PERIOD/2-0.01 ns));
  report "A2 period violation";
   E1:B1<=X;
```

VERILOG

```
`timescale 1 ns/ 100 ps
 module F_T_CTL (A1, A2,B1, B2);

   input A1,A2;
  output B1,B2;

   parameter T_PERIOD= 2;
   parameter T_AND= 1;
   parameter T_NOT= 1;
   wire X;integer T_E;
   initial T_E=0;
  always @ (A1) begin: Proverka_A1
   if ($time> 0)
    if(( $time-T_E)<=T_PERIOD/2)
    $display ("A1 period violation");

   T_E=$time;
  end //always
 specify
 $period(A2, T_PERIOD);
 endspecify
  assign  B1=X;
```

```
  E2:B2<= not(X) after T_NOT;                assign #(T_NOT) B2=!(X);
   X<= A1 and  A2  after  T_AND;             assign #(T_AND) X= A1 & A2;
end F_B_TIME_CTRL;                         endmodule   // F_B_TIME_CTRL
```

VHDL-расширение в виде VITAL-пакетов IEEE.VITAL_TIMING и VITAL_PRIMITIVE с такой же легкостью, как системные функции VERILOG, позволяют описывать временные соотношения (подробнее см. приложения 1 и 2), а ниже для иллюстрации дан только фрагмент, использующий процедуру VitalPeriodPulseCheck пакета VITAL_TIMING (порт A1 должен быть типа Std_logic).

```
library IEEE; USE IEEE.Std_Logic_1164.ALL; use IEEE.vital_timing.all;
architecture F_B_TIME_CTRL_V      of F_T_CTL  is  begin
PROVERKA_A1: process(A1)
-- вспомогательные переменные для проверки периода A1
    VARIABLE Pviol_A1          : X01 := '0';
    VARIABLE PD_A1             : VitalPeriodDataType := VitalPeriodDataInit;
begin
-- проверяется период и длительности 1 и 0 состояний сигнала A1.
    VitalPeriodPulseCheck (                  -- обращение к процедуре
       TestSignal      => A1,                -- контролируемый сигнал
       TestSignalName  => "A1",              -- имя для печати при нарушении
       Period          => T_PERIOD,          -- минимальный период
       PulseWidthHigh  => T_PERIOD/2,        -- минимальная длительность 1
       PulseWidthLow   => T_PERIOD/2,        -- минимальная длительность 0
       PeriodData      => PD_A1,             -- вспомогательная переменная
       Violation       => Pviol_A1);         -- переменная для фиксации нарущений
end process; end;
```

3.1.7. VERILOG-описание, использующее примитивы

Как уже отмечалось, VERILOG имеет **встроенные примитивы** — многовходовые И, ИЛИ и др., — а также позволяет пользователю создавать собственые-**примитивы пользователя**. Ниже описание схемы на рис. 3.2 на встроенных примитивах ИНЕ-nand с указанными разными задержками: фронта — 2 ns и среза — 3 ns.

```
`timescale 1 ns/100 ps
 module STRUCT_F_PRIM
   `include "F_INTERFACE.v"
    nand #(2,3) E1 (B2,A1, A2);  //  выход примитива - первый в списке
    nand #(2,3) E2 (B1, B2, B2);  // его параметров
  endmodule // STRUCT_F_PRIM
```

Ниже описание пользовательского комбинационного примитива INE2.

```
primitive INE2(Y,X1,X2);
  output Y;//единственный выход должен быть первым в списке параметров
  input X1,X2;//все порты (их до 10 штук) должны быть однобитовые
   table // таблица функции комбинационной схемы
// X1   X2  : Y
   0    ?   : 1  // ? - безразличное значение аргумента
   ?    0   : 1
   1    1   : 0
  endtable
endprimitive
```

Далее его использование в модуле STRUCT_F_USER_PRIM

```
`timescale 1 ns/100 ps
 module STRUCT_F_USER_PRIM
   `include "F_INTERFACE.v"
    INE2  E1 (B2,A1, A2);  //  E1
    INE2  E2 (B1, B2, B2);  //  E2
  endmodule // STRUCT_F_USER_PRIM
```

Примеры HDL-описаний сумматоров были даны ранее в главе 1, а в главе 5 приведены синтезабельные (составленные в соответствии с определенными ограничениями языка) описания других комбинационных схем.

Упражнения

1. Постройте поведенческое описание архитектуры схемы F, приведенной на рис. 3.2,*d*.
2. Постройте структурное описание архитектуры схемы F, приведенной на рис. 3.2,*d*.
3. Постройте описание конфигурации для упражнения 2, интерфейс и архитектуру объекта nor (ИЛИ-НЕ).
4. Измените HDL-описание объекта проекта ine2, установив входные порты по умолчанию в 1, и соответственно в структурном описании объекта проекта F один из портов ine2 оставьте не задействованным.
5. Измените VERILOG-2000 описание F_CASE в соответствии с возможностями старой версии VERILOG-1995.

3.2. Схемы с памятью

Эти примеры иллюстрируют использование HDL-средств описания фронтов сигналов, массивов, контроля времен предустановки-удержания и т. д.

3.2.1. D-триггер

D-триггер с информационным входом DIN и тактовым сигналом CLK. Триггер принимает информацию по фронту CLK.

Иллюстрируются средства описания фронтов сигналов.

VHDL

```
entity dff is
 port (CLK,DIN: in bit;
        DOUT: out bit);
end;
architecture beh of dff is
 begin
process (CLK)
begin
  if (CLK'event and CLK='1') then
    DOUT <= DIN ;
  end if;
end process;
end;
```

VERILOG

```
module dff( clk,din,dout);
  input clk,din;
  output dout;
   reg dout;


 always @(posedge clk)
      begin

        dout <=  din;
      end
endmodule //dff
```

3.2.2. D-триггер со сбросом

D-триггер с информационным входом DIN, асинхронным инверсным сбросом RST_N и тактовым сигналом CLK. Триггер принимает информацию по фронту CLK. VERILOG-описание выполнено в стиле VERILOG-2000, VHDL использует пакет STD_LOGIC_1164. Задержка выхода — 6 ns.

VHDL

```
Library IEEE;
Use IEEE.std_logic_1164.all;
entity dffr is
  port (CLK,DIN,RST_N
        : in STD_LOGIC;
        DOUT: out STD_LOGIC);
end;
architecture beh of dffr is
 begin
process (CLK, RST_N)

begin
 if RST_N='0' then
     DOUT <= '0' after 6 ns;
elsif (CLK'event and CLK='1') then
     DOUT <= DIN after 6 ns;
  end if;
end process;
end;
```

VERILOG-2000

```
timescale 1 ns/100 ps
  module dffr
    (input wire clk,
     din,RST_N,
    output reg dout);

  always @(posedge clk,
           negedge RST_N)
  begin
    if(!RST_N)
    dout <= #(6) 0;
    else
     dout <= #(6) din;
  end
endmodule //dffr
```

3.3.3. Схема D-триггера на вентилях ИНЕ

Триггер с информационным входом Din, асинхронным инверсным сбросом RST_N и тактовым сигналом CLK. Триггер принимает информацию по фронту CLK. Описание в потоковом стиле. Задержка каждого вентиля равна 1 ns. Выход триггера в VHDL-описании объявлен как INOUT, так как с него происходится считывание внутри архитектуры. В VERILOG-описании для последних двух операторов непрерывного присваивания применена сокращенная форма записи.

VHDL

```
entity dffr_c is
  port (CLK,DIN,rst_n: in bit;
        DOUT: inout bit);
end;
architecture dffr_c of dffr_c is
 signal n1,n2,n3,n4,n5,n6,
   q_n,c_n,d_n: bit;
   begin
    n2<= d_n and c_n after 1 ns;
    n3<= not(n1 and n4)after 1 ns;
   dout<= not(n5 and q_n) after 1 ns;
    n1<= not(dout and c_n and rst_n)
            after 1 ns;
```

VERILOG

```
timescale 1 ns/100 ps
module dffr_c(clk,din,rst_n,dout);
 input clk,rst_n,din;
output dout;
 reg dout;

wire n1,n2,n3,n4,n5,n6,
     q_n,c_n,d_n;

assign #(1)n2= d_n & c_n;
assign #(1)n3=~( n1 & n4);
assign #(1)dout=~( n5 & q_n);

assign #(1)n1=~(dout & c_n & rst_n);
```

```
  n4<= not(n2 and n3 and rst_n)
            after 1 ns;                  assign #(1)n4=~(n2 & n3 & rst_n);
  n5<= not(n3 and clk)after 1 ns;        assign #(1)n5=~(n3 & clk);
  n6<= not(n4 and clk)after 1 ns;        assign #(1)n6=~(n4 & clk);
  q_n<= not(n6 and rst_n and dout)
                after 1 ns;              assign #(1)q_n=~(n6 & rst_n & dout);
  c_n<= not(clk)after 1 ns;              assign #(1)c_n=~(clk),
  d_n<= not(din)after 1 ns;                          d_n=~(din);
end;                                     endmodule //dffr_c
```

Читателю в качестве упражнений предлагается построить структурное описание триггера.

3.2.4. D-триггер как примитив VERILOG

D-триггер с информационным входом d, асинхронным инверсным сбросом rst_n и тактовым сигналом clk, реализованный на VERILOG как примитив пользователя. Триггер принимает информацию по фронту clk.

```
primitive DFF (q,d,clk,rst_n);
  output q; input d,clk,rst_n;  reg q;
  table
// d   clk   rst_n  : q  : next_q;
   0   r     1     : ? :   0; //прием входа  d  по фронту clk
   1   r     1     : ? :   1;
   ?   ?     0     : ? :   0; // асинхронный сброс в 0
   ?   f     ?     : ? :   -; //
  endtable
endprimitive
```

3.2.5. Модель RS-триггера-защелки

Иллюстрируется использование контролирующих утверждений (оператор утверждения см. в гл. 1).

На рис. 3.3 дано условное графическое изображение RS-триггера-защелки на вентилях И-НЕ с двумя инверсными входами S и R и одним выходом Q и таблица, иллюстрирующая его функционирование.

Комбинация S="0" и R="0" на входах считается запрещенной и переводит триггер в неопределенное состояние «X», так как при последующей подаче сигнала хранения «11» в схеме возможны гонки. Qt — старое состояние и значение вы-

S	R	Qt	Qt+1
0	1	X	1
1	0	X	0
1	1	Qt	Qt
0	0	Qt	X

Рис. 3.3. RS-триггер-защелка

хода триггера, Qt+1 — новое состояние. Задержка установки — 17 ns, сброса — 13 ns. Архитектурой POVED поведение триггера RST представлено в двоичном алфавите с учетом задержек и контролем запрещенных комбинаций сигналов.

VHDL

```
entity RST is
  port(S, R: in bit; Q: out bit);
 -- ниже параллельный
-- оператор утверждения
  begin postponed
   assert not((R='0') and (S='0'))
   report "RS - возможны гонки"
   severity warning;
end RST;
architecture POVED of RST is
  begin process (R, S)
    begin
      if (R='1') and (S='0') then
        Q<='1' after 17 ns; -- в 1
      elsif (R='0') and (S='1') then
        Q<='0' after 13 ns;
      end if;
   end process;
  end;
```

VERILOG

```
timescale 1 ns/100 ps
 module RST(S,R,Q);
  input S,R;
  output Q; reg Q;
 always @(R or S) begin

  if ( ((R==0) && (S==0)))
  $display("RS - возможны гонки");

  if ((R==1) && (S==0))
      Q<=#17 1;
 else if((R==0) && (S==1))
      Q<=#13 0;
end
endmodule
```

Обратите внимание, что оператор assert срабатывает при ложном (false) значении условия!

Упражнения

1. Опишите поведение RS-триггера с двумя выходами Q и NQ.
2. Опишите функциональную схему RS-триггера на элементах 2ИЛИ-НЕ.
3. Опишите поведение RS-триггера в алфавите пакета STD_LOGIC_1164, вырабатывая значение «X» при запрещенной комбинации сигналов на входе.
4. Реально гонки возникают при смене входов с 00 на 11. Отразите это в контролирующем утверждении.

3.2.6. Модель Т-триггера

Иллюстрируется прием определения режима переключения и вычисления задержки сигналов.

Приведем пример поведенческого описания Т-триггера с асинхронными инверсными S и R входами. На рис. 3.4 представлено условное графическое изображение и таблица переходов и выходов триггера. В ней символ X соответствует произвольному значению сигнала. Триггер срабатывает по фронту (F) сигнала C. В описании используется двоичный алфавит представления сигналов, контролируется запрет одновременного поступления комбинаций S=0 и R=0, учитываются представленные как параметры настройки задержки: установки в 1 по сигналу S=0 (T1S), сброса в 0 по сигналу R=0 (T0R); переключения в 1 (T1C) и в 0 (T0C) по счетному входу C. Задержки для Q и NQ приняты равными.

S	R	C	Qt	Qt+1	Пояснения
0	1	X	X	1	Установка в 1
1	0	X	X	0	Сброс в 0
1	1	F	Qt	$\overline{Qt}$	Счет по фронту
0	0	X	Qt	X	Запретный режим

F - фронт сигнала

Рис. 3.4. Т-триггер, условное графическое изображение и таблица переходов

Обычно комбинация S=0, R=0 не является запретной [24] и в нашем примере просто используется для иллюстрации возможностей оператора контроля.

Внутреннее состояние триггера представлено переменной T, объявленной в процессе. При отображении переключения по C входу использован атрибут сигнала Stable, равный false, когда сигнал меняется. По умолчанию значение параметров настройки равно «0», начальное значение выходов Q=0, NQ=1.

VHDL-модель

Ниже описание интерфейса и архитектуры TFF:

```
    entity TFF is
      generic (T1S, T0R, T1C, T0C: time:=0 ns);
      port (S: in bit:='1'; C, R: in bit:='0';
           Q: inout bit:='0';          -- inout, так как в архитектуре BEHAVIOR
           NQ: inout bit:='1');        -- выходы считываются
    end TFF;
-- начальное состояние триггера «0» (T:='0')
architecture BEHAVIOR of TFF is
  begin
    process (S, C, R)
    variable T: bit:='0';            -- T отображает внутреннее состояние триггера
    variable DELAY: time:=0 ns;        -- отображает задержку
  begin
--контроль S=0 и R=0 реализуется последовательным оператором утверждения
    assert not (S='0' and R='0')
      report "одновременный 0 на S и R входе T-триггера"
      severity warning;
                    -- определение нового состояния триггера
   if S='0' and R='1' then T:='1';            -- установка
   elsif S='1' and R='0' then T:='0';         -- сброс
   elsif R='1'and S='1'and C='1'and not C'stable then          -- счет по фронту
         T:=NQ;
   else T:=Q;                                 -- хранение
   end if;
                -- определение нового значения задержки переключения
      if T='0' and Q='1' then
        if R='0' then DELAY:=T0R;             -- сброс в 0 по R входу
        else DELAY:=T0C;                      -- сброс в 0 по C входу
        end if;
```

```
      elsif T='1'and Q='0' then
          if S='0' then DELAY:=T1S;              -- установка в 1 по S
          else DELAY:=T1C;                       -- установка в 1 по C
          end if;
      end if;
    -- ниже  передача внутреннего состояния T на выходы Q  и NQ
    Q<=T after DELAY;
    NQ<=not T after DELAY;
  end process;
end BEHAVIOR;
```

VERILOG-модель

```
module TFF  (S, C, R,Q,NQ);
        input S, C, R; output Q,NQ;       reg Q,NQ;
        parameter T1S=0, T0R=0, T1C=0, T0C=0;
        reg T; initial T=0; // T  внутреннее состояние триггера
        integer DELAY; initial DELAY =0;  //отображает задержку
always @( negedge S or  posedge C or  negedge R)
    begin
    -- контроль S=0 и R=0
      if( (S=0  && R=0))
        $display("одновременный 0 на S и R входе T-триггера");
      // определение нового состояния триггера
      if(( S==0 ) &&( R==1) )          T   =1;        -- установка
     else   if ((S==1) && ( R==0))       T=0;          -- сброс
            else   if( (R=1) &&( S=1) &&(C==1) )        -- счет по фронту
                          T     =NQ;
     // определение нового значения задержки переключения
    if( (T==0) &&(Q==1))begin
      if( R==0)  DELAY =T0R;              -- сброс в 0 по R входу
      else DELAY=T0C;                     -- сброс в 0 по C входу
   end
     else if ((T==1)&&( Q==0)) begin
         if( S==0)  DELAY:=T1S;             -- установка в 1 по S
        else DELAY:=T1C;                     -- установка в 1 по C
    end
   // ниже  передача внутреннего состояния T на выходы Q  и NQ
   Q<=#(DELAY)  T;
   NQ<= #(DELAY) ~ T;
  end
  endmodule
```

Упражнения

1. Опишите VHDL-модель T-триггера:
   а) с разными задержками на выходах Q и NQ;
   б) без использования оператора process;
   в) в многозначном алфавите;
   г) используя вместо атрибута stable атрибут event.
2. Опишите VERILOG-модель T-триггера:
   а) с разными задержками на выходах Q и NQ;
   б) в многозначном алфавите с выдачей X при запретной комбинации входов.

3.2.7. VHDL — оператор блока в модели триггера типа «защелка»

Иллюстрируется применение оператора блока. Оператор блока в книге подробно не рассмотрен, некоторые детали — см. приложение 1.

Описание функционирования защелки выполнено без учета задержек, но с использованием ранее не употреблявшейся конструкции оператора блока (block) языка VHDL и оператора условного назначения сигналу. Блок содержит параллельные операторы. В его заголовке может использоваться «охранное» (guarded) выражение, имеющее тип boolean (в примере C='1' or R='1'). Оператор охраняемой конструкции в нашем примере — это защищенный оператор назначения сигналу с дополнением guarded — будет выполняться, если охранное выражение имеет значение истинно (TRUE) и сигнал в правой части оператора назначения меняется или если охранное выражение меняет значение из ложного (FALSE) в истинное (TRUE).

```
entity DTR is
  port (D, C, R: in bit; Q: inout bit);
end DTR;
architecture BEH_DTR of DTR is
   begin
     D_LATH: block (C='1' or R='1');
            begin          -- ниже оператор условного назначения
              Q<=guarded '0' when R='1' else
                        D  when C='1' else   Q;
     end block D_LATH;
end BEH_DTR;
```

Упражнения

1. Опишите поведение D-триггера в форме оператора процесса.
2. Отобразите поведение D-триггера в многозначном алфавите.

3.3. Модель блока синхронной памяти

Иллюстрируется использование массивов и векторов.

Поведение реальных блоков синхронной памяти несколько отличается от приведенного ниже — обычно по тактовому сигналу clk защелкиваются не значение выходного порта, а значения входных управляющих сигналов: выборки кристалла — EN, чтения-записи — WE, адреса — ADDR, а затем, если режим чтения, на выход DO асинхронно выдаются данные из памяти. В главе 7 приведено более точное описание двухпортовой синхронной памяти.

Пример модели асинхронной памяти рассмотрен в конце главы 4.

3.3.1. VHDL-модель

Модель иллюстрирует возможность описания функции преобразования типа данных пользователем, хотя можно было использовать подобную функцию из стандартного пакета, например, IEEE.numeric_std или std_unsigned.

```
library IEEE;use IEEE.STD_LOGIC_1164.all;
entity ramb4_s8p is
  generic ( TW,TR: time :=3 ns);
  port (DI      : in STD_LOGIC_VECTOR (7 downto 0);
        EN      : in STD_ULOGIC; WE     : in STD_ULOGIC;
        RST     : in STD_ULOGIC; CLK    : in STD_ULOGIC;
        ADDR    : in STD_LOGIC_VECTOR (8 downto 0);
        DO      : out STD_LOGIC_VECTOR (7 downto 0)
       );
end;
architecture beh of ramb4_s8p  is
   constant data_width : integer := 8; constant addr_width: integer := 9;
   constant depth : integer := 2** addr_width;
   type mem_type is array (depth-1 downto 0) of std_logic_vector
      (data_width-1 downto 0);
-- функция a_value преобразует std-вектор в целое
        function a_value (BV:in std_logic_vector) return natural is
            variable E: natural:=0;
            begin
     --  цикл по числу разрядов вектора
              for i in BV'LOW to BV'HIGH loop
                 E:=E*2;          -- старший разряд - LOW
                 if BV(I)='1' then E:=E+1;
                 elsif BV(I) /='0' then
                   report "there is wrong bit val in the  mem addr";
                    E:=0;exit;
                 end if;
              end loop;
        return E;
        end a_value;
begin
process (clk)
    variable mem: mem_type;
    begin
        if rising_edge(clk) then --- функция выделения фронта из пакета std_logic
           if (en = '1') then
              if (rst = '1')     then       do <=(others=>'0');
              elsif (we = '1') then    mem(a_value(addr)) := di;
              elsif (we = '0') then   do <= mem(a_value(addr))after TR;
              end if;
           end if;
       end if;
   end process;
end beh;
```

Объявление VHDL-массива памяти как переменной — variable mem: mem_type, а не как сигнала позволяет в несколько раз сократить расход памяти моделирующей ЭВМ.

3.3.2. VERILOG-модель

```
module  ramb4_s8p(di, en, we, rst, clk, addr, do);
   parameter        data_width = 8; parameter        addr_width= 9;
```

```
// parameter        depth  =  2 **addr_width // операция ** в  VERILOG- 2000
 parameter        depth = 512;//
// parameter        depth = 1<< addr_width;// можно и так
parameter        tw=3;  parameter tr       =3;
input  [data_width -1 : 0]di;    input     en;
input     we;  input     rst;   input     clk;
input   [data_width  : 0 ] addr;
output [addr_width -1 : 0]   do;    reg [data_width -1 : 0]   do;
reg[data_width -1 : 0] mem[ depth-1 : 0];//блок памяти
always @ ( posedge clk)
   begin
     if (en == 1) begin
        if (rst ==1)    do <=0;
        else if (we ==1)begin
               mem[addr] <= #(tw)di;  do <=di;
            end
            else if (we ==0)
                       do <=#(tr) mem[addr];
   end   //if en==1;
end  //always
endmodule
```

3.3.3. VERILOG — модель памяти с учетом задержек и контролем временых параметров сигналов в блоке specify

```
// Описание интерфейса опущено
always @ ( posedge clk)
  begin
     if (en == 1) begin
        if (rst ==1)       do <=0;
        else if (we ==1)begin   //  запись
               mem[addr] = #(TW)di ; do <=di;
            end
            else if (we ==0)
                  do <=#(TR) mem[addr];
   end   //if en==1;
 end  //always
//ниже проверка времени предустановки и удержания входных данных
 specify
    specparam tsetup=0.7, thold= 0.25;
   $setup (di,clk,tsetup);//проверка времени предустановки
   $hold(clk,di,thold);//проверка удержания
 endspecify
endmodule
```

Читателю в качестве упражнений предлагается включить в модель утверждения, которые могут проверить максимальную частоту тактового сигнала, время предустановки и удержания сигналов адреса и разрешения (см. приложение 2).

3.3.4. VHDL — модель памяти с общим регистром входных-выходных даных

У памяти изменен не только выходной регистр, но и адресный — адрес фиксируется на нем по фронту clk.

```
library IEEE;use IEEE.STD_LOGIC_1164.all;  IEEE.STD_LOGIC_Unsigned.all;
entity ram is
  generic (TW,TR: time :=3 ns;TD,TDZ: time:=1 ns);
  port (D     : inout STD_LOGIC_VECTOR (7 downto 0);
        EN    : in STD_ULOGIC; WE     : in STD_ULOGIC;
        DE    : in STD_ULOGIC; CLK    : in STD_ULOGIC;
        ADDR  : in STD_LOGIC_VECTOR (8 downto 0)
       );
end;
architecture beh of ram  is
constant data_width : integer := 8; constant addr_width: integer := 9;
constant depth : integer := 2** addr_width;
type mem_type is array (depth-1 downto 0) of std_logic_vector
                                              (data_width-1 downto 0);
signal do: std_logic_vector (data_width-1 downto 0);
signal AD_REG: std_logic_vector (addr_width-1 downto 0);
begin
   ADR_FIX:  process (clk) begin -- защелкивается адрес
       if rising_edge(clk) then  AD_REG<=addr; end if;
   end process;
  process (AD_REG,en,we)
      variable mem: mem_type;
   begin
         if (en = '1') then
             if (we = '1') then          mem(conv_integer(ad_reg)) := d ;
              elsif (we = '0') then      do <= mem(conv_integer(ad_reg))after TR;
             end if;
         end if;
  end process;
      d<= do when de'stable and de='1' else
             do after td when de'event and de='1' -- переключение de в 1
             else (others=>'Z')   after tdz ;--  de =0 -- выход в высокий импеданс
end beh;
```

Читателю в качестве упражнений предлагается уточнить описание, например добавив в условия возможные ситуации с неопределенными значениями управляющих сигналов (при de ='X' будет d=(others=>'X') и т. п.).

Глава 4
Функциональная верификация HDL-описаний

Верификация

Верификация проекта — это процесс доказательства того, что он функционирует согласно его спецификации.

Различают методы **имитационной** и **формальной верификации**.

Имитационная верификация — традиционная идеология верификации с использованием моделирования. Она предполагает моделирование HDL- описания проектируемого объекта и воспроизведение в ходе имитационного эксперимента всех (или части) возможных внешних входных воздействий и внутренних состояний моделируемой системы.

Формальная верификация (см., например, пакеты фирмы Verplex — www.verplex.com) обычно предполагает использование HDL-описания как формальной модели с целью доказательств ее корректности и эквивалентности или неэквивалентности спецификации проекта.

Ниже рассматривается только имитационная верификация.

Кто такой «инженер-верификатор»?

Перечень типичных требований 1998 г. к инженеру-верификатору на примере фирмы Raycer Graphics представлен ниже. Он включает, помимо общих: навыки работы в коллективе, стаж не менее 3 лет, умение работать с технической документацией, знание ОС UNIX и WINDOW и т. п., следующие требования:

— разработка и применение верификационных сред;

— разработка, отладка и верификация моделей систем;

— разработка тест-планов и тестов;

— создание инструментальных подсистем автоматической генерации тестов;

— формальная верификация и проверка модели;

— опыт работы с языками и системами VERILOG, VERILOG-PLI (интерфейс с C);

— C/C++ (часть тестов пишется на них);

— Make, PERL, TCL (языки создания управляющих скриптов);

— OpenGL (стандарт графического программирования — специфика фирмы Raycer).

В дополнение к этому перечню в настоящее время подобная фирма может также потребовать опыт работы с современными системами поддержки процесса верификации типа VERA (OpenVera), SPECMAN и т. п.

Этапы верификации

Процесс верификации начинается с разработки тест-плана (как предполагается тестировать, каким инструментарием, в каком порядке и когда, простые и граничные тесты, запрещенные и провокационные режимы и т. п.). Затем следуют:

— создание верификационной среды (test bench);
— отладка тестирующей программы и модели исследуемого объекта;
— профилирование (выявление и по возможности устранение сильно снижающих быстродействие участков теста);
— проведение регрессионных (многократных, исчерпывающих) экспериментов.

4.1. Пример верификации описания простого объекта проекта F

Модельный (имитационный) эксперимент позволяет осуществить проверку алгоритмов и схем проектируемой аппаратуры, выявить риски и гонки в схемах, провести анализ контролирующих тестов на полноту и корректность, получить временные диаграммы сигналов, которые могут использоваться в процессе настройки аппаратуры и ее эксплуатации.

Путь для верификации результатов первого этапа проектирования объекта проекта F (см. главу 3) требуется произвести модельный эксперимент, например, по проверке архитектуры F_CASE, отображающей функцию объекта проекта F, на предмет, соответствует ли она спецификации- функции, заданной таблицей рис. 4.1, *б*.

а)

Входы		Выходы	
A1	A2	B1	B2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

б)

Рис. 4.1. Объект проекта F: а — условное графическое изображение объекта F; б — реализуемая им функция

VHDL

```
entity F is
  port (A1, A2:in bit; B1, B2: out bit);
end F;
architecture F_CASE of F is
 begin

   process (A1, A2)
   variable EE:bit_vector(0 to 1);
     begin -- & - конкатенация
       EE:=A1& A2;
```

VERILOG

```
module F_CASE(A1,A2,B1,B2);
  input A1,A2;
  output B1,B2;reg B1,B2;
  always @(A1 or A2)

 begin //{} конкатенация
```

```
    case EE is
      when "11" => B1<='1'; B2<='0';
      when others => B1<='0'; B2<='1';
    end case;
  end process;

end F_CASE;
```

```
case ({A1, A2} )
  2'b11: begin B1=1; B2=0;end
  default: begin B1=0; B2=1;end
endcase
end

endmodule //F_CASE
```

Для этого следует создать новый объект проекта, назовем его TB_F, содержащий, помимо модели объекта F, еще две составляющие — модель внешней среды — генератор внешних сигналов и модель наблюдателя — (рис. 4.2). TB — сокращение английского термина Test bench (тест-стенд).

Генератор сигналов Исследуемая схема Регистратор

Рис. 4.2. Структура объекта проекта TB_F, реализующего модельный эксперимент по верификации объекта проекта F

Тест — набор внешних сигналов, подаваемых на проверяемый объект. Он должен быть полным, т. е. обнаруживать все возможные ошибки проекта (в данном случае функциональные), корректным, т. е. не содержать запрещенных комбинаций входных сигналов, компактным, т. е. проверять тестируемый объект за приемлемое машинное время, и т. д.

В нашем случае на первый взгляд достаточно подать на вход модели объекта F все возможные 4 комбинации входных сигналов и сравнить выходы проверяемого объекта с эталоном, заданным таблицей (см. рис. 4.1). Для подачи сигналов на входы проверяемого объекта в структуре теста (см. рис. 4.2) используется генератор внешних воздействий (GEN). Один из возможных вариантов временной диаграммы его выходных сигналов дан на рис. 4.3.

Сигналы Y1 и Y2 пробегают комбинации значений от 00 до 11 за 40 наносекунд. Можно заметить, что Y1 — периодический сигнал с периодом 30 ns.

Рис. 4.3. Временная диаграмма выходных сигналов Y1, Y2 объекта GEN

Тестирующая программа (testbench)

VHDL

```
 entity TB_F is
-- у теста порты отсутствуют
end TB_F;
USE STD.TEXTIO.all;-- ввод-вывод
architecture NOT_STRUCT_TB of TB_F is
   component F port
      (A1, A2: in bit; B1, B2: out bit);
   end component;
   signal I1, I2, O1, O2: bit;
     begin
GEN_V1 :process begin
           I1<='0'; I2<='0' ;
          wait for 10 ns; I1<='1';
          wait for 20 ns; I1<= '0' ; I2<='1';
          wait for 10 ns; I1<='1' ;
           wait for 10 ns;
           wait; -- бесконечное ожидание
end process GEN_V1;
  -- ниже связи компонент объекта TB_F
  C2: F port map (I1, I2, O1, O2);
WRITER1 : process (O1,O2)
   variable L:LINE;-- из пакета TEXTIO
          begin
            write (L,NOW);

           write (L,I1); write (L,I2);
           write (L,O1); write (L,O2);

           writeline (output,L);
         end process;
  end NOT_STRUCT_TB;
```

VERILOG

```
module NOT_STRUCT_TB ();

   reg I1, I2;
   wire O1,O2;
   initial begin :GEN_V1
    I1=0;I2=0;
   #10; I1=1;
   #20; I1=0;I2=1;
   #10; I1=1;
   #10;
   $finish;//останов модели
  end// GEN_V1

F_CASE  C2 (I1, I2, O1, O2);
always @(O1 or O2);

  begin :WRITER1
   $display("t=%t",$time);
   $display("I1=%bI2=%b",I1,I2);

  $display("O1=%b",O1);
  $display("O2=%b",O2);

  end //WRITER1
endmodule //NOT_STRUCT_TB
```

В VHDL-описании процесса вывода WRITER_1 использован стандартный пакет TEXTIO, содержащий декларацию текстового файла OUTPUT и стандартную процедуру записи строки WRITELINE. Каждый раз, когда один из сигналов (O1 или O2) изменяет свое значение, процесс запускается и выполняет оператор записи значений модельного времени NOW и значений сигналов I1, I2, O1, O2 в стандартный текстовый файл OUTPUT.

В VERILOG-описании первый оператор форматируемого вывода $display выводит время, второй — значения входных сигналов, третий и четвертый выводят значения выходов.

Обратите внимание на останов моделирования в момент времени 50 наносекунд.

В VHDL-модели процесс GEN_V1 попадает в оператор wait — вариант безусловного ожидания — и зависает в нем. В модели пропадает источник событий.

В VERILOG системный оператор $finish останавливает моделирование.

VHDL-описание объекта проекта TB_F для модельного эксперимента можно не дополнять конфигурационным описанием, так как имя компоненты F совпадает в нашем случае с именем объекта проекта F (когда оно совпадает, то конфигурирование происходит по умолчанию).

На рис. 4.4 представлен экран системы моделирования ACTIVE_HDL после прогона VHDL-модели TB_F.

В нижней части экрана видны результаты вывода на консоль значении сигналов и моментов времени, когда выходы объекта проекта F изменялись

В левой верхней части экрана видны временные диаграммы сигналов.

Справа виден состав файлов и библиотек проекта По результатам печати и временным диаграммам видно, что поведение моделируемого объекта совпадает с его табличной спецификацией.

Нетрудно ввести в тестирующую программу модуль автоматического сравнения выходных сигналов с эталоном, например с файлом, содержащим исходную таблицу (см. рис. 4.1), использованную при проектировании объекта F. Можно улучшить описание TB_F за счет его структуризации путем выделения в отдельные модули генератора входных сигналов и блока вывода (это упростит в дальнейшем их повторное использование и модификацию), вместо константных значений использовать параметры или именованные константы (упрощение проведения экспериментов на различных частотах) и т д. Эти вопросы будут рассмотрены позднее. Сначала рассмотрим вопросы полноты тестов.

Рис. 4.4. Экран системы моделирования

4.2. Стратегия функциональной верификации

4.2.1. Типы тестов

Для проверки правильности функционирования проектируемых систем в основном используются методы детерминистского (deterministic), случайного (random) и транзакционного (transaction level) тестирования.

Детерминистский тест проверяет обычные режимы работы системы, а также ее функционирование в граничных условиях с помощью фиксированного набора входных воздействий.

Метод случайного тестирования предполагает наличие генератора случайных входных воздействий.

Метод транзакционного тестирования предполагает фиксацию внимания на проверке интерфейсов блоков системы и проверке прохождения пакетов данных через нее.

Различают тестирование объекта как:

— «черного ящика» — инженер-верификатор при разработке теста не использует знаний о внутренней организации описания объекта;

— «серого ящика» — инженер-верификатор частично использует знание внутренней организации описания объекта;

— «прозрачного ящика» — инженер-верификатор базируется на знании внутренней структуры описания объекта.

4.2.2. Полнота теста

Целью процесса верификации проекта является проверка его соответствия спецификации на всех уровнях проектирования.

Имитационная верификация — трудоемкий процесс. Обычно в реальных случаях полный перебор невозможен (например, для такой проверки 16-разрядного сумматора требуется перебрать 2^{32} степени наборов!) и используется ограниченное множество входных наборов теста. Например, для того же сумматора сначала при тестировании его как «серого ящика» проверяют простейшие случаи (0 + 0 = 0,2 + 2 = 4, 1111111111+0,0+ 111111111111 и т. п.), затем пробег переносов (111111111+1,1+111111111 и т. д.) а затем, рассматривая его как «черный ящик», используют выборку из случайных чисел.

Требуется оценить полноту таких тестов с учетом минимизации цены тестирования и максимизации вероятности обнаружения возможных ошибок в проекте.

4.3. Оценка полноты функциональных тестов

4.3.1. Эвристические метрики

Эвристические метрики основаны на текущей статистике обнаружения ошибок в проекте. Подобные метрики включают:

— календарное время между моментами обнаружения ошибок проекта (к концу процесса верификации оно увеличивается);

— общее количество промоделированных тактов работы проектируемого устройства;

— общее число обнаруженных ошибок в проекте и т. д.

В свое время, когда автор работал в компании Raycer Graphics над проектом мощного профессионального графического ускорителя, состоящего из трех функциональных блоков-кристаллов, шеф верификационной команды Рик Авра (Rick Avra) оценил вероятное число ошибок в 3000, и, когда этот рубеж был достигнут в условиях успешной работы всех трех RTL-моделей блоков на сотнях тестов (от изображения простого голубого треугольника до изображений автомобиля с текстуровкой, тенями и полутонами), все вздохнули с облегчением. Психологический барьер был пройден и стал виден конец работы, так как темп обнаружения ошибок коллективом из десятков человек стал менее одной ошибки в день. Однако такие метрики весьма необъективны, и возможно, что часть функций проектируемого устройства осталась бы неверифицированной.

4.3.2. Программные метрики

В настоящее время большинство коммерческих систем оценки полноты функциональных тестов COVERSCAN, COVERMETER и т. п., а также развитые системы моделирования типа SILOS, ACTIVE-HDL, MODELSIM, VSS базируются на метриках, используемых для верификации программ (программо-метрический подход).

В числе таких метрик можно указать, например, такие:

— полнота покрытия тестом строк кода модели — количество исполнений каждой строки описания;

— полнота покрытия переходов — число исполнений ветвей операторов if и case;

— полнота покрытия путей — число исполнений всех возможных путей в графе программы;

— полнота покрытия выражений — низкоуровневая метрика, основанная на оценке числа вычислений выражений на различных наборах данных;

— полнота переключений (0 ->1 и 1 ->0) каждого бита данных.

Пример подобных статистических данных, получаемых системой SILOS-demo при прогоне VERILOG-варианта вышеприведенного теста описания объекта проекта F, представлен на рис. 4.5. Видно, что все строки кода тест-программы исполнялись, однако тест представляется неполным, так как выход B2 не перебрасывался из 1 в 0, а B1 из 0 в 1.

Ясно, что, если после прогона теста собранная статистика говорит, что часть операторов модели вообще ни разу не исполнялась или некоторые сигналы ни разу не переключались, имеет смысл попытаться понять причину этого явления и при необходимости дополнить тест. Несмотря на очевидную полезность и объективность этих метрик, никто не может гарантировать, что достижение, допустим, 100% покрытия тестом программного кода позволяет обнаружить все ошибки проекта устройства.

Следует вспомнить о понятиях управляемости (controllability) и наблюдаемости (observability) тестируемых объектов и их блоков. То, что тест создал условия, когда некоторая строка кода (оператор) исполняется (управляемость), не гарантирует, что результат этого исполнения будет влиять на выход моделируемой схемы (наблюдаться на выходе и сравниваться с эталоном — наблюдаемость).

Рис. 4.5. Экран системы моделирования SILOS-3 demo: покрытие строк кода модели объекта проекта F

Статистика говорит, что 100%-ное покрытие тестом строк HDL-кода функциональной модели дает примерно 70% наблюдаемости этого покрытия.

Второй недостаток программо-метрик в том, что они не имеют прямой связи с оценкой функциональной корректности модели исследуемой системы и ее спецификации.

4.3.3. Автоматно-метрический подход

Если проектируемое устройство является конечным автоматом, то можно использовать для оценки полноты теста статистику о числе достижений автоматом различных состояний (вершин) и переходов (дуг на графе автомата). Однако, так же как и для программо-метрик, автомато-метрическая полнота — это необходимое (управляемость), но не достаточное (наблюдаемость) условие полноты теста. Более мощные результаты дают системы формальной верификации автоматов, обнаруживающие зависания, недостижимые состояния и т. п.

4.3.4. Моделирование неисправностей

Эти методы базируются на идее обнаружения тестом множества возможных неисправностей в моделируемой схеме (система моделирования SILOS 3 и др.).

В модель вносится неисправность, и если выходы неисправной системы не отличаются от эталона, т. е. на всем времени моделирования неисправность не обнаруживается тестом, то тест неполный. Эти методы хорошо зарекомендовали себя в моделях вентильного уровня (неисправности типа тождественный 0 или 1 на входе/выходе вентиля), но для моделей функционального уровня они недостаточны. Неясно, например, как моделировать неисправности оператора CASE, и трудно представить, что все возможные неисправности в микросхеме процессора INTEL сводятся к установке одного из внешних контактов микросхемы в 0 или 1.

4.3.5. Мониторинг событий и проверка контрольных соотношений в модели

Эти верификационные методики базируются на том, что для разработчика модели устройства она является «прозрачным ящиком» и он имеет возможность включить в модель дополнительные фрагменты текста (assertion — мониторы и утверждения), контролирующие соблюдение функциональных ограничений.

Пример некорректных режимов — запрет одновременного прихода сигналов сброса и установки на вход триггера или наличия двух открытых буферов на тристабильной шине и т. п.

Пример обязательного режима для протокола «рукопожатие» — за фронтом сигнала запрос (req) должно следовать подтверждение (фронт ack), после чего запрос снимается (срез req), после среза req должен следовать срез ack. Это уже более сложное утверждение, содержащее последовательность событий во времени.

В VHDL-моделях для контроля можно использовать оператор утверждения — assert, реагирующий на ложность условия.

VERILOG не имеет подобного встроенного средства, и его приходится создавать с помощью обычных операторов.

Пример VHDL-утверждения о запрете одновременной 1 на двух входах set, reset:

```
signal set, reset : bit;
assert ((set and reset) / = '1')   report "wrong  set_reset condition"
severity level error;
```

Его VERILOG-эквивалент:

```
//пусть assert_on — переменная периода компиляции, управляющая включением
контроля,`ifdef и `endif — директивы компиляции. Если в модели есть строка
`define assert-on то в модель включается текст блока always.
`ifdef assert_on
            always @(set or reset) begin
                 if (set && reset)
                 $display ("assert set && reset, %b, %b, time= %t",
                              set, reset, $time);
            end
 `endif
```

Второй вариант организации VERILOG-утверждений — оформление соответствующего утверждения в виде процедуры (task) или модуля, включение его в библиотеку исходных модулей (процедур) и использование в тексте модели.

Пример такого модуля:

```
`define assert_on  1
 module assert_set_reset (clk, set, reset, assert_id);
 input clk, set, reset;
 input [7:0] assert_id;
    `ifdef assert_on
        always @(posedge clk) begin
           if (set  && reset)
              $display
    ("assert set_reset, set = %b, reset = %b, assert_id = %d, time = %t",
                                  set, reset, assert_id, $time);
           end
    `endif
 endmodule
```

Рассмотрим пример использования подобного VERILOG-модуля в тексте описания устройства. Из-за соображений эффективности контроль включается только при определении (`define assert_on) переменной периода компиляции assert_on, а для пропуска утверждения синтезатором (см. главу 5) применяются прагмы-комментарии rtl_synthesis on и off.

```
// rtl_synthesis off
 `ifdef assert_on
  assert_set_reset   ff1_assert (clk, set, reset, 1);
  `endif
 // rtl_synthesis on
```

В библиотеке модулей-утверждений (ряд фирм поставляет библиотеки из 100—300 подобных модулей) обязательно имеются утверждения типа «всегда», «никогда», «при определенных условиях», «только один разряд» и т. п. Система OpenVera 2 (www.open_vera.com) включает специальный язык утверждений и верификационные среды на этой базе.

4.4. Компоненты тестирующей программы

В типичной тестирующей программе (testbench) мы обычно имеем:
— генератор тактовых сигналов (clk);
— генератор сигнала сброса (reset);
— генератор (источник) векторов тестовых данных (test-vectors);
— вызов модели тестируемого устройства (Unit Under Test-UUT);
— компаратор — сравнение выходов устройства с эталоном;
— диагностическую печать (if (debug =1) $display(););
— проверку контрольных соотношений (assert);
— описание условий останова модели и печать итоговых сообщений.

При необходимости в этот список включаются: сохранение результатов моделирования в файле, средства реконфигурации модели для регрессионных экспериментов и т. п. Развитые системы создания верификационных сред типа SPECMAN фирмы VERYSITY включают библиотеки сложных типовых верификационных компонент (например, шины PCI, Gigabit Ethernet и др.).

Ниже приведены примеры типичных простых компонент тестовых программ.

4.4.1. Тактовый генератор

В разделе дан пример генератора сигналов с периодом 10 ns и неравной длительностью сигнала 1 и 0:

VHDL

```
signal Y: bit:='0';
GEN1:process
begin
    wait for 3 ns;
   Y<= '1';
   wait for 7 ns;
   Y<= '0';
end process;
```

VERILOG

```
`timescale 1 ns/1 ns
  reg Y; initial Y=1'b0;
 always
   begin: GEN1
    #3;
    Y<=1;
    #7;
    Y<=0;
end //GEN1
```

Пример типичной ошибки при описании генератора (программа зациклится, так как процесс не содержит оператора задержки):

VHDL

```
WRONG_GEN1:process
   begin
  Y<= '1' after 3 ns,
         '0'  after 7 ns;
end process WRONG_GEN1;
```

VERILOG

```
always
   begin: WRONG_GEN1
    Y<= #3 1;
    Y<=#7 0;
end // WRONG_GEN1
```

4.4.2. Генератор сигнала сброса

Пример генератора сигналов сброса длительностью 10 тактов:

VHDL

```
signal RST: bit:='1';

r1:process  variable I:integer;
  begin
   for I in 1 to 10 loop
 wait until clk='1' and clk'event;
  end loop;
  RST<= '0'; wait;
end process;
```

VERILOG

```
reg RST;
 initial begin   RST=1'b1;
 repeat( 10) @(posedge clk);

 RST<=0;


end
```

4.4.3. Входные векторы

Источник входных воздействий на тестируемый объект может быть реализован в виде алгоритма (детерминированный или псевдослучайный) или набора данных, считываемых из файла. Работа с файлом существеннее медленнее.

4.4.3.1. Случайные входные наборы и задержки

VERILOG

Наиболее просто случайные процессы реализуются в языке VERILOG, имеющем встроенный датчик равномерного случайного распределения 32-разрядных случайных чисел — системную функцию $random.

Пример генерации случайных адресов и данных длиной до 64 bit:

```
always @(posedge clk) begin
     data ={$random, $random); addr={$random, $random);
end
```

Пример случайной задержки данных:

```
initial  begin
    #($random % Max_del)  data =($random);
end
```

VHDL

Случайные функции VHDL реализованы в пакетах.

Приводим фрагмент использования одного из них - DISTRIBUTION.

Пример генерации случайных вещественных адресов (0:255) и данных (0:1023) процедурой uniform (потом их надо преобразовать в целые):

```
Library Synopsys; use Synopsys.distribution.all;
-- в теле архитектуры--------------------
Rnd:process (clk)
 Variable NN:real:= 20.0;
 begin
if (clk'event and clk ='1')then
     Uniform(NN, 0.0,1023.0,  data);Uniform (NN,0.0,255.0,addr);
end if;
```

4.4.3.2. Входные наборы из файла

VERILOG

```
Integer I;
reg  [width-1:0]pat_mem[0:depth-1];//буферная память векторов
initial begin
readmemh ("test_pat.dat", pat_mem);//считывание из файла в буфер
  for (I=0;  I< Max_pat_num;  I=I+1) begin @(posedge clk);
  #(pat_del) in_vec=pat_mem[I];// подача вектора на входы
end
end
```

VHDL

Пример см. в разделее 4.6.

4.4.4. Сравнение выходов модели с эталоном (VERILOG)

Для сравнения в условии оператора if следует использовать операцию ===(тождественно), а не ==(равенство), так как, например:

```
I1= ( 4'b10xz !=4'b1011 );  // дает 1'bx (не истинно)
I2=  (4'b10xz  ==4'b1011)  ;//дает 1'bx (не истинно)
I3=   (4'b10xz ===4'b1011);//  дает 1'b0  (ложно)
I4=    (4'b10xz !==4'b1011);//  дает 1'b1  (истинно)
```

4.5. Быстродействие и расход памяти инструментальной ЭВМ

Кроме полноты тестирования, одними из важных характеристик тестирующей программы являются **требуемая память** и **расход машинного времени**.

4.5.1. Расход памяти

Расход памяти зависит от числа данных в программе, их размера (особенно больших векторов и массивов), а также количества и сложности операторов.

Например, объем памяти инструментальной ЭВМ, необходимый для хранения VHDL-данных вида signal, на порядок превышает объем для variable, и объявлять модель блока памяти лучше с помощью variable.

Пример фрагмента упрощенной модели асинхронной RAM (адрес — целое!).

```
entity OZU is
  generic (width:positive:=10;    depth:positive := 1024);
    port (rw,CS:in bit;
            DI,Do:inout bit_vector(width-1 downto 0);  addr:in integer
            );
 end;
architecture BEH of OZU is
 type mem_arr is array (0: depth-1) of bit_vector( width-1 downto 0);
  begin
   process read_write(rw,CS,DI)
    variable Ram:mem_arr;
      begin
        if CS='1' then
          if rw='1' then   Do<=Ram(addr);
          else  Ram(addr) :=DI;
          end if;
        end if;
   end process;
  end;
```

VERILOG расходует обычно в 4 раза меньше памяти на хранение одного бита данных, чем VHDL. Данные вида переменные в этом смысле всегда лучше, чем вида сигналы и соединения.

4.5.2. Быстродействие тестирующей программы

Уменьшить время модельного эксперимента можно различными способами, среди которых:

1. Эффективная организация тестирующей программы (**профилирование** — выявление наиболее часто исполняемых фрагментов модели и их оптимизация, исключение ненужных печатей и т. п.).

2. Использование систем поддержки верификации и языков программирования для создания верификационной среды (это часто не только сокращает время разработки и отладки теста, но и повышает его быстродействие).

3. Выбор более быстрой системы моделирования (примененный вместо интерпретатора компилятор повышает быстродействие в 5—10 раз).

4. Использование более мощной инструментальной ЭВМ (недостаток оперативной памяти приводит к свопингу, практически останавливающему моделирование).

5. Разумное ограничение множества входных наборов теста (ничего нет мощнее здравого смысла).

6. Использование аппаратных ускорителей моделирования (повышение скорости порой на два-три порядка).

7. Использование прототипов схемы для прогона тестов на них (повышение скорости на три-четыре порядка).

На этапе отладки теста основное время тратится на вывод результатов и временных диаграмм сигналов, а в период прогона длинных — регрессионных тестов на воспроизведение событий в моделируемом объекте, и в первую очередь зависящих от тактового сигнала.

Например, три отдельных процесса требуют больше машинного времени, чем те же действия, объединенные в один процесс.

VHDL	**VERILOG**

P1: Process (clk) begin
if clk'event and clk='1' then
a<= a+1;
end if;
end process;
	always @(posedge clk)
	a= a+1;

P2: Process (clk) begin
if clk'event and clk='1'then
b<= b and c;
end if;
end process;
	always @(posedge clk)
	b= b & c;

P3: Process (clk) begin
if clk'event and clk='1' then
d<= e or f;
end if;
end process;
	always @(posedge clk)
	d= e | f;

Ниже дан объединенный процесс----------	

P : Process (clk) begin
if clk'event and clk='1' then
a<= a+1;
b<= b and c;
d<= e or f;
end if;
end process;
	always @(posedge clk)begin
	a= a+1;
	b= b & c;
	d= e | f;
	end

Использование языков программирования (C,C++ и т. п.) для написания фрагментов теста позволяет в несколько раз снизить расход памяти и машинного времени, однако это требует соответствующей квалификации и предварительных усилий.

Еще больший эффект при массовом использовании дают специализированные системы автоматизации разработки и прогона тестов.

Такие системы, как VERA (фирмы Synopsys, базируется на VERILOG и C++), SPECMAN (фирмы Verisity, специальный язык E), Testbilder (фирмы Cadence),

Quick Bench (фирмы Forte, графический ввод временных диаграмм теста), включают развитые объектно-ориентированные языки, наборы типовых модулей и функций, развитые средства отладки и мониторинга.

Фирма Axis (www.axis.com) выпускает специализированные ЭВМ и программное обеспечение, которые аппаратно реализуют HDL-модель исследуемой системы и тестовую программу. HDL-описание транслируется в коды настройки массивов ПЛИС типа FPGA специализированной ЭВМ. Аппаратная поддержка на несколько порядков (до 10 000 раз) повышает скорость моделирования. С возможностями FPGA читатель может ознакомиться в главе 6 и в литературе [29—31].

4.6. Отладка тестирующей программы

4.6.1. Порядок отладки

Отладка начинается с этапа компиляции модулей проекта и теста. Напомним, что VHDL подразумевает строгий порядок компиляции — сначала первичные модули — описание объекта (entity), описание пакета (package), описание конфигурации (configuration), а затем описание архитектуры (architecture) и тела пакета (package body). VERILOG не так строг.

Компилятор выдает сообщение о синтаксических ошибках (типа лишней скобки или пропущенной запятой) и семантических ошибках (типа дважды объявленного идентификатора), сопровождаемое номером строки исходного текста и кодом ошибки. По такому сообщению пользователь находит место в исходном тексте и вносит соответствующие исправления, не забывая сохранить (save) текст в файле перед повторной компиляцией. Хорошим подспорьем на этом этапе является прогон синтезабельных модулей проекта через reuse-checker'ы, осуществляющие дополнительный статический контроль описаний.

Как только все модули прошли безошибочную компиляцию, они оказываются в рабочей библиотеке проекта (work) и осуществляется их сборка (elaboration). Редактор связей выдает сообщения об ошибках типа отсутствия некоторого модуля в рабочей библиотеке или несоответствия числа портов или их размера в модуле и его конкретизации. После устранения этих ошибок можно приступать к отладке динамики работы модели.

Общий подход здесь такой же, как и при отладке аппаратуры, — от простого к сложному, от ядра системы к его окружению, от входов к выходам и т. п.

Возможность использования отладочной печати, наблюдения временных диаграмм любых внутренних сигналов, методов интерактивной, пошаговой работы модели существенно облегчает этот процесс.

Если самопроверяющийся тест сразу не пошел («а вдруг случится чудо?»), то при моделировании используется программа-просмотрщик временных диаграмм (VIEWER). Обычно пользователем сначала проверяется работа тактового генератора (изменяются ли тактовые сигналы (clk) в тесте, проходят ли они внутрь модели объекта) и генератора сигнала начального сброса (reset).

Если с ними все в порядке, просматривается влияние сигнала сброса на систему — не остались ли триггеры и регистры в неопределенном состоянии ('u' — VHDL, 1'bx в VERILOG). После этого просматривается прохождение первого простейшего тестового вектора данных через систему, затрагивающего минимальную часть оборудования (ядро), затем следующего и т. д.

К числу относительно тяжело диагностируемых ошибок относится, например, конфликт на общей шине (рис. 4.6).

Рис. 4.6. Пример общей шины D, связывающей два модуля M и тест

Фрагмент HDL-тестирующей программы. Пример содержит ошибку — объект проекта M выдает значение D, когда на шине новые данные из теста в течение t_data.

VHDL

```
signal D:std_logic:='Z';
 -- Ниже тестовый сигнал на шину
      D<=DATA(i)when clk='0'

            else 'Z';
-- ниже два экземпляра объекта проекта M
K1: M port map(clk,    D);
K2: M port map(clk,    D);
----------------------------------------------------------------
entity M is
generic(t_data:time:=5 ns);
port (clk: in std_logic;
      D: inout std_logic);
end;
architecture BEH of M is
 begin
  D<= not D after t_data when clk='1'
        else 'Z' after t_data;
end;
```

VERILOG

```
wire  D;

assignD<=(clk==1'b0)?
          DATA[I]:
           1'bz;

M K1(clk,    D);
M K2(clk,    D);
----------------------------------------------------------------
module M(clk,D);

input clk;
inout D;
parameter t_data=5;

assign #(t_data) D=
(clk==1'b1)?~D:1'bz;
endmodule
```

Если состояние шины D неопределенно (x), а из теста на нее подается определенное значение, не равное Z, приходится предполагать неотключение выхода одного из модулей (K1, K2), например, из-за большой задержки t_data.

В таком случае полезны специальные средства обнаружения конфликтов на шине — компонент программы-просмотрщика временных диаграмм.

Как только простейший короткий (несколько тактов) тест прошел, прогоняются более длинные тесты и тесты, заставляющие срабатывать ядро в более сложных режимах, а также ранее не функционировавшую часть модели и т. п.

Затем подаются тесты проверки граничных условий, затем идет прогон случайных наборов данных и регрессионные тесты. По мере усложнения самопроверяющихся тестов и увеличения времени их прогона пользователь уменьшает объем отладочной печати и отказывается от просмотрщика временных диаграмм, тем самым резко увеличивая скорость моделирования.

4.6.2. Общие рекомендации

Отладочная печать

Сообщение отладочной печати должно содержать время, место выдачи сообщения, имена данных и их значения. Например, при отладке некоторого арифметического блока следует в начале печатать значения входных данных, а затем — выходных.

Анализ временных диаграмм

Указав системе моделирования множество сигналов, для которых следует накапливать временную диаграмму, пользователь после останова модели может прослеживать их временные диаграммы, используя средства поиска определенных событий, автоматического сравнения сигналов и т. п. Одна из типичных техник поиска ошибок в проекте — **обратный просмотр (back tracking)** временной диаграммы. Некоторый сигнал, допустим A, в момент Tn принял неверное значение, например X. Прослеживаем (используя исходное HDL-описание), в каком блоке формируется значение A. Допустим, это блок BB с входными сигналами C и D. Проверяем по диаграмме значения сигналов C и D в моменты времени, непосредственно предшествовавшие Tn. Если один из них в этот момент равен X, он является предполагаемым источником ошибки, и весь процесс обратного просмотра следует повторить для него.

Интерактивный режим отладки

В самых тяжелых случаях, когда более быстрые (в смысле затрат машинного времени) способы отладки не помогают, применяется интерактивный способ отладки. Пошаговый способ (step) — после каждого оператора модель останавливается с отметкой места останова в исходном тексте и возможностью отследить значения сигналов.

Учитывая возможность параллельного развития в модели множества процессов, иногда непросто отследить состояние одного из них. Более быстрый способ интерактивной отладки использует останов модели по контрольным точкам, предварительно указанным пользователем.

Примеры VERILOG-описаний повторяющихся отладочных остановов в тесте:

```
forever #10 $stop;// останов каждые 10 интервалов времени
forever repeat(10) @(posedge clk) $stop; //останов каждые 10 тактов
```

4.7. Автоматизация построения тестирующих программ

На примере VHDL-модели объекта проекта F имеем такое объявление интерфейса объекта проекта F:

```
entity F is
  port (A1, A2:in bit; B1, B2: out bit);
end F;
```

Подавая это описание на вход подсистемы генерации тестирующих программ системы моделирования ACTIVE — HDL, получаем следующую заготовку (ее легко получить и вручную простым троекратным копированием описания интерфейса объекта проекта и заменой в первой копии entity на component, во второй оставив только описания сигналов, а в третьей — только имена сигналов в карте портов):

```
-----------------------------------------------------------------------------
-- Title       : Test Bench for f
-- Design      : F1-- Author      : AKP -- Company     : MPEI
-----------------------------------------------------------------------------
-- Description : Automatically generated Test Bench for f_tb
-----------------------------------------------------------------------------
      -- Add your library and packages declaration here ...
entity f_tb is
end f_tb;
architecture TB_ARCHITECTURE of f_tb is
      -- Component declaration of the tested unit
      component f
      port(
            A1 : in BIT;A2 : in BIT;
            B1 : out BIT;B2 : out BIT);
      end component;
  -- Stimulus signals - signals mapped to the input and inout ports of tested entity
  signal A1 : BIT;  signal A2 : BIT;
  -- Observed signals - signals mapped to the output ports of tested entity
  signal B1 : BIT;signal B2 : BIT;
  -- Add your code here ...
begin
  -- Unit Under Test port map
  UUT : f
        port map (
              A1 => A1,A2 => A2,
              B1 => B1,  B2 => B2
             );
  -- Add your stimulus here ...

end TB_ARCHITECTURE;
configuration TESTBENCH_FOR_f of f_tb is
  for TB_ARCHITECTURE
        for UUT : f use entity work.f(f_behavior);
        end for;
  end for;
end TESTBENCH_FOR_f;
```

После включения в эту заготовку описания процесса генерации входных наборов получим:

```
entity f_tb is
end f_tb;
architecture TB_ARCHITECTURE of f_tb is
  -- Component declaration of the tested unit
  component f
  port(
        A1 : in BIT;A2 : in BIT;
        B1 : out BIT;      B2 : out BIT );
```

```
  end component;
  -- Stimulus signals - signals mapped to the input and inout ports of tested entity
  signal A1 : BIT;signal A2 : BIT;
  -- Observed signals - signals mapped to the output ports of tested entity
  signal B1 : BIT;signal B2 : BIT;
  -- Add your code here ...

begin
  -- Unit Under Test port map
  UUT : f
        port map (
              A1 => A1,A2 => A2,
              B1 => B1,  B2 => B2
           );
  -- Add your stimulus here ...
   GEN_V1 :process--
         begin--
         A1<='0'; A2<='0' ;
          wait for 10 ns;  A1<='1';
          wait for 20 ns;   A1<=  '0' ; A2<='1';
          wait for 10 ns;  A1<='1' ;
           wait for 10 ns;    A1<='0'; A2<='0' ;
         wait;
end process GEN_V1;
end TB_ARCHITECTURE;
```

Еще больший эффект дает использование в тестовой программе верифицированных типовых компонент, например, моделей шин PCI, USB, RAM, памяти FIFO и т. п.

4.8. Структурированный тест объекта проекта F

Допустим, архитектура структурного теста будет названа нами TB_F_STRUCT и будет описываться структурно, как схема, состоящая из компонент GENSIG, F и WR.

4.8.1. Генератор сигналов GEN

VHDL

```
entity GENSIG is
    generic (T_C :time :=10 ns;
       T_STOP :time:=50 ns);
   port (Y1, Y2: out bit);
end GENSIG;

architecture GEN_V1 of GENSIG is
     begin
        Y1<='0' after 0 ns, '1' after T_C,
         '0' after T_C *3, '1' after T_C*4;
```

VERILOG

```
`timescale  1 ns/ 100 ps
module  GEN_V1 (Y1, Y2);


output Y1, Y2; reg Y1,Y2;
parameter  T_C=10;
parameter T_STOP=50 ;
 initial begin
 Y1=0;Y2=0;
   #(T_C); Y1=1;
   #(T_C * 2);
 Y1=0;Y2=1;
```

```
    Y2<='0' after 0 ns,  '1' after T_C *3 ns;
    STOP: process  begin
              wait for T_STOP;
              wait;
    end process STOP;
end GEN_V1;
```

```
  #(T_C); Y1=1;
 end //initial for signals
 initial begin
  #(T_STOP) $finish;
 end // process STOP
endmodule // GEN_V1
```

VHDL-архитектура объекта проекта GENSIG описана в потоковой форме. Параллельные операторы назначения сигнала задают временную диаграмму сигналов Y1, Y2, соответствующую рис. 4.3.

Объект проекта GENSIG имеет два выхода: Y1 и Y2. Вектор выходных сигналов пробегает значения от «00» до «11» в течение 4 * T_C ns.

Процесс STOP останавливает моделирование через время T_STOP. Нетрудно заметить, что Y1 — это периодический сигнал, и для иллюстрации способов описания периодических генераторов приводится вариант GEN_V2:

VHDL

```
architecture GEN_V2 of GENSIG is
   begin
   CLK_gen:  process  begin
             Y1<='0'; wait for T_C;
             Y1<='1'; wait for 2*T_C;
       end process CLK_gen;
   Y2<='0' after 0 ns,  '1' after T_C *3 ns;

end GEN_V2;
```

VERILOG

```
`timescale 1 ns /100 ps
module  GEN_V2 (Y1, Y2);
   output Y1, Y2;
   reg Y1. Y2;
   parameter  T_C=10;
   parameter T_STOP=50 ;
   initial forever begin
        Y1=0;
        #(T_C);
       Y1=1;
        #(T_C * 2);
   end// CLK gen
  initial begin  Y2=0;
       #(T_C*3); Y2=1;
   end

 initial begin
       #(T_STOP) $finish;
 end // process STOP
endmodule // GEN_V2
```

4.8.2. Регистратор сигналов WRITER

Вывод результатов моделирования производит объект проекта WRITER. Описание регистрирующего объекта WRITER может быть таким:

VHDL

```
entity WRITER is
  port (X1, X2: in bit);
end WRITER;
-- объект WRITER не имеет выходов
-- ниже описание его архитектуры
 use STD.TEXTIO.ALL;
-- подключается    пакет  TEXTIO
```

VERILOG

```
module WRITER_1 (X1, X2);
input (X1, X2: in bit);
```

```
architecture WRITER_1 of WRITER is
      begin          process
       variable L:LINE;
-- тип LINE из пакета TEXTIO
          begin
           wait on  X1, X2;                    always  @ (X1 or X2)begin
           write (L,NOW);                      $display(t=%t X1=%b",$time,X1);
                                               $display("X2=%b",X2);
           write (L,X1);write(L,X2);           end
           writeline  (output,L);
         end process;
end WRITER_1;                                  endmodule // WRITER_1
```

Архитектура объекта проекта WRITER представлена на поведенческом уровне и оформлена как процесс.

В VHDL-описании архитектуры WRITER_1 объекта WRITER использован стандартный пакет TEXTIO, содержащий описание текстового файла OUTPUT и стандартную процедуру записи строки WRITELINE. Каждый раз, когда один из сигналов (X1 или X2) изменяет свое значение, процесс запускается и выполняет оператор записи значений модельного времени NOW и значений сигналов X1 и X2 в стандартный текстовый файл OUTPUT.

4.8.3. Архитектура теста — структурное описание

Вариант 1

Описание интерфейса и архитектуры объекта проекта TB_F (см. рис. 4.2), состоящего из прямых порождений (VHDL) экземпляров объектов GENSIG, F, WRITER с именами конкретизаций C1, C2 и C3:

VHDL **VERILOG**

```
architecture TB_F_STRUCT of TB_F is          module TB_F_STRUCT();
   signal I1, I2, O1, O2: bit;                wire I1, I2, O1, O2;
-- внутренние сигналы для общности поименованы иначе, чем прежде
   begin
-- ниже связи компонент
       C1:entity work.GENSIG(GEN_V1)
          generic map( 10 ns,50 ns)           GEN_V1 #(10,50)
                port map (I1, I2);                   C1 (I1,I2);
       C2: entity work.F(F_CASE)
            port map (I1,  I2,  O1,  O2);     F_CASE     C2 (I1,  I2,
                                                            O1, O2);
       C3: entity work.WRITER(WRITER_1)
                port map (O1, O2);            WRITER_1    C3 (O1, O2);
end ; --TB_F_STRUCT                          endmodule //TB_F_STRUCT
```

Вариант 2 (VHDL)

Описание интерфейса и архитектуры объекта проекта MODEL_F (см. рис. 3.1), состоящего из компонент GEN, WRI и F1:

```
architecture TB_F_STR_COMP of TB_F is
   component GEN
    generic (T_C :time :=10 ns;
```

```
                T_STOP : time:=50 ns);
    port (X1, X2: out bit);
    -- генератор сигналов
    end component;
    component WRI port (Y1, Y2: in bit); -- регистратор выходных сигналов
    end component;
    component F1 port (X1, X2: in bit; --  компонента F1 соответствует F
                  Y1, Y2: out bit);
    end component;
    signal I1, I2, O1, O2: bit; -- внутренние сигналы
    begin
        -- ниже связи компонент
        C1: GEN generic map( 10 ns,50 ns)
                  port map (I1, I2);
        C2: F1 port map (I1, I2, O1, O2);

        C3: WRI port map (O1, O2);
end ; --TB_F_STR_COMP
```

Структурное VHDL-описание объекта TB_F с архитектурой TB_F_STR_COMP должно быть дополнено конфигурационным описанием, так как названия компонентов не совпадают с именами объектов проекта. Объявление конфигурации TB_F_config отождествляет компоненту GEN с объектом GENSIG и его архитектурой GEN_V1, компонент WRI с объектом WRITER и его архитектурой WRITER_1 и компоненту F1 с объектом F и его архитектурой F_B, хранящихся в рабочей библиотеке проекта WORK.

```
Library WORK; use WORK.ALL;
configuration TB_F_config of TB_F is
  for TB_F_STRUCT
    for C1:GEN
      use entity GENSIG (GEN_V1) port map (Y1=>X1, Y2=>X2);
    end for;
    for C2: F1
      use entity F (F_CASE) port map (A1=>X1, A2=>X2,
                                      B1=>Y1, B2=>Y2);
    end for;--C2
    for C3: WRI
      use entity WRITER (WRITER_1) port map (X1=>Y1, X2=>Y2);
    end for;--C3
  end for;
end TB_F_config ;
```

Упражнения

1. Измените генератор входных сигналов, сделав сигнал Y периодическим с периодом 40 ns.

2. Организуйте наблюдение за всеми сигналами в модели.

4.9. Модельный эксперимент с самопроверкой

Модельный эксперимент иногда может упроститься, если тест (входные воздействия) задавать не в виде констант в операторах назначения, а считывать их значения из файла или описать алгоритм, генерирующий эти воздействия.

Что касается анализа результатов моделирования, то тест должен быть самопроверяющимся. Для этого в него в нашем примере надо ввести сравнение выходов объекта проекта F с эталоном, а также использовать операторы утверждений, анализирующие возникновение запрещенных режимов работы исследуемого объекта.

4.9.1. VHDL-вариант

В качестве примера ниже приводится VHDL-вариант TB_F_ETALON описания архитектуры объекта проекта TB_F, в котором входные воздействия (тест) и эталонные значения выходов объекта проекта F (см. табл. рис. 4.1) считываются из стандартного устройства INPUT и осуществляется сравнение выходных сигналов объекта проекта F, отображаемого архитектурой F_B с эталоном. Архитектура F_B_TIME соответствует схеме, описанной с учетом задержек (см. гл. 3).

В связи с тем что сравнение имеет смысл только после того, как оканчиваются переходные процессы в схеме объекта проекта F, в модель введена задержка сравнения выходов F_B с эталоном относительно момента подачи входных воздействий (задержка строба), равная 30 ns. Периодический синхросигнал S (период 40 ns) служит для привязки процессов чтения тестовых векторов и сравнения выходов с эталоном.

```
LIBRARY WORK, STD;
use STD.TEXTIO.all;
architecture TB_F_ETALON of TB_F is
    signal I1, I2, O1, O2: bit;
    signal S: bit:='0';
    constant  T_synchro: TIME:=40 ns;
  component F1 is port (X1, X2: in bit; Y1, Y2:out bit);
  end component;
        -- ниже спецификация конфигурации компоненты F1
        for C2: F1 use entity F(F_B_TIME) port map
            (A1=>X1, A2=>X2, B1=>Y1,B2=>Y2);
      --описание архитектуры F_B_TIME см. в гл.3.
      begin
    -- ниже генератор синхросигнала стробирования S
      S<=transport not(S) after T_synchro;
 -- ниже считывание входных воздействий с клавиатуры по фронту сигнала S
      GEN:process(S)
       variable L:LINE; variable I1_V,I2_V:bit;
          begin
            if S='1' and S'EVENT   then          -- по фронту S
                      readline (INPUT, L); read(L,I1_V); read(L,I2_V);
              I1<=I1_V; I2<=I2_V
            end if;
      end process GEN;
```

```
    -- ниже конкретизирована компонента F1
      C2: F1 port map(I1, I2, O1, O2);
    -- ниже анализ результатов сравнением с эталоном
      compare: process
       variable L:LINE; variable K1, K2: bit;
        begin wait on S;
         if S='1' and S'EVENT  then
          wait for 30 ns;
          readline( INPUT, L);read(L,K1);read(L,K2);
     -- считываются эталонные значения выходов
            if (K1/=O1) or (K2/=O2) then
            write(L,"несовпадение с эталоном");
            write(L,K1);write(L,K2);write(L,O1);write(L,O2);
            write(L,"на тесте");write(L,I1);write(L,I2);write(L,NOW);
            writeline(output, L);
            end if;
          end if;
      end process compare;
end TB_F_ETALON;
```

4.9.2. VERILOG-вариант

В VERILOG-варианте использован дополнительный модуль COMPARATOR, осуществляющий сравнение выходов объекта проекта F (вариант F_CASE) с эталоном, который предварительно считывается из входного файла f_table в массив. При каждом изменении входных сигналов объекта проекта F через время задержки T_STROBE, превышающей задержку в схеме F, компаратор сравнивает выход F с эталоном (обратите внимание на отличия этого варианта организации сравнения с эталоном от VHDL-варианта).

```
`timescale 1 ns /100 ps
module  COMPARATOR (X1,X2,Y1, Y2);
input X1,X2,Y1, Y2;
parameter T_STROBE= 9;//задержка строба = 9 ns
reg [1:0] mem[1:4]; // массив для хранения эталона
reg  [1:0]tmp;// для хранения очередного вектора эталона
integer i;initial i=0;//индекс очередного эталонного вектора
initial $readmemb("f_table.v",mem);//read file
always @(X1 or X2)begin
     #(T_STROBE);
    T= $time;i=i+1;tmp=mem[i];
     $display("i=%0d",i);
    if ({Y1,Y2}!== tmp)
     $display ("miscomparing, t =%0t Y1=%b  Y2=%b, expected= %b",
                  $time,Y1,Y2,tmp);
  end
```

Теперь VERILOG-описание модуля тестирующей программы, осуществляющей сравнение выходов объекта проекта F с эталоном выглядит так:

Дополнительным параметром может быть уровень оптимизации, т. е. уровень усилий синтезатора: чем он выше, тем больше требуется затрат машинного времени, но и тем лучше получаемая схема. На начальных стадиях синтеза обычно используется минимальный уровень усилий синтезатора, на конечных — максимальный.

Мощные синтезаторы типа Synopsys Design Compiler обеспечивают задание большего числа пользовательских параметров, включая даже форму тактовых импульсов, свойства внутренних проводников (wire load) и т. п. Другие синтезаторы, упрощая работу пользователя или ориентируясь на определенный тип элементной базы, большинство параметров задают по умолчанию.

Например, системе Synplify, предназначенной для проектирования устройств на программируемых интегральных схемах (ПЛИС — FPGA, CPLD), достаточно в диалоговом режиме указать семейство FPGA, например XILINX XC4000E, тип кристалла, например XC4013E, тип корпуса, например PC84, группу, определяющую его скорость, например 1, тактовую частоту — требуемое быстродействие проекта, — например 30 МГц, максимальную нагрузку на выходе, например 10, и запустить синтезатор.

При необходимости дополнительно задаются задержки запаздывающих входных сигналов, указания по синтезу конвейерной (pipeline) или обычной схемы, группировке или разгруппировке (flattening logic) отдельных фрагментов проекта, использованию системы синтеза автоматов (FSM compiler), способу кодировки состояний автомата и т. п.

Порядок работы программы-синтезатора

Сначала синтезатор осуществляет синтаксический контроль HDL-описаний. Хотя после моделирования сообщения о грубых синтаксических ошибках невероятны, но сообщения о запрещенных с точки зрения синтезабельности (см. ниже) HDL-конструкциях и неполноте списков чувствительности процессов относятся к типичным.

Если ошибок нет, синтезатор генерирует отчет о работе, включающий предупреждения о появлении в итоговой схеме триггеров-защелок, параметры полученной схемы (предельную частоту и задержки критического пути, число триггеров и т. п.). Синтезатор строит также тексты структурных описаний схем (форматы HDL (см. ниже примеры), EDIF, XNJ или др.). Программа просмотра полученной схемы позволяет видеть ее условное графическое изображение (в образах базовых узлов (RTL view) и после покрытия заданной системой элементов (в технологическом базисе). Просматривая отчет синтезатора, пользователь порой находит такие неприятные сообщения, как появление непредвиденных триггеров-защелок (latch), критических путей с задержкой больше предусмотренной (отрицательный запас по времени — slack) и т. п. Учитывая, что после трассировки величины задержек в схеме обычно возрастают на 10—20% (при заполненности кристаллов FPGA более 80%, трассировка часто увеличивает задержки на 30%), проектировщику порой приходится вносить большие изменения в исходный HDL-текст и, жертвуя оборудованием, вводить в проект конвейерные и параллельные структуры. Новейшие системы синтеза (MAGMA и др.), ориентированные на проектирование сверхбыстродействующих схем, в которых задержки проводников сопоставимы или превышают задержки элементов ведут синтез с учетом результатов работы встроенной в них подсистемы предварительной трассировки.

Пример синтеза схемы объекта проекта F, рассмотренного в главе 3 и тестированного в главе 4.

```
        $display ("miscomparing, t =%0t B1=%b  B1_S=%b, B2=%b  B2_S=%b",
                  $time,B1,B1_S, B2,B2_S);
    end
  endmodule //TB_F_COMP
```

Вопросы

1. Зачем в тестовую программу на VHDL введен синхросигнал?
2. Зачем в компаратор VERILOG-модели (модуль COMPARATOR) введена задержка сравнения выходов с эталоном?

4.10. VHDL-модель и простой тест микросхемы памяти

4.10.1. Микросхема К134РУ6

Методика описания асинхронных микросхем ОЗУ и ПЗУ иллюстрируются на примере микросхемы К134РУ6, которая является одной из типичных микросхем малой степени интеграции, используемых при проектировании полупроводниковых запоминающих устройств (ЗУ).

Ее условное графическое изображение и таблица функционирования представлены на рис. 4.7, а временная диаграмма работы — на рис. 4.8.

Информационная емкость микросхемы 1024 бита. Максимальное время выборки относительно сигнала адреса (ta) не более 700 ns, а время выборки микросхемы (tcs) 500 ns. Минимальные времена: установления сигнала выборки микросхемы относительно адреса tsu_a_cs=200 ns; установления сигнала записи относительно CS — tsu_cs_wr=30 ns; установления CS относительно входных данных tsu_di_cs=200 ns; сохранения сигнала адреса после сигнала CS tv_cs_a=250 ns; сохранения данных после сигнала CS tv_cs_di=250 ns; сохранения сигнала записи после сигнала CS tv_cs_wr=0; длительность сигнала CS tw_cs=550 ns; длительность интервала между сигналами CS trec_cs=450 ns; длительность цикла tcy_a=1000 ns.

Ниже приводится обобщенная VHDL-модель микросхемы памяти, подобной К134РУ6, но с информационной емкостью в N слов и разрядностью K. Это объект проекта с именем SK134RU6 и параметрами настройки N и K.

CS	WR/RD	A0 - A9	DI	DO	Режим работы
1	X	X	X	1	Хранение
0	0	A	0	1	Запись 0
0	0	A	1	1	Запись 1
0	1	A	X	**	Считывание

Рис. 4.7. Микросхема К134РУ6 — условное графическое изображение и таблица функционирования: A — адрес; X — произвольное значение; ** — данные в прямом коде

Рис. 4.8. Временная диаграмма работы микросхемы К134РУ6. Время t1 на диаграмме соответствует времени tv_cs_a

4.10.2. Описание интерфейса микросхемы

В объявлении интерфейса объекта SK134RU6 количество слов (N) и разрядность слова (K) указаны как параметры настройки, что позволяет описывать аналогичные микросхемы произвольной емкости. Параметрически также заданы и временные характеристики.

В текст введены утверждения, контролирующие некоторые временные соотношения для сигналов в микросхеме.

```
entity SK134RU6 is
    generic (N: integer:=10;          -- размерность шины адреса
             K: integer:=1;           -- размерность шины данных
      ta, tcs, tsu_a_cs, tsu_wr_cs, tsu_di_cs, tv_cs_a, tv_cs_di, tv_cs_wr,
      tw_cs, trec_cs, tcy_a: time);
        port (A: in bit_vector (0 to N-1);
              DI: in bit_vector (0 to K-1); CS, WRRD: in bit;
              DO: out bit_vector (0 to K-1));
      begin
        assert                        -- параллельный оператор утверждения
          tw_cs>550 ns  -- может располагаться и в описании интерфейса
        report "параметр длительности сигнала CS меньше 550 ns"
          severity warning;
        assert tv_cs_di>250 ns
        report "параметр tv_cs_di меньше 250 ns"
          severity error;
end SK134RU6;
```

4.10.3. Архитектура объекта SK134RU6

Ниже приводится упрощенный вариант описания процесса функционирования микросхемы (работа по CS). Он соответствует идеализированной диаграмме, представленной на рис. 4.8, и не отражает ряд возможных ситуаций, когда, например, работа микросхемы инициируется не срезом сигнала выборки микросхемы (CS), а адресным (A) сигналом или сигналом чтение-запись (WRRD). Кроме того, используемый двоичный алфавит не позволяет учитывать возможность записи и чтения из памяти неопределенных значений сигналов, например, из-за начальной неустановки или при нарушении временных соотношений в управляющих воздействиях и т. д. Кроме того, не отражено, что выход D0=1 во всех решениях, кроме чтения.

В теле архитектуры описана функция преобразования битовой строки адреса в целочисленное значение, необходимое для выборки элемента массива, отображающего массив ячеек памяти. Обычно такие функции включаются в соответствующие пакеты.

```
architecture SIMPLE of SK134RU6 is
      constant NW:integer:=2**N;          -- число слов памяти
      type MEM is array (0 to NW-1)
                of bit_vector (0 to K-1);
      signal M:MEM;
    -- функция value преобразует бит-вектор в целое
      function value (BV:in bit_vector) return natural is
          variable E: natural:=0;
          begin
    --  цикл по числу разрядов вектора
            for i in BV'LOW to BV'HIGH loop
               E:=E*2;          -- старший разряд - LOW
               if BV(I)='1' then
                   E:=E+1;
               end if;
            end loop;
      return E;
      end value;
 begin
  process (CS) begin
    -- по срезу CS инициируется работа микросхемы
    if CS='0' and not CS'stable  then
         if A'stable(tsu_a_cs) and WRRD='0' and WRRD'stable
              (tsu_wr_cs) and DI'stable(tsu_di_cs)     then
       -- запись по срезу WRRD и предустановке A и DI
                  M(value(A))<=DI after tcs;
        elsif A'stable(tsu_a_cs) and WRRD='1'
           and WRRD'stable (tsu_a_cs) then
       -- чтение при WRRD=1 и предустановке A и WRRD
                  DO<=M(value(A)) after tcs;
        end if;
    end if;
  end process;
end SIMPLE;
```

Упражнения

1. Опишите работу микросхемы К134РУ6 в трехзначном алфавите («X», «0», «1») или в алфавите пакета STD_LOGIC_1164 и правильно отразите состояние выхода D0.

2. Введите в модель более полный контроль временных параметров сигналов.

3. Дополните модель описанием режима чтения, когда сигнал адреса может поступать после среза CS или когда подается WRRD после CS.

4. Опишите микросхему памяти с трехстабильным выходом (типа К134РУ2) и микросхему с совмещенным входом-выходом.

4.10.4. Модельный эксперимент с микросхемой ОЗУ

Использование объекта SK134RU6 для моделирования иллюстрируется объектом RAM_SIM, включающим его как компоненту модели. В микросхему памяти емкостью 1024 бита производится запись «1» по адресу «010» и чтение этого слова согласно временной диаграмме рис. 4.9. Этот тест далек от полного и иллюстрирует только первый шаг верификации — проверку простейшего режима функционирования. Более развитую верификационную среду читатель может найти в гл. 7.

```
entity RAM_SIM is   -- модель объекта RAM_SIM
    end RAM_SIM;
    use WORK.SK134RU6;
      architecture M1 of RAM_SIM is
        constant N: integer:=10;constant K: integer:=1;
       component RAM
         generic (N, K: integer;            -- K - разрядность
         ta, tcs, tsu_a_cs, tsu_di_cs, tv_cs_a, tv_cs_di,
         tv_cs_wr, tw_cs, trec_cs, tcy_a:time);
         port (A: in bit_vector(0 to N-1);
              DI: in bit_vector(0 to K-1); WRRD, CS: in bit;
              DO: out bit_vector(0 to K-1));
       end component;
    subtype BVN is bit_vector ( 1 to N);
    subtype BVK is bit_vector ( 1 to K);
    signal DI, DO: BVK;   signal A: BVN;
    signal CS: bit:='0'; signal WRRD: bit;
    -- спецификация конфигурации внутри архитектуры M1
    for all: RAM use entity SK134RU6 (SIMPLE);
      begin
      C1:RAM          -- конкретизация компоненты RAM
      generic map (10,1,700 ns, 500 ns, 200 ns, 30 ns, 200 ns,
                 250 ns, 250 ns, 0 ns, 550 ns, 450 ns, 1000 ns)
      port map (A, DI, CS, WRRD, DO);
-- входные сигналы теста
      CS <= not CS after 1000 ns;   -- период CS=2000 ns
      A<= (others=>'0'),"0000000010" after 2650 ns,
       (others=>'0')after 4800 ns,"0000000010" after 7000 ns;
      WRRD <= '0', '1' after 4800 ns;
      DI<=(others=>'0'),(others=>'1') after 2600 ns;
    end M1;
```

Рис. 4.9. Временная диаграмма тестовых сигналов в RAM SIM

Упражнения

1. Выполните модельный эксперимент с моделью микросхемы К134РУ6 с нарушением временных соотношений для сигналов A и CS.

2. Проведите эксперимент на модели микросхемы емкостью 1024 однобитовых слов с занесением и чтением «0» и «1» в каждое сотое слово и проверкой истинности считываемых данных.

3. Постройте полный тест-эксперимент для верификации модели микросхемы.

Вопросы к главе 4

1. Что является критерием полноты функционального теста?

2. Как уменьшить время процесса верификации?

3. Что делать, если на временной диаграмме некоторый выходной сигнал стал равен X?

4. Что такое «профиль» тестирующей программы?

Упражнения

1. Организуйте модельный эксперимент по сравнению результатов работы поведенческой модели (архитектура F_) и структурной модели (архитектура F_B) объекта проекта F.

2. В приведенном VHDL-примере верификации объекта проекта F при сравнении его выходов с эталоном из файла реализован так называемый синхронный стенд контроля, в котором изменение входов и сравнение выходов с эталоном привязано к синхросигналу S. Реализуйте асинхронный стенд контроля, где осуществляется привязка к изменениям выходных сигналов.

3. Опишите вариант модельного эксперимента с использованием оператора утверждения (VHDL-assert, VERILOG — другие операторы) для контроля совпадений выходных сигналов проверяемого объекта проекта F с эталоном.

4. Введите в тестовую программу параметр debug, изменение которого с 1 на 0 позволяет отключить вывод результатов моделирования на печать (этот прием позволяет повысить скорость моделирования).

5. В конце главы 3 были рассмотрены модели триггеров. Предложите свой тест-план и постройте тестирующую программу.

6. В конце этой главы и главы 3 были рассмотрены модели блока памяти. Включите в нее контрольные утверждения. Оцените на функциональную полноту следующий план тестирования — запись в одну из ячеек всех 1, считывание со сравнением с 1, то же всех 0 со сравнением, повторить эти два действия для ячейки с адресом 0 и с максимальным адресом, проверить, не пойдет ли запись-чтение при невыбранном кристалле.

7. При реализации тестирующей память (упражнение 6) программы создайте процедуры write и read с параметрами: адрес, значение данных, выбор кристалла, такт, сброс, сравнение (для «чтение»).

Глава 5
Синтезабельность HDL-описаний

Автоматизированный синтез

На рис. 5.1 представлена схема потоков информации в процессе автоматизированного синтеза. Исходное HDL-описание проекта преобразуется программой-синтезатором в структурное описание вентильного уровня (список соединений — netlist) в форматах HDL, EDIF и др.

Рис. 5.1. Блок-схема потоков информации в процессе синтеза

Требования к специалисту

Перечень типичных требований 1998 г. к инженеру-схемотехнику на примере фирмы Raycer Graphics включает помимо общих — навыки работы в коллективе, стаж не менее 4 лет, умение работать с технической документацией, опыт разработки больших быстродействующих схем(>150 МГц), знание ОС UNIX и Windows и т. п. — следующие:

— разработку микроархитектуры устройств;

— разработку HDL-описаний на уровне RTL;

— автоматизированный синтез;

— статический временной анализ проектов;

— опыт работы с языками и системами — VERILOG, VERILOG-NC, чекерами типа VERILINT и т. п.

Спецификация проекта

Спецификация является начальным блоком в схеме процесса проектирования (см. рис. 5.2). Она должна включать описание функции и ограничения на внешние параметры проектируемой системы, в том числе функциональные характеристики, временные параметры, производительность, интерфейс с другим оборудованием и программным обеспечением, физические и конструктивные ограничения — размер кристалла, потребляемую мощность и т. п.

Рис. 5.2. Идеализированная схема проектирования «сверху вниз»

Известны два метода спецификации: формальная и исполняемая [20].

Методология формальной спецификации предполагает задание требуемых характеристик в абстрактной неалгоритмической математической форме соотношений.

Методология исполняемой (executable) высокоуровневой спецификации предполагает функциональное алгоритмическое описание проекта на одном из языков программирования типа C, C++ или на HDL типа VHDL, VERILOG. Наличие такой программной модели, как уже отмечалось в главе 3, позволяет на самых ранних стадиях проекта отработать не только функциональные характеристики, но и интерфейсы системы, а также использовать ее как эталон при верификации результатов последующих этапов проектирования. Пример исполняемой спецификации дан в главе 6 (модель шифро-алгоритма RC4 на языке C).

Критерии синтеза и ограничения, задаваемые пользователем

При своей работе программа-синтезатор (далее коротко — синтезатор) пытается учесть два типа ограничений и параметров, заданных пользователем:

1) критерии оптимизации (быстродействие, объем — площадь схемы);

2) параметры элементной базы и условия работы схемы (система элементов, нагрузочная способность выходов, диапазон температур и т. п.).

Дополнительным параметром может быть уровень оптимизации, т. е. уровень усилий синтезатора: чем он выше, тем больше требуется затрат машинного времени, но и тем лучше получаемая схема. На начальных стадиях синтеза обычно используется минимальный уровень усилий синтезатора, на конечных — максимальный.

Мощные синтезаторы типа Synopsys Design Compiler обеспечивают задание большего числа пользовательских параметров, включая даже форму тактовых импульсов, свойства внутренних проводников (wire load) и т. п. Другие синтезаторы, упрощая работу пользователя или ориентируясь на определенный тип элементной базы, большинство параметров задают по умолчанию.

Например, системе Synplify, предназначенной для проектирования устройств на программируемых интегральных схемах (ПЛИС — FPGA, CPLD), достаточно в диалоговом режиме указать семейство FPGA, например XILINX XC4000E, тип кристалла, например XC4013E, тип корпуса, например PC84, группу, определяющую его скорость, например 1, тактовую частоту — требуемое быстродействие проекта, — например 30 МГц, максимальную нагрузку на выходе, например 10, и запустить синтезатор.

При необходимости дополнительно задаются задержки запаздывающих входных сигналов, указания по синтезу конвейерной (pipeline) или обычной схемы, группировке или разгруппировке (flattening logic) отдельных фрагментов проекта, использованию системы синтеза автоматов (FSM compiler), способу кодировки состояний автомата и т. п.

Порядок работы программы-синтезатора

Сначала синтезатор осуществляет синтаксический контроль HDL-описаний. Хотя после моделирования сообщения о грубых синтаксических ошибках невероятны, но сообщения о запрещенных с точки зрения синтезабельности (см. ниже) HDL-конструкциях и неполноте списков чувствительности процессов относятся к типичным.

Если ошибок нет, синтезатор генерирует отчет о работе, включающий предупреждения о появлении в итоговой схеме триггеров-защелок, параметры полученной схемы (предельную частоту и задержки критического пути, число триггеров и т. п.). Синтезатор строит также тексты структурных описаний схем (форматы HDL (см. ниже примеры), EDIF, XNJ или др.). Программа просмотра полученной схемы позволяет видеть ее условное графическое изображение (в образах базовых узлов (RTL view) и после покрытия заданной системой элементов (в технологическом базисе). Просматривая отчет синтезатора, пользователь порой находит такие неприятные сообщения, как появление непредвиденных триггеров-защелок (latch), критических путей с задержкой больше предусмотренной (отрицательный запас по времени — slack) и т. п. Учитывая, что после трассировки величины задержек в схеме обычно возрастают на 10—20% (при заполненности кристаллов FPGA более 80%, трассировка часто увеличивает задержки на 30%), проектировщику порой приходится вносить большие изменения в исходный HDL-текст и, жертвуя оборудованием, вводить в проект конвейерные и параллельные структуры. Новейшие системы синтеза (MAGMA и др.), ориентированные на проектирование сверхбыстродействующих схем, в которых задержки проводников сопоставимы или превышают задержки элементов ведут синтез с учетом результатов работы встроенной в них подсистемы предварительной трассировки.

Пример синтеза схемы объекта проекта F, рассмотренного в главе 3 и тестированного в главе 4.

VHDL

```
entity F is
  port (A1, A2:in bit; B1, B2: out bit);
end F;
architecture F_CASE of F is
 begin

   process (A1, A2)
   variable EE:bit_vector(0 to 1);
     begin  -- & - конкатенация
        EE:=A1& A2;
       case  EE  is
         when "11" => B1<='1'; B2<='0';

         when others => B1<='0'; B2<='1';

       end case;
   end process;
end F_CASE;
```

VERILOG

```
module F_CASE(A1,A2,B1,B2);
input A1,A2;
output B1,B2;reg B1,B2;
always @(A1 or  A2)

  begin   //{} конкатенация

  case ({A1, A2} )
            2'b11: begin B1=1;
                   B2=0;end
           default: begin B1=0;
                   B2=1;end
  endcase
  end
endmodule //F_CASE
```

На рис. 5.3 представлено условное графическое изображение схемы F, полученное системой синтеза Synplify. Синтезатор исключил вспомогательную переменную (EE) из VHDL-кода и изобразил схему из двух вентилей — одну из возможных схем, рассмотренных в главе 3. Но это условное изображение (RTL-view) несколько отличается от реализации схемы в технологическом базисе кристалла FPGA типа VIRTEX.

Реализация схемы F_CASE в технологическом базисе кристалла ПЛИС типа VIRTEX фирмы Xilinx (использовалась микросхема xcv50bg256-4) выглядит так (сравните с вариантами рис. 3.2, гл. 3):

```
B1 <= A1 and A2 after 100 ps;
B2 <= not A1 or not A2 after 100 ps;
```

Фрагмент протокола работа синтезатора:

```
##### START TIMING REPORT ####--ВРЕМЕННЫЕ ПАРАМЕТРЫ
        Performance Summary
        ***********************
          Requested    Estimated    Requested    Estimated
Clock      Frequency    Frequency    Period       Period     Slack
          Заданное     Полученное   Период                  Запас
-------------------------------------------------------------
System    120.0 MHz    129.6 MHz    8.3          7.7        0.6
=============================================
```

Полученное быстродействие — 129 МГц с учетом задержек в буферах ввода-вывода

```
Resource Usage Report -- - РАСХОД ОБОРУДОВАНИЯ
Mapping to part: xcv50bg256-4 -- - используемый кристалл
Cell usage:
GND          1 use      VCC          1 use - питание и земля
I/O primitives:
OBUF         2 uses     IBUF         2 uses - буфера ввода-вывода
        Mapping Summary:
Total  LUTs: 2 (0%) - использовано всего два логических элемента кристалла.
```

Рис.5.3 Построенное синтезатором графическое изображение схемы F

5.1. Общие принципы построения синтезабельных описаний

5.1.1. Повторнопригодность проектов

В 2001 г. большие интегральные схемы (БИС) достигли уровня сложности, превышающего 10 миллионов вентилей на кристалле и тактовой частоты более 1 ГГц. При производительности одного разработчика в 100 вентилей в день создание проекта такой схемы потребует гипотетических 500 человеко-лет и обойдется более чем в 75 миллионов долларов.

Один из путей преодоления «барьера размерности» лежит в использовании методологии так называемого повторнопригодного проектирования (reuse-based design methodology). Возможность легкого включения в проекты новых систем проектов предыдущих систем или их фрагментов значительно упрощает задачу разработчика.

Руководства по повторнопригодному HDL-проектированию (reuse methodology manual — RMM) [19] появились в конце 90-х годов прошлого века. Они содержат ряд положений по практическому применению этой методологии, частично

приведенных ниже. Некоторые из положений этого руководства пересекаются с рекомендациями по созданию верификабельных (проще — легко формально верифицируемых) HDL-проектов [20], также использованными ниже.

5.1.2. Твердые и мягкие макросы

Практика создания **систем на одном кристалле-чипе** (System-On a Chip — SoC) предполагает частое использование одних и тех же так называемых макросов (macros) в разных проектах (иногда вместо термина «макрос» применяется термин core — ядро). Отсюда термин повторнопригодность HDL-описаний.

Например, процессор ARM (название одного из встраиваемых в кристаллы микропроцессоров) может быть частью нейрочипа или чипа управления стиральной машиной и т. п. Соответственно проектировщик чипа покупает **интеллектуальную собственность** (intellectual property — IP) в виде VHDL- или VERILOG-описания у фирмы ARM и включает макрос в описание своей системы, наряду, допустим, с купленным у другой фирмы VHDL-описанием последовательного порта, аналого-цифрового преобразователя и т. п.

Этот макрос может быть «жестким» (hard macros) или «мягким» (soft macros).

Мягкий макрос предполагает использование синтезабельного HDL- описания на регистровом уровне (RTL).

Жесткий макрос предполагает применение описания в виде GDSII-файла, т. е. в виде полностью размещенного и оттрассированного поставщиком схемного компонента.

В дальнейшем нас будут интересовать в первую очередь мягкие макросы.

5.1.3. Что такое «хороший проект макроса»

Современная методология автоматизированного проектирования предполагает следование трем главным правилам:

1. Проектируемое устройство должно быть **полностью синхронным**, входы и **выходы блоков (макросов) должны быть регистрами**. Это позволяет сделать локальной задачей оптимизацию временных параметров.

2. Полностью верифицируйте проектные модули перед их включением в модули более высокого ранга иерархии, так как с повышением сложности проектных задач трудоемкость поиска ошибок растет непропорционально быстрее.

3. Планируйте, прежде чем исполнять, разрабатывайте проектные спецификации перед началом проектирования, параллельно с проектированием блоков разрабатывайте средства их верификации (верификация по трудоемкости часто превышает разработку).

Обычные критерии, которым соответствует «хороший проект», а именно:

— исчерпывающая документация;

— хороший стиль HDL-кодирования и ясный комментарий;

— удачно спроектированная верификационная среда (test benches) и исчерпывающие тестовые наборы (test vectors);

— развитые и ясные скрипты (scripts — последовательности команд синтезатора, используемых для управления процессом проектирования).

Повторнопригодный проект макроса дополнительно должен быть:

1) легко реконфигурируемым для использования в различных приложениях;

2) малозависимым от типов технологий (различных библиотек) его логической и конструкторской реализации, например, и на ПЛИС (FPGA), и на элементах заказных БИС;

3) портативным — пригодным для верификации на различных типах формальных верификаторов или моделирующих систем, например, моделироваться на Modelsim и на VERILOG-XL и пригодным для синтеза на различных типах САПР, например, САПР ПЛИС фирм Synopsys, Modeltech, Altera и Xilinx;

4) пригодным для автономной верификации, независимо от кристалла, в котором он будет использоваться;

5) должен включать исчерпывающие рекомендации по реконфигурации и ограничениям значений параметров.

Здесь приведены лишь некоторые из проблем, возникающих при повторном использовании макросов:

— несоответствие языка описания макроса (например, VERILOG) описанию остальных компонент проектируемого кристалла (например, VHDL);

— системы проектирования, использовавшиеся ранее при разработке макроса, в настоящее время не поддерживаются или отсутствуют или у проектировщиков макроса и кристалла были разные операционные системы и инструментальные ЭВМ. Иногда сказываются даже проблемы разрядности этих ЭВМ.

Вопросы к разделу 5.1

1. Что такое хороший макрос?
2. Что такое повторнопригодность проекта?

5.2. Рекомендации по стилю кодирования HDL-описаний

Обычно разработчик сразу кодирует свой проект, придерживаясь синтезабельного стиля, для того чтобы верификация и синтез не требовали дополнительных перекодировок.

Отклонение от приведенных в разделе правил и рекомендаций в большинстве случаев обнаруживается системой синтеза или на более ранних стадиях специальными программными средствами, так называемыми чекерами (checker, lint tool). Примерами таких средств могут служить: Reuse-Corrector фирмы Intrinsix, созданный под руководством А. Сохацкого (демоверсия свободно доступна на сервере midc.miem.edu.ru), система Verilint (фирма Cadence) и др. Число правил, которые они контролируют, обычно 200—400. Ниже приведена только малая часть из них.

5.2.1. Рекомендации общего плана

Разработайте общие для всех участников проекта правила выбора имен объектов проекта:

1) имена должны быть мнемоничны, и, например, вместо ra используйте ram_addr;

2) в именах сигналов и переменных используйте только строчные буквы, прописные буквы оставьте для имен констант и меток:

— для тактовых сигналов используйте имя clk (clk1, clk2, если их много);
— для сигналов с активным низким (0) уровнем используйте суффикс _n, например vv1_n, rst_n, reset_n;
— для сигналов сброса используйте имя rst (rst_n).

Применение строчных букв позволит, например, легко отличать имена исходных объектов проекта от дополнительных (прописные буквы в именах), генерируемых системой синтеза (см. приведенные ниже тексты синтезированных объектов сумматора и счетчика);

3) заголовок-комментарий HDL-описания должен быть у каждого файла проекта и включать имя файла, автора, реализуемую функцию, дату создания, историю модификаций, их суть (см., например, главу 6);

4) на каждые 2—3 строки-оператора следует давать комментарий, который должен быть кратким, выразительным, ясным;

5) каждому оператору — отдельную строку. Длина строки до 72 символов (это упростит документирование проекта);

6) используйте отступы — 2—4 пробела, а не клавишу Tab при структуризации кода (число пробелов, создаваемых Tab, может различаться на разных машинах);

7) не используйте зарезервированные слова одного HDL в описании на другом HDL (это затруднит перекодировку в другой HDL);

8) рекомендуемый порядок портов в описании интерфейса объекта проекта: сначала входы, потом выходы:

— входы — сначала такты (clk), потом сброс (rst), разрешение (en), управление, данные;
— выходы — такты, сброс, разрешение, управление, данные;

9) список конкретизации — рекомендуется ключевое сопоставление порт — сигнал: a=>a, .b(b), а не позиционное;

10) используйте функции и процедуры вместо многократного повторения одних и тех же фрагментов кода;

11) используйте векторы, массивы, операторы генерации и циклы для повышения компактности описаний;

12) в VHDL-описаниях используйте IEEE стандартные типы std_logic, std_logic_vector,std_ulogic, std_ulogic_vector;

13) используйте поименованные константы вместо указания непосредственных значений в тексте модели.

5.2.2. Рекомендуемая структура и примеры имен сигналов

Назначение (тип) сигнала	Примеры рекомендуемых имен: VHDL, VERILOG	Примечания
Сброс	rst, rst1	Строчными буквами
Тактовый	clk, clk1	
Активный уровень нуля	rst_n, x_n	
Выход регистра	a_r	
Имя константы	constant ZERO.bit'='0'; parameter ZERO=1'b0;	Имена констант прописными буквами
Имя метки процесса	M1_PROC;	Суффикс _PROC

Назначение (тип) сигнала	Примеры рекомендуемых имен: VHDL, VERILOG	Примечания
Имя конкретизации	U_2	Префикс U_
В списке портов — формальный — фактический порт	a=>a (VHDL) .a(a) (VERILOG)	По возможности стремиться, чтобы имена совпадали
Векторный сигнал — нисходящая нумерация разрядов	signal b (10 downto 0):bit; (VHDL) reg [10:0] b; (VERILOG)	Порядок нумерации разрядов — слева наибольший номер, справа — 0
Имя компоненты и имя модуля	Хорошо, когда имя компоненты совпадает с именем объекта (VHDL)	
Имя архитектуры (модуля), отражающее тип описания (VHDL) и имя теста	Rtl beh str tb	architecture rtl of A is architecture beh of A is architecture str of A is module tb_A;

На рис. 5.4 дан пример протокола проверки программой REUSE_CHECKER описания простого объекта F, подробно рассмотренного ранее (предупреждения о нарушении рекомендаций по стилю — в именах прописные буквы и т. п.).

Из примера видно, что большинство моделей предыдущих глав содержат отступления от рекомендуемого стиля синтезабельного и повторнопригодного кодирования.

Рис. 5.4. Экран программы REUSE_CHECKER

5.2.3. Организация базы данных проекта

1. Описание каждого модуля помещается в одноименный файл, имеющий расширение .v для VERILOG и .vhd для VHDL. VHDL-описание интерфейса (entity) и архитектуры (architecture) обычно содержится в одном файле.

2. Для VERILOG хорошим тоном является выделение описаний всех общих (директивой компилятора `define) констант и помещение их перед заголовком модуля или в отдельные файлы и включение таких файлов в модули с помощью директивы `include. Рекомендуемое расширение для имен таких файлов- .vh или .h.

3. Проектные библиотеки (lib).

Библиотеки содержат описания многократно используемых модулей проекта. Например, описания мультиплексоров, сумматоров, триггеров, блоков памяти,

тристабильных буферов и т. п. или общих процедур различных тестов и т. п. Для VERILOG-библиотек исходных модулей рекомендуется расширение .vlib, VHDL — .vhlib.

Проектные директории

Проектные директории обычно состоят из поддиректорий типа:

RTL или src — они хранят исходные RTL-описания;

SIM или tb — используется для моделирования — хранения теста и результатов моделирования;

GATE — синтезированные описания вентильного уровня.

База данных

При коллективном проектировании для защиты файлов от несанкционированного доступа и сбоев оборудования обычно используются системы типа RCS (Revision Control System), RRCS (Recursive RCS), TDM (Team Design Manager) и т. п.

Вопросы к разделу 5.2

1. Почему рекомендуется использовать строчные буквы в именах синтезабельных описаний?
2. Стоит ли в VERILOG-описании использовать имя IF?

5.3. Что такое «хорошие» модули-макросы

Этот раздел содержит «выжимку» из правил проектирования [19, 20].

5.3.1. Общие рекомендации

Размер модуля

Рекомендуется ограничивать размер модуля объемом в 4000—5000 вентилей - это позволяет затрачивать приемлемое время на его синтез.

Правда, сделать эту оценку до синтеза порой непросто.

Система синхронизации

Рекомендуется использовать минимальное число тактовых генераторов (clk) и частот. Это уменьшает риск нежелательных фазовых сдвигов сигналов в сети распространения синхроимпульсов.

Сигнал сброса — синхронный или асинхронный сброс?

Синхронный сброс (rst) упрощает синтез, предполагает независимо работающий тактовый генератор.

Асинхронный сброс не обладает этими преимуществами. Он предполагает наличие специального источника сигнала сброса с повышенной нагрузкой (на стадии синтеза-трассировки строится дерево разветвляющих буферов сети сброса). Однако при наличии в проекте тристабильных шин, в которых одновременно мо-

жет быть включен только один источник, асинхронный сброс, вырабатываемый сразу при подаче напряжения питания, является одним из простейших решений

Шины — тристабильные или мультиплексные

Тристабильные шины популярны при проектировании плат, так как они уменьшают число проводников. Однако на уровне кристалла часто рекомендуется применять мультиплексные шины, т. е. ставить мультиплексоры на входе каждого приемника, соединенные со всеми необходимыми источниками сигналов.

Учет потребляемой мощности

Портативные устройства (мобильные телефоны, пейджеры, блокнотные ЭВМ и т. п.) — быстрорастущий сектор БИС Динамическая мощность (P), рассеиваемая КМОП БИС, описывается формулой

$$P = \sum_{i}^{k} afcv^2,$$

где k — число ключей в схеме; a — число переключений узла; f — тактовая частота; c — емкость нагрузки; v — напряжение питания.

Понижение напряжения питания с 5 до 1,1 В уменьшает расход мощности в 21 раз, но, естественно, за счет быстродействия элементов. Поэтому современная мобильная аппаратура использует 3- и 2,5-вольтовое напряжение

Понижение потребляемой мощности за счет уменьшения нагрузочной емкости в 3—4 раза достигается путем использования специальных библиотек элементов при синтезе.

Управляемые тактовые генераторы (временное отключение подачи тактов на неработающие блоки) дают 2—3-кратное уменьшение мощности, но за счет больших усилий при проектировании и синтезе подобных систем (системы синтеза не любят управляемые внутри модуля синхросигналы).

Структура модулей

При проектировании больших систем их разбивают на модули-блоки. В числе учитываемых при этом факторов (помимо функциональных, конструктивных и т. п.) следующие:

1) на выходах рекомендуется иметь регистры — иначе больше вероятность того, что задержка в КС на выходе блока плюс задержка в межблочных соединениях плюс в комбинационных схемах на входе следующего блока превысят время между синхроимпульсами. Рекомендуемые структуры блоков и синхронизации см. на рис. 5.5;

2) запрещается объединять в одном модуле триггеры, управляемые фронтом, и триггеры, управляемые срезом тактового сигнала clk;

3) избегайте управляемых тактовых сигналов в RTL-коде модуля, как и генерируемых внутри модуля тактов (примеры плохих решений приведены на рис. 5.6);

4) при проектировании маломощных схем, когда не обойтись без периодического отключения пассивных фрагментов устройств, описания управляемых тактовых сигналов, а также внутреннегенерируемых тактов и сбросов, выделяйте в отдельный модуль и выносите его на верхний уровень проекта.

Рис. 5.5. Идеализированная схема связи и тактирования двух блоков-модулей (КС — комбинационная часть, reg — регистровая часть блока)

Рис. 5.6. Примеры плохих решений — управляемые синхросигналы

5.3.2. Дополнительные замечания

1. Применяйте и строго придерживайтесь рекомендованных в документации по системе синтеза HDL-форм образов узлов. Например, для процессных форм при описании регистров: сначала описывается сброс, потом установка (см. раздел 5.7), потом прием и обработка данных на регистре. Это соответствует обычным схемам, в которых сигнал сброса сильнее других сигналов.

2. Избегайте использования триггеров-защелок (latch) (см. раздел 5.7) (исключение — регистровые файлы).

3. Список чувствительности процесса должен быть полным, иначе RTL-модель будет неверно функционировать и ее поведение будет отлично от поведения структурной модели, получаемой в результате синтеза.

4. В синтезабельных VERILOG-описаниях триггеров и регистров внутри блока always не стоит применять блокирующее присваивание (=)-, лучше использовать <= (это позволит упростить верификацию).

5. Используйте case оператор вместо if при описании мультиплексоров (подробности в следующих разделах).

6. При описании автоматов используйте единообразный стиль, например, разделяйте HDL-код на 2 части (2 процесса) — один для описания комбинационной части автомата, другой — триггеров.

7. Кодировка состояний автоматов. В VHDL-описаниях вводите свой перечислимый тип для кодировки состояний, в VERILOG используйте директиву `define для записи констант-кодов состояний.

8. По возможности выделяйте описания автоматов (FSM-Finite State Mashine) в отдельные модули. В этом случае облегчается их оптимизация с использованием специальных средств системы синтеза автоматов (FSM compiler).

9. Старайтесь облегчить читателям вашего HDL-текста (вы сами в первую очередь будете многократным, до отвращения частым читателем) различение фрагментов текста, соответствующих комбинационным узлам и схемам с памятью. Например, в одном из проектов, где довелось участвовать автору, комбинационные узлы были описаны на VHDL в потоковом стиле, а регистры — в процессном; в другом VERILOG-проекте имена связей имели суффикс _w, а имена регистров — _r.

Вопросы к разделу 5.3

1. Почему синтезабельное подмножество HDL не включает операторы ввода-вывода?
2. Как уменьшить потребляемую схемой мощность?
3. Почему рекомендуется иметь регистры на выходах проектируемых блоков?

5.4. RTL-описание

Проектирование ЭВМ и цифровых систем (ЦС), как уже отмечалось, условно можно разбить на несколько этапов.

1. На этапе системного проектирования определяется структура будущей системы, состав устройств и их основные характеристики: производительность, надежность и др.

2. На этапе структурно-алгоритмического проектирования отрабатывается система команд, алгоритмы функционирования и структурные схемы устройств.

3. На этапе функционально-логического проектирования разрабатываются функциональные и принципиальные электрические схемы, тесты контроля устройств и узлов.

4. На конструкторском этапе производится привязка схем к конструктивным элементам: стойкам, панелям, типовым элементам замены. Осуществляется расстановка корпусов микросхем и трассировка соединений на печатных платах.

Эта последовательность этапов характерна и для разработки больших интегральных схем (БИС). Она отображает процесс проектирования объекта от спецификации его внешних характеристик («черного ящика») до схемы и конструкции, представляющих объект как «прозрачный ящик».

Проектирование «сверху вниз» на базе HDL предполагает переход от поведенческого описания системы к синтезабельному описанию на уровне регистров (RTL-описанию). Этот переход обычно осуществляется проектировщиком вручную на базе заданной библиотеки типовых узлов или рекомендуемых типовых форм HDL-описаний регистров (образов регистров, образов узлов).

Поясним процесс перехода от поведенческого к регистровому уровню HDL-описания на примере.

В документации на БИС типа микропроцессор, микросхема программируемого контроллера прерывания, контроллер прямого доступа к памяти и т. п. функ-

ционирование и структура системы обычно описывается на поведенческом уровне. Если говорить про микропроцессор, то его функционирование описывается в виде набора описаний режимов его работы, описаний параллельных процессов функционирования отдельных устройств и режимов выполнения команд.

Например:

— режим (процесс) инициализации — сбрасываются в 0 регистры флагов, счетчик команд и т. д.;

— режим (процесс) выборки команд — обращение за командой в память, команда поступает на регистр команд, дешифрируется, формируется исполнительный адрес операндов и т. д.;

— режим выборки операндов из памяти;

— режим исполнения команды, например команды безусловного перехода: адрес, содержащийся в поле адреса команды, передается на счетчик команд, и начинается выбор следующей команды, если нет прерывания;

— вход в режим прерывания — в стеке сохраняются флаги и счетчик команд, после чего в счетчик команд устанавливается адрес прерывающей программы и т. д.

Если сопоставить описанию каждого из режимов описание процесса, то в тот же счетчик команд будет происходить присваивание новых значений в разных процессах.

Хотя семантика HDL допускает такое описание и предусматривает возможность разрешения конфликтов при записи в один и тот же регистр (сигнал) новых значений в разных процессах, такое описание систем обычно непригодно для использования для САПР-синтеза, т. е. не является синтезабельным.

Следует построить так называемое синтезабельное описание на уровне регистров. Например, выделить для того же счетчика команд все ситуации, связанные с изменением его состояния, сгруппировать их в одном месте и описать поведение счетчика как отдельный процесс на HDL. Аналогичным образом следует поступить с другими регистрами и узлами.

Итак, для того чтобы HDL-описание хорошо воспринималось системой логического синтеза САПР, оно должно быть синтезабельным. Для большинства современных САПР это описание на уровне регистровых передач (в дальнейшем — уровне регистров, RTL), выполненное с использованием ограниченного (синтезабельного) подмножества HDL, а также ограниченного подмножества форм описаний (синтезабельных форм).

Следует отметить определенное совпадение идей ограничения, используемого подмножества HDL и изобразительных форм, вытекающих из нужд синтеза, с ограничениями так называемого верификабельного подмножества, обусловленными требованиями систем формальной верификации и контроля эквивалентности разноуровневых описаний (например, регистрового и вентильного уровней) [20].

При этом чем мощнее САПР, тем больше свободы и разнообразия языковых форм она допускает, но проектировщик, широко пользующийся этим богатством, сильно ограничивает мобильность своих разработок — его описание становится непригодным для маломощных САПР.

5.5. Синтезабельное подмножество HDL

Для разных САПР это подмножество несколько отличается, следует подробно ознакомиться с документацией соответствующей САПР. Кроме того, можно ука-

зать САПР, чтобы она пропустила без обработки некоторые фрагменты описаний с несинтезабельным текстом с помощью специального комментария.

Для САПР SYNOPSYS это комментарий:

VHDL	VERILOG
-- synopsys translate off - отключить синтез	// synopsys translate off
-- synopsys translate on - включить синтез	// synopsys translate on

Рекомендуемые в описании синтезабельного подмножества VHDL:
-- synthesis off - отключить синтез
-- synthesis on - включить синтез

5.5.1. Основные синтезабельные конструкции

Ключевые слова

Список основных синтезабельных ключевых слов и конструкций HDL:

VHDL	VERILOG
alias, all, and, architecture, array, attribute, begin, block, buffer, case, component, constant, downto, else, elsif, end, entity, exit, for, function, generate, generic, if, in, inout, is, library, loop, map, package, mod, nand, nor, next, null, of, on, open, or, others, out, then, port, procedure, process, range, rem, return, select, signal, subtype, to, type, until,use, variable, wait, when, while, with, xor	always, assign, begin, case, casex, casez, default, defparam, disable, else, end, endcase, endfunction, endmodule, endtask, for, forever, function, if, inout, input, integer, module, negedge, or, output, parameter, posedge, reg, task, wire, wand, wor, supply1, supply0 встроенные примитивы: and, nand, or, nor, xor, buf, not, bufif1, bufif0, notif1, notif0

Типы и виды данных

	VHDL	VERILOG
Скаляры и векторы	типы сигналов и переменных: Std_logic, Std_ulogic	типы связей: wire,wand,wor, supply0,supply1
	Boolean, bit,integer,	типы переменных: reg, integer

Описатели

	VHDL	VERILOG
Описание объекта	entity, architecture	module, macromodule
Описание параметров	generic	parameter
Описание портов	in, out, inout buffer	input, output, inout

Описание связей	signal	wire, wand, wor, supply0, supply1
Описание констант	constant	parameter
Описание переменных	variable	reg, integer
Описание типа	type, subtype	
Описание функций и процедур	function procedure	function task

Операторы

Процесса	process	always

Конкретизации компонент — допускается как позиционное так и поименованное соответствие портов сигналам

Конкретизации встроенных примитивов	----------	and, nand, or, nor, xor, buf, not, buff1, buff0, notif1, notif0
Параллельное назначение	<=	assign ИМЯ =
Присваивание переменной (блокирующее)	:=	=
Назначение сигналу (неблокирующее)	<=	<=
Условный	if, then, elsif, else, end if	if else
Цикла	for, while, loop, end loop	for, while
Выбора	case, end case	case, casez, casex, endcase
Процедуры		
Группы	----------	begin, end
Операции	Допускаются все операции, но при подключении нужных пакетов	Допускаются все операции и выражения, за исключением === и !== , часто вместо операций && и || применяют & и |

Дополнительные ограничения и пояснения

Задержки -
Система синтеза обычно игнорирует задержки.

Неопределенные и высокоимпедансные значения сигналов

Запрещается использовать значения X и Z в выражениях, только при описании тристабильных буферов разрешается присваивать значение Z, а в VERILOG-операторах casex, casez значения X и Z воспринимаются как безразличные.

Начальные значения сигналов

Запрещается указывать их в объявлениях сигналов и с помощью initial — в реальной схеме при включении питания и до поступления сигнала сброса значение триггеров не определено.

5.5.2. Синтезабельные библиотеки типовых узлов

Существенное повышение производительности труда проектировщиков может дать применение в новых проектах фрагментов предыдущих разработок, в частности, использование HDL-библиотек высокопараметризованных и технологически независимых описаний макросов и элементов. (В программировании аналогом может служить применение библиотек стандартных функций и процедур.)

Проектирование «снизу вверх» предполагает возможность построения проектировщиком структурных RTL-описаний систем с использованием библиотек моделей типовых узлов.

Например, проектная HDL-библиотека может включать:

— одно- и n-разрядный вентиль И_НЕ на произвольное число входов (а также вентили И, ИЛИ, НЕ и т. п.);

— одно- и n-разрядные мультиплексоры 2-1,3-1, 4-1,8-1 (а также дешифраторы 1-2,1-3,1-4 и т. п.);

— одно- и n-разрядный сумматор (вычитатель, умножитель);

— одно- и n-разрядный трехстабильный буфер;

— одно- (триггер) и n-разрядный регистр с асинхронным сбросом (соответственно с синхронным сбросом, установкой, разрешением и т. п.);

— n-разрядный счетчик с синхронным (асинхронным) сбросом и установкой;

— n-разрядный сдвигатель и т. д.;

— регистровый файл (n*m), fifo (n*m), ram (n*m) и т. д. (Обычно 20—90 подобных компонент.)

Структурное описание проекта строится как композиция из заданного набора синтезабельных компонент.

Положительными свойствами такого подхода могут быть:

— привычность инженерам-схемотехникам;

— высокая степень синтезабельности;

— надежность и эффективность моделей узлов;

— возможность использования графического ввода описаний при наличии графического редактора в САПР и т. д.

Недостатки:

— затруднения при описании нестандартных фрагментов систем, например, автоматов;

— излишняя громоздкость при большом количестве функционально простых узлов и низкая наглядность описаний.

Пример конкретизации библиотечного n-разрядного мультиплексора типа mux2_1 в описании некоторой системы с сигналами sig1, sig2,sel,out_m1.

VHDL

```
m1: mux2_1
   port map(sig1,sig2,sel,out_m1);
```

VERILOG

```
mux2_1   m1
      (sig1,sig2,sel,out_m1);
```

5.5.3. Синтезабельные образы узлов

Можно строить не структурные, а поведенческие (RTL-уровня) синтезабельные описания проектов, используя рекомендованные в документации САПР типовые синтезабельные формы (образы — template). Чем мощнее САПР, тем больше разнообразие таких форм.

Эти формы как бы являются формами записи архитектур соответствующих обобщенных узлов.

Пример синтезабельного, использующего потоковую форму описания, порождающего мультиплексор 2_1 при синтезе.

VHDL

```
out_m1<= sig1 when sel='1'
              else  sig2;
```

VERILOG

```
assign out_m1=( sel==1'b1)?sig1:
                           sig2;
```

Следует (как уже было отмечено) тщательно придерживаться рекомендуемых форм, так как не каждое RTL-описание системы с точки зрения конкретной САПР является «синтезабельным». Оно может быть непригодно для синтеза и «покрытия» определенным элементным базисом, если не учитывает дополнительных, присущих данной САПР ограничений на стиль описания и подмножество HDL. Даже порядок проверки условий в условных операторах может влиять на «синтезабельность».

Например, для порождения D-триггеров и регистров с асинхронным инверсным сбросом (reset_n) желательно описывать их процессы так, чтобы в условном операторе (if) был сначала описан сброс в 0 при наличии сигнала сброса =0 и лишь потом прием нового значения по фронту синхросигнала. В синтезабельных VERILOG-описаниях (отличие от VHDL) подобных триггеров в списке чувствительности процесса обязательно должно быть указано событие среза сигнала сброса reset_n, хотя реально асинхронный сброс управляется потенциалом (примеры см. ниже).

Кроме того, приходится учитывать также ряд рекомендаций, упрощающих отладку моделей.

Например, полезно вводить задержку в модели триггеров и регистров хотя бы на один дельта-цикл за счет применения оператора <=, а лучше явную, например, на 1 ns или на параметр reg_del.

Это упрощает прослеживание причинно-следственных отношений сигналов в процессе отладки моделей, а система синтеза игнорирует задержки.

VHDL

```
process (clk,reset_n)

  begin
  if reset_n='0' then
    q<= '0' after 1 ns;
  elsif clk'event and clk='1'then
    q<=d after 1 ns;
  end if;
end process;
```

VERILOG

```
always @(negedge reset_n or
            posedge clk)
  begin  //
   if (reset_n == 1'b0)
    q<= #1 1'b0;
   else
     q<= #1  d;
end //always
```

Ниже приведены примеры синтезабельных (для большинства САПР) форм RTL-описаний узлов. Их можно воспринимать как примеры описаний компонент проектной библиотеки обобщенных узлов, а описания их архитектур — как рекомендуемые образы.

5.6. Синтезабельные описания комбинационных узлов

Описание комбинационного элемента: при любых значениях входных аргументов узла оно должно исполнять одну из ветвей, присваивающих значение его выходу (не должно быть запоминаний выхода).

Все VHDL-описания, приведенные далее, подразумевают подключение пакета std_logic_1164.

```
library ieee; use ieee.std_logic_1164.all;
```

5.6.1. Мультиплексоры

Варианты описаний комбинационных схем подробно иллюстрируются на примере мультиплексоров. В примерах имеются отклонения от «хорошего стиля кодирования».

Пример 1. Процессная форма описания n-разрядного мультиплексора 2-1 (входы A, B, выход Y).

VHDL

```
entity mux_2_1 is
generic (n:positive:=1);
 port (a,b:in std_logic_vector
   (n-1 downto 0);
    c:in std_logic;
   y:out std_logic_vector (0 to 2)
    );
end;
architecture behavior of mux_2_1 is
constant one : std_logic:='1';
begin
   process (a,b,c)
    begin
     if c=one then
        y<=a;
    else
        y<=b;
    end if;
  end process;
end behavior;
```

VERILOG

```
module mux_2_1(a, b, c, y );
   parameter n=1;

   input [n-1:0] a,b;
   input c;
   output [n-1:0]  y;
   reg [n-1:0]  y;

always (a or b or c )
   begin
     if (c == 1'b1)
        y<=a;
   else
        y<=b;
   end // always

endmodule  // MUX_2_1
```

Обратите внимание на то, что синтезабельному описанию мультиплексора присуща в строгом рассмотрении некоторая поведенческая неточность, так называемый X-оптимизм. При неопределенном (X) значении управляющего сигнала c (c=X) выход приведенного примера все равно будет равен b, а не X). Это затруднит при моделировании такого описания обнаружение ошибок в цепи сигнала c, однако упростит сравнение синтезированной схемы с оригиналом, так как системы моделирования на уровне вентилей часто оперируют в двоичном алфавите.

Пример 2. Потоковая форма описания n-разрядного мультиплексора 2-1 (входы a, b, выход y). В описании в «потоковом» (dataflow) стиле используют краткую форму записи процесса.

VHDL

```
architecture beh_short of mux2_1 is
 constant  one : std_logic:='1';
      begin
       y<= a when c=one else b;
end beh_short;
```

VERILOG

```
module mux_2_1_S(a,b,c,y);
 input c;parameter n=1;
 input[n-1:0]  a,b; output[n-1:0] y;
 assign  y =( c==1'b1)? a: b;
endmodule  // mux_2_1_S
```

Пример 3. Одноразрядный мультиплексор 4_1.

```
entity mux4 is
  port (c, d, e, f : in std_logic;
        s : in std_logic_vector
                    (1 downto 0);
       muxout : out std_logic );
end mux4;
architecture beh of mux4 is
begin
mux: process (s, c, d, e, f)
begin
  case s is
    when "00" => muxout <= c;
    when "01" => muxout <= d;
    when "10" => muxout <= e;
    when "11" => muxout <= f;
    when others => muxout <= 'X';
  end case;
end process mux;
end beh;
```

```
module mux4 (c,d,e,f,s,muxout);
    input c,d,e,f;

    input [1 :0]s; output muxout;
     reg muxout;

always (s or c or  d or  e or  f)
    begin
   case (s)
        2'b00 : muxout <= c;
        2'b01 : muxout <= d;
        2'b10 : muxout <= e;
        2'b11: muxout <= f;
      default: muxout <= 1'bx;
    endcase
  end // always
endmodule //mux4
```

Хотя в данном примере и присваивается X, но система синтеза SYNPLIFY правильно распознает мультиплексор (другие могут это не делать).

Пример 4. Описание n-разрядного мультиплексора 3-1 с поразрядным кодированием селектора (one hot). Несинтезабельная часть описания используется при моделировании для контроля присутствия только одной единицы в коде селектора. Эта часть ограничена комментариями отключения и включения синтезатора.

В описание также включены комментарии, которые указывают на синтез мультиплексора с полным набором путей. Стиль комментариев САПР фирмы Synopsys является одним из стандартов.

Остается надеяться, что синтезатор минимизирует схему, исключив промежуточные переменные tmp, которые также замедляют моделирование.

VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux3oh is

generic(size :integer:=1;
check:std_logic:='1');
```

VERILOG

```
module mux3oh(o, d0, d1, d2,  s0,
                                s1,s2);

parameter size = 1, check = 1;
```

```
port( o:out std_logic_vector
        (size-1 downto 0);
   d0,d1,d2:in std_logic_vector
        (size-1 downto 0);
  s0, s1, s2: in std_logic);
end;
architecture beh of mux3oh is
constant one :std_logic:='1';
signal onehotcheck:std_ulogic;
signal tmp0,tmp1,tmp2:
std_logic_vector(size-1 downto 0);
begin
tmp0,tmp1,tmp2;
process (d0,d1,d2,s0,s1,s2)
 begin
  if s0=one then tmp0 <= d0;
   else tmp0 <= (others=>'X');
  end if;
  if s1=one then tmp1 <= d1;
      else tmp1 <= (others=>'X');
        end if;
  if s2=one then  tmp2 <= d2;
   else tmp2 <= (others=>'X');
   end if;

-- one hot контроль ниже  --

-- synopsys translate_off
 onehotcheck<= (s0 and s1) or
   (s0 and s2) or (s1 and s2);
 -- synopsys translate_on
 end process;
process(tmp0, tmp1, tmp2,
  onehotcheck)

 begin
 -- synopsys translate_off
  if (check =one and
     onehotcheck='1'
       ) then
        o <= (others=>'X');
  else
  -- synopsys translate_on
     o <= tmp0 or tmp1 or tmp2;
    end if;
end process;
end;
```

```
output [size-1:0] o; reg [size-1:0] o;
input [size-1:0] d0, d1, d2;

input s0, s1, s2;

                    reg [size-1:0]

always @(d0 or d1 or d2 or s0 or s1 or s2),
    begin

if(s0) tmp0 = d0; else tmp0 = 0;

if(s1) tmp1 = d1; else tmp1 = 0;

 if(s2) tmp2 = d2; else tmp2 = 0;
   end
// one hot check
   wire  onehotcheck;
// synopsys translate_off
  assign onehotcheck=(s0 & s1) |
  (s0 &  s1) |(s1 & s2);
// synopsys translate_on

always @(tmp0 or tmp1 or
      tmp2 or onehotcheck)
  begin
// synopsys translate_off
  if (check == 1 &&
     onehotcheck == 1)

   o <= {size{1'bx}};
   else
// synopsys translate_on
     o <= tmp0 | tmp1 | tmp2;
   end
endmodule
```

Пример 5. Структурное VERILOG-описание 8-входового n-разрядного мультиплексора, построенного из 4- и 2-разрядных.

Читателю предлагается самостоятельно построить аналогичное VHDL-описание, а также более быстрое с точки зрения моделирования процессное описание с использованием одного case (8 ветвей) в процессе.

```
module mux8n(yo, d0, d1, d2, d3, d4, d5, d6, d7, s);
parameter size = 1;
output [size-1:0] yo;
input [size-1:0] d0, d1, d2, d3, d4, d5, d6, d7;
input [2:0] s;
   wire [size-1:0] t0;
   mux4n #(size) i0(t0, d0, d1, d2, d3, {s[1], s[0]});  wire [size-1:0] t1;
   mux4n #(size) i1(t1, d4, d5, d6, d7, {s[1], s[0]});  wire [size-1:0] t2;
   mux2n #(size) i2(t2, t0, t1, s[2]);
   assign yo = t2;
endmodule
```

Пример 6. Описание 2^n-входового 1-разрядного мультиплексора.

VHDL

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity mux_n is
 generic(n:integer:=5);
  port (I_bus : in std_logic_vector
                  (2** n-1 downto 0);
          s : in std_logic_vector
                   (n-1 downto 0);
      muxout : out std_logic );
end;
architecture beh of mux_n is
begin
mux: muxout<=I_bus((conv_integer( s)));
end;
```

VERILOG

```
module mux_n (I_bus,s,muxout);
   parameter n=5;

  input[1 <<n -1:0] I_bus;

  input [n-1 :0]s;
  output muxout;

  assign muxout=I_bus[s] ;
endmodule
```

5.6.2. Дешифраторы (демультиплексоры)

Пример 1. Простой дешифратор 2-4.

VHDL

```
   entity decod2_4   is
    port(a:in std_logic_vector
                  (1 downto 0);
         sel:out std_logic_vector
                  (3 downto 0);
  end decod2_4   ;
architecture  synt of decod2_4 is
 begin
 process (a)
   begin
    case (a) is
     when "00" => sel<="0001";
     when "01" => sel<="0010";
     when "10" => sel<="0100";
     when others => sel<="1000";
    end case;
 end process; end;
```

VERILOG

```
module vdecod2_4(a,sel);

 input [1:0]a;

 output [3:0]sel;
 reg[3:0] sel

 always @(a)

 case(a)
     2'b00: sel=4'b0001;
     2'b01: sel=4'b0010;
     2'b10: sel=4'b0100;
     default: sel=4'b1000;
  endcase
endmodule
```

(Читателю предлагается построить более точную модель без X-оптимизма: при a, содержащем X, выход sel поставить в X и посмотреть, что даст синтезатор.)

Более компактно описание процесса со сдвигом кода 0001 влево.

```
sel<=shl (B"0001",a) ;                    sel<=4'b0001 << a;
```

(VHDL — надо подключить пакет типа IEEE.STD_logic_unsigned.)

Пример 2. Энергосберегающий дешифратор 2-4 с сигналом разрешения.

Такой дешифратор активизируется только в те моменты, когда его выход задействован, т. е. подан сигнал разрешения en, что полезно в смысле энергосбережения.

VHDL

```
entity decod2_4en   is

    port(en: std_logic;
          a:in std_logic_vector
                    (1 downto 0);
         sel:out std_logic_vector
              (3 downto 0));
  end decod2_4en   ;
 architecture  synt of decod2_4en is
  begin
   process (en,a)
   variable xx: std_logic_vector
                 (2 downto 0);
  begin
    xx:= (en & a);
    case xx is
     when "100" => sel<="0001" ;
     when "101" => sel<="0010" ;
     when "110" => sel<="0100";
     when "111" => sel<="1000" ;
    when others => sel<= "0000";
   end case;
  end process;
end;
```

VERILOG

```
module  decod2_4en
              (en,a,sel);
    input en;

     input [1:0]a;

     output [3:0]sel;
     reg[3:0] sel;


      always @(en or a)

    begin

     case({en,a})
          3'b100: sel=4'b0001;
          3'b101: sel=4'b0010;
          3'b110: sel=4'b0100;
          3'b111: sel=4'b1000;
          default :sel=4'b0000;
      endcase
  end
endmodule
```

Пример 3. Дешифратор 3-8.

Его описание выполнено в более компактной, но менее наглядной форме.

VHDL

```
library ieee;
use ieee.numeric_std.all;
entity decoder3_8 is
  port (
    a : in std_logic_vector(2 downto 0);
    q : out std_logic_vector(7 downto 0));
end decoder3_8;

architecture behavior of decoder3_8 is
   signal n_a : unsigned(2 downto 0);
   signal n_q : unsigned(7 downto 0);
```

VERILOG

```
module decoder3_8 (a,q);

    input [2:0]a;
    output [7:0]q;
          reg[7:0]q;
    integer i;
```

```
begin
  n_a <= unsigned(a);
  process (n_a)
   variable i: integer;
  begin

    for i in 7 downto 0 loop
      if (n_a=i) then
        n_q(i) <= '1';
      else
        n_q(i) <= '0';
      end if;
    end loop;
  end process;
  q <= std_logic_vector(n_q);
end behavior;
```

```
always @(a)

for( i=7;i>=0;i=i-1)begin
if(a==i)

  q[i] <= 1;
else
  q[i] <= 0;

end

endmodule
```

5.6.3. Тристабильный буфер-ключ

Потоковое описание n-разрядного ключа.

VHDL

```
entity buf is
 generic (size:natural:=1);
 port (buf_out: out std_logic_vector
                    (size-1 downto 0);
       buf_in :in  std_logic_vector
                    (size-1 downto 0);
       e        :in std_logic);
end;
architecture beh of buf is
  begin
  buf_out <= buf_in when e='1'
            else   ( others=> 'Z');
 end beh;
```

VERILOG

```
module buf (buf_out,buf_in,e);
    parameter size=1;
    output[size-1:0] buf_out;

    input  [size-1: 0] buf_in;
    input   e;

assign buf_out= (e==1)?buf_in:
                        {size{1'bz}};
endmodule
```

Напоминаем, что в VHDL функция разрешения конфликтов на общей шине (resolved) ассоциирована с типом сигналов std_logic пакета std_logic_ll64, а в VERILOG — с соединениями вида wire и tri.

5.6.4. n-разрядный компаратор

VHDL

```
entity compare is
  generic (size n:positive:=32);
  port(compout: out std_logic;
      a,b :std_logic_vector
      (size -1 downto 0));
end;
architecture beh of compare is
   begin
      compout <=(a=b);
end;
```

VERILOG

```
module compare(compout, a, b);
   parameter size = 32;
    output compout;
    input [size-1:0] a;
    input [size-1:0] b;

 assign compout = (a == b);

endmodule // compare
```

5.6.5. Типичные ошибки в описании комбинационных узлов

5.6.5.1. X-пессимизм и X-оптимизм

Следует не забывать о погрешности (с точки зрения возможностей моделирования в многозначном алфавите) синтезабельных описаний комбинационных узлов, связанной с отождествлением, например, неопределенного значения управляющих сигналов с 0. Оптимизм игнорирования возможного X-значения сигнала селектора был отмечен в описании мультиплексора. X-пессимизм арифметических операций также был указан ранее (0001 + 000X дает XXXX, а не 00XX). При моделировании это иногда затрудняет поиск ошибок в проекте, вызывающих неопределенность управляющих сигналов, ибо X порой не «проходит» через синтезабельные описания узлов на внешние входы схем, которые обычно сравниваются с эталоном.

5.6.5.2. Неожиданные триггеры-защелки

Пример возможной ошибки в описании комбинационных схем — неожиданное появление триггеров-защелок. Отсутствие (см. ниже) одной ветви в нашем примере описания мультиплексора в операторе case (неполный CASE) или прочего выбора (others — VHDL, default — VERILOG) является ошибкой (при отсутствии специальных указаний системе синтеза), влекущей появление триггера-защелки при синтезе.

VHDL

```
process (s, c, d, e, f)
begin
  case s is
    when "00" => muxout <= c;
    when "01" => muxout <= d;
    when "10" => muxout <= e;
  end case;
```

VERILOG

```
always (s or c or d or e or f)

  case (s)
     2'b00 : muxout <= c;
     2'b01 : muxout <= d;
     2'b10 : muxout <= e;
  endcase
```

Поэтому при описании комбинационных схем целесообразно всегда проверять, все ли возможные альтернативы условий учтены, и если нет — имеет место хранение — синтезируется защелка.

5.6.5.3. Непредвиденные различные задержки по входам

Ниже пример описания мультиплексора с приоритетным дешифрированием управляющего кода. Соответственно синтезируется приоритетная схема, имеющая различные задержки на выходе для входов c, d, e, f.

VHDL

```
entity priority_mux is
  port (pout : out std_logic;
          c, d, e, f : in std_logic;
          s : in std_logic_vector
           (1 downto 0);
end;
architecture pr_arch of
                    priority_mux is
begin
myif_pro: process (s, c, d, e, f)
begin
   if s = "00" then
```

VERILOG

```
module priority_mux(pout, c, d, e, f, s);
    output pout;
    input c, d, e, f; input[1:0]s;

    reg pout;

//VERILOG-2000
always (s, c, d, e, f)
    begin
      if (s == 2'b00)
```

```
    pout <= c;
  elsif s = "01" then
    pout <= d;
  elsif s = "10" then
    pout <= e;
  else  pout <= f;
  end  if;
 end process myif_pro;
   end;
```

```
    pout <= c;
  else if (s = 2'b01)
    pout <= d;
  else if (s = 2'b10)
    pout <= e;
  else
    pout <= f;
 end
endmodule //
```

5.6.5.4. Неполный case может породить приоритетный шифратор

Для ряда САПР ПЛИС характерно покрытие синтезируемой схемы сначала некоторым набором обобщенных узлов, например, обобщенный мультиплексор для FPGA типа VIRTEX фирмы Xilinx может иметь от 4 до 256 входов. Однако для этого порождения требуется, чтобы в его образе в CASE было бы задействовано не менее 75% возможных ветвей или же присутствовало указание-комментарий системе синтеза о порождении мультиплексора (в этом случае можно задействовать даже 40% ветвей).

VHDL

```
entity mux8_1 is
port (a,b,c,d,e,f: in std_logic;
        sel: in std_logic_vector(2 downto 0);
        mux_out: out std_logic);

end;
architecture mux_infer of mux8_1 is
 begin
  process (a,b,c,d,e,f,sel)
   begin
     case (sel) is--synopsys infer_mux
       when "000" =>mux_out<= a  and b,
       when "001" =>mux_out<= a  or c;
       when "010" =>mux_out<= d  or e;
       when others =>mux_out<= d  xor f;
     end case
   end process;
 end;
```

VERILOG

```
module mux8_1
   (a,b,c,d,e,f,sel,mux_out);
  input a,b,c,d,e,f;
  input [2:0]sel;
  output mux_out;
  reg mux_out;

 //VERILOG-2000
 always @( a,b,c,d,e,f,sel)

  case (sel) //synopsys infer_mux
    3'b000: mux_out=a & b;
    3'b001: mux_out=a | b;
    3'b010: mux_out=d | e;
    default: mux_out=d ^ f;
  endcase

endmodule
```

В данном случае на входе мультиплексора стоят вентили И, ИЛИ, XOR.

5.6.5.5. Излишний расход аппаратных ресурсов

В предыдущем примере по каждой ветви case предусматривалось выполнение логических операции, в нашем случае они простые и не нужно предусматривать специальных мер, но в случае сложных операций лучше не рисковать и сначала получить промежуточные сигналы типа tmp1<=a and b; tmp2=a or c; и т. д., а уж потом подать их на вход обычного четырехвходового мультиплексора.

Совмещенное использование аппаратных ресурсов уменьшает затраты оборудования.

Пример фрагмента HDL-описания процесса, порождающего в синтезируемой схеме два сумматора на входе мультиплексора.

VHDL

```
signal sum:std_logic;
if(select='1') then
  sum<= a+b;
else
  sum<=c+d;
end if;
```

Verilog

```
reg sum;
if (select)
  sum<= a+b;
else
  sum<= c+d;
```

Пример оптимизированного описания, порождающего два мультиплексора и один сумматор (в VHDL необходимо добавить пакет для сложения std_logic).

```
variable tmp1, tmp2:std_logic;
if (select='1') then
  tmp1 :=a;
  tmp2 :=b;
else
  tmp1 :=c;
  tmp2 :=d;
end if;
sum <=tmp1+tmp2;
```

```
reg tmp1, tmp2;
if (select) begin
  tmp1 = a;
  tmp2 = b;
end
else begin
  tmp1 = c;
  tmp2 = d;
end
sum <=tmp1 + tmp2;
```

Циклы, порождающие много подсхем — копий тела цикла

Этот пример взят из практики автора. Стиль кодирования далек от идеала.

Поведенчески описывается комбинационный узел, выделяющий старшую единицу в коде 8-разрядного регистра IR, причем номер разряда, с которого начинается поиск старшей единицы в IR, определяется трехразрядным сигналом INT_MAX. Результат поиска заносится в регистр INT_NUM. Используется САПР Synopsys. Подключаются IEEE пакеты, содержащие функции преобразования типов std_logic_unsigned.all;std_arith.all; std_logic_misc.

В архитектуре контроллера описаны регистры IR, INT_NUN и INT_MAX.

VHDL

```
signal IR: std_logic_vector (7 downto 0)
signal INT_NUM, INT_MAX:
              std_logic_vector
                    (2 downto 0);
PI: process (IR, INT_MAX)
variable N,j: integer;
  begin
-- получение целочисленного значения
N:=CONV_integer (INT_MAX);
-- проводится поиск с единицы
for j In 0 to 7 loop    -- цикл от j-0 до 7

   If IR(N)='1' then
     INT_num <=
          CONV_std_logic_vector (N,3);
     Exit;
  end If;
-- изменение номера  по модулю 8
  If N=7 then
    N:=0;
```

VERILOG

```
reg [7:0] ir;

reg [2:0] int_max, int_num;
integer I,j;
reg [2:0[ n;

n=int_num;
for(j=0;j<=7; j=j+1)
  begin
  if(ir[n]==1)begin

  int_num= n;
  break;
end
  if(n==7) n=0;
  else        n=n+1;
```

```
    else N:=N+1;
    end if;
  end loop;                                    end //for
  end process;
```

Хотя это описание узла верно с точки зрения HDL и синтезабельно с точки зрения САПР Synopsys, но порожденная в ходе синтеза схема получилась крайне неэффективной по оборудованию. Система синтеза обрабатывает цикл for, копируя 8 раз его тело в порождаемую схему, т. е. порождает 8 подсхем. Каждая подсхема содержит не только схему проверки значения соответствующего разряда на 1, но и сумматор — инкрементор, увеличивающий значение в регистре N на 1. Обнаружить неэффективность схемы оказалось возможным только после анализа данных о числе вентилей в синтезированной схеме контроллера (этот узел являлся его небольшим фрагментом) и просмотра с помощью графического редактора синтезированной схемы, где 8 сумматоров оказались неприятным сюрпризом. Пришлось для этого узла продумать сначала его схемную реализацию на уровне вентилей, а потом описать ее на HDL. Читатель может поупражняться в построении подобной схемы и ее описании. Так что на автоматизированный синтез надейся, а сам не плошай!

5.6.6. Результаты синтеза одноразрядного сумматора

Здесь приводится пример потокового VHDL-описания 1-разрядного сумматора, рассмотренного в главе 1.

```
library ieee;use ieee.std_logic_1164.all;
entity  adder is
  port  (a, b, c        :std_logic;   sum , cout        :out std_logic);
end  adder;
architecture  behave of adder is
begin
   sum  <= (a xor b) xor c;
   cout   <=  (a and b) or (a and c) or (b and c);
end  behave;
```

Описание сумматора было обработано синтезатором системы синтеза SINPLIFY. Его графическое представление синтезатором до покрытия технологическим базисом на рис. 5.7, а после покрытия технологическим — на рис. 5.8. Просматривая синтезированный VHDL-текст, можно заметить, что если для суммы имеем схему из XOR, то перенос реализуется так:

```
N2<= a and b; N3<= a or b;  cout<=N2 when c='1' else N3;
```

Далее представлен протокол работы системы. Комментарий на русском языке поясняет результаты. Синтез был задан в базисе FPGA VIRTEX фирмы Xilinx (подробнее этот тип FPGA рассмотрен в главе 6), но в первом приближении этот кристалл имеет буферные элементы для подключения к внешним сигналам (IO buf) и логические блоки (LUT), реализующие логические функции сложностью до 4 входных переменных. Проектировщик задал очень заниженные параметры синтезируемой схемы (частота 1 МГц) по сравнению с возможностями кристалла.

Рис. 5.7. Графическое представление схемы сумматора

Рис. 5.8. Схема сумматора в технологическом базисе

Полученная задержка — 7.7ns включает задержку входных (1.3 ns) и выходных буферов (5.7 ns). Естественно, что, когда такой сумматор используется внутри кристалла, задержки буферов не имеют места.

```
VHDL syntax check successful!- - синтаксис в порядке
Synthesizing work.adder.behave
Process took 0.0599999 seconds realtime, 0.06 seconds cputime - затраченное время
Setting fanout limit to 100 - проектировщик задал максимальную нагрузку на
выходе=100
List of partitions to map:
  view:work.adder(behave)
Net buffering Report for view:work.adder(behave):
No nets needed buffering.
        ##### START TIMING REPORT #####
     --Отчет о задержках в полученной схеме
        Performance Summary
        ************************
частота   Требование   Полученная   Период
          проектировщика  частота   затребованный полученный  Запас
          Requested      Estimated  Requested     Estimated
Clock     Frequency      Frequency  Period        Period      Slack
-----------------------------------------------------------------------
System    1.0 MHz        129.6 MHz  1000.0        7.7         992.3
=======================================================================
        Interface Information
        **********************
Input Ports:
Port      Reference   User        Arrival   Required
Name      Clock       Constraint  Time      Time      Slack
-----------------------------------------------------------------------
a         System      0.0         0.0       992.3     992.3
b         System      0.0         0.0       992.3     992.3
c         System      0.0         0.0       992.3     992.3
=======================================================================
Output Ports:
Port      Reference   User        Arrival   Required
Name      Clock       Constraint  Time      Time      Slack
-----------------------------------------------------------------------
cout      System      0.0         7.7       1000.0    992.3
sum       System      0.0         7.7       1000.0    992.3
=======================================================================
        Detailed Timing Report for clock : System
        *****************************************
Requested Period     1000.0 ns
Estimated Period     7.7 ns
Worst Slack          992.3 ns
-- Дальнейшая часть отчета о задержках опущена
        ##### END TIMING REPORT #####
-----------------------------------------------
Resource Usage Report - затраты оборудования
Mapping to part: xcv50bg256-4
Cell usage:
GND          1 use    VCC          1 use
```

```
I/O primitives:
OBUF        2 uses    IBUF        3 uses
I/O Register bits:                0
Register bits not including I/Os:  0 (0%)
Mapping Summary:Total  LUTs:  2 (0%)   -- затрачено 2 LUT
Mapper successful!
Process took 0.66 seconds realtime, 0.66 seconds cputime
```

Ниже VHDL-описание полученной в результате синтеза схемы. Обычно формат EDIF, а не VHDL.

Схема включает в цепи суммы один компонент типа LUT3_96 с XOR на три входа и компонент типа LUT3_E8, содержащий двухвходовые вентили И и ИЛИ. Обратите внимание на появление задержек сигналов в элементах — это может быть одной из причин несовпадения результатов совместного моделирования исходного функционального и синтезированного описаний.

Текст отредактирован для уменьшения числа страниц — в строке несколько операторов.

```
library ieee;use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library synplify;use synplify.components.all;
-- ниже элемент формирования переноса LUT3_E8
entity LUT3_E8 is
port( I0 :  in std_logic; I1 :  in std_logic;
  I2 :  in std_logic; O :  out std_logic);
end LUT3_E8;
architecture beh of LUT3_E8 is
  signal N_2 : std_logic ; signal N_3 : std_logic;
  signal GND : std_logic ; signal VCC : std_logic;-- ПИТАНИЕ
begin
  N_2 <= I2 and I1 after 100 ps; N_3 <= I2 or I1 after 100 ps;
  O <= N_2 after 100 ps when I0 = '0' else N_3 after 100 ps;
  GND <= '0'; VCC <= '1';
end beh;
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
-- ниже элемент формирования суммы---------- LUT3_96
entity LUT3_96 is
port( I0 :  in std_logic;  I1 :  in std_logic;
  I2 :  in std_logic;  O :  out std_logic);
end LUT3_96;
architecture beh of LUT3_96 is
  signal GND : std_logic ; signal VCC : std_logic ;
  signal N_18 : std_logic ;  signal I0_I_0 : std_logic ;
  signal I2_I_0 : std_logic ;  signal G_11_I : std_logic ;
begin
  GND <= '0';  VCC <= '1';
  O <= not N_18; I0_I_0 <= not I0; I2_I_0 <= not I2;
  G_11_I <= I1 xor I0_I_0 xor I2_I_0 after 100 ps;
  N_18 <= not G_11_I;
end beh;
--
```

```
library ieee;use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity IBUF is
port( O : out std_logic; I : in std_logic);
end IBUF;
architecture beh of IBUF is -- ниже буфера ввода-вывода IBUF, OBUF
 signal NN_1 : std_logic ; signal NN_2 : std_logic ;
begin
 O <= I; NN_1 <= '1'; NN_2 <= '0';
end beh;
--
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity OBUF is
  port( O : out std_logic; I : in std_logic);
end OBUF;
architecture beh of OBUF is
 signal NN_1 : std_logic ; signal NN_2 : std_logic ;
begin
 O <= I; NN_1 <= '1'; NN_2 <= '0';
end beh;
library ieee;use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity GND is port( G : out std_logic); -- ниже питание Vcc и земля GND
end GND;
architecture beh of GND is   begin G <= '0';
end beh;
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity VCC is port( P : out std_logic);end VCC;
architecture beh of VCC is
begin P <= '1';end beh;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library synplify;
use synplify.components.all;
entity adder is --ниже сумматор adder
port( a : in std_logic; b : in std_logic;
 c : in std_logic; sum : out std_logic;
 cout : out std_logic);
end adder;
architecture beh of adder is
 signal SUM_C : std_logic; signal A_C : std_logic;
 signal B_C : std_logic; signal C_C : std_logic;
 signal N_2_I : std_logic; signal NN_1 : std_logic;
 signal NN_2 : std_logic;
 component GND port(G : out std_logic);
```

```
  end component;
  component LUT3_E8 port(I0 : in std_logic;I1 : in std_logic;
     I2 : in std_logic; O : out std_logic);
  end component;
  component VCC port(P : out std_logic);
  end component;
  component IBUF port(O : out std_logic; I : in std_logic);
  end component;
  component OBUF
   port(O : out std_logic; I : in std_logic);
  end component;
  component LUT3_96
   port(I0 : in std_logic; I1 :in std_logic;
     I2 : in std_logic; O : out std_logic);
  end component;
begin
  II_GND:  GND port map (G => NN_1);
  II_N_2_I: LUT3_E8 port map (I0 => C_C,
     I1 => A_C, I2 => B_C, O => N_2_I);
  II_VCC:  VCC port map (P => NN_2);
  II_A_IBUF: IBUF port map (O => A_C, I => a);
  II_B_IBUF: IBUF port map (O => B_C, I => b);
  II_C_IBUF: IBUF port map (O => C_C, I => c);
  II_COUT_OBUF: OBUF port map (
     O => cout, I => N_2_I);
  II_SUM: LUT3_96 port map (I0 => B_C,
     I1 => C_C,I2 => A_C,O => SUM_C);
  II_SUM_OBUF: OBUF port map (O => sum, I => SUM_C);
end beh;
```

Вопросы к разделу 5.6

1. Как экономить оборудование при описании схем?
2. В каких случаях синтезатор порождает нежелательные триггеры-защелки?
3. Почему рекомендуется использовать case в описании мультиплексоров?

5.7. Триггеры и регистры

Общая структура описаний

При синтезабельном HDL-описании схем с памятью обычно используются процессные формы со списком чувствительности и условным оператором, однократно включающим фронтовое условие (фронта или среза) тактирующего сигнала clk (в общем виде ниже обозначено как ..edge) или процессные формы без списка чувствительности, но с оператором ожидания (wait), включающим фронтовое условие.

Примеры фронтовых условий:

Условие	VHDL	VERILOG
Фронт	clk'event and clk='1' rising_edge (clk) not clk'stable and clk='1'	posedge clk -в списке чувствительности
Срез	clk'event and clk='0' falling_edge (clk) not clk'stable and clk='0'	negedge clk --в списке чувствительности

Ниже приведен пример обобщенной структуры описания с условным оператором. Многоточием обозначен префикс перед egde (falling_edge, rizing_edge, posedge, negedge).

Узлы (триггеры, регистры) с асинхронным сбросом (установкой):

VHDL

```
process ( clk,
   rst)
 begin
if  (rst='1')  then
   операторы
elsif   ...edge (clk) then
   операторы
end if;
end process;
```

VERILOG

```
always @(...edge clk,
   ...edge rst)
  begin
  if (rst)begin
     операторы        end
  else              begin
     операторы        end

  end
```

Узлы памяти с синхронным сбросом (установкой):

VHDL

```
process (clk)
 begin
   if ... egde (clk)  then


      if rst='1' then
        операторы
     else
        операторы
    end if;
  end if;
end process;
```

VERILOG

```
always @(... egde clk)
      begin
      if (rst) begin
        операторы
      end
      else begin
        операторы

      end

end//always
```

Теперь приведем пример обобщенной структуры описания узлов без сброса (установки) с применением оператора ожидания wait.

VHDL

```
process
 begin
 wait until ... edge (clk) then
 операторы
end process;
```

VERILOG

```
always
      begin
      @ (...edge clk);begin
      операторы    end
end
```

Как уже отмечалось, приведенные ниже синтезабельные VHDL-описания предполагают подключение пакета std_logic-1164, а VERILOG — заданный масштаб времени 1 ns с точностью до 100 ps. Сравните приведенные ниже описания триггеров с описаниями узлов, построенными синтезатором при синтезе схемы счетчика (VHDL) и схемы автомата управления светофором (VERILOG) — стили несколько отличаются, но идеи те же.

5.7.1. D-триггер-асинхронный сброс-установка

Пример 1. D-триггер с асинхронным установочным входом set, сбросом reset, информационным входом D и стробом CLK.

VHDL

```
Library IEEE;
Use IEEE.std_logic_1164.all;
entity dffa is
 port (clk, set, reset,din:
                in std_logic;
          dout: out std_logic);
end;
architecture beh of dffa is
 begin
process (clk,set, reset)


begin
   if reset='1' then
      dout <= '0';
   elsif set ='1' then
       dout <= '1';
   elsif (clk'event and clk='1') then
      dout <= din after 1 ns;
   end if;
end process;
end;
```

VERILOG

```
 timescale 1 ns/100 ps
module dffa( clk,set,reset,din,dout),

 input clk,set,reset,din;
 output dout;
 reg dout;


always @(posedge clk or
            posedge set
           or posedge reset)
  begin
    if (reset ==1'b1)
     dout<= 1'b0;
     else if (set ==1'b1)
     dout<= 1'b1;
       else
         dout <= #1 din;
   end
endmodule //dff
```

Следует еще раз пояснить использование операции <= (а не =) и явной задержки в описаниях триггеров. Задержка обычно игнорируется системой синтеза, и она нужна только для упрощения поиска ошибок в схеме в процессе моделирования (триггеры имеют задержки, видные на временных диаграммах, а комбинационные схемы — нет).

5.7.2. Триггер-синхронный сброс и установка

Пример 2. D-триггер с синхронным (по фронту clk) установочным входом set, сбросом reset, информационным входом din.

VHDL

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity dffs is
 port (clk,set,reset,din:
                in std_logic;
```

VERILOG

```
 `timescale 1 ns/100 ps
module dffs( clk,set,reset,din,dout);

 input clk,set,reset,din;
```

```
        dout: out std_logic);
end;
architecture beh of dffs is
begin
process (clk)
begin
  if clk'event and clk='1' then
   if reset='1' then
     dout <= '0';
   elsif set ='1' then
      dout <= '1';
   else
     dout <= din after 1 ns;
   end if;
end process;
end;
```

```
output dout;
reg dout;


always @(posedge clk )
  begin
    if (clk==1'b1)begin
       if (reset ==1`'b1)
          dout<= 1'b0;
       else if (set ==1'b1)
          dout<= 1'b1;
       else
          dout <= #1 din;
    end //if clk
end
endmodule
```

Нетрудно построить на базе приведенного примера описания триггеров без сброса или установки и путем добавления параметра n описания n-разрядных регистров.

Следует еще раз обратить внимание на X-оптимизм с точки зрения моделирования. В VHDL-описании триггера правильнее было бы использовать имеющуюся в пакете std_logic_1164 функцию rising_edge при описании условия фронта clk, которая дает значение TRUE (истина) только при смене clk из «0» в «1» и не дает его, в отличие от (clk'event and clk='1'), при смене clk, например из «X» в «1».

5.7.3. Регистры с разрешающим входом

Пример 3. n-разрядный регистр с асинхронным сбросом и разрешением/приема.

VHDL

```
entity regenn is

  generic (size:positive:=32);
  port ( rout: out std_logic_vector
              (size-1 downto 0);
         rin : in std-logic_vector
              (size-1 downto 0);
       en,clr_n,clk: in std_logic);
  end;
  architecture beh of regenn is

 begin
   process(clk,clr_n) begin
     if (clr_n ='0' ) then
         rout <= (others=>'0');
    elsif clk'event and clk='1' then
         if  en='1' then
     rout <= rin after 1 ns;
        end if;
   end if;
 end process;
 end;
```

VERILOG

```
module regenn(rout, rin,
                 en, clr_n, clk);
  parameter size = 32;

  output [size-1:0] rout;
  reg [size-1:0] rout;
  input [size-1:0]  rin;
  input en,clr_n,clk;

  always @(posedge clk or
          negedge clr_n)

  begin
     if (!clr_n)
          rout <= 0;
     else
          if (en)
           rout <= #(1) rin;

  end

endmodule
```

5.7.4. Защелки

Пример 4. Триггер-защелка (регистр).

Как уже отмечалось, не рекомендуется применять защелки в обычных схемах (может быть, только при синтезе блоков регистровой памяти для экономии оборудования).

VHDL

```
entity rlatchn    is
 generic(size: integer:=32);
 port( rout:out std_logic_vector
            (size-1 downto 0);
        rin:in std_logic_vector
            (size-1 downto 0);
        en: in std_logic);
end;
architecture beh of rlatchn is
begin
  process (en,rin)
   begin
   if en ='1' then
       rout <=  rin after 1 ns;

  end if;
end process;
end;
```

VERILOG

```
module rlatchn(rout,rin,en);
  parameter size = 32;

  output [size-1:0] rout;

  input [size-1:0] rin;
  input en;

  reg [size-1:0] rout;

always @(en or rin)
  begin
     if (en)

       rout <= #1 rin;
  end

endmodule
```

5.7.5. Сдвигатели

Пример 5. Образ 4-разрядного сдвигового регистра с последовательным входом и выходом (в комментариях предполагаемые описания входов-выходов).

VHDL

```
--        CLK,din: in STD_LOGIC;
--        DOUT: out STD_LOGIC;

process (clk)
  variable r_s: std_logic_vector
              (3 downto 0);
begin
  if clk'event and clk='1' then
    r_s := din & r_s(3 downto 1);
  end if;
  dout <= r_s(0) after 1 ns;
end process;
```

VERILOG

```
  //input clk,din;
  //output dout;reg dout;
  //reg [3:0] r_s;
always @(posedge clk)
begin

 if (clk==1'b1)
   r_s= {din,r_s[3:0]};

  dout<= #1 r_s[0];
end //always
```

5.7.6. Счетчики

Пример 6. n-разрядный счетчик с асинхронным сбросом.

VHDL

```
library IEEE; use IEEE.std_logic_1164.all;
entity count_up is
```

```
  generic(n: integer:=7);
  port (reset,clk: in std_logic;
        count:out  std_logic_vector(n-1 downto 0));
end;
library IEEE; use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all; use IEEE.std_logic_unsigned.all;
architecture beh of count_up is
  begin
  process (reset,clk)
 variable count_tmp:  integer;
 begin
     if reset ='1' then      count_tmp :=0 ;
     elsif clk'event and clk='1' then
                 count_tmp:= count_tmp+1;
     end if;
     count<=conv_std_logic_vector( count_tmp,n) after 1 ns;
   end process;
   end;
```

VERILOG

```
 module count_up(reset,clk,count);
 parameter n=7;
  input reset,clk;  output [n-1:0] count;
  reg [n-1:0] count;
  always @(posedge reset or  posedge clk )
   begin
             if (reset==1)begin
               count=0;
             end
             else begin
               count=count+1;
             end
        end
 endmodule
```

5.7.7. Регистровые файлы и блоки памяти

Обычно система синтеза описание памяти реализует как массив регистров, построенных на триггерах. Для того чтобы она использовала блочную память, следует дать ей специальные указания в виде установки параметров синтеза или в HDL-тексте в форме специального комментария. Для САПР Synplify это значение директивы syn_black_box и атрибута syn_ramstyle (последний у автора не срабатывал).

Пример 7.

VERILOG

Описание блока памяти, реализуемое на триггерах
а) (см. специальный комментарий)

```
module ram8 (datain,dataout,clk,addr,wr);
output[31:0] dataout;reg[31:0] dataout;
input clk,wr;input[31:0] datain;input[2:0] addr;
reg [31:0] mem  [7:0] /* synthesis syn_ramstyle="registers" */;
always @( posedge clk)begin
```

```
if (wr)  mem[addr]<= datain; // и так далее
```

b) Описание блока памяти (подробно этот блок памяти описан в главе 3), реализуемое блоками памяти FPGA VIRTEX (см. гл. 6).

```
module  ramb4(di, en, we,r st, clk, addr, do) /* synthesis syn_black_box */;
parameter        data_width = 8,   addr_width= 9, depth=256;
input  [data_width -1 : 0]di;  input     en,    we, rst,clk;
input   [data_width  : 0] addr;
output [data_width -1 : 0]   do;   reg [data_width -1 : 0]   do;
reg[data_width -1 : 0] mem[ depth-1 : 0] ;
// synthesis syn_ramstyle="block_ram" - не срабатывал у автора
```

VHDL

Описание блока памяти (подробно этот блок памяти описан в главе 3), реализуемое блоками памяти FPGA VIRTEX (см. гл. 6).

Подключается библиотека атрибутов и задается значение нужного.

```
library ieee;use ieee.std_logic_1164.all;
entity ram4 is
  port (d : in std_logic_vector(7 downto 0);
                addr : in  std_logic_vector(2 downto 0);
                we ,clk : in  std_logic;
                ram_out : out std_logic_vector(7 downto 0));
end ram4;
library synplify;use synplify.attributes.all;
architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector (7
  downto 0);
signal mem : mem_type;
-- mem is the signal that defines the RAM
attribute syn_ramstyle of mem : signal is "block_ram";
begin  --и т.д. end;
```

При использовании ram_4 как компоненты надо объявить attribute black_box: boolean; attribute black_box of ram_4: component is TRUE;.

5.7.8. Типичные ошибки в описаниях триггеров и регистров

Отсутствие нужных указаний — комментария системе синтеза.

Описание как бы RS-триггера защелки, на самом деле порождающее D-триггер.

VHDL

```
process (reset, set)
begin
  if reset='1' then
       dout <= '0';
  elsif set='1' then  --
       dout <= 1;
  end if;
end process;
```

VERILOG

```
always  @( reset or set)
     begin
     if( reset=1)
        dout <= 0;
     else if (set=1)
              dout <= 1;
end
```

При синтезе в САПР SYNOPSYS ему сопоставляется не RS-триггер, а D-триггер-защелка, так как это образ D-триггера (см. выше). До полностью син-

тезабельного RS-триггера ему недостает указания — комментария системе синтеза (one hot) о том, что выходы не могут быть одновременно в 1.

Несоблюдение синтезабельных рекомендаций типа: только один раз указывать фронт (срез), и тем более нельзя в одном условии задавать фронт, а в другом срез (это уже не триггер, а что-то невероятное).

Пример VHDL — не D-триггер. Заодно в приведенном примере полное игнорирование указаний по стилю кодирования.

```
Library IEEE;Use IEEE.std_logic_1164.all;
  entity no_dffa is
 port (clk,reset,din:     in std_logic;    dout: out std_logic);
 end;
architecture beh of no_dffa is
   begin
process (clk,set, reset)
begin
            if reset='1' and (clk'event and clk='1') then
                    dout <= '0';
           elsif set ='1' and  (clk'event and clk='0')  then
                     dout <= '1';
          elsif (clk'event and clk='1') then
            dout <= din after 1 ns;
          end if;
       end process;
end;
```

5.7.9. Пример синтеза счетчика

VHDL

```
-- 2-разрядный счетчик с асинхронным сбросом
-- описание фронтового условия с функцией Rising_edge
library ieee;use ieee.std_logic_1164.all;
entity counter is
   port (clk, reset: in std_logic;
        count: out std_logic_vector (1 downto 0) );
end counter;
library ieee;use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;use ieee.std_logic_arith.all;
architecture behave of counter is
   signal count_i : std_logic_vector (1 downto 0);
begin
   process (clk, reset)
    begin
      if (reset = '0') then
          count_i <= "00";
      elsif rising_edge(clk) then
                count_i <= count_i + '1';
      end if;
   end process;
   count <= count_i;
end behave;
```

Его графическое представление синтезатором до покрытия технологическим базисом на рис. 5.9, а после на рис. 5.10.

Рис. 5 9. Графическое представление схемы счетчика

Рис 5.10. Представление счетчика в технологическом базисе

Фрагмент протокола работы синтезатора (режим невключения IO буферов):

```
-- Синтезатор использовал в схеме два триггера (FDC )
Setting fanout limit to 10 - нагрузка 10 установлена пользователем
List of partitions to map: view:work.counter(behave)
Net buffering Report for view:work.counter(behave):
No nets needed buffering. Made 0 timing clusters
@N|The option to pack flops in the IOB has not been specified
Found clock clk with period 50 ns
 ##### START TIMING REPORT #####
 Performance Summary
*************************
```

Clock	Requested Frequency	Estimated Frequency	Requested Period	Estimated Period	Slack
clk	20.0 MHz	205.7 MHz	50.0	4.9	45.1

Порожденный VHDL текст описания схемы:

```
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity LUT2_6 is --схема XOR
port(I0 :  in std_logic;  I1 :  in std_logic;
  O :  out std_logic);
end LUT2_6;
architecture beh of LUT2_6 is
  signal I0_I : std_logic;  signal GND : std_logic;
  signal VCC : std_logic;  signal N_6 : std_logic;
begin
  I0_I <= not I0; GND <= '0';
  VCC <= '1';  O <= not N_6;
  N_6 <= I1 xor I0_I after 100 ps;
end beh;
-- D-триггер FDC из примитива prim_dff
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity FDC is
port(Q :  out std_logic;  D :  in std_logic;
  C :  in std_logic;  CLR :  in std_logic);
end FDC;
architecture beh of FDC is
begin
  I1_Q: prim_dff port map (Q, D, C, CLR, '0');
end beh;
--
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity INV is
port(I :  in std_logic;  O :  out std_logic);
```

```
end INV;
architecture beh of INV is begin  O <= not I;
end beh;
--- ОПИСАНИЯ обьектов GND;VCC опущены
--- они совпадают с приведенными в
-- результатах синтеза сумматора
library ieee;
use ieee.std_logic_1164.all;use ieee.numeric_std.all;
library synplify;use synplify.components.all;
entity counter is -- счетчик counter
port(clk :  in std_logic; reset :  in std_logic;
  data : in std_logic_vector(1 downto 0);
  count : out std_logic_vector(1 downto 0));
end counter;
architecture beh of counter is
  signal COUNT_I_0 : std_logic_vector(0 to 0);
  signal COUNTZ : std_logic_vector(1 downto 0);
  signal UN3_COUNT_I_AXBXC1 : std_logic ;
  signal RESET_I_0 : std_logic ;
  signal NN_1 : std_logic ; signal NN_2 : std_logic ;
  component GND port(G :  out std_logic);
  end component;
  component VCC  port(P :  out std_logic);
  end component;
  component FDC port(Q :  out std_logic;    D :  in std_logic;
     C :  in std_logic;     CLR :  in std_logic);
  end component;
  component INV
    port(I :  in std_logic;   O :  out std_logic);
  end component;
  component LUT2_6
    port(I0 :  in std_logic;     I1 :  in std_logic;
      O :  out std_logic  );
  end component;
begin
  II_GND: GND port map (G => NN_2);
  II_VCC: VCC port map (P => NN_1);
  \II_COUNT[0]\: FDC port map (
     Q => COUNTZ(0), D => COUNT_I_0(0),
     C => clk, CLR => RESET_I_0);
  \II_COUNT[1]\: FDC port map (
     Q => COUNTZ(1), D => UN3_COUNT_I_AXBXC1,
     C => clk,CLR => RESET_I_0);
  \II_COUNT_I[0]\: INV port map (
     I => COUNTZ(0), O => COUNT_I_0(0));
  II_RESET_I: INV port map ( I => reset, O => RESET_I_0);
  II_UN3_COUNT_I_AXBXC1: LUT2_6 port map (
     I0 => COUNTZ(1), I1 => COUNTZ(0),
     O => UN3_COUNT_I_AXBXC1);
  count(0) <= COUNTZ(0);count(1) <= COUNTZ(1);
end beh;
```

5.8. HDL-описания автоматов

5.8.1. Автоматы Мили и Мура

Известны два типа конечных автоматов (FSM-Finite State Machine): автомат Мура (Moore) и автомат Мили (Mealy). В автомате Мура выходные сигналы (out_syg) зависят (f) от текущего состояния (state) автомата. В автомате Мили выходы являются функцией (f) как текущего состояния, так и входных сигналов (in_syg). Состояние автомата изменяется по синхросигналу (clk). Ф — функция переходов.

Автомат Мили

```
out_sig=f (state);
nexstate=Ф (in_syg, state);
state = nexstate when posedge of clk;
```

Автомат Мура

```
out_syg=f (state, in_syg);
nexstate=Ф (in_syg, state);
state = nexstate when posedge of clk;
```

HDL-описание автомата в общем случае может состоять из трех частей-процессов, соответствующих отдельным частям автомата, как это показано выше, но часто состоит из двух или одного процесса.

Кодирование состояний автомата

Среди способов кодирования состояний автомата различают минимальное, случайное, противогоночное (код Грея) и позиционное (onehot) кодирование — каждому состоянию сопоставляется отдельный триггер.

Например, для хранения четырех состояний при минимальном кодировании достаточно двух триггеров, при противогоночном эти состояния могут меняться так: 00, 01, 11, 10; при позиционном необходимы четыре триггера и смена состояний может быть такой: 1000, 0100, 0010, 0001.

В качестве примера рассмотрим автомат управления светофором. При сигнале reset = 1 светофор не работает, если reset = 0, светофор начинает работать, зажигая зеленый и красный свет длительностью 60 секунд и желтый — 10 секунд. Допустим, тактовый генератор автомата выдает импульс синхронизации каждые 10 секунд. Каждая лампочка (красный, зеленый, желтый) управляется отдельным выходным сигналом автомата. Автомат имеет три рабочих состояния: зеленый, желтый, красный и одно нерабочее. При использовании позиционного способа кодирования состояний автомата достаточно трех триггеров (регистр с: зеленый, желтый, красный); при минимальном кодировании — двух.

5.8.2. VERILOG — описание и тест автомата управления светофором

Пример VERILOG-описания автомата управления светофором с разной длительностью периода красного, зеленого и желтого свечения.

```
//Описание счетчика COUNT_UP БЫЛО ВЫШЕ
//Ниже  VERILOG-описании упрощенного автомата управления
// пример последовательного кодирования состояний - ЭТО не лучший способ
// кодировки, но система синтеза может изменить способ кодирования
`timescale 1ns/100 ps
`define IDLE_ST    0                // не работает
```

```
`define RED_ST    1                // горит красный
`define GREEN_ST  2                // горит зеленый
`define YELLOW_ST 3                // желтый после зеленого
module simple_svetofor2 (clk, rst, red, green, yellow);
        input clk, rst; output red, green, yellow;
      parameter n = 7;// cnt size
      parameter T_RED = 5;//25; - другие задержки
      parameter T_GREEN = 3;//12;
      parameter T_YELLOW = 2;
   parameter state_size = 2;
   reg[state_size-1:0] svet_state, next_svet_state;
      reg cnt_rst,next_cnt_rst;   wire [n-1 :0]count;
     //автомат состоит из счетчика времени и автомата состояний
       count_up #(n) cnt (.reset(cnt_rst),.clk(clk),.count(count));
      //ниже комбинационная часть - определение следующего
      //   состояния автомата)
       always @( svet_state or count)
         begin
            next_svet_state = svet_state; // неизменное состояние
            next_cnt_rst= 0;
             case (svet_state)
                `IDLE_ST:  begin
                        next_cnt_rst=1;
                        next_svet_state=`YELLOW_ST;
                      end
                 `RED_ST: if(count==T_RED)begin
                        next_svet_state=`GREEN_ST;
                        next_cnt_rst=1;
                      end
                 `GREEN_ST: if(count==T_GREEN)begin
                         next_svet_state=`YELLOW_ST;
                         next_cnt_rst=1;
                        end
                 `YELLOW_ST: if(count==T_YELLOW)begin
                      next_svet_state=`RED_ST;
                      next_cnt_rst=1;
                    end
                 default:    next_svet_state= `IDLE_ST;
              endcase
            end // always
 // ниже смена состояния автомата по фронту clk
       always @ (posedge clk)
         if(rst==1)begin
                  svet_state<=`IDLE_ST;
                 cnt_rst<=1;
               end
         else begin
            svet_state<=next_svet_state;
          cnt_rst<= next_cnt_rst;
      end
 // ниже выработка выходных сигналов, зависящих
 // от состояния автомата
```

```
    assign red= (svet_state==`RED_ST)? 1: 0;
    assign green= (svet_state==`GREEN_ST)? 1 : 0;
   assign yellow = (svet_state==`YELLOW_ST)? 1 : 0;
endmodule // simple-svetofor ----------------------------------------
// Verilog   тест автомата светофора
// ниже
`timescale 1ns/100 ps
module TB_svetofor  (); // ---------------------------------------------
    reg  clk, rst;
   wire red, green, yellow; parameter n=7;
     always begin clk =0; #10; clk=1; #10; end // генератор тактов
     initial begin rst =1; #20; rst =0;#800;$finish; end // сброс-запуск
  simple_svetofor2 #(n) svet (.rst(rst),.clk(clk),
            .yellow(yellow),.red(red),.green(green));
endmodule
```

5.8.3. VHDL-описание и тест автомата управления светофором

Приведем пример VHDL-описания упрощенного автомата управления светофором с разной длительностью периода красного, зеленого и желтого светов. Способ кодирования состояний автомата в VHDL-описании не определен, он определяется ЗАДАНИЕМ РЕЖИМА ПРИ ЗАПУСКЕ СИНТЕЗАТОРА ИЛИ В ТЕКСТЕ ОПИСАНИЯ специальным комментарием, влияющим на работу системы синтеза.

```
--Ниже описание автомата
library IEEE;use IEEE.STD_LOGIC_1164.all;
entity simple_svetofor2 is
  generic (n: positive := 7;T_RED: positive:=5;
      T_green : positive:= 5; T_YELLOW: positive:= 3);
  port (clk, rst: in std_logic;
       red, green, yellow: out std_logic);
  end;
library IEEE;
use IEEE.STD_LOGIC_1164.all;use IEEE.STD_LOGIC_UNSIGNED.all;
  architecture beh of   simple_svetofor2 is
  type st_type is (IDLE_ST,RED_ST,GREEN_ST,YELLOW_ST);
  signal  svet_state, next_svet_state: st_type;
     signal  cnt_rst,next_cnt_rst: std_logic;
     signal   count: std_logic_vector (n-1 downto 0);
  --автомат состоит из счетчика времени и автомата состояний
   begin
  cnt_inst:entity       count_up generic map( n=> n)
          port map (reset =>cnt_rst,clk=>clk,count=>count);
       --ниже комбинационная часть - определение следующего
                --   состояния автомата)
         process (svet_state, count)
           begin
               next_svet_state <= svet_state;
               -- неизменное состояние
                next_cnt_rst <= "0";
```

```
            case (svet_state) is
                when   IDLE_ST => next_cnt_rst<='1';
                            next_svet_state<=YELLOW_ST;
                when RED_ST => if(count=T_RED ) then
                              next_svet_state<=GREEN_ST;
                              next_cnt_rst<='1';
                           end if;
                when GREEN_ST=>if(count=T_GREEN)then
                            next_svet_state<=YELLOW_ST;
                            next_cnt_rst<='1';
                           end   if;
                when YELLOW_ST=> if(count=3) then --T_YELLOW)then
                              next_svet_state<=RED_ST;
                              next_cnt_rst<='1';
                             end if;
              when others=>    next_svet_state<= IDLE_ST;
            end case;
        end  process;
  -- ниже смена состояния автомата по фронту clk
        process  (clk,rst)   begin
          if(rst='1')then
                    svet_state<=IDLE_ST;
                  cnt_rst<='1';
          elsif clk'event and clk='1' then
               svet_state<=next_svet_state;
               cnt_rst<=next_cnt_rst;
           end if;
        end process;
-- ниже выработка выходных сигналов, зависящих
-- от состояния автомата ----------------------------------
   red    <='1' when (svet_state=RED_ST)else'0';
   green  <='1' when (svet_state=GREEN_ST)else '0';
   yellow <='1' when (svet_state=YELLOW_ST)else '0';
end;
-- НИЖЕ ТЕСТ-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity simple_svetofor2_tb is
  -- Generic declarations of the tested unit
        generic(
        n : POSITIVE := 7;T_RED : POSITIVE := 5;
        T_green : POSITIVE := 3;T_YELLOW : POSITIVE := 2 );
end simple_svetofor2_tb;
architecture TB_ARCHITECTURE of simple_svetofor2_tb is
  -- Component declaration of the tested unit
  component simple_svetofor2
    generic(
        n : POSITIVE := 7;T_RED : POSITIVE := 5;
        T_green : POSITIVE := 3;T_YELLOW : POSITIVE := 2);
    port(
        clk : in std_logic;rst : in std_logic;
        red : out std_logic;green : out std_logic;
```

```
        yellow : out std_logic);
  end component;
  signal clk : std_logic;
  signal rst : std_logic;
  signal red : std_logic;
  signal green : std_logic;
  signal yellow : std_logic;
   begin
  -- Unit Under Test port map
  UUT : simple_svetofor2
        generic map (n => n,T_RED => T_RED,
              T_green => T_green,T_YELLOW => T_YELLOW)
        port map (clk => clk,rst => rst,
              red => red,green => green,
              yellow => yellow);
  -- Add your stimulus here ...
clkp:   process begin
          clk <='0' ; wait for 10 ns; clk<='1';
          wait for 10 ns ;
  end process;
rstp:   process begin
          rst <='0'; wait for 10 ns; rst<='1';
          wait for 20 ns ; rst <='0' ;wait;
  end process;
end TB_ARCHITECTURE;
configuration TESTBENCH_FOR_simple_svetofor2 of simple_svetofor2_tb is
  for TB_ARCHITECTURE
        for UUT : simple_svetofor2
              use entity work.simple_svetofor2(beh);
        end for;
  end for;
end TESTBENCH_FOR_simple_svetofor2;
```

В данном случае объявление конфигурации избыточно.

5.8.4. Синтез VERILOG-описания автомата управления светофором

Графическое представление автомата управления светофором синтезатором до покрытия технологическим базисом представлено на рис. 5.11, а его увеличенный фрагмент — на рис. 5.12.

Виден (справа) прямоугольник блока выработки состояний автомата, прямоугольник 7-разрядного счетчика, триггер сигнала сброса счетчика, цепи (слева) выработки сигнала сброса, сравнивающие состояние счетчика с заданными периодами свечения.

Фрагмент протокола работы синтезатора:

```
$ Start of Compile    #Sat Jun 15 16:19:56 2002
@I::"c:\program files\simucad\silos2001demo\pol_des\svetofor_ver2.v"
@I::"c:\program files\simucad\silos2001demo\pol_des\count_up.v"
Verilog syntax check successful!
```

Рис. 5.11. Графическое представление схемы автомата светофора

Рис. 5.12. Фрагмент схемы автомата управления светофором

File c:\program files\simucad\silos2001demo\pol_des\svetofor_ver2.v changed - recompiling
Selecting top level module simple_svetofor2
Synthesizing module count_up
n=32'b00000000000000000000000000000111
Generated name = count_up_7
Synthesizing module simple_svetofor2
Trying to extract state machine for register svet_state
Extracted state machine for register svet_state
State machine has 4 reachable states with original encodings of:
00 01 10 11 -- исходная кодировка состояний
@END
Process took 0.17 seconds realtime, 0.17 seconds cputime

```
Setting fanout limit to 100
List of partitions to map:
  view:work.simple_svetofor2(verilog)
Automatic dissolve at startup in view:work.simple_svetofor2(verilog) of cnt(count_up_7)
Encoding state machine work.simple_svetofor2(verilog)-svet_state_h.svet_state[3:0]
original code -> new code
  00 -> 00  01 -> 01  10 -> 10  11 -> 11  -- новые коды состояний
@N:"c:\program files\simucad\silos2001demo\pol_des\count_up.v":9:2:9:7|Found
counter in view:work.simple_svetofor2(verilog) inst cnt.count[6:0]
Clock Buffers:
 Inserting Clock buffer for port clk,   TNM=clk
Net buffering Report for view:work.simple_svetofor2(verilog):
No nets needed buffering.
Made 0 timing clusters
@N|The option to pack flops in the IOB has not been specified
Found clock clk with period 1000ns
        ##### START TIMING REPORT #####--Временные параметры
        Performance Summary
       ************************
          Requested    Estimated    Requested    Estimated
Clock     Frequency    Frequency    Period       Period       Slack
-----------------------------------------------------------------------
clk       1.0 MHz      97.0 MHz     1000.0       10.3         989.7
=======================================================================
##### END TIMING REPORT #####
-----------------------------------------------------------------
Resource Usage Report -- затраты оборудования
Mapping to part: xcv50bg256-4
Cell usage:
FDC          7 uses - счетчик времени - семь триггеров
GND          1 use     VCC          1 use
XORCY        7 uses-сравнение
MUXCY_L      6 uses    FDS          1 use
FDR          2 uses  --состояния автомата - два триггера
I/O primitives:
OBUF         3 uses    IBUF         1 use
BUFGP        1 use
I/O Register bits:                 0
Register bits not including I/Os:   10 (0%)
Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)
Mapping Summary: Total  LUTs: 18 (1%)
Found clock clk with period 1000ns
Mapper successful!
// Source file index table:
// Object locations will have the form <file>:<line>
// file 0 "noname"
// file 1 "\c\simucad\silos2001demo\pol_des\svetofor_ver2.v"
// file 2 "\c:\simucad\silos2001demo\pol_des\count_up.v"
Ниже  фрагменты VERILOG-описания синтезированной схемы --------
`timescale 100 ps/100 ps
module GND ( G);output G;wire G ; assign G = 1'b0;
```

```
endmodule /* GND */
module LUT2_8 ( I0, I1, O);
input I0;input I1;output O;wire I0 ;wire I1 ;
wire O ;wire GND ;wire VCC ;
//@1:62
 assign #(1) O = I1 & I0 ;
//@1:62
 assign GND = 1'b0;
//@1:62
 assign VCC = 1'b1;
endmodule /* LUT2_8 */
module LUT4_0040 ( I0, I1, I2, I3, O);
input I0;input I1;input I2;input I3;
output O;wire I0 ;wire I1 ;wire I2 ;wire I3 ;
wire O ;wire I3_i ;wire I0_i ;wire GND ;wire VCC ;
//@1:62
  assign #(1) I3_i = ~ I3;
//@1:62
  assign #(1) I0_i = ~ I0;
//@1:62
  assign #(1) O = I3_i & I2 & I1 & I0_i ;
//@1:62
  assign GND = 1'b0;
//@1:62
  assign VCC = 1'b1;
endmodule /* LUT4_0040 */
/* ОПУЩЕНЫ ОПИСАНИЯ ЭТИХ МОДУЛЕЙ
 module LUT4_FFFE ( I0, I1, I2, I3, O);
 module LUT4_FDFF (I0, I1, I2, I3, O);
  module LUT3_EA ( I0, I1, I2, O);
 module LUT4_0D5D ( I0,I1, I2, I3, O);
*/
module VCC ( P); // питание
output P;wire P ; assign P = 1'b1;
endmodule /* VCC */
module BUFGP ( I, O);
input I;output O;wire I ;wire O ; assign #(1) O = I;
endmodule /* BUFGP */
//триггер ----------------------------------------------------
module FDC ( Q, D, C, CLR);
output Q;input D;input C;input CLR;
wire Q ;wire D ;wire C ;wire CLR ;
  reg r_e_g0; // dffr
  always @(posedge C or posedge CLR ) r_e_g0 = #1 CLR ? 1'b0 : D ;
   assign Q = r_e_g0;
endmodule /* FDC */
// МУЛЬТИПЛЕКСОР ---------------------------------------
module MUXCY_L ( DI, CI, S, LO);
input DI;input CI;input S;output LO;
wire DI ;wire CI ;wire S ;wire LO ;
  assign #(1) LO = (!S & DI ) |
     (S & CI ) | (DI & CI );
```

```
endmodule /* MUXCY_L */
// ОПУЩЕН module LUT2_6 ( I0, I1, O);
module XORCY ( LI, CI, O);
input LI;input CI;output O;wire LI ;wire CI ;wire O ;
  assign #(1) O = LI ^ CI ;
endmodule /* XORCY */
// НИЖЕ СЧЕТЧИК----------------------------------------
module count_up_7 ( count, cnt_rst, clk_c);
output [6:0] count;input cnt_rst;input clk_c;
wire [6:0] count;wire cnt_rst ;wire clk_c ;wire [5:0] count_cry;
wire [6:0] count_s;wire [6:0] count_qxu;wire GND ;wire VCC ;
  GND GND_Z (.G(GND)); VCC VCC_Z (.P(VCC));
  FDC \count_Z[0] (.Q(count[0]), .D(count_s[0]),
  .C(clk_c), .CLR(cnt_rst));
 // ТРИГГЕРЫ 1-5 опущены
  FDC \count_Z[6] (.Q(count[6]), .D(count_s[6]), .C(clk_c),
  .CLR(cnt_rst));
  MUXCY_L \count_cry_Z[0] (.DI(GND),.CI(VCC),
  .S(count_qxu[0]), .LO(count_cry[0]));
// МУЛЬТИПЛЕКСОРЫ 1-4 опущены
  MUXCY_L \count_cry_Z[5] (.DI(GND),.CI(count_cry[4]),
  .S(count_qxu[5]), .LO(count_cry[5]));
  LUT2_6 \count_qxu_Z[0] (.I0(count[0]), .I1(GND),
  .O(count_qxu[0]));
 // ЭЛЕМЕНТЫ LUT2_6 сумматора 1-5 опущены---------------------
// @2:9
  LUT2_6 \count_qxu_Z[6] (.I0(count[6]),.I1(GND),.O(count_qxu[6]));
// @2:9
  XORCY \count_s_Z[0] (.LI(count_qxu[0]),.CI(VCC),
.O(count_s[0]));
// ЭЛЕМЕНТЫ XORCY сумматора 1-5 опущены
  XORCY \count_s_Z[6] ( .LI(count_qxu[6]),
  .CI(count_cry[5]),.O(count_s[6]));
endmodule /* count_up_7 */
//НИЖЕ D-ТРИГГЕР с установкой-------------------------------------
module FDS ( Q, D, C, S);
output Q;input D;input C;input S;
wire Q ;wire D ;wire C ;wire S ;wire fbs ;
  assign #(1) fbs = S | D ;
  reg r_e_gl; // dff
  always @(posedge C) r_e_gl = #1 fbs ;
    assign Q = r_e_gl;
endmodule /* FDS */
module OBUF ( O, I);
output O;input I;
wire O ;wire I ;wire true ;wire false ;
  assign #(1) O = I; assign true = 1'b1;
  assign false = 1'b0;
endmodule /* OBUF */
module IBUF ( O, I);
output O;input I;
wire O ;wire I ;wire true ;wire false ;
```

```
  assign #(1)  O = I; assign true = 1'b1;
  assign false = 1'b0;
endmodule /* IBUF */
// ОПУЩЕН module LUT4_4F7F ( I0, I1, I2, I3, O);
//НИЖЕ триггер со сбросом----------------------------------------
module FDR ( Q, D, C, R);
output Q;input D;input C;input R;
wire Q ;wire D ;wire C ;wire R ;
wire ri ;wire fbr ;
  assign #(1)  ri = ~ R;
  assign #(1)  fbr = ri  & D ;
  reg r_e_g2; // dff
  always @(posedge C) r_e_g2 = #1 fbr ;
    assign Q = r_e_g2;
endmodule /* FDR */
// ОПУЩЕН module LUT4_FCFB ( I0, I1, I2, I3, O);
// ОПУЩЕН  module LUT2_2 ( I0, I1, O);
// ОПУЩЕН module LUT2_4 ( I0, I1, O);
input I0;input I1;
//НИЖЕ АВТОМАТ СОСТОЯНИЙ - 2 триггера и управление ----------
module cell_svet_state_3_0__h ( svet_state,
  rst_c, clk_c, N_76, N_60, G_52, N_11_1, N_12_1);
output [1:0] svet_state;
input rst_c;input clk_c;input N_76;input N_60;input G_52;
output N_11_1;output N_12_1;
wire [1:0] svet_state;wire rst_c ;wire clk_c ;wire N_76 ;
wire N_60 ;wire G_52 ;wire N_11_1 ;wire N_12_1 ;
wire [0:0] svet_state_ns_0_iv_i;wire N_36_i_0 ;
// @1:62
  LUT4_4F7F N_36_i (.I0(G_52),.I1(svet_state[1]),
  .I2(svet_state[0]),.I3(N_60),    .O(N_36_i_0));
// @1:62
  FDR \svet_state_Z[0]  (.Q(svet_state[0]),
   .D(svet_state_ns_0_iv_i[0]),.C(clk_c),      .R(rst_c));
// @1:62
  FDR \svet_state_Z[1]  (.Q(svet_state[1]),
  .D(N_36_i_0),.C(clk_c),        .R(rst_c));
// @1:62
  LUT4_FCFB \svet_state_ns_0_iv_i_0[0]  (
  .I0(N_60),  .I1(svet_state[0]),
  .I2(N_76),  .I3(svet_state[1]),
  .O(svet_state_ns_0_iv_i[0]));
// @1:62
  LUT2_2 svet_state_tr2_1_0_and2 (.I0(svet_state[0]),
  .I1(svet_state[1]), .O(N_11_1));
// @1:62
  LUT2_4 svet_state_tr3_6_1_0_and2 (
  .I0(svet_state[0]), .I1(svet_state[1]),
  .O(N_12_1));
endmodule /* cell_svet_state_3_0__h */
// ОПУЩЕН module LUT4_0100 ( I0, I1, I2, I3, O);
// НИЖЕ СВЕТОФОР----------------------- СВЯЗИ КОМПОНЕНТ --------
```

```
module simple_svetofor2 ( clk, rst, red,
 green, yellow);
input clk;input rst;
output red;output green;output yellow;
wire clk ;wire rst ;wire red ;wire green ;
wire yellow ;wire [3:3] svet_state_d;
wire [1:0] \svet_state_h.svet_state ;
wire [6:0] count;
wire N_11_1 ;wire N_12_1 ;wire N_57 ;
wire N_76 ;wire N_60 ;wire N_85_1 ;
wire G_54 ;wire G_52 ;wire rst_c ;wire clk_c ;
wire cnt_rst ;wire VCC ;wire GND ;
// @1:62
 GND GND_Z ( .G(GND));
// @1:62
 LUT2_8 G_32 (.I0(\svet_state_h.svet_state [0]),
 .I1(\svet_state_h.svet_state [1]),
 .O(svet_state_d[3]));
// @1:62
 LUT4_0040 G_33 (.I0(count[2]),
 .I1(count[1]),.I2(count[0]),.I3(N_57),.O(N_76));
// @1:62
 LUT4_FFFE G_37 (.I0(count[3]),.I1(count[4]),
 .I2(count[5]),.I3(count[6]),     .O(N_57));
// @1:62
 LUT4_FDFF G_40 (.I0(count[0]),.I1(N_57),
 .I2(count[1]),.I3(count[2]),.O(N_60));
// @1:62
 LUT3_EA G_54_Z (.I0(N_85_1),.I1(G_52),
 .I2(svet_state_d[3]),.O(G_54));
// @1:62
 LUT4_0D5D G_54_1 (.I0(\svet_state_h.svet_state [1]),.I1(N_76),
 .I2(\svet_state_h.svet_state [0]),       .I3(N_60),
 .O(N_85_1));
// @1:62
 VCC VCC_Z (.P(VCC));
// @1:15
 BUFGP clk_ibuf (        .I(clk),
 .O(clk_c));
//НИЖЕ КОНКРЕТИЗАЦИЯ СЧЕТЧИКА----------------------------
// @1:29
 count_up_7 cnt (
 .count({count[6], count[5], count[4], count[3],
    count[2], count[1], count[0]}),
 .cnt_rst(cnt_rst),.clk_c(clk_c));
//НИЖЕ триггер сброса счетчика----------------------------
// @1:62
 FDS cnt_rst_Z (.Q(cnt_rst),.D(G_54),
        .C(clk_c),.S(rst_c));
//НИЖЕ выход на внешние контакты кристалла----------------
// @1:16
 OBUF green_obuf (.O(green),.I(N_12_1));
```

```
// @1:16
  OBUF red_obuf ( O(red),      .I(N_11_1));
// @1:15
  IBUF rst_ibuf (.O(rst_c),      .I(rst));
// @1:62---Управление состоянием автомата
  cell_svet_state_3_0__h svet_state_h (
  .svet_state({\svet_state_h.svet_state [1],
               \svet_state_h.svet_state [0]}),
  .rst_c(rst_c),.clk_c(clk_c),
  .N_76(N_76),.N_60(N_60),
  .G_52(G_52),.N_11_1(N_11_1),
  .N_12_1(N_12_1));
// @1:62
  LUT4_0100 \un1_un1_rst_i_0_and2_1 G_52  (
  .I0(N_57),.I1(count[0]),.I2(count[2]),
  .I3(count[1]),      .O(G_52));
// @1:16
  OBUF yellow_obuf (     .O(yellow),  .I(svet_state_d[3])
);
endmodule /* simple_svetofor2 */
```

Вопросы и упражнения

1. Как реализован счетчик в светофоре — с помощью сумматора или как счетчик с последовательным переносом?

2. При моделировании будущее состояние светофора next_svet_state порой меняется нечетко (хотя конечное svet_state верно). Не является ли это следствием обычного кодирования состояний, может, противогоночное кодирование улучшит картину?

Вопросы к главе 5

1. Какие параметры и критерии обычно надо задать системе синтеза?

2. В каких случаях синтезатор порождает нежелательные триггеры-защелки?

3. Почему рекомендуется иметь регистры на выходах проектируемых блоков?

4. К чему приводит неполный список чувствительности процессов при верификации и как это влияет на результаты синтеза?

Глава 6. Реализация шифроалгоритма RC4 на ПЛИС

Данная глава иллюстрирует подходы к проектированию устройств на HDL. В нее включены результаты курсовой работы студентов МЭИ, которые схемно реализовали шифроалгоритм RC4 базе ПЛИС серии VIRTEX фирмы XILINX.

6.1. Шифроалгоритм RC4

Шифроалгоритм RC4 был предложен одним из отцов современной криптографии профессором Риверстом (Riverst). Он широко используется наряду с другими шифроалгоритмами в Netscape и других Интернет-браузерах. Описание RC4 автор взял из опубликованной в Интернет статьи, посвященной сравнительной оценке эффективности схемной и программной (на PC) реализации систем взлома шифроалгоритмов (www.cs.berkley.edu/~iang/isaac/hardware). Оно совпадает с приведенным в [21] описанием RC4.

Шифроалгоритм RC4 предполагает наличие 256-байтной постоянной памяти ключей K (в случае, если, как обычно, ключ имеет длину меньше 256 байт, его дополняют до 256 повторением), 256-байтной рабочей памяти S с произвольным доступом и двух 8-разрядных регистров i и j .

Работа начинается с инициализации ключом рабочего массива S

```
    j=0
for i = 0 to 255                    // заполнение ячеек S
    S(i) = i                        // их адресами
for i = 0 to 255
    j= (j + S(i) + K(i))mod 256
    swap S(i) and S(j)              // обмен S(i) и S(j)
```

После этого i и j устанавливаются в 0.

Как только инициализация ключом массива S закончена, возможно побайтовое шифрование (дешифрование) поступающих данных по следующему алгоритму.

Сначала извлекается байт рабочего ключа (output)

```
i = (i + 1) mod 256
j = (j + S(i)) mod 256
swap S(i) and S(j)
output S(S(i) + S(j)) mod 256
```

Извлеченный из памяти S на выход (output) байт рабочего ключа складывается по модулю 2 (XOR) с шифруемым (дешифруемым) байтом данных и образует шифрокод (дешифрокод).

Ниже приведена найденная в Интернет спецификация алгоритма на языке PERL.

```
#!/usr/bin/perl -0777
@k=unpack('C*',pack('H*',shift));for(@t=@s=0..255){$y=($k[$_%@k]+$s[$x=$_
]+$y)%256;&S}$x=$y=0;for(unpack('C*',)){$x++;$y=($s[$x%=256]+$y)%256;
&S;print pack(C,$_^=$s[($s[$x]+$s[$y])%256])}sub S{@s[$x,$y]=@s[$y,$x]}
```

Авторы программы пишут, что проверяли ее на тексте «test message», используя 32-разрядный шестнадцатеричный ключ «12abcdef» (закодированное сообщение перенаправлялось в файл «test.rc4» таким способом: % echo test message | rc4 12abcdef > test.rc4).

При дешифровании процесс был обратный и использовался тот же ключ и закодированный файл % rc4 12abcdef < test.rc4.

Эквивалентный С код программы RC4 приводится ниже. Его можно отыскать на многих сайтах, например ftp://ftp.ox.ac.uk /pub/crypto/misc/rc4.tar.gz

Версия С-программы (автор John Allen).

```
#define S ,t=s[i],s[i]=s[j],s[j]=t /* rc4 hexkey <file */
unsigned char k[256],s[256],i,j,t;main(c,v,e)char**v;{++v;while(++i)s[
i]=i;for(c=0;*(*v)++;k[c++]=e)sscanf((*v)++-1,"%2x",&e),while(j+=s[i]
+k[i%c]S,++i);for(j=0;c=~getchar();putchar(~c^s[t+=s[i]]))j+=s[++i]S;}
```

Запуск программы: % rc4 hexkey < input > output

Компиляция: % gcc -o rc4 -O4 rc4.c

Ниже — цитата из Интернет-статьи (www.cs.berkley.edu/~iang/isaac/hardware), посвященной шифроалгоритму RC4.

«SSL, одна из известных систем шифрования, является несколько усложненной версией RC4. Вместо простой переписи ключа в массив K при инициализации он сначала обрабатывается хеш-функцией MD5, и результат MD5 записывается в K. Так как запись и выборка из K осуществляется в строго определенном порядке, его схемная реализация необязательно требует памяти произвольного доступа. Пусть память — самый медленный блок устройства. Подсчитаем число обращений к памяти, необходимых для инициализации ключа. Первый цикл — 256 обращений, каждый шаг второго цикла — 4 обращения. Всего инициализация требует 1281 обращение. Аналогично, обработка каждого шифруемого байта данных требует 5 обращений в память». Авторы вышеприведенной статьи реализовали схему шифроалгоритма RC4 на ПЛИС фирмы Альтера, а память в виде подсоединенного блока внешней памяти с циклом 10 ns. В ходе сопоставлений возможностей реализованного ими аппаратного подхода и программного, на базе использования простаивающих в праздничные дни и ночи сетей организаций, они сделали вывод в пользу программного по цене стоимость — производительность.

На базе приведенных описаний можно составить спецификацию шифроалгоритма RC4 на HDL. Это описание предполагалось использовать при совместном моделировании с синтезабельным RTL-описанием и потом при совместном моделировании с результатами синтеза — схемным решением на базе ПЛИС типа FPGA серии VIRTEX фирмы XILINX.

Упражнения

1. Проверьте соответствие PERL- и С-спецификаций RC4.
2. Является ли тест, предложенный авторами PERL- и С-программ, полным и как его улучшить?

6.2. HDL-спецификация алгоритма RC4

Так как в дальнейшем предполагалось использовать HDL-спецификацию как эталон при верификации схемной реализации RC4, в нее добавлены два сигнала — ask и ready. По первому (ask) начинается инициализация ключом массива памяти. Когда инициализация закончена и устройство готово, оно выдает сигнал ready и может работать в режиме приема (по фронту сигнала strobe) очередного входного байта (code_in) и его шифрования (дешифрования) с результатом в code_out.

6.2.1. Verilog

VERILOG-спецификация

Основное отличие от эталона — остаток массива K заполняется нулями при длине ключа, меньшей 256.

```
module RC4_specification   (ask, ready,strobe, code_in, code_out),
       parameter n=256          parameter debug=1,
       input ask,strobe, input  [7 0] code_in,
     output ready, output [7 0] code_out reg [7 0] code_out
     integer i, j,reg [7 0] tmp,reg[7 0] tmp_out,
     reg ready,reg [7 0] S [0 255], reg [7 0] K [0 255],
     initial begin // инициализация K, S массивов
       ready=0, j=0,
        for (i=n,i<=255,i=i+1) K[i]=0,// массив K заполняется 0
       @(posedge ask) ,
         // ниже чтение ключа  из файла key_file txt в K,
          // ключ в шестнадцатеричном коде
       $readmemh ( key_file txt , K,0,n-1),//ниже контрольная печать ключа
       if(debug)
             for (i=0,i<=n-1,i=i+1) $display( check key file i=%0d,K[i]=%0d ,
               i,K[i]),                          // ниже инициализация S
        for (i = 0, i <= 255, i = i+1)   S[i] = i,
        for (i=0, i <= 255, i = i+1) begin
             j = (j+S[i] + K[i]) % 256, // %  деление по модулю - остаток
             tmp = S[i],
                 if(debug)
                  $display( tmp=%b ,tmp),
              S[i] = S[j], S[j] = tmp,
           if(debug)
                  $display( i=%d,s[i]=%b,j=%d, s[j]=%b ,i S[i],j,S[j]),
      end
      ready=1,j=0,
      end
    /*     //Простой тест- массив S заполнен  числами 0-255
    for (i = 0  i <= 255, i = i+1) begin
      S[i] = i,
        if(debug)
             $display( i=%d, S[i]=%d   ,i,S[i] ),
    end
       #10,   ready =1, j=0,       end
```

```
    */
      // ниже работа
      always @ (posedge strobe) begin
            i = (i+1) % 256,
            j= (j+S[i]) % 256
            tmp = S[i],
             if(debug)
                 $display( i=%d j=%0d  S[i]=%0d S[j]=  %d , i,j,S[i],S[j])
            S[i] = S[j],
          S[j] = tmp,
          tmp_out = S[(S[i] + S[j]) % 256]
             if(debug)
               $display( tmp_out=%0b ,tmp_out)
            code_out = code_in   ^  tmp_out
        end
endmodule //RC4_spec
module tb_spec_ver_check,
parameter  code_in=  8'b00001111 parameter n=3
 reg ask,strobe,wire ready, wire [7 0] code_out
    RC4_specification  #(n) uut (ask, ready,strobe, code_in, code_out),
    initial begin
      #10,ask=1,strobe=0,
      wait (ready==1),$display( READY was ),
      repeat(20)
       begin  #10 strobe =1 #10,strobe=0
         #10, $display ( code_out=%h  t=%0t ,code_out,$time),
       end
      $finish,
    end
endmodule
```

Тест

Ниже приведен пример модуля тестирующей программы (tb_spec_ver_check), файла с ключем (key_file.txt) и листинг первых 5 циклов работы в режиме шифрования. Тестирование проводилось на двух примерах — в одном варианте массив S заполнялся числами 0—255 (это упрощало расчет эталона вручную), в другом в файле первые три байта ключа имели значения 1, 2, 3, а остальные 0.

```
module tb_spec_ver_check,
parameter  code_in=  8 b00001111,parameter n=3,
 reg ask,strobe,wire ready, wire [7 0] code_out,
    RC4_specification  #(n) uut (ask, ready,strobe, code_in, code_out)
    initial begin
      #10,ask=1,strobe=0,
      wait (ready==1),$display( READY was )
      repeat(20)
       begin  #10,strobe =1,#10,strobe=0
         #10, $display ( code_out=%h, t=%0t  code_out,$time),
       end
      $finish,
    end
endmodule
```

Файл ключа key_file.txt

```
1 //K[0]
2 //K[1]
3 //K[2]
```

Фрагмент листинга вывода

```
READY was
i=          1 j=4  S[i]=4 S[j]=  250
tmp_out=10110001
code_out=be, t=40
i=          2 j=229  S[i]=225 S[j]=   41
tmp_out=11111001
code_out=f6, t=70
i=          3 j=152  S[i]=179 S[j]=  131
tmp_out=10001111
code_out=80, t=100
i=          4 j=156  S[i]=4 S[j]=  202
tmp_out=11100110
code_out=e9, t=130
i=          5 j=58  S[i]=158 S[j]=  168
```

6.2.2. VHDL

VHDL-спецификация

Она развернута в тест проверки алгоритма на простом ключе и с печатью промежуточных результатов.

```
library IEEE,
use IEEE std_logic_1164 all,
entity RC4_specification is
  port(ask in std_logic,
       ready out std_logic,
       strob in bit,
         code_in in std_logic_vector(7 downto 0),
       code_out out std_logic_vector(7 downto 0)),
end RC4_specification,
library IEEE,
use std textio all,use IEEE std_logic_textio all,
use IEEE std_logic_unsigned all,use IEEE std_logic_arith all,
architecture behaviour of RC4_specification is
type memory is array(0 to 255 )of std_logic_vector(7 downto 0),
shared variable K,S memory,
signal tmp_out std_logic_vector(7 downto 0),
begin
  --key initialization array K
start_proc process(ask)
variable i,j natural =0,
 variable tmp std_logic_vector(7 downto 0),
variable str line,
begin
```

```
-----------------Подготовка ключа----------------------
if(strob='0')and(ask='1') then
          ready<='0';  report "ready set to 0";
     --Заполняем  S последовательно значениями от 0 до 255
 for i in 0 to 255 loop
    S(i):=CONV_STD_LOGIC_VECTOR(i,8);
end loop;
 --------------------- некоторое значение ключа----------
 K(0):="00000001"; K(1):="00000010";  K(2):="00000011";
 for i in 3 to 255 loop
    K(i):="00000000";
 end loop;
 ----------------------------------------------------------
 write(str,"START-S initialisation");
 for i in 0 to 255 loop
--каждый очередной элемент S обменивается местами с элементом
--номер которого определяется элементом ключа key
--самим элементом и суммой номеров элементов, с которыми
--  происходил обмен на предыдущих итерациях
  j:=(j+CONV_INTEGER(S(i))+CONV_INTEGER(K(i)))mod 256;
  tmp:=S(i);
   write(str," i=  ");write(str,i,right,0); write(str," S(I)=   ");
    write(str,s(i),right,0);
  S(i):=S(j);
  write(str," j=");write(str,j,right,0);write(str," S(j)=  ");
   write(str,s(j),right,0); writeline(output,str);
  S(j):=tmp;
end loop;
--закоментирован простой тест с S(i)=i
--for i in 0 to 255 loop S(i):= conv_std_logic_vector(i,8); end loop;
 ready<='1';
 write(str," READY SET TO !  "); writeline(output,str);
end if;
end process start_proc;
--------------------шифрование----------------------
code_process:process(strob)
--c code_in тоже самое получается
variable i,j:natural:=0;variable tmp:std_logic_vector(7 downto 0);
variable code_tmp:std_logic_vector(7 downto 0);variable str:line;
begin
  write( str,"START CODING "); writeline(output,str);
  if strob'event and strob='1' then
      i:=(i+1)mod 256;
    j:=(j+CONV_INTEGER(S(i)))mod 256;
    tmp:=S(i);
    write(str,"  i=  ");write(str,i,right,0);
    write(str,"  S(i)=  "); write(str,s(i),right,0); writeline(output,str);
   S(i):=S(j);
   write(str,"   j=   "); write(str,j,right,0);
   write(str,"   S(J)=   "); write(str,s(j),right,0); writeline(output,str);
      S(j):=tmp;
```

```
    code_tmp:=S((conv_integer(S(i))+conv_integer(S(j)))mod 256);
       tmp_out <=code_tmp;
        write( str,"  code_tmp=  ");write(str,code_tmp,right,0); writeline(output,str);
    code_out<=code_in xor code_tmp;
  end if;
end process;
end;
 Тест RC4_coder
      --Подключение библиотек
library     IEEE;
use IEEE.std_logic_1164.all;use IEEE.std_logic_arith.all;
use IEEE.std_logic_textio.all;use std.textio.all;
entity test_bench_RC4 is
end;
architecture behaviour of test_bench_RC4 is
signal ask,ready,clk,rst:std_logic;signal out_of_code:std_logic;
signal code_in,code_out:std_logic_vector(7 downto 0);
signal strob:bit;
--Описание тестируемого кодера RC4
begin
  coder:entity RC4_specification
  port map(ask,ready,strob,code_in,code_out);
--Тестовый набор
test:process
 variable i:integer:=0;variable counter:integer;variable str:line;
 type t_data      is array (1 to 10) of std_logic_vector(7 downto 0);
 constant data:t_data:=(( "00000001"),("00000010"),("00000011"),
 ("00000100"),("00000101"), ("00000110"),("00000111"),("00001000"),
 ("00001001"),("00001010"));
--constant data:t_data:=(("00000000"),("00000001"),("00000010"),("00000011"));
begin
--Кодирование массива данных data
   strob<='0';wait for 200 ns;
   ask<='1'; --начало инициализации
   wait for 200 ns;
   ask<='0'; report"wait until ready='1'";write(str,ready); writeline (output,str);
--wait until ready='1';report "READY WAS";
   wa  for 200 ns;
   for counter in 1 to 10 loop
  -- code_in<=data( counter );--переменный код на входе
      code_in<="00001111";--постоянный код на входе
      wait for 200 ns;  strob<='1';  wait for 200 ns;
      write(str, "Input Information- code-in=  ") ; write(str,code_in,right,0);
      write(str, "Output Code Information  =");write(str,code_out,right,0);
      writeline(output,str);
      strob<='0';
   end loop;
   wait;
 end process;
end;
```

Фрагмент листинга вывода

```
  READY SET TO !
# : NOTE   : wait until ready='1'
#  Time: 400 ns,  Iteration. 0,  TOP instance.
# : 1
# : START CODING
#  :   i=  1  S(i)=  00000100   j=   4   S(J)=   11111010
#  :   code_tmp=  10110001
# : Input code-in=  00001111    Output Code   =10111110
# : START CODING # : START CODING
#  :   i=  2  S(i)=  11100001   j=   229   S(J)=   00101001# :   code_tmp=  11111001
# . Input- code-in=  00001111   Output Code   =11110110
# : START CODING # : START CODING
#  :   i=  3  S(i)=  10110011   j=   152   S(J)=   10000011# .   code_tmp=  10001111
# : Input  code-in=  00001111   Output Code   =10000000
# : START CODING # . START CODING
#  :   i=  4  S(i)=  00000100   j=   156   S(J)=   11001010 # .   code_tmp=  11100110
# : Input  code-in=  00001111   Output Code  =11101001
```

Упражнение

1. Проверьте соответствие VHDL- и VERILOG-спецификаций RC4.
2. Проверьте соответствие VHDL- и PERL- или C-спецификаций RC4 (с учетом разницы дополнения ключа до 256 байт).

6.3. ПЛИС семейства Virtex

6.3.1. Возможности

ПЛИС типа FPGA серии VIRTEX фирмы XILINX интересны своими большими логическими возможностями и тем, что позволяют включать в проект внутреннюю память кристалла, необходимую для реализации RC4. Хотя более современные ПЛИС VIRTEX 2 имеют примерно в 2—4 раза лучшие параметры, ниже приводятся данные VIRTEX в форме цитат из [22] и фирменной документации как пример возможностей относительно дешевых устройств (подробную информацию читатель может найти в [22, 23, 29, 30] и на сервере www.xilinx.com).

Параметры кристаллов VIRTEX:

— сложность реализуемых схем — от 50 К до 1 М эквивалентных вентилей;
— тактовая частота — до 200 МГц;
— иерархическая система элементов памяти:
  а) на базе 4-входовых таблиц преобразования (4-LUT — Look-Up Table), конфигурируемых либо как 16-битовое ОЗУ (RAM-Random Access Memory), либо как 16-разрядный сдвиговый регистр;
  б) на базе встроенной блочной памяти, каждый ее блок конфигурируется как синхронное двухпортовое ОЗУ емкостью 4 Кбит.

В состав семейства Virtex входят девять микросхем, отличающихся логической емкостью (табл. 1).

Таблица 1. Основные характеристики семейства Viftex

Кристалл	Число вентилей	Матрица КЛБ	Логические ячейки	Число в/в	Блочная память	Память на LUT [бит]
XCV50	57 906	16 × 24	1 728	180	32 768	24 576
XCV100	108 904	20 × 30	2 700	180	40 960	38 400
XCV150	164 674	24 × 36	3 888	260	49 152	55 296
Промежуточные микросхемы опущены						
XCV800	888 439	56 × 84	21 168	512	114 688	301 056
XCV1000	1 124 022	64 × 96	27 648	512	131 072	393 216

6.3.2. Архитектура семейства Virtex

Матрица КЛБ

Кристаллы состоят из матрицы КЛБ (КЛБ — Конфигурируемый Логический Блок — CLB), которая окружена программируемыми блоками ввода-вывода (БВВ — IOB). Все соединения между основными элементами (КЛБ, БВВ) осуществляются с помощью набора иерархических высокоскоростных программируемых трассировочных ресурсов (рис. 6. 1).

Конфункционирование кристаллов определяется загружаемыми во внутренние ячейки памяти конфигурационными данными. Они управляют настройкой логических элементов и коммутаторами трасс, осуществляющих межсоединения в схеме. Эти коды загружаются в ячейки кристалла после включения питания и могут перезагружаться в процессе работы, если необходимо изменить реализуемые микросхемой функции.

Рис. 6.1. Кристалл VIRTEX

Блок ввода-вывода (БВВ — IOB)

Отличительным свойством БВВ семейства Virtex является поддержка широкого спектра стандартов сигналов ввода-вывода. На рис. 6.2 представлена структурная схема БВВ.

Рис. 6.2. Блок ввода-вывода (БВВ — IOB)

БВВ содержит три запоминающих элемента, функционирующих либо как D-триггеры, либо как триггеры-защелки. Каждый БВВ имеет входной сигнал синхронизации (CLK), распределенный на три триггера, и независимые для каждого триггера сигналы разрешения тактирования (Clock Enable — CE).

Кроме того, на все триггеры заведен сигнал сброса/установки (Set/Reset — SR). Для каждого триггера этот сигнал может быть сконфигурирован независимо как синхронная установка (Set), синхронный сброс (Reset), асинхронная предустановка (Preset) или асинхронный сброс (Clear).

Конфигурируемый логический блок (КЛБ — CLB)

Базовым элементом КЛБ является логическая ячейка (Logic Cell — LC). Она состоит из 4-входового функционального генератора, логики ускоренного переноса и запоминающего элемента. Выход каждого функционального генератора каждой логической ячейки подсоединен к выходу КЛБ и к D-входу триггера. Каждый КЛБ серии Virtex содержит четыре логические ячейки, организованные в виде двух одинаковых секций. На рис. 6.3 представлено детальное изображение одной секции с LUT.

В дополнение к четырем базовым логическим ячейкам КЛБ серии Virtex содержит логику, которая позволяет комбинировать ресурсы функциональных генераторов для реализации функций от пяти или шести переменных.

Функциональные генераторы реализованы в виде 4-входовых таблиц преобразования (Look-Up Table — LUT). Кроме использования в качестве функциональ-

Рис. 6.3. Конфигурируемый логический блок (КЛБ — CLB)

ных генераторов каждый LUT-элемент может быть также сконфигурирован как синхронное ОЗУ размерностью 16 × 1 бит. Более того, из двух LUT-элементов в рамках одной секции можно реализовать синхронное ОЗУ размерностью 16 × 2 бита или 32 × 1 бит либо двухпортовое синхронное ОЗУ размерностью 16 × 1 бит.

Арифметическая логика

Арифметическая логика включает в себя элемент, реализующий функцию исключающего ИЛИ (xor), который позволяет реализовать однобитовый сумматор в одной логической ячейке.

В каждой логической ячейке имеется элемент, реализующий функцию И (AND), который предназначен для построения быстродействующих умножителей

Блочная память (Block Select RAM)

В FPGA Virtex встроена особая блочная память (Block Select RAM) из блоков большой емкости (4 кбит) — см. рис. 6.4. Она создана в дополнение к распределенной памяти небольшой емкости (Select RAM), реализованной на таблицах преобразования (Look Up Table RAM — LUTRAM).

Блоки блочной памяти (Block Select RAM) организованы в виде столбцов. Все устройства Virtex содержат два таких столбца, по одному вдоль каждой вертикальной стороны кристалла. Каждый блок памяти равен по высоте четырем КЛБ, таким образом, микросхема Virtex, имеющая 64 КЛБ по высоте, содержит 16 блоков памяти на колонку и 32 блока памяти в целом. В табл. 3. приводятся емкости блочной памяти для различных кристаллов Virtex.

Каждый блок памяти, как показано на рис. 6.4, это полностью синхронное двухпортовое ОЗУ с независимым управлением для каждого порта. Размерность шины данных для обоих портов может быть сконфигурирована независимо.

В табл. 4 показаны возможные соотношения размерностей шин данных и адреса блока блочной памяти.

Таблица 3. Емкость блочной памяти

Кристалл Virtex	Число блоков	Общий объем блочной памяти [бит]
XCV50	8	32 768
XCV100	10	40 960
XCV150	12	49 152
Промежуточные микросхемы опущены		
XCV600	24	10 304
XCV800	28	114 688
XCV1000	32	131 072

Таблица 4 Соотношение размеров шин адреса и данных блока памяти

Разрядность	Число слов	Шина адреса	Шина данных
1	4 096	ADDR<11:0>	DATA
2	2 048	ADDR<10:0>	DATA<1:0>
4	1 024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

Рис. 6.4. Блочная память (**Block Select RAM**)

6.4. VHDL-вариант реализации автомата RC4

Структура устройства (автомата RC4) включает блок памяти 512 × 8 (в дальнейшем память ключей K и рабочая память S объединяются) и автомат управления, осуществляющий инициализацию массива S и затем обработку входных байтов. Инициализация массива S начинается по сигналу запроса ask и при готовности к работе выдается сигнал ready. При реализации схемы на FPGA типа Virtex фирмы XILINX можно использовать встроенные в микросхему блоки памяти 512 × 8 или 256 × 16, используя их как две памяти 256 × 8.

В микросхеме Virtex-1000 таких блоков (256 × 8) 64 и много логических блоков (LUT), каждый из которых реализует произвольную пятивходовую логическую функцию. Это наводит на мысль, что возможна попытка реализации на одном кристалле более 40 параллельно работающих схем RC4.

Анализ документации по микросхеме Virtex-1000 показывает, что задержка выборки-записи в память (~ 3—5 ns) сравнима с задержкой 8-разрядного счетчика или сумматора, и поэтому возможно, что целесообразно разнести по тактам операции вычисления адреса памяти и обращения в память.

6.4.1. Блок памяти

Ниже дано описание модели блока синхронной памяти 512 × 8, используемого для функциональной верификации автомата RC4. Оно соответствует стилю, необходимому для реализации памяти встроенными в кристалл VIRTEX примитивами синхронной блочной памяти.

```
library ieee;use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned all;
entity ramb4_s8p is
  generic (TW,TR. time  =3 ns ),
port (
      di : in std_logic_vector(7 downto 0);
      adar : in std_logic_vector(8 downto 0);
      we : in std_logic,
         en      : in std_logic,  rst    : in std_logic;
      clk : in std_logic,do   out std_logic_vector(7 downto 0)),
end ;
library synplify;use synplify.attributes.all,
architecture rtl of ramb4_s8p is
type mem_type is array (255 downto 0) of  std_logic_vector (7 downto 0),
signal mem· mem_type;
attribute syn_ramstyle of mem : signal is "block_ram",
--атрибуты-один из способов управления системой синтеза. "block_ram"
--если синтезировать память на блоках блочной памяти кристалла
--"register"- если  на триггерах
signal read_add · std_logic_vector(8 downto 0),
begin
   process (clk)
      begin
      if rising_edge(clk) then
```

```
            if (en =  1 ) then
               if (rst =  1 ) then
                    read_add  <=(others=> 0 ),
               else
                   read_add <= addr
              end if,
              if (we =  1 ) then
                  mem(conv_integer(addr)) <= di,
              end if,
          end if,
       end if,
  end process
  do <= mem(conv_integer(read_add)),
end rtl ,
```

6.4.2. Распределение микроопераций алгоритма по тактам

Возможны различные варианты распределения микроопераций по тактам. Например, в VERILOG-версии автомата каждый такт основного процесса шифрования содержит обращение в память и цикл шифрования состоит из пяти тактов. В VHDL-версии автомата, исходя из предположения, что задержка выборки из памяти плюс задержка сложения в сумматоре при формировании следующего адреса j может стать критической, эти микрооперации разнесены и цикл шифрования содержит большее число тактов, но зато при большей тактовой частоте. Возможен, например, следующий расклад по тактам микроопераций и состояний автомата.

RSI и RSJ — регистры хранения данных считываемых из S(i) и S(j);

Этап подготовки массива S

```
i= 0,            //состояние IDLE-ST
while  I<= 255 begin // каждая ячейка S обрабатывается за 1 такт
   S (i) = I,  // состояние автомата S_WSI_ST, сигнал WS
   i = i +1,
end
i = 0, while i <= 255 begin //каждая ячейка S инициализируется за 6 тактов
   RSI = S(i), //такт 1 // S_RSI_ST  сигнал RS, load_rsi
   RKI = K(i) // такт 2    S_RKI_ST сигнал RK
   j = j +RSI,  j = j + RKI, // такт 3 // S_newj_ST
  RSJ = S(j), // такт 4 // S_RSJ_ST сигнал RS  load_rsi
  S(i) = RSJ, // такт 5 // S_WSI_RSJ_ST  сигнал WS
  S(j) = RSI // такт 6 // S_WSJ_ST
   i = i+1,
end  // конец инициализации памяти - далее в таком же стиле
     //расписывается шифрование
```

6.4.3. VHDL-описание автомата RC4

Ниже дано VHDL-описание автомата RC4.

```
--Иванов И  И  уч  группа А-10-98
--Описание автомата реализующего алгоритм кодирования RC4
--Входными данными является ключ (задается пользователем) и
--собственно шифруемые данные, на выходе получается зашифрованная информация
--Поступление нового ключа и выдача выходных данных не сопровождаются --сигналами синхро-
низации
--Сигналы управления clk - тактовая частота  rst - асинхронный
--сброс, ask - сигнал запроса инициализации памяти,
--end_of_code - сигнал завершения поступления на вход - не используется
--шифруемых данных(конец пакета)
--key_write - строб записи байта ключа в память K
--ready - сигнал готовности инициализации
--Подключение библиотек
library     IEEE use IEEE std_logic_1164 all
--Описание интерфейса RC4_coder
entity RC4_coder is
      port (clk rst,ask in std_logic,end_of_code in std_logic
            ready out std_logic,key_write in std_logic
            code_in in std_logic_vector(7 downto 0)
            code_out out std_logic_vector(7 downto 0)),
end,
library     IEEE,
use IEEE std_logic_1164 all,--use IEEE std_logic_textio all
use IEEE std_logic_arith all,use Ieee std_logic_unsigned all,
--use std textio all
--Описание архитектуры   RC4_coder
architecture behaviour of RC4_coder is
--Описание типа - возможных состояний автомата RC4_coder
  type staterc4 is
    (RST_state,IDLE_state,KEYWRITE_state,INIT_RSi_state,
      INIT_RKi_state,OutofCode_state
     INIT_NEWj_state INIT_RSj_state INIT_WSi_state,
      INIT_WSi_RSj_state
     INIT_WSj_RSi_state,READY_state INITFIN_state,RSi_state
     NEWj_state,RSj_state,WSi_state,WSj_state,TMPout_state),
signal state,next_state staterc4,
 --Текущее и следующее состояние автомата
signal RSi,next_RSi,RSi_ff,RSj,RKi std_logic_vector(7 downto 0),
 --Выбранные из памяти S значения
signal i,j,next_i,next_j std_logic_vector(7 downto 0)
 --Текущие и следующие значения индексов
signal TMP,next_TMP std_logic_vector(7 downto 0),
--Промежуточные индексы
 signal wes ens std_ulogic,
-- Сигналы чтения/записи и разрешения по выходу памятей
signal datain_s,dataout_s std_logic_vector(7 downto 0)
 --Входные и выходные данные из памятей S
signal address std_logic_vector(8 downto 0),
--Сигнал расширенного(9 бит) адреса
```

```
-- (по которому из памяти выбираются данные)
--Описание констант 0-го и 1-го векторов
constant vec_0 std_logic_vector(7 downto 0) = 00000000 ,
constant vec_1 std_logic_vector(7 downto 0) ='11111111 ,
--Описание компоненты - памяти RAM с организацией 512 x 8
-- Для варианта реализации памяти на регистрах **********
--component RAMB4_S8p
-- generic (TW,TR  time  =3 ns ),
-- port (
--   DI   in std_logic_vector (7 downto 0),
--   EN   in std_ulogic,
 --  WE   in std_ulogic,
 ---  RST   in std_ulogic,
--   CLK   in std_ulogic,
--   ADDR   in std_logic_vector (8 downto 0)
--   DO   out std_logic_vector (7 downto 0)),
---end component,
--attribute syn_black_box of RAMB4_S8p   component is true,
--Описание компоненты - памяти RAM с организацией 512 x 8
-- Для варианта реализации памяти на триггерах **********
component RAMB4_S8
 port (
   DO   out std_logic_vector (7 downto 0),
   ADDR   in std_logic_vector (8 downto 0),
   DI   in std_logic_vector (7 downto 0),
   EN   in std_logic,
   CLK   in std_logic,
   WE   in std_logic,
   RST   in std_logic
 )
end component,
attribute black_box  boolean,
attribute xc_props  string,
attribute black_box of RAMB4_S8   component is TRUE,

begin
--ОБРАЩЕНИЕ к памяти хранения ключа и промежуточных рез-тов
--кодирования S
--ВАРИАНТ ДЛЯ ФУНКЦИОНАЛЬНОЙ ВЕРИФИКАЦИИ И СИНТЕЗА
-- ПАМЯТИ НА РЕГИСТРАХ
--S RAMB4_S8p
--port map(datain_s,ens,wes,rst,clk,address,dataout_s),
--ВАРИАНТ ДЛЯ СИНТЕЗА ПАМЯТИ НА  БЛОКАХ ПАМЯТИ КРИСТАЛЛА

u1  RAMB4_S8 port map (dataout_s,address,datain_s,ens,clk,wes,rst),

-- Основной процесс смены состояний автомата(Реализация алгоритма RC4)
process (state,ask,key_write,i,RSi,code_in,dataout_s,TMP,j)
begin
next_state<=state,wes<= 0 ,ens<= 0 ,ready<= 0
next_i<=i,datain_s<=vec_0,RSj<=vec_0,
next_TMP<=TMP,next_RSi<=RSi,
address<= 0 & vec_0,next_j<=j;code_out<= ZZZZZZZZ',
case (state) is
```

```
-- Состояние сброса автомата  RST_state
when RST_state =>  next_i<=vec_0;  -- Обнуление индексов
                        next_j<=vec_0,
                              address<='0  & vec_0;
                             ready<='0',
                       next_state<=IDLE_state,
-- Состояние готовности IDLE_state
when IDLE_state => if key_write='1' then
                        -- Если пришел запрос на запись ключа,
                                next_state<=KEYWRITE_state,-- то  запись
                         elsif ask='1' then
                           -- Если пришел запрос на инициализацию,
                          next_i<=vec_0, -- Обнуление индексов
                          next_j<=vec_0,
                       next_state<=INIT_WSi_state,
                           -- то начать инициализацию,
                  end if,
-- Состояние запись ключа в память K(память S адреса от 00h до FFh)
when KEYWRITE_state =>  address<='0'& i,
                      --Установка адреса очередного байта ключа
                       datain_s<=code_in,
                      --Очередной байт ключа поступает по входу code_in
                        wes<='1',--Запись
                        ens<='1',
                        next_i<=i+1,
                        next_state<=IDLE_state,
-- Состояние - конец входного кода
when OutofCode_state =>if (ask='1 ) then
                                                -- Если пришел запрос,
                                next_state<=RSi_state,
                                             - -то продолжить работу
                                           --(без инициализации памяти),
                                     end if,
   -- Инициализация памяти(подготовка ключа)
   -- Состояние - 1-й этап подготовки ключа
     --(заполнение последовательными числами от 0 до 255)
when INIT_WSi_state =>
              --Запись в память S(память S адреса от 100h до 1FFh)
             -- числа i по адресу i
                  address<='1 &i,              --}
                 datain_s<=i;      --          }  S(i)<=i,
                 wes<='1', --Запись              }
                    ens<='1';                       --}
                 next_i<=i+1,
                    next_state<=INIT_WSi_state,
                           if (i=vec_1) then
                           next_state<=INIT_RSi_state,
                 end if,
-- Состояние - 2-й этап подготовки ключа()
when INIT_RSi_state => -- Чтение из памяти S значения по адресу i
                 address<='1'&i,              --}
                 --Чтение                       }  RSi<=S[i],
                  ens<='1';                 --}
```

```
                    next_state<=INIT_RKi_state;
when INIT_RKi_state => next_RSi<=dataout_s;
                      --Прочитанное значение из памяти S(пред.такт)
                        --RSi_ff<=dataout_s,
                        -- Чтение из памяти K значения по адресу i
                        address<='0'&i,
                        --Чтение                        --}
                       ens<='1',--                      }RKi<=K[i],
                        next_j<=j+ dataout_s,                --}
                         --заменить на RSi_ff
                       next_state<=INIT_NEWj_state,

when INIT_NEWj_state =>RKi<=dataout_s,
                    --Прочитанное значение из памяти K(пред.такт)
                       next_j<=j+ dataout_s,
                        --заменить на Rki
                               next_state<=INIT_RSj_state;

when INIT_RSj_state => -- Чтение из памяти S значения по адресу j
                       address<='1'&j;                --}
                        --Чтение                  .   } RSj<=S[j],
                       ens<= 1';                  --}

                        next_state<=INIT_WSi_RSj_state,

when INIT_WSi_RSj_state => RSj<=dataout_s,
                        --Прочитанное значение из памяти S(пред.такт)
                           -- Запись в память S значения RSj по адресу i
                        address<='1 &i,        --}
                        datain_s<=dataout_s, --RSj,-- }     S[i]<=RSj,
                         wes<='1',--Запись          }
                        ens<='1',              --}
                                 next_state<=INIT_WSj_RSi_state,

when INIT_WSj_RSi_state =>
                       -- Запись в память S значения RSi по адресу j
                        address<='1'&j,        --}
                         datain_s<=RSi;--    }   S[j]<=RSi;
                         wes<='1',--Запись          }
                         ens<='1',             --}
                         next_i<=i+1,
                                   next_state<=INIT_RSi_state,
                        if (i=vec_1) then
                           next_i<="00000001 , --!!!--
                           next_j<=vec_0;
                           next_state<=INITFIN_state,
                        end if,

when INITFIN_state => -- сообщение об окончании подготовки ключа
                       -- synopsys synthesis_off
                     report "You key is prepare(array S is set)",
                        -- synopsys synthesis_on
                              ready<= 1',
                       next_state<=RSi_state,
```

```
-- Кодирование входных данных пользователя
--поступающих по линии code_in
when RSi_state =>
            -- Чтение из памяти S значения по адресу i
                address<= 1 &i,                    --}
               --Чтение                      }  RSi<=S[i],
                ens<= 1 ,                     --}
                      next_state<=NEWj_state,

when NEWj_state =>next_RSi<=dataout_s
           --Прочитанное значение из памяти S(пред такт)
                 RSi_ff<=dataout_s
             next_j<=j+ dataout_s,
                  --заменить на RSi_ff
                 next_state<=RSj_state,

when RSj_state => -- Чтение из памяти S значения по адресу j
                 address<=1 &j,                       --}
                  --Чтение                      }  RSj<=S[j]
                   ens<= 1 ,                     --}
                          next_state<=WSi_state

when WSi_state => RSj<=dataout_s
                          --Прочитанное значение из памяти S(пред такт)
                           next_TMP<=RSi+dataout_s,
                  -- Запись в память S значения RSj по адресу i
                  address<= 1 &i,      --}
                  datain_s<=RSj --          }   S[i]<=RSj,
                  wes<= 1 ,--Запись             }
                  ens<= 1                    --}
                 next_state<=WSj_state,

when WSj_state => -- Запись в память S значения RSi по адресу j
                        address<= 1 & j           --}
                        datain_s<=RSi,--        }    S[j]<=RSi
                        wes<= 1  --Запись            }
                        ens<= 1                  --}
                        next_state<=TMPout_state

when TMPout_state => -- Чтение из памяти S значения по адресу TMP
                           address<= 1 & TMP,              --}
                           --Чтение                } TMPout<=S[TMP],
                           ens<= 1  --                       }
                             --}
                          next_i<=i+1
                                 next_state<=READY_state

when READY_state => --TMPout =dataout_s
                  --Прочитанное значение из памяти S(пред такт)
                   --Выдача текущего закодированного байта
                 code_out<=code_in xor dataout_s,--TMPout,
                 next_state<=RSi_state,
  end case,
end process,
-----------------------------------------------------------------------
```

```
-- ниже смена состояний автомата на новые
process (clk,rst,end_of_code)
begin
    if rst='1' then
      state<=RST_state,
   elsif end_of_code='1' then
      state<=OutofCode_state;
   elsif clk 'event and clk='1' then
       state<=next_state,
       i<=next_i;    j<=next_j;
      TMP<=next_TMP;   RSi<=next_RSi;
  end if;
end process;
end,
```

6.4.4. VHDL-тест автомата RC4

Ниже описание тестирующей программы

```
-- Тест RC4_coder
--Подключение библиотек
library     IEEE;use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all,use IEEE.std_Logic_Unsigned.all;
use IEEE.std_logic_textio.all;use std.textio.all,
entity test_bench_RC4 is
end;
architecture behaviour_test of test_bench_RC4 is
signal ask,ready,clk,rst:std_logic,signal out_of_code:std_logic,
signal code_in,code_out:std_logic_vector(7 downto 0);
signal end_of_code:std_logic,signal key_write:std_logic;

--Описание констант 0-го и 1-го векторов
constant vec_0:std_logic_vector(7 downto 0):="00000000";
constant vec_1:std_logic_vector(7 downto 0):="11111111";
--Описание тестируемого кодера RC4
begin
coder:entity RC4_coder
   port map(clk,rst,ask,end_of_code,ready,key_write,code_in,code_out);
--Процесс- генератор сигнала тактовой частоты clk
generator: process
    begin
      clk<='0'; wait for 10 ns; clk<='1'; wait for 10 ns;
      if now>= 1000 us then wait;
      end if,
end process;
--Тестовый набор векторов
test:process
variable k:integer:=0; -- Счетчик числа байтов
variable str:LINE,          -- Строка для вывода
type t_key is array (1 to 3) of std_logic_vector(7 downto 0);
```

```
type t_data is array (1 to 10) of std_logic_vector(7 downto 0);
constant key:t_key:=(("00000001"),("00000010"),("00000011"));
constant
data:t_data:=(( "00000001"),("00000010"),("00000011"),("00000100"),("00000101"),
("00000110"),("00000111"),("00001000"),("00001001"),("00001010"));
begin
ask<='0';wait for 15 ns,
--Предварительный сброс
rst<='1';
wait for 40ns;    --должен появиться сигнал ready
rst<='0';wait for 40ns;
--Заполнение массива K(ключа)
for k in 1 to 256 loop
  wait until clk='0';
  if k < 4 then
    code_in<=key(k);
      key_write<='1';   wait for 40 ns;
  else
    code_in<=vec_0;
      key_write<='1';wait for 40 ns;
  end if;
end loop;
key_write<='0';
--Кодирование массива данных data
wait for 25 ns;
ask<='1';
--После подачи ask идет заполнение памяти S числами от 0 до 255
--(1-й этап инициализации)с учетом что запись одной ячейки памяти
--происходит за 1 такт, то на это потребуется 256 тактов
--Далее происходит последовательное 'перемешивание' памяти S с
--учетом содержимого памяти K(ключа), на инициализацию одной ячейки
--памяти S происходит за 6 тактов(всего на инициализацию всей памяти
--требуется 256*6 тактов)
--Затем должен появиться сигнал ready
wait for 35840 ns;
for k in t_data'range loop
  wait until clk='0';
  code_in<=data(k);
  --Распечатка значения байта информации
  report "Information byte is";
  write(str,code_in,right,0); writeline(output,str);
  wait for 6*20 ns;--6 тактов на кодирование одного байта данных
  --Распечатка значения байта закодированной информации
  report "Encoding byte is";
  write(str,code_out,right,0); writeline(output,str);
end loop;
end process;
end;
```

Упражнения

1. При инициализации массива ключей в VHDL-автомате используется не один, а два такта на запись одного значения — оптимизируйте описание.

2. Измените тестовую программу для совместного прогона алгоритма спецификации и автомата с целью сравнения результатов в ходе эксперимента.

3. Введите в тестовую программу параметр debug, изменение которого с 1 на 0 позволяет отключить вывод результатов моделирования на печать (этот прием позволяет, в частности, повысить скорость моделирования).

6.4.5. Результаты синтеза с памятью на триггерах

На рис. 6.5 показано общее представление проекта схемы системой SINPLIFY до покрытия схемы технологическим базисом (RTL-view). К сожалению, масштаб не позволяет разглядеть имена сигналов и блоков.

На рис. 6.6 и 6.7 даны увеличенные фрагменты этой схемы. Правая часть схемы (рис. 6.6) представляет блочную память — восьмиразрядную схему XOR, на которую приходит код из памяти и шифруемый код и выходные буфера вывода результата шифрования.

Левая нижняя часть (рис. 6.7) регистры RSI и TMP. На входе TMP виден сумматор и мультиплексор.

При тестировании описания автомата и синтезе использовалось закомментированное (в тексте описания автомата, см. выше) обращение к модели памяти, воспринятой системой синтеза как память, реализуемая на логических блоках — LUT.

Ниже фрагмент протокола работы синтезатора для случая реализации памяти на триггерах, входящих в логические блоки (LUT). Для кодирования состояний автомата (18) использовано 18 триггеров (one hot).

Рис. 6.5. Автомат RC4 до покрытия технологическим базисом (RTL-view)

Рис. 6.6. Фрагмент схемы автомата RC4 — память u1

Рис. 6.7. Фрагмент схемы автомата RC4 — регистры

```
##### START TIMING REPORT #####
        Performance Summary
        *********************
          Requested     Estimated     Requested     Estimated
Clock     Frequency     Frequency     Period        Period        Slack

clk       71.0 MHz      66.0 MHz      14.1          15.1          -1.1
==========================================
-- Предельная частота 66 мегагерц в отличие от заданной=70.
        Interface Information
        **********************
Input Ports:
Port        Reference       User          Arrival     Required
Name        Clock           Constraint    Time        Time        Slack

EN          clk [rising]    0.0           0.0         1.4         1.4
RST         clk [rising]    0 0           0 0         1.4         1.4
addr[0]     clk [rising]    0 0           0.0         3.1         3.1
addr[1]     clk [rising]    0.0           0 0         4.3         4.3
addr[2]     clk [rising]    0.0           0.0         4.3         4 3
addr[3]     clk [rising]    0.0           0.0         4.3         4.3
addr[4]     clk [rising]    0.0           0 0         4 3         4.3
addr[5]     clk [rising]    0.0           0.0         1.4         1.4
addr[6]     clk [rising]    0 0           0.0         5 7         5 7
addr[7]     clk [rising]    0.0           0.0         5.7         5.7
addr[8]     System          0.0           0.0         >2000.0     >2000.0
clk         System          0.0           0.0         >2000.0     NA
-------------------------------------
Resource Usage Report
Mapping to part: xcv1000cg560-5
Cell usage:
BUF          24 uses          FDE              2169 uses
GND          1 use            VCC              1 use
MUXF5        512 uses         MUXF6            172 uses
I/O primitives:
OBUF_F_24    8 uses           IBUF             19 uses
BUFGP        1 use            I/O Register bits:          0
Register bits not including I/Os:   2169 (8%)
Global buffer usage summary  BUFGs + BUFGPs  1 of 4 (25%)
Mapping Summary:
Total  LUTs: 1682 (6%)
Found clock clk with period 14.0845ns
Mapper successful!
```

Выводы

Результаты синтеза не впечатляют — при реализации памяти на триггерах затрачено 1682 LUT (6 % от ресурсов кристалла), а быстродействие ниже 66 мегагерц при 7 тактах на один байт шифрируемого кода.

6.4.6. Результаты синтеза с использованием блочной памяти

Реализация с использованием XILINX-примитива RAMB4_S8.

(В описании архитектуры объекта rc4_coder_p раскомментированы операторы объявления и конкретизации компоненты RAMB4_S8.) При синтезе задана более высокая целевая частота — 120 мегагерц.

Ниже фрагменты протокола работы системы синтеза:

```
VHDL syntax check successful! - синтаксис в порядке
Synthesizing work.rc4_coder.behaviour
@N:"c:\synplicity\synplify\examples\vhdl\xilinx\rc4coder_p.vhd":35:14:35:15|Using onehot
encoding for type staterc4 (rst_state="1000000000000000000")
-- Кодировка состояний автомата позиционная( onehot)-наиболее скоростная
@W:"c:\synplicity\synplify\examples\vhdl\xilinx\rc4coder_p.vhd".90:0:90:8|black_box at-
tribute has been renamed to syn_black_box. Change black_box usage to synthesis
syn_black_box for upward compatibility.- имя атрибута syn_black_box было бы --точнеее
Synthesizing work.ramb4_s8.syn_black_box-память реализована на блоке
Post processing for work.ramb4_s8.syn_black_box
Post processing for work.rc4_coder.behaviour
@W:"c:\synplicity\synplify\examples\vhdl\xilinx\rc4coder_p.vhd":304:0:304:1|Feedback mux
created for signal i[7:0]. Did you forget the set/reset assignment for this signal?
--для 6 сигналов ,включая  сигнал i  надо было дать уточняющие указания
 --синтезатору и он бы не  встраивал мультиплексора  на входах этих
--регистров
Setting fanout limit to 100
List of partitions to map:
   view:work.RC4_coder(behaviour)
@N:"c:\synplicity\synplify\examples\vhdl\xilinx\rc4coder_p vhd":304 0:304:1|Found counter
in view:work.RC4_coder(behaviour) inst i[7:0]
-- I -счетчик
Clock Buffers:
  Inserting Clock buffer for port clk,    TNM=clk
@N:"c:\synplicity\synplify\examples\vhdl\xilinx\rc4coder_p.vhd":19:7:19:14|Changing pad
type from OBUFT to OBUFT_F_24 for pad code_out_obuft[7] to improve timing.
@N|Automatic conversion of slower pads to faster pads can be turned off using attribute
xc_fast_auto
-- реализованы быстрые буфера на входах-выходах кристалла
Made 0 timing clusters
@N|The option to pack flops in the IOB has not been specified
Found clock clk with period 8.3333ns-таким был задан такт
   ##### START TIMING REPORT #####
            Performance Summary
           *********************
          Requested     Estimated     Requested     Estimated
Clock     Frequency     Frequency     Period        Period        Slack

clk       120.0 MHz     101.8 MHz     8.3           9.8           -1.5
========================================
            Interface Information
           ************************
Input Ports:
Port            Reference        User           Arrival     Required
```

```
Name            Clock          Constraint     Time        Time        Slack

ask             clk [rising]   0.0            0.0         1.1         1.1
clk             System         0.0            0.0         >2000.0     NA
code_in[0]      clk [rising]   0.0            0.0         1.6         1.6
code_in[7]      clk [rising]   0.0            0.0         1.6         1.6
end_of_code     clk [rising]   0.0            0.0         0.4         0.4
key_write       clk [rising]   0.0            0.0         1.1         1.1
rst             clk [rising]   0.0            0.0         0.4         0.4
=========================================
Output Ports:
Port            Reference      User           Arrival     Required
Name            Clock          Constraint     Time        Time        Slack

code_out[0]     clk [rising]   0.0            9.8         8.3         -1.5
code_out[7]     clk [rising]   0.0            9.8         8.3         -1 5
ready           clk [rising]       0.0            6.4         8.3         2.0
=========================================
            Detailed Timing Report for  clock : clk
            ****************************************
Requested Period    8.3 ns
Estimated Period    9.8 ns
Worst Slack              -1.5 ns
Start Points for Paths with Slack Worse than -0.1 ns :
                                                  Arrival
Instance            Type      Pin     Net         Time        Slack
-------------------------------------------------------------------
state_fast[0]       FDC       Q       N_799       2.4         -1.5
state_fast[12]      FDC       Q       N_800       2.4         -1.5
state[13]           FDCPE     Q       state[13]   2.4         -1.5
code_out_obuft[7]       OBUFT_F_24     T               In      6.7
code_out_obuft[7]       OBUFT_F_24     O               Out     9.8         3.1
code_out[7]             Net                                                    1
code_out[7:0]           Port           code_out[7]     In      9.8
=========================================
This path has no setup requirement
A Critical Path with worst case slack = -0.6 ns:
The start and the end point of this path are clocked by the clk [rising]
Instance/Net                           Pin             Pin     Arrival     Delta     Fan
Name                    Type           Name            Dir     Time        Delay     Out

                        RC4_coder      View
state[5]                FDC            Q               Out     2.4         2 4
state[5]                Net                                                    4
code_out[7]             Net                                                    1
code_out[7:0]           Port           code_out[7]     In      9.8
 ##### END TIMING REPORT #####
-------------------------------------
Resource Usage Report
Mapping to part: xcv1000cg560-5
Cell usage
FDE             32 uses --триггера
FDC             28 uses-- триггера
```

```
FDCP            1 use      GND            1 use
VCC             1 use
MUXF5           2 uses --мультиплексоры
XORCY           22 uses     MUXCY_L        21 uses
FDCPE           1 use
RAMB4_S8        1 use  -- один блок памяти
I/O primitives'-- буфера входов-выходов
OBUFT_F_24     8 uses
IBUF           12 uses
OBUF_F_24      1 use     BUFGP          1 use
I/O Register bits                  0
Register bits not including I/Os    62 (0%)
RAM/ROM usage summary-один блок памяти
Block Rams . 1 of 32 (3%)
Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)
Mapping Summary.
Total  LUTs  136 (0%)----а не 1600 как было раньше
Found clock clk with period 8 3333ns
Mapper successful!
Process took 3 18 seconds realtime, 3 18 seconds cputime
```

Выводы

Результаты синтеза гораздо лучше — при реализации памяти на блоках блочной памяти кристалла затрачено один блок из 32 и 136 (а не 1682) LUT, быстродействие выше 100 мегагерц (а не 66 мегагерц) при 7 тактах на один байт шифрируемого кода.

В свое время автору удавалось достичь такой же частоты при 3—5 тактах, так что поле для оптимизации VHDL-проекта RC4_coder открыто.

6.5. VERILOG-описание автомата RC4

6.5.1. Описание автомата

Описание автомата RC4 упрощено. Оно не отражает фазу инициирования массива S ключем. Состояния автомата закодированы последовательно.

В тесте описания закомментирован вариант использования пользовательской модели памяти ramb4_s8p, которая применялась при тестировании. Синтез с ее использованием дал реализацию памяти S на триггерах при отсутствии директивы синтеза /* synthesis syn_black_box */.

```
 `timescale 1 ns/100 ps
module  ramb4_s8p( di, en  ,we  ,rst  ,clk ,addr ,do ); /* synthesis syn_black_box */ж
     parameter        data_width = 8; parameter       addr_width= 9;
    // parameter         depth  =  2 **addr_width // операция ** в  VERILOG- 2000
     parameter         depth = 512;//
    // parameter         depth = 1< addr_width,// можно и так
    parameter    TW=1,  parameter TR   =1 ;
    input  [data_width -1 : 0] di;    input     en   ;
    input     we   ,  input  rst    ,   input     clk  ;
```

```
    input   [addr_width -1 : 0 ] addr  ;
    output [data_width -1 : 0]   do;
    reg[data_width -1 : 0] mem[0: depth-1 ]
      /* synthesis sin_ramstyle="register" */;//блок памяти
      reg   [addr_width -1 : 0 ] addr_reg  ;
    integer i;
    always @ (posedge clk ) begin
            if( en==1) begin
                    if (rst )addr_reg=0;
                      else addr_reg=addr;
                           if (we ==1)begin
                           mem[addr_reg] = #(TW)di ; do=di; end
                           else begin do=#(TR) mem[addr_reg];
                           end
            end
     end
   assign  #(TR) do= mem[addr_reg];
  endmodule //RAM
// ниже автомат----------------------
 `define REG_DELAY        1    // условная задержка регистров
`define IDLE_ST     1 //последовательность кодов состояний
`define R_SI_ST     2// автомата
`define R_SJ_ST     3//
`define W_SJ_ST     4//
`define W_SI_ST     5//
`define R_SIJ_ST    6//
`include "c:\synplicity\synplify\lib\xilinx\virtex.v"
module rc4_ver_pol (clk, rst,datain,dataout,done );
  input rst, clk ; input [7:0] datain;//входной байт
  output [7:0] dataout;//результат шифрования-дешифрования
  output done;//сигнал готовности результата
  reg done;reg [7:0] dataout;reg [7:0] i_reg,next_i_reg,j_reg,next_j_reg;
  reg [7:0] tmp_si,next_tmp_si,tmp_sj,next_tmp_sj;//tmp_output;
  reg [5:0] state,next_state;
  reg [7:0] addr; reg we,en,next_en;
  reg[7:0]di;wire [7:0]do;
wire [8:0]addr_w={ 1'b0,addr};

  //НИЖЕ КОНКРЕТИЗАЦИЯ КОМПОНЕНТЫ ramb4_s8p
 /*    ramb4_s8p ss (.we( we),.en(en),.rst(rst),.clk(clk),
                              .addr(addr_w),.di(di),.do(do));
*/
//НИЖЕ КОНКРЕТИЗАЦИЯ КОМПОНЕНТЫ RAMB4_S8
//из библиотеки  синтезатора Sinplyfy - она подключена include
    RAMB4_S8 ss (.WE( we),.EN(en),.RST(rst),.CLK(clk),
                              .ADDR(addr_w),.DI(di),.DO(do));
//FSM
always @(state or do or rst or en or tmp_si or
        i_reg or j_reg or datain or tmp_sj) begin
     next_state=state;
     next_i_reg=i_reg;next_j_reg=j_reg;
     next_tmp_si=tmp_si;
```

```
      next_tmp_sj=tmp_sj;
       next_en =en ;done=0;we=0,dataout=0;
        addr=0;di=0,
if (rst==0)begin
case(state)
  `IDLE_ST:begin
            next_state=`R_SI_ST;
            addr=i_reg;
            we=0;
            next_en=1;
            next_i_reg=1; next_j_reg=0;
            end

 `R_SI_ST: begin
                    next_state=`R_SJ_ST;
                  we= 0 ; // далее read пойдет
                   next_tmp_si=  do;     //tmp=S[i]
               addr=  j_reg + do ;//J=J+S[i]
                next_j_reg=addr;
               end
 `R_SJ_ST: begin
                next_state=`W_SJ_ST;
                 next_tmp_sj =  do;  //tmp=S[ j]
           addr=i_reg,
                 di=  do;
                we= 1 ; // далее запись пойдет
           end
 `W_SJ_ST: begin
           next_state=`W_SI_ST;      // S[ j]=S[i]
           addr= j_reg;    //j_reg;
           di= tmp_si;
        we= 1 ; // далее запись пойдет
          end

 `W_SI_ST: begin
           next_state=`R_SIJ_ST;       // S[ i]=S[j]
           addr= tmp_si +tmp_sj, //S[ j]+S[i]
           we= 0 ; // read
        next_i_reg=  i_reg +1 ;
        end

  `R_SIJ_ST. begin
             next_state=`R_SI_ST;
           we=0 , // далее read пойдет
                dataout=  datain ^ do;// кодирование ^ xor
          // $display( "dataout=%b,time=%t",dataout,$time);
            done= 1;//выход готов,вход заблокирован
            //i_reg= #(`REG_DELAY) i_reg +1 ;
             addr=  i_reg ;
        end

  default: begin  next_state=`IDLE_ST;   //    $display("ERROR STATE \n ");
         end
```

```
   endcase
   end//rst
  end  //fsm
// ниже смена состояния автомата
always @( posedge clk)begin
     if (rst==1)begin state='IDLE_ST;
          //next_state='IDLE_ST;
          i_reg=1;j_reg=0;en=1;
     end
     else begin state=next_state;
          i_reg=next_i_reg; en=next_en;
             j_reg=next_j_reg;
             tmp_si= next_tmp_si;
             tmp_sj=next_tmp_sj;
     end
  end
endmodule
```

6.5.2. Тест

При тестировании простым заполнением массива S адресами использован прием доступа из внешнего модуля к внутренним переменным компоненты.

```
`timescale  1ns/100ps
module tb_rc4_ver_pol;
parameter debug=1;
reg rst, clk ; reg [7:0] datain;//входной байт
  wire [7:0] dataout;//результат шифрования-дешифрования
  wire done;//сигнал готовности результата
   //ниже конкретизация компоненты
rc4_ver_pol   uut (clk, rst,datain,dataout,done );
 integer i;
initial begin clk =0 ;repeat (100 )begin #10;clk=~clk;end end
initial begin rst =0 ; #10;rst=1; #42; rst=0; end
initial datain=8'b00001111;
       initial for (i=0;i<=255;i=i+1)begin
            uut.ss.mem[i]=i;
           // uut.ss.RAMB4_S8[i]=i;
            if( debug==1)
                  $display(" i=%d,uut.ss.mem[i]=%d",i,uut.ss.mem[i]);//mem load
          // $display("mem[i]=%0d",mem[i]);
   end
endmodule//tb_rc4_ver_pol
```

Упражнения

1. Добавьте в VERILOG-описание автомата инициализацию массива S.
2. Постройте иной тест проверки VERILOG-описания автомата.

6.5.3. Результаты синтеза

Ниже фрагмент протокола системы синтеза при использовании блочной памяти кристалла.

```
@I::"c:\synplicity\synplify\examples\verilog\xilinx\rc4_ver_pol_las\rc4_ver_pol.v"
@I:"c:\synplicity\synplify\examples\verilog\xi-
linx\rc4_ver_pol_las\rc4_ver_pol v".'c:\synplicity\synplify\lib\xilinx\virtex.v"
Verilog syntax check successful!
File c:\synplicity\synplify\examples\verilog\xilinx\rc4_ver_pol_las\rc4_ver_pol.v changed
- recompiling
Selecting top level module rc4_ver_pol
Synthesizing module RAMB4_S8
Synthesizing module rc4_ver_pol
Extracted state machine for register state
State machine has 6 reachable states with original encodings of:
   000001
   000010
   000011
   000100
   000101
   000110
@END
Process took 0.38 seconds realtime, 0.38 seconds cputime
Setting fanout limit to 100
List of partitions to map:
   view:work.rc4_ver_pol(verilog)
Encoding state machine work.rc4_ver_pol(verilog)-state_h.state[5 0]
original code -> new code // новая кодировка состояний !!!!!!
   000001 -> 000001
   000010 -> 000010
   000011 -> 000100
   000100 -> 001000
   000101 -> 010000
   000110 -> 100000
@N:"c:\synplicity\synplify\examples\verilog\xi-
linx\rc4_ver_pol_las\rc4_ver_pol.v":138.0.138:5|Found counter in
view:work.rc4_ver_pol(verilog) inst i_reg[7.0]
Clock Buffers:
  Inserting Clock buffer for port clk,    TNM=clk
Net buffering Report for view.work.rc4_ver_pol(verilog):
No nets needed buffering.
Found clock clk with period 8.3333ns
             ##### START TIMING REPORT #####
             Performance Summary
            *********************
          Requested     Estimated     Requested     Estimated
Clock     Frequency     Frequency     Period        Period        Slack

clk       120.0 MHz     85.8 MHz      8.3           11.7          -3.3
    ==========================================

Resource Usage Report,
Mapping to part: xcv1000cg560-4
```

```
Cell usage:
FDE             25 uses        FD              2 uses
GND             1 use          VCC             1 use
XORCY           22 uses    MUXCY_L         21 uses
FDRE            8 uses         FDR             5 uses
RAMB4_S8        1 use
I/O primitives:   OBUF_F_24      9 uses
IBUF           9 uses      BUFGP          1 use
I/O Register bits:                  0
Register bits not including I/Os:   40 (0%)
RAM/ROM usage summary
Block Rams : 1 of 32 (3%)  //один блок из 32
Global buffer usage summary
BUFGs + BUFGPs: 1 of 4 (25%)
Mapping Summary: Total  LUTs: 94 (0%)
Found clock clk with period 8.3333ns
Mapper successful!
```

Выводы

Результаты синтеза упрощенного VERILOG-проекта RC4 удовлетворительны. При реализации памяти на блочной памяти затрачено: один блок памяти (512 × 8) и 94 LUT (<1 % от ресурсов кристалла), а тактовая частота выше 85 мегагерц при 5 тактах на один байт шифрируемого кода. При реализации памяти автомата на триггерах получаются худшие результаты — затрачено 2800 LUT и тактовая частота 38 мегагерц.

Глава 7. Функциональная модель микросхемы двухпортовой синхронной памяти

В данной главе рассматривается разработка и верификация высокопараметризированной функциональной модели микросхемы памяти, иллюстрируются вопросы описания сложных процессов с контролем временных соотношений (VHDL-пакет Vital_Timng, VERILOG-блок specify), рассмотрена тестирующая программа с развитым набором процедур, позволяющих иммитировать различные режимы работы двух блоков VHDL и двух блоков VERILOG-моделей микросхем.

Показано, что сложность тестирующей программы в реальных проектах совместима со сложностью модели моделируемой системы. Трудоемкость такого проекта примерно 1,5 человеко-месяца (при тиражировании общих фрагментов в серии моделей, подобных RAM, она снижается в 3—4 раза.

7.1. Состояние вопроса

Функциональные модели микросхем памяти: RAM, FIFO и т. п., доступные вне фирм-разработчиков микросхем, обычно не являются синтезабельными и в основном предназначаются для использования при верификации моделей систем, включающих эти компоненты.

Различают два типа функциональных моделей памяти:

— VITAL-совместимые, пригодные для верификации схем после этапа трассировки с учетом задержек в проводниках;

— модели уровня шинного интерфейса (будем называть их просто функциональными или моделями уровня шинного интерфейса (bus functional model), в основном предназначенные для функциональной верификации RTL-описаний.

Некоторые производители микросхем памяти включают в число свободно распространяемых через Интернет ресурсов не только технические описания (data sheet), но также и функциональные VERILOG- и VHDL-модели своих микросхем. В табл. 7.1 в качестве примера приведены данные на конец 2000 года о HDL-моделях микросхем памяти произвольного доступа (RAM) на веб-сайтах таких компаний, как Cypress (www.cypress.com), IDT (www.idt.com) и Micron (www.micron.com).

Многие другие компании предоставляют подобный продукт лишь платно. В их числе компания DENALYsoft — разработчик моделей микросхем памяти, производимых ведущими фирмами. Ее библиотека включает более 3000 моделей RAM, FIFO и т. п.

Бесплатные модели привлекательны, но лишь при определенных гарантиях их надежности и точности. Подобные гарантии мы в первую очередь ожидаем от моделей, получаемых от производителей микросхем. Однако в этой части ожидания как «халявных», так и платных пользователей не всегда оправдываются. Разработчики микросхем, боясь утечки секретов, не дают потребителям свои синтезабельные модели и разрабатывают функциональные отдельно, часто независимо от синтезабельных и соответственно не всегда полностью функционально совместимые с реальными схемами.

Хотя для части микросхем (особенно последнего выпуска) модели отсутствуют (см. табл. 7.1), но имеющиеся обычно высоко параметризированы и каждая модель покрывает несколько функционально подобных микросхем разной емкости или быстродействия. В табл. 7.2 приведены параметры моделей RAM.

Таблица 7.1. Функциональные HDL-модели микросхем RAM

Фирма	Общее число типов производимых RAM	Количество VERILOG-моделей	Количество VHDL-моделей
Cypress	200	50	50
IDT	150	30	VHDL — не разрабатывают
Micron	200	60	30

Таблица 7.2. Некоторые параметры функциональных моделей RAM

Параметр модели/Фирма-разработчик	Cypress	IDT(ver)	Micron
Объем памяти	+ (ver)	—	+
Длина слова, длина адреса	+ (ver)	—	+
Значения задержек	+	+	+
Время выборки	+	+	+
Контроль нарушений рабочих условий (включен постоянно)	есть	есть	есть
Возможность варьирования задержек внутренних регистров	—	—	—
Модель выходных задержек	max	max	max
Наличие отладочного режима	—	—	—
Управление выдачей отладочных сообщений на печать	—	—	—
Наличие неопределенного (X) значения на выходе во время переходных процессов	—	—	—

«+» — означает наличие легко варьируемого параметра;
«—» — означает отсутствие такого параметра;
«on» — параметр присутствует постоянно;
(ver) — имеется только в VERILOG-моделях;
max — модель отражает только максимальную задержку (наихудшим считается максимальная задержка памяти.

Параметр «размер памяти» (Memory size) полезен тем, что помогает, например, моделировать неполный объем памяти кристалла и, соответственно, снизить требования к памяти инструментальной ЭВМ.

Параметр «режим отладки» (debug, message on/off) представляется полезным, так как во многих случаях не исключено, что модель памяти неверно работает в особых условиях.

Интересно, что несмотря на усилия по стандартизации HDL-описаний моделей памяти, в том числе разработке новой версии VHDL-стандарта VITAL-2000, включающей пакеты моделирования RAM , общий стиль пока не наблюдается. VITAL-пакеты используются в моделях RAM нечасто (см. табл. 7.3). Возможная причина — вносимое ими снижение скорости моделирования.

Таблица 7.3 Характеристики VHDL-моделей RAM

Характеристика /Производитель VHDL-моделей	Cypress	Micron
Применение Vital	нет	нет
Применение Vital memory package	нет	нет
Отображение слова памяти типом	Std_logic_vector	Std_logic_vector

Таблица 7 4 Характеристики VERILOG-моделей RAM

Характеристика /Производитель моделей	Cypress	Idt	Micron
Применение системных процедур типа $setuphold для контроля временных соотношений	Да	Да	Да
Применение системных процедур для вычисления задержек на путях сигналов	Нет	Да	Да

Как видно из табл 7.3 и 7.4, Verilog-модели относительно Vital более стандартизованы.

7.2. Некоторые свойства моделей RAM

Защита фирменных секретов не способствует надежности моделей

Модели плохо работают в граничных режимах.

Постоянное отставание сроков выпуска моделей от сроков выпуска микросхем

В частности, оно определяется большой трудоемкостью их создания и верификации (четырехпортовые RAM с встроенными функциями JTAG требуют 2—4 человекомесяца).

Привязка модели к одному из компиляторов

Например, Cypress Verilog-модели с трехстабильными выходными буферами, описанными без применения Verilog-примитивов типа buff, bufif, pmos, не проходят на Verilog-XL-симуляторе.

Неэффективное использование памяти инструментальной ЭВМ VHDL-моделями

Verilog-модели расходуют 2 разряда на представление одного бита массива памяти.

VHDL-модели обычно тратят 8 разрядов (байт) на каждый бит моделируемой памяти при отображении массива как переменной и еще больше в случае отображения сигналом (Cadence-NC VHDL и ModelTech-Modelsim), хотя 4 бит кажется вполне достаточным даже при использовании 9-значного алфавита std_logic_1164 (см. табл. 7.5). Например, модель 9 mbit RAM (256 k * 36) требует 10 000—12 000 килобайт памяти с учетом дополнительных затрат, вызванных неэффективной работой супервизора управления памятью операционной системы.

Одним из методов преодоления этого ограничения является моделирование слова памяти не вектором, а целым значением.

Таблица 7.5. Способы отображения слова памяти средствами VHDL

Способ отображения слова n-разрядного памяти	Требуемое число бит	Точность модели	Скорость моделирования
STD_LOGIC_VECTOR	8*n	высшая	высшая
BIT_VECTOR	8*n	низкая	высшая
INTEGER	1*n	низкая	низкая
LONGX_INTEGER	1*n +1	средняя	низкая

(Низкая скорость означает, что присваивание при наличии функции преобразования типа примерно в 20 раз медленнее) variable sdram: mem_std_logic;

```
sdram( I):=IO;

type mem_int is array (0 t0 10) of integer;
type mem_std_logic is array (0 t0 10) of std_logic_vector( 10 to 0);
variable int_sdram:mem_int;
int_sdram(I):=STD_MEMDATA_TO_longx_integer(IO);
```

Для большей точности моделирования можно один из разрядов при отображении слова памяти целым (например знаковый) отводить под признак неопределенности. LONGX_INTEGER — это название подразумевает использование подобного приема.

Если хоть один из разрядов записываемого в память слова равен X, в память записывается целое число < 0. При чтении такое число конвертируется в код «все X» (OTHERS=>'X').

Ниже пример конвертирования

При записи:

```
Std_logic_vector значение "00000111"  конвертируется        в      7
                                          "00001X01"        в     -1(initial U  )
                                          "00XX1101"        в     -1
```

При чтении:

```
Integer          значение          7        в     "00000111"
                                  -1        в     'XXXXXXXX"
```

Если слово памяти больше 31 разряда, то можно использовать структуру из нескольких целых. Пример реализации такого пакета см. ниже.

Нестандартные методы контроля запрещенных режимов и реализации задержек

Примеры реализации контроля времени предустановки/удержания входного сигнала средствами HDL.

VHDL

В примере: CLK-тактовый сигнал, IO-шина данных, Tdh-время предустановки данных на шине, Tdh- время удержания, (CE_n = '0') и (WR = '0') и (MRST_n ='1')--условие записи в память.

VHDL

a) Без использования пакета Vital_timing

Текст взят из модели микросхемы памяти фирмы Cypress.

```
-- проверка удержания
    Process (CLK DELAYED(Tdh))
  -- процесс запускается задержанным сигналом CLK
     BEGIN
        IF (CLK DELAYED(Tdh) =  1 ) AND  (CLK DELAYED(Tdh) EVENT)
           THEN
          ASSERT (IO LAST_EVENT = 0 ns) OR  (IO LAST_EVENT > Tdh)
          REPORT  Hold time violation on Data     SEVERITY Error,
        END IF,
    END Process,
-- проверка времени предустановки
       Process (CLK)
       BEGIN
          ASSERT (IO LAST_EVENT >= Tds)
          REPORT  Setup time violation on Data bus
          SEVERITY Error,
        END IF,
       END Process,
```

Это вариант самой быстродействующей проверки, но зато требующий определенной квалификации от программиста и иногда не свободен от ошибок. Например, если IO — это порт типа INOUT, то эти процессы запускаются не только при записи в память, но и при чтении. Поэтому возможны ложные выдачи предупреждений о нарушении временных соотношений при чтении.

b) Вариант с использованием стандартной процедуры пакета Vital_Timing.

```
TimingChecks  PROCESS ( CLK, IO)
--ниже обьявлены вспомогательные переменные
     VARIABLE Tviol_IO_CLK        X01  =  0  ------set  to  X   when violation
     VARIABLE TD_IO_CLK           VitalTimingDataType
  BEGIN
    IF  (TimingChecksOn)    THEN -- если проверка включена
      VitalSetupHoldCheck (
         TestSignal            =>   IO,-- проверяемый сигнал
         TestSignalName  =>  IO ,--имя  для сообщения  о нарушении
         RefSignal             => CLK -- опорный сигнал
         RefSignalName   =>  CLK ,--имя
         SetupHigh            => Tds,  --время предустановки 1
         SetupLow             => Tds,-- время предустановки 0
         HoldHigh             => Tdh,--время удержания 1 -hold time
         HoldLow              => Tdh,
         CheckEnabled      => (CE_n =  0 ) AND (WR =  0 )AND (MRST_n = 1 ),
                                        --условие записи в память
         RefTransition       =>  / , --проверять при фронте CLK
         HeaderMsg          => HeaderMsgText,
                         --значение этой переменной добавляется к сообщению
         TimingData         => TD_IO_CLK,
         XOn                    => XOn,--включен режим записи X при нарушении
         MsgOn                => MsgOn,--включен режим выдачи сообщений
```

```
        Violation            => Tviol_IO_CLK),
    END IF,
END PROCESS;
```

Этот вариант отличается стандартностью, простотой для пользователя. Однако он громоздок (в смысле объема текста) и существенно замедляет моделирование — эта процедура медленнее примерно на порядок двух предыдущих процессов, если выполняется на системе моделирования типа modelsim 5.5c. Более подробно вопросы методики реализации VITAL-совместимых VHDL- и VERILOG -моделей микросхем памяти и FIFO рассмотрены в [25, 26].

VERILOG----------

На языке VERILOG вариант проверки, использующий системную процедуру $setuphold в блоке specify, выглядит намного компактнее: пусть `Tds-, `Tdh-константы, заданные в `define,reg notifier-переменная-индикатор нарушения временных соотношений.

```
wire  writecond= (CE_n` == 1 b0)  && (WR == 1'b0) && (MRST_n ==1 b1),
$setuphold(posedge CLK &&& writecond, IO, `Tds, `Tdh, notifier),
```

7.3. Двухпортовая синхронная память

На рис. 7.1 представлено условное графическое изображение синхронной двухпортовой памяти.

Входные управляющие сигналы: выборки кристалла (CS), чтения-записи (R/W) и адреса защелкиваются в триггерах по тактовому сигналу clk и используются в следующем такте или через один. Микросхема имеет две модификации, различающиеся предельной частотой синхросигналов (быстродействием) — 83 и 67 мегагерц.

Временные параметры микросхемы (модификация f83):

tCYC2- время цикла (минимальное-12ns);

tCH2- время в течение которого тактовый синал равен 1(4.0 ns -- min);

Рис. 7.1. Блок-схема двухпортовой синхронной памяти

tCL2- время в течение которого тактовый синал равен 0 (4.0 ns -- min);

tR,tL-время фронта и среза тактового сигнала –(3 ns мах)

(в модели не отображается);

-- ЗАДЕРЖКИ выходов----

tCD2-время выборки на шину данных после фронта тактового сигнала (6.0 ns max);

tDC- время удержания на шине данных после фронта тактового сигнала -2 ns;

tCKLZ время отключения шины данных в низкое Z (0 ns -- min);

TCKHZ время отключения шины данных в высокое Z;

tOE время отключения шины данных по сигналу OE (6.0 ns-- max);

tOLZ-(1 ns -- min);

tCCS- минимальное время сдвига синхросигналов правого и левого портов при котором не возникают коллизии одновременного обращения в одну и ту же ячейку.

Времена предустановки и удержания сигналов адреса (tSA, tHA), данных (при записи — tSD, tHD), выборки кристалла (tSC, tHC), чтения-записи (tSW, tHW) см. ниже в тексте модуля контроля временных соотношений.

Временная диаграмма одного из режимов чтения дана на рис. 7.2.

Наряду с нормальным чтением показан режим отключения выхода шины памяти по сигналу разрешения выхода OE.

Временная диаграмма одного из режимов чтения с недоступным периодом при переходе к записи дана на рис. 7.3, 7.4.

При верификации модели эти режимы воспроизводятся соответствующими процедурами теста.

Рис. 7.2. Диаграмма процесса чтения

Read-to-Write-to-Read ($\overline{OE}$ = LOW)[8,11,12,13]

Notes.

Рис. 7.3. Диаграмма процессов чтения и записи

Switching Waveforms (continued)

Read-to-Write-to-Read ($\overline{OE}$ Controlled)[8,11,14]

Рис. 7.4. Диаграмма процессов чтения и записи (сигнал OE)

Принятые в модели допущения

А) Коллизии

Они возникают при одновременном (с точностью до tCCS) обращении в одну и ту же ячейку из двух портов.

1. При коллизии запись/чтение (write/read) модель шлет X на выход памяти, однако при этом пишет верное значение в память.

2. При коллизии запись/запись (write/write) модель печатает предупреждающее сообщение и шлет X в память.

Б) Задержки

В модели реализована так называемая распределенная модель задержек, т. е задержки приписаны к отдельным узлам устройства. В VERILOG-версии модели присутствует в закомментированном виде (см. блок specify) другая реализация – путевая модель задержек, в которой рассматриваются задержки отдельных путей сигналов — от входа до выхода, а отдельные узлы при этом задержек не имеют. Относительно значений параметров:

1. Параметр tCKLZ принят ненулевым (0.1 ns).

2. OE min длительность принята равной tOHZ, и более короткий сигнал не меняет выход шины памяти.

В) X-пессимизм

Документация на микросхему гарантирует максимальное время задержки чтения данных по отношению к тактовому сигналу и время его удержания. В остальное время можно устанавливать шину данных в X при пессимистическом режиме работы модели. То же самое относится к обнаружению X-значений входных сигналов. X-значение на шине адреса вызывает печать предупреждений.

Файлы модели

--VHDL ----------

mem_pac.vhd —пакет отображения слов памяти целыми

dual_port_ramv_timing_data.vhd ---—пакет значений временных параметров

dual_port_ramv_timing_check.vhd--модуль проверки временных ограничений

dual_port_ramv_vh.vhd -- модель микросхемы

--VERILOG -

dual_port_ramv.v модель микросхемы

Файлы теста-

-- - совместно исследуются VHDL и VERILOG модели памяти

dual_port_ramv_timing_data.v -- пакет значений временных параметров

tb_dual_port_ramv.v -- тест

Для задания в тесте быстродействия моделируемой микросхемы необходимо установить параметр DeviceType

например `define DeviceType f67

Для верификации без нарушений временных соотношений тестом надо задать параметр tDelta = 0;

Для верификации с нарушениями временных соотношений-

tDelta = 100; // (100 ps)

7.4. VHDL-модель блока памяти

7.4.1. Интерфейс

```
-- Пакеты        :IEEE :std_logic_1164,  vital_timing
--                пакет с временными параметрами
--                  dual_port_ramv_timing_data.vhd
--               ,пакет с функциями преобразования типов
--                   mem_pac.vhd
--для экономии памяти  инструментальной ЭВМ вы можете  использовать
 --не STD_LOGIC , а  biginteger представление данных  пакета  mem_pac.vhd
--------------------- ----------------------------------------**
LIBRARY ieee;   USE ieee.std_logic_1164.ALL; USE ieee.std_logic_arith.ALL;
                USE ieee.numeric_std ALL; USE ieee.vital_timing.ALL;
                USE work.dual_port_ramv_timing_data.ALL;
                USE work.mem_pac.ALL;USE STD.TEXTIO.all;
                USE ieee.std_logic_textio ALL;
ENTITY  dual_port_ramv_vh is
   GENERIC (    Device_Type   :MaxFreqVal  :=f83;--can be  f83 or 67 mhz
            data_length   : INTEGER    :=36,
            addr_length   : INTEGER    :=18;
            data_num      : INTEGER    := 2 ** 18;
--при моделировании слова  памяти как STD_logic mem word
-- значение этого параметра должно быть =1
            -- параметра int_data_num=1.
            int_data_num  : INTEGER := 1 ;
--при отображении слова памяти целым значение
--этого параметра должно быть >1 а data_num =1.
           -- параметры управления  режимами работы модели
             debug           : BOOLEAN :=FALSE,
                                --ложно исключает отладочную печать
            CollisioCheckOn: BOOLEAN :=TRUE,
                               --ложно исключает контроль коллизий
            OutXon          : BOOLEAN :=TRUE;  --ложно исключает X пессимизм
           InstancePath    : STRING  := " U1 ",
           --дополнительная часть сообщений о нарушениях - имя конкретизации
           partID          : String  :=  "dual_port_ramv.vhdl  ",
           TimingChecksOn : BOOLEAN := TRUE ,
             --ложно исключает проверки временных соотношений
           MsgOn           : BOOLEAN := TRUE ;
          --ложно- исключает печать сообщений о нарушениях соотношений
           XOn             : BOOLEAN := TRUE
      );
--ports order not the same as in the datasheet-----------------------
PORT ( AL   :     IN std_logic_vector(addr_length-1 DOWNTO 0);
       AR  :   IN std_logic_vector(addr_length-1 DOWNTO 0);
       RWL  :     IN    std_logic ,
       RWR  :     IN    std_logic ;
       OEL_n :    IN    std_logic ;
       OER_n :    IN    std_logic ;
       IOL  :   INOUT std_logic_vector(data_length-1 DOWNTO 0):=(others=>'Z');
       IOR  :   INOUT std_logic_vector(data_length-1 DOWNTO 0):=(others=>'Z'),
```

```
      CLKL :     IN    std_logic ;
      CLKR :     IN    std_logic ;
      CE0L_n:  IN    std_logic ;
      CE1L :     IN    std_logic ;
      CE0R_n:  IN    std_logic ,
      CE1R :     IN    std_logic
      );
END dual_port_ramv_vh;
```

7.4.2. Архитектура

```
ARCHITECTURE  vhdl_behavioral OF dual_port_ramv_vh IS
-- ВРЕМЕННЫЕ ПАРАМЕТРЫ-----------------------
 ATTRIBUTE vital_level1 OF vhdl_behavioral : ARCHITECTURE IS False;
    --CONSTANT  data_num:INTEGER:=2**addr_length;
    CONSTANT  tCYC2:TIME:= tCYC2_arr(DEVICE_TYPE);    --- (  12 ns);   --min
    CONSTANT  tCH2:TIME := tCH2_arr(DEVICE_TYPE);    -- ( 4.0 ns ); -- min
    CONSTANT  tCL2:TIME := tCL2_arr(DEVICE_TYPE);    -- ( 4.0 ns ); -- min
    --ВЫХОДНЫЕ ЗАДЕРЖКИ----
    CONSTANT   tCD2:TIME:=  tCD2_arr(DEVICE_TYPE);     -- (  6.0 ns); --max
    CONSTANT   tDC:TIME:=   tDC_arr(DEVICE_TYPE);      -- (2 ns); -- m
    CONSTANT   tCKLZ:TIME:= tCKLZ_arr(DEVICE_TYPE);   -- ( 0 ns);  -- min
    CONSTANT   tCKHZ:TIME:= tCKHZ_arr(DEVICE_TYPE);   -- ( 5.5.0 ns); -- max
    CONSTANT   tOE:TIME:=   tOE_arr(DEVICE_TYPE),     -- ( 6.0 ns); -- max

    CONSTANT   tOLZ:TIME:= tOLZ_arr(DEVICE_TYPE);   -- ( 1 ns);     -- min
    CONSTANT   tOHZ:TIME:=  tOHZ_arr(DEVICE_TYPE);   -- ( 3.0 ns); -- max
     CONSTANT   tCCS.TIME:=  tCCS_arr(DEVICE_TYPE);       -- ( 5.0 ns); -- min
   -- internal reg delay for debug---------------------------
    CONSTANT   tREG_DEL : TIME :=0 ns;
-- НИЖЕ ПОЛЕЗНЫЕ КОНСТАНТЫ--------------------
--- ***********************************************************---
-- constants
CONSTANT        HIZ         :  STD_LOGIC_VECTOR(data_length-1 DOWNTO 0)
 := (OTHERS => 'Z');
CONSTANT        HIX         :  STD_LOGIC_VECTOR(data_length-1 DOWNTO 0)
 := (OTHERS => 'X')
CONSTANT        HIZA        :  STD_LOGIC_VECTOR(addr_length-1 DOWNTO 0)
 := (OTHERS => 'Z');
CONSTANT        HIXA        :  STD_LOGIC_VECTOR(addr_length-1 DOWNTO 0)
  := (OTHERS => 'X');
--CONSTANT        BYTE_HIZ  :  STD_LOGIC_VECTOR(byte_length-1 DOWNTO 0)
  := (OTHERS => 'Z');
--CONSTANT        BYTE_X    :  STD_LOGIC_VECTOR(byte_length-1 DOWNTO 0)
 := (OTHERS => 'X');
CONSTANT        TRUE        :  std_logic := '1';
CONSTANT        FALSE       :  std_logic := '0';
CONSTANT        ALL_ONE     :  STD_LOGIC_VECTOR(data_length-1 DOWNTO 0)
  := (OTHERS => '1');
CONSTANT        ALL_ZERO    :  STD_LOGIC_VECTOR(data_length-1 DOWNTO 0)
  := (OTHERS => '0');
```

```
CONSTANT        ALL_ONEA    :  STD_LOGIC_VECTOR(addr_length-1 DOWNTO 0)
  := (OTHERS =>'1');
CONSTANT        ALL_BITMEMDATA_ONE  :  BIT_VECTOR(data_length DOWNTO 0)
  := (OTHERS => '1');
CONSTANT        BIGINT_HIU    :biginteger:=(-1,-2);
-- ДЛЯ ИНИЦИАЛИЗАЦИИ МАССИВА ПАМЯТИ ПРИ biginteger
--ВНУТРЕННИЕ РЕГИСТРЫ
TYPE mem IS ARRAY (0 TO data_num-1) OF
        STD_LOGIC_VECTOR (data_length-1 DOWNTO 0),
---For big mem better to use big integer representation !!!!!!!!!!
--TYPE bit_mem IS ARRAY (0 TO bit_data_num-1) OF
1.    BIT_VECTOR (data_length DOWNTO 0);
2.    --1 bit additional in the mem word!!
---For big mem better to use biginteger word representation !!!!!!!!!!
TYPE int_mem IS ARRAY (0 TO int_data_num-1) OF
        biginteger; -- bit 32 used for X val representation!!
-----------------------------------------------------------
--SIGNAL sdram :mem;
SIGNAL al_reg,ar_reg :std_logic_vector(addr_length-1 DOWNTO 0);
SIGNAL iol_in_reg,ior_in_reg :std_logic_vector(data_length-1 DOWNTO 0);
SIGNAL iol_out_reg,ior_out_reg :std_logic_vector(data_length-1 DOWNTO 0);
SIGNAL rwl_reg,rwr_reg :std_logic;
SIGNAL cs1l_reg,cs2l_reg,cs1r_reg,cs2r_reg :std_logic:='0';
SIGNAL csl,csr        :std_logic;
SIGNAL datal_reg:std_logic_vector(data_length-1 DOWNTO 0):= (OTHERS=>'X');
SIGNAL datar_reg:std_logic_vector(data_length-1 DOWNTO 0):= (OTHERS=>'X');
SIGNAL rwl_w,rwr_w:std_logic;
--SIGNAL posl_time,posr_time,tdelta_p:TIME;
SIGNAL clk_collision_reg :STD_LOGIC;---,addr_collision,leadingx.std_logic;
SIGNAL iol_int,ior_int:std_logic_vector(data_length-1 DOWNTO 0);
SIGNAL      oeld_n,oerd_n:std_logic;
SIGNAL tx_viol,ty_viol  :std_logic:='0'; -- for change funct when violation
SIGNAL tx_viol_reg,ty_viol_reg  :std_logic:='0'; -- for change funct
SIGNAL      cel_n,cer_n:std_logic;
SIGNAL      corr_l_reg,corr_r_reg:std_logic;    -- for data corruption check
SIGNAL acorr_l_reg,acorr_r_reg :std_logic_vector(addr_length-1 DOWNTO 0);
 begin
-------ФУНКЦИОНАЛЬНАЯ ЧАСТЬ-------------------------------------
--НИЖЕ ЗАЩЕЛКИВАЮТСЯ ВХОДНЫЕ СИГНАЛЫ ЛЕВОГО ПОРТА  l и правого r  - порта-
csl<='1' when CEOL_n='0' AND CE1L='1' else '0';
csr<='1' when CEOR_n='0' AND CE1R='1' else '0';
cel_n <=not csl;
cer_n <=not csr;
--Управление левого порта ----------------------
csl_p: PROCESS(CLKL)--chip select
begin
IF CLKL='1' THEN
   cs1l_reg<=csl after tREG_DEL;
  -- cs2l_reg<=cs1l_reg;
   IF cs1l_reg='1' AND cs2l_reg='0' THEN
          cs2l_reg<='1'  after tCKLZ ; --tDC;
   ELSIF cs1l_reg='0' AND cs2l_reg='1' THEN
```

```
      cs2l_reg<='0'  after tCKHZ ;
    ELSE  cs2l_reg<=cs1l_reg;
   END IF;
 END IF;
END PROCESS;
iol_p: PROCESS(CLKL)--data in
begin
   IF CLKL='1' THEN
      iol_in_reg<=IOL  after tREG_DEL;
   END IF;
END PROCESS;
al_p: PROCESS(CLKL)--addr in
begin
 IF CLKL='1' THEN
    al_reg<=AL  after tCCS;-- tREG_DEL;
  END IF;
END PROCESS;
rwl_w<= '1' when( RWL='0' and csl='1') else '0';
    --RWL=1:means read,0:write,
rwl_p: PROCESS(CLKL)-- w/r
begin
  IF CLKL='1'  THEN
     IF rwl_w='1' THEN --write
       rwl_reg<=rwl_w  after tCKLZ ; --tDC;  ;--??not hZ !!
     ELSE
       rwl_reg<=rwl_w  after tCKHZ;
     END IF;
  --corruption check
     IF CollisioCheckOn  THEN  --after read -write
       IF  rwl_reg='0' AND  cs1l_reg='1'AND rwl_w='1' AND oeld_n='0'  THEN
                --after read next write
         corr_l_reg<='1';acorr_l_reg<= AL;
       ELSE
         corr_l_reg<='0';
       END IF;
       IF corr_l_reg='1' AND (rwl_w='0' OR
            (rwl_w='1' AND acorr_l_reg /=AL))    THEN
        print_addrmsg (text =>InstancePath & partID &
        "Possible Data CORRUPTION when WRITE after Read  to Left mem port",
                     addr =>al_reg);
       END IF,
    END IF;
  END IF;
END PROCESS;
--Управление правого  порта  опущено----------------------
--(для его восстановления замените  l на r  в именах сигналов управления
--левого порта и скопируйте на это место
--НИЖЕ ОПИСАНИЕ ЧТЕНИЯ-ЗАПИСИ**************
 rwx_p:PROCESS(CLKL,CLKR)
  VARIABLE sdram :mem;
 -- VARIABLE bit_sdram :bit_mem:=(OTHERS=>"111");
  VARIABLE int_sdram :int_mem:=(OTHERS=> BIGINT_HIU);
```

```
 VARIABLE data_vx,data_vy:std_logic_vector(data_length-1 DOWNTO 0);
-- VARIABLE bit_data_vx,bit_data_vy:bit_vector(data_length DOWNTO 0);
 VARIABLE int_data_vx,int_data_vy:biginteger;
 VARIABLE clk_collision :STD_LOGIC;
 VARIABLE tdelta_p, posr_time, posl_time:TIME,
 VARIABLE left_collision,right_collision:BOOLEAN;
 VARIABLE  time_l_read, time_r_read,time_l_write,time_r_write.TIME:=0 ns,
 VARIABLE  addr_l_write, addr_r_write,addr_l_read,
           addr_r_read:std_logic_vector(addr_length-1 DOWNTO 0);
 VARIABLE start_check_col:STD_LOGIC:='0';
BEGIN
--ПРОВЕРКА ПАРАМЕТРОВ-  какой способ отображения слова
-- памяти используется- целыми числами или векторами -data_num and int_data_num
 IF now = 0 ns THEN
    assert (( data_num =1 AND int_data_num>= 1)OR
            (data_num >=1  AND int_data_num= 1))
      REPORT "MEMORY GENERIC val IS WRONG "
      SEVERITY ERROR;
 END IF;
 ЛЕВЫЙ ПОРТ---------------------------
 IF (  CLKL'EVENT AND CLKL  = '1') THEN
   IF (  rwl_reg   = '0'  AND  csll_reg='1') THEN
    --ЧТЕНИЕ ЛЕВОГО---------
        time_l_read:=now;
        addr_l_read.=al_reg;
       -- start_check_col:='1';
        IF int_data_num= 1 THEN -- std_logic memory
         data_vx  := sdram(address_trans  (al_reg));
         datal_reg  <=   data_vx;
         IF DEBUG THEN
           print_msg (text =>" READ from LEFT mem port",
                     data =>sdram(address_trans  (al_reg)) ,
                addr =>al_reg);
         END IF;
       ELSE                    -- biginteger memory
          int_data_vx  .= int_sdram(address_trans  (al_reg));
          datal_reg  <=  BIGINTEGER_MEMDATA_TO_STD (int_data_vx,data_length),
       END IF;
     SIF (  rwl_reg   = '1'   AND  csll_reg='1' )THEN
     --ЗАПИСЬ В ЛЕВЫЙ-----
       addr_l_write:=al_reg,
       time_l_write.=now;
       start_check_col:='1',
       IF int_data_num= 1 THEN -- std_logic memory
          sdram( address_trans (al_reg)):=iol_in_reg;
         IF DEBUG THEN
           print_msg (text =>' WRITE ->to LEFT mem port",
                     data =>sdram( address_trans (al_reg)) ,
                addr =>al_reg),
         END IF;
      ELSE
         int_sdram(address_trans
```

```
(al_reg)):=STD_MEMDATA_TO_BIGINTEGER(iol_in_reg);
      END IF;
    END IF;
  END IF;
 --правый порт-----------------------------
 IF (  CLKR'EVENT AND CLKR  = '1') THEN
   IF (  rwr_reg   = '0'  AND cs1r_reg='1')THEN
   ---ЧТЕНИЕ ПРАВОГО---
       time_r_read:=now;
       addr_r_read:=ar_reg;
       IF int_data_num= 1 THEN -- std_logic memory
          data_vy  := sdram(address_trans(ar_reg));
          datar_reg  <=data_vy;
         -- start_check_col:='1';
         IF DEBUG THEN
            print_msg (  text =>" READ from RIGHT mem port",
                       data => sdram(address_trans(ar_reg)),
                       addr =>ar_reg);
        END IF;
      ELSE                    -- biginteger memory
            int_data_vy  := int_sdram(address_trans  (ar_reg));
            datar_reg  <= BIGINTEGER_MEMDATA_TO_STD (int_data_vy,data_length);
      END IF;
   ELSIF (  rwr_reg   = '1'  AND cs1r_reg='1')THEN
    --ЗАПИСЬ В ПРАВЫЙ-----
     addr_r_write:=ar_reg;
     time_r_write:=now;
     start_check_col:='1';
     IF int_data_num= 1 THEN -- std_logic memory
        sdram(address_trans(ar_reg)):=ior_in_reg;
        IF DEBUG THEN
            print_msg (text =>" WRITE ->to RIGHT mem port",
                       data =>sdram(address_trans(ar_reg)),
                       addr =>ar_reg),
       END IF;
    ELSE
          int_sdram( address_trans (ar_reg)):=STD_MEMDATA_TO_BIGINTEGER(ior_in_reg);
    END IF;
  END IF;
 END IF;
--ПРОВЕРКА КОЛЛИЗИЙ-----------------------------------
  if CollisioCheckOn AND  (start_check_col='1' ) AND
     (( CLKR'EVENT AND CLKR  = '1')OR (  CLKL'EVENT AND CLKL  = '1'))THEN
    --ПРИ ОДНОВРЕМЕННОЙЗАПИСИ   WRITE, шлем в память X!!!
    if  (now-time_r_write)<tCCS AND (now-time_l_write)<tCCS
                      AND (addr_r_write =addr_l_write) THEN
        IF XOn AND (iol_in_reg /= ior_in_reg)              THEN
          IF int_data_num= 1 THEN -- std_logic memory
              sdram(address_trans(al_reg)):=HIX;
              print_addrmsg (text =>InstancePath & partID &
            "Collision when WRITE to BOUTH mem ports,'X' val was written",
                     addr =>al_reg);
```

```
        ELSE
           int_sdram( address_trans (ar_reg)):=STD_MEMDATA_TO_BIGINTEGER(HIX);
           print_addrmsg (text =>InstancePath & partID &
              "Collision when WRITE to BOUTH mem ports,'X' val was written",
                   addr =>al_reg);
        END IF;
     END IF;
  END IF;
  -- параллельно ЗАПИСЬ В ПРАВЫЙ -ЧТЕНИЕ ИЗ ЛЕВОГО
 IF ( (now-time_r_write)<tCCS) AND (( now -time_l_read)<tCCS )
             AND (addr_r_write =addr_l_read) THEN
       IF XOn        THEN
          data_vx  := HIX;
          datal_reg  <=   data_vx;
        END IF;
        print_addrmsg (text =>InstancePath & partID &
           "**** Collision when READ from LEFT mem port,OUT  DATA =X",
                   addr =>al_reg);
 --  одновременно запись в левый  -чтение из правого
 ELSIF  (now-time_l_write)<tCCS AND (now -time_r_read)<tCCS
        AND (addr_l_write =addr_r_read) THEN
       IF XOn THEN
          data_vx  := HIX;
          datar_reg  <=   data_vx;
        END IF,
        print_addrmsg (text =>InstancePath & partID &
         "**** Collision when READ from RIGHT mem port ,OUT DATA =X",
                   addr =>ar_reg);
  END IF;
END IF;
-----------------------
END PROCESS;
 ----ВЫХОДНЫЕ ДРАЙВЕРЫ шины ПАМЯТИ
--возможны варианты с заданием X  на выходе памяти
-- в переходном режиме в промежутках между
  --предустановкой  -hold  и удержанием setup
process(datar_reg)
 begin
    if(( datar_reg /= ior_int) AND outXon)  then
       ior_int<=(others=>'X')after tDC, datar_reg after tCD2,
    else
       ior_int<=datar_reg after tCD2;
    end if;
end process;
process(datal_reg)
 begin
    if( (datal_reg /=iol_int)AND outXon ) then
       iol_int<=(others=>'X')after tDC, datal_reg after tCD2;
    else
       iol_int<= datal_reg after tCD2;
    end if;
end process,
```

```
--ЗАДЕРЖКА  OE -----------------------
oeld_n <=  1  after tOHZ when OEL_n = 1  ELSE  0  after tOLZ, 0  after tOE,
oerd_n <=  1  after tOHZ when OER_n = 1  ELSE  0  after tOLZ, 0  after tOE,
-----ВЫХОДНЫЕ ТРЕХСТАБИЛЬНЫЕ БУФЕРА--------------------------------
--ЛЕВЫЙ ПОРТ --------------------
process(oeld_n,cs2l_reg,rwl_reg,iol_int, tx_viol_reg)
begin
   if  (cs2l_reg= 1 ) and( oeld_n= 0 )and( rwl_reg= 0 ) then--read
       iol<=iol_int,
   elsif(( cs2l_reg= 0 ) OR ( oeld_n= 1 )OR ( rwl_reg= 1 ) )  then
       iol<=(others=> Z ) ,
   else
      iol<= (others=> X ),
   end if,
  -- if(tx_viol_reg /= 0 )    then
  --     iol<=(others=> X )  --change dout when timing viol
  -- end if,
end process,
PROCESS (tx_viol,CLKL)
 BEGIN
   IF (tx_viol EVENT   AND tx_viol = X ) AND  XOn THEN
        tx_viol_reg<= 1  ,
   ELSIF  CLKL EVENT AND CLKL= 1  THEN
        tx_viol_reg<= 0  ,
   END IF,
 END PROCESS,
--ПРАВЫЙ ПОРТ - описание опущено-оно подобно левому порту
-- но в именах сигналов буква  r  вместо l
---- ВЫЗОВ( КОНКРЕТИЗАЦИЯ) МОДУЛЯ ВРЕМЕННЫХ ПРОВЕРОК---
TIME_VIOL_CHECK  entity dual_port_ramv_timing_check

GENERIC MAP (
            Device_Type=> Device_Type,
            data_length => data_length
            addr_length=>addr_length,
    -- timing check control parameters
    InstancePath   => InstancePath,
    partID         => partID,
    TimingChecksOn =>TimingChecksOn,
    MsgOn          => MsgOn,
    XOn            => XOn
         )
PORT MAP(
      AL =>AL,  AR=> AR  RWL=>RWL,
       RWR=>  RWR   OEL_n=> OEL_n,
       OER_n=>OER_n,  IOL =>IOL,
       IOR =>IOR,  CLKL=>CLKL,
       CLKR=>CLKR,   CEOL_n=>CEOL_n,
       CE1L=> CE1L, CEOR_n=>CEOR_n,
       CE1R =>CE1R,
 ----ADDITIONAL PORTS FOR change funct when violation------
      tx_viol =>tx_viol  ty_viol =>ty_viol
      ),
END vhdl_behavioral ,  -- OF dual_port_ramv_vh
```

7.4.3. Пакет со значениями временных параметров

```
-- File name : dual_port_ramv_timing_data vhd
PACKAGE dual_port_ramv_timing_data IS
 ---------------------------------------------------
  -- Common Types to pass timing parameters
  ----------------------------------------
    TYPE MaxFreqVal IS (f83,f67);                    -- MHZ
    TYPE DevTimeArr IS ARRAY (MaxFreqVal) OF TIME;
    --CLOCK----
    CONSTANT tCYC2_arr   . DevTimeArr := (12 ns,  15 ns);   --min
    CONSTANT tCH2_arr     : DevTimeArr := ( 4 ns,6.0 ns ), -- min
    CONSTANT tCL2_arr     : DevTimeArr := ( 4 ns,6 0 ns ); -- min
     CONSTANT tR_arr     : DevTimeArr := (3 ns,3 ns),   -- max
     CONSTANT tF_arr     : DevTimeArr := (3 ns,3 ns);   -- max
    --addr SETUP hold TIME---------
    CONSTANT tSA_arr     : DevTimeArr := (2.5 ns,2.5 ns);   -- min
    CONSTANT tHA_arr      . DevTimeArr := (0.5 ns,0.5 ns);   -- min
  --chip enable
    CONSTANT tSC_arr     : DevTimeArr .= (2 5 ns,2.5 ns),   -- min
    CONSTANT tHC_arr      . DevTimeArr := (0.5 ns,0.5 ns);   -- min
   --RW
    CONSTANT tSW_arr     : DevTimeArr .= (2.5 ns,2.5 ns);   -- min
    CONSTANT tHW_arr      : DevTimeArr := (0.5 ns,0.5 ns);   -- min
  -- data delay
    CONSTANT tSD_arr     . DevTimeArr := (2 5 ns,2.5 ns), -- min
    CONSTANT tHD_arr      : DevTimeArr := (0.5 ns,0.5 ns);   -- min
    --OUTPUT DELAY TIMES----

    --oe to data valid
CONSTANT tOE_arr    : DevTimeArr := (6.0 ns, 6.5 ns); -- max
      --oe to low z
    CONSTANT tOLZ_arr   : DevTimeArr := (1 ns, 1 ns);     -- min
      --oe to h z
    CONSTANT tOHZ_arr   : DevTimeArr := (3.0 ns, 3.0 ns); -- max
      -- clk to data valid
    CONSTANT tCD2_arr    : DevTimeArr := (6.0 ns, 6 5 ns), -- max
       --data out hold after clk hi
    CONSTANT tDC_arr     : DevTimeArr := (2 0 ns,  2.0 ns); --min
     -- clk hi to out hi Z
    CONSTANT tCKHZ_arr   : DevTimeArr := (5.5 ns, 6.0 ns); -- max
     -- clk hi to out lo Z
    CONSTANT tCKLZ_arr   : DevTimeArr := (0.1 ns, 0.1 ns); -- min
    --      port to port del
    CONSTANT tCCS_arr     : DevTimeArr := (5.0 ns, 6 0 ns), -- min
```

7.4.4. Модуль контроля временных параметров

```
-- File name : dual_port_ramv_timing_check.vhd
-----------------------------------------------------------
 LIBRARY ieee;   USE ieee.std_logic_1164.ALL;
                 USE ieee.std_logic_arith.ALL;  USE ieee.vital_timing.ALL;
                 USE work.dual_port_ramv_timing_data.ALL;
ENTITY dual_port_ramv_timing_check IS
GENERIC (
-----------------------------------------------------------
--non -VITAL generics
-----------------------------------------------------------
   Device_Type    :MaxFreqVal  :=f83;
          data_length   : INTEGER :=36;
          addr_length   : INTEGER :=18;
    -- timing check control parameters
    InstancePath  : STRING   := " UAT "    ;
    partID        : String   := "dual_port_ramv.vhdl  ";
    TimingChecksOn : BOOLEAN := TRUE ;--:= DefaultTimingChecks;
    MsgOn          : BOOLEAN := TRUE ;--:= DefaultMsgOn;
    XOn            : BOOLEAN := TRUE --:= DefaultXOn;
          );
-------------------PORT DECLARATION-----------------
PORT (
      AL   :  IN std_logic_vector(addr_length-1 DOWNTO 0);
        AR   :  IN std_logic_vector(addr_length-1 DOWNTO 0);
       RWL  :     IN    std_logic ;
       RWR  :     IN    std_logic ;   OEL_n :   IN    std_logic ;
       OER_n :    IN    std_logic ;
       IOL  :  IN std_logic_vector(data_length-1 DOWNTO 0);
       IOR  :   IN std_logic_vector(data_length-1 DOWNTO 0);
       CLKL :     IN    std_logic ;  CLKR :     IN    std_logic ;
       CEOL_n:  IN    std_logic ;  CE1l : IN    std_logic ;
       CEOR_n:  IN    std_logic ;  CE1R : IN    std_logic ;
       tx_viol: OUT   std_logic ;  ty_viol: OUT   std_logic
      );
END ;
-------************************************************
ARCHITECTURE vhdl_behavioral of  dual_port_ramv_timing_check IS
  -- clock times
    CONSTANT  tCYC2:TIME:= tCYC2_arr(DEVICE_TYPE);--- (10 ns,  12 ns); --min
    CONSTANT  tCH2:TIME:= tCH2_arr(DEVICE_TYPE); -- ( 3.5 ns,4.0 ns ); -- min
    CONSTANT  tCL2:TIME:= tCL2_arr(DEVICE_TYPE);-- ( 3.5 ns,4.0 ns ); -- min
    CONSTANT  tR  :TIME:= tR_arr(DEVICE_TYPE);
    CONSTANT  tF  :TIME:= tF_arr(DEVICE_TYPE);
    --SETUP -HOLD TIMES---------
    CONSTANT   tSA:TIME:= tSA_arr(DEVICE_TYPE);    --   -- min
    CONSTANT   tSC:TIME:= tSC_arr(DEVICE_TYPE);    --    -- min
    CONSTANT   tSW:TIME:= tSW_arr(DEVICE_TYPE);    --   -- min
    CONSTANT   tSD:TIME:= tSA_arr(DEVICE_TYPE);    --    -- min
    --HOLD ----------------
    CONSTANT   tHA:TIME:= tHA_arr(DEVICE_TYPE);        -- min
    CONSTANT   tHC:TIME:= tHC_arr(DEVICE_TYPE);         -- min
```

```
    CONSTANT   thW:TIME:= tHW_arr(DEVICE_TYPE);    -- min
    CONSTANT   tHD:TIME:= tHA_arr(DEVICE_TYPE);    -- min
    --OUTPUT DELAY TIMES----
    CONSTANT   tOE:TIME := tOE_arr(DEVICE_TYPE);         --max
    CONSTANT   tOLZ:TIME.= tOLZ_arr(DEVICE_TYPE);        -- min
    CONSTANT   tOHZ:TIME:= tOHZ_arr(DEVICE_TYPE);       -- min
    CONSTANT   tCD2:TIME:= tCD2_arr(DEVICE_TYPE);       -- max
    CONSTANT   tDC:TIME := tDC_arr(DEVICE_TYPE);        -- max
    CONSTANT   tCKHZ:TIME:= tCKHZ_arr(DEVICE_TYPE);   --- min
    CONSTANT   tCKLZ:TIME:= tCKLZ_arr(DEVICE_TYPE);   --- max
    CONSTANT   tCCS:TIME:= tCCS_arr(DEVICE_TYPE);       --  -- min
---END OF CONTANT DELAY SECTION-----------------------------
-------additional signals for use in the CHIP select check and delay proc
signal cel_n,cer_n.std_logic:='0',
-----------------------------------------------------------
--ВЫЗЫВАЕМЫЕ VITAL-ПРОЦЕДУРЫ
-----------------------------------------------------------
BEGIN
  -- ВЫЧИСЛЕНИЕ ПРОМЕЖУТОЧНЫХ СИГНАЛОВ
      cel_n<= '0' when( CEOL_n='0') AND ( CE1l='1') else '1';
      --выбор кристалла- cel_n -левый порт
      cer_n<= '0' when ( CEOR_n='0') AND (      CE1R='1') else '1',
-- Timing Check Section
     TimingChecks: PROCESS ( AL, AR,  CLKL, CLKR,  OEL_n ,  OER_n ,
        cel_n,cer_n,    -- chip select
        RWL,RWR, IOL , IOR
          )
    -- Timing Check Variables
    -- Pulse Width Check Variables
      VARIABLE Pviol_CLKL          : X01 := '0';
      VARIABLE PD_CLKL             : VitalPeriodDataType
                                 := VitalPeriodDataInit,
      VARIABLE Pviol_CLKR          : X01 := '0';
      VARIABLE PD_CLKR             : VitalPeriodDataType
                         := VitalPeriodDataInit;
      VARIABLE Pviol_OEL_n          : X01 := '0';
      VARIABLE PD_OEL_n             : VitalPeriodDataType
                         := VitalPeriodDataInit,
      VARIABLE Pviol_OER_n          : X01 := '0';
      VARIABLE PD_OER_n             : VitalPeriodDataType
                         := VitalPeriodDataInit;
      -- Setup/Hold Check Variables
      VARIABLE Tviol_AL_CLKL       : X01 := '0';     --------L
      VARIABLE TD_AL_CLKL          : VitalTimingDataType;
      VARIABLE TViol_AR_CLKR       : X01 := '0';    -------R
      VARIABLE TD_AR_CLKR          : VitalTimingDataType,
  -------------------------------------------------------
----RWL
      VARIABLE Tviol_RWL_CLKL      . X01 := '0';    --------A
      VARIABLE TD_RWL_CLKL         : VitalTimingDataType;
----RWR
      VARIABLE Tviol_RWR_CLKR      : X01 := '0';   ------B
      VARIABLE TD_RWR_CLKR         : VitalTimingDataType;
```

```
    -----------------------------------------------------------
    -- CHIP ENABLE
      VARIABLE Tviol_cel_n_CLKL        X01  =  0',------------L
      VARIABLE TD_cel_n_CLKL           VitalTimingDataType;
      VARIABLE Tviol_cer_n_CLKR       X01  = '0', ------------R
      VARIABLE TD_cer_n_CLKR         · VitalTimingDataType;
  ---------------------------------------------------------------
  -- IO DATA---
      VARIABLE Tviol_IOL_CLKL      : X01  =  0',--------A
      VARIABLE TD_IOL_CLKL         : VitalTimingDataType,
      VARIABLE Tviol_IOR_CLKR       X01 '= '0'; ------------B
      VARIABLE TD_IOR_CLKR        : VitalTimingDataType,
      -----------------------------------------------------------

      -- Violation variable (used to OR all individual violation variables)
      VARIABLE XViolation , YViolation        : X01  = '0',
BEGIN
--Проверки временных соотношений- только левый порт-
-- правый по аналогии с измененными именами - буква l на r
 IF  (TimingChecksOn)  ,
    THEN
      -- CLKL period and pulse width check(high & low)
      VitalPeriodPulseCheck (
          TestSignal      => CLKL, TestSignalName  => 'CLKL",
          Period    => tCYC2, PulseWidthHigh  => tCH2,
          PulseWidthLow   => tCL2,
           CheckEnabled    => (cel_n = '0 ),--TRUE,
          HeaderMsg       => InstancePath & partID,
          PeriodData      => PD_CLKL,
          XOn             => XOn,
          MsgOn           => MsgOn,
          Violation       => Pviol_CLKL),
 -- OE  pulse width check(high )
      VitalPeriodPulseCheck (
          TestSignal      => OEL_n,
          TestSignalName  => 'OEL_n',
          Period    => tOHZ+tOLZ,
          PulseWidthHigh  => tOHZ,
          PulseWidthLow   => tOLZ,
          CheckEnabled    => TRUE,
          HeaderMsg       => InstancePath & partID,
          PeriodData      => PD_OEL_n,
          XOn             => XOn,
          MsgOn           => MsgOn,
          Violation       => Pviol_OEL_n);
-----------------------------------
  -- ADDRESS CHECK
   -- AL/CLKL setup/hold time checks
      VitalSetupHoldCheck (
          TestSignal      => AL,
          TestSignalName  => "AL',
          RefSignal       => CLKL,
          RefSignalName   => 'CLKL",
```

```
        SetupHigh       => tSA,
        SetupLow        => tSA,
        HoldHigh        => thA,
        HoldLow         => thA,
        CheckEnabled    => (cel_n = '0'),
        RefTransition   => '/',
        HeaderMsg       => InstancePath & partID,
        TimingData      => TD_AL_CLKL,
        XOn             => XOn,
        MsgOn           => MsgOn,
        Violation       => Tviol_AL_CLKL),
    -- RWL/CLKL setup/hold time check
    VitalSetupHoldCheck (
        TestSignal      => RWL,
        TestSignalName  => "RWL",
        RefSignal       => CLKL,
        RefSignalName   => "CLKL",
        SetupHigh       => tSW,
        SetupLow        => tSW,
        HoldHigh        => thW,
        HoldLow         => thW,
        CheckEnabled    => (cel_n = '0'),--True,
        RefTransition   => '/',
        HeaderMsg       => InstancePath & partID,
        TimingData      => TD_RWL_CLKL,
        XOn             => XOn,
        MsgOn           => MsgOn,
        Violation       => Tviol_RWL_CLKL),
--------------CHIP ENABLE CHECK   ----------------------
    -- cel_n/CLKL setup/hold time check
    VitalSetupHoldCheck (
        TestSignal      => cel_n,
        TestSignalName  => "cel_n= not(not CEOL_n and CE11)  ,
        RefSignal       => CLKL,
        RefSignalName   => "CLKL",
        SetupHigh       => tSC,
        SetupLow        => tSC,
        HoldHigh        => thC,
        HoldLow         => thc,
        CheckEnabled    => True,
        RefTransition   => '/',
        HeaderMsg       => InstancePath & partID,
        TimingData      => TD_cel_n_CLKL,
        XOn             => XOn,
        MsgOn           => MsgOn,
        Violation       => Tviol_cel_n_CLKL);
          --   IOL/CLKL setup/hold time check
    VitalSetupHoldCheck (
        TestSignal      =>   IOL,
        TestSignalName  => "IOL",
        RefSignal       => CLKL,
        RefSignalName   => "CLKL",
        SetupHigh       => tSD,
```

```
        SetupLow        => tSD,
        HoldHigh        => thD,
        HoldLow         => thd,
        CheckEnabled    => (cel_n = '0') AND (RWL = '0'),--WRITE
                   --True,
        RefTransition   => '/',
        HeaderMsg       => InstancePath & partID,
        TimingData      => TD_IOL_CLKL,
        XOn             => XOn,
        MsgOn           => MsgOn,
        Violation       => Tviol_IOL_CLKL);
  XViolation := Pviol_CLKL          OR   Pviol_OEL_n        OR
                Tviol_RWL_CLKL      OR  Tviol_cel_n_CLKL    OR
                Tviol_IOL_CLKL      OR   Tviol_AL_CLKL      ;
  YViolation :=
                Pviol_CLKR          OR
                Pviol_OER_n        OR
                Tviol_RWR_CLKR      OR
                Tviol_cer_n_CLKR   OR
                Tviol_IOR_CLKR      OR
              Tviol_AR_CLKR

  tx_viol<= XViolation,
  ty_viol<= YViolation,
  END IF;
 END PROCESS TimingChecks,
END ;
```

7.4.5. Пакет функций преобразования типов данных

```
LIBRARY ieee,   USE ieee.std_logic_1164.ALL;USE ieee.std_logic_arith.ALL;
                USE STD.TEXTIO.all; USE ieee.std_logic_textio.ALL;
PACKAGE mem_pac is
CONSTANT MEM_WARNINGS_ON  BOOLEAN:=TRUE,
-- when false, no warning msg about
   --  u or x bit in the addr val
TYPE biginteger is RECORD
    lowbit:INTEGER; higbit:INTEGER;
END RECORD;
procedure  print_msg (text:IN STRING; data:IN std_logic_vector;
                      addr: IN STD_LOGIC_VECTOR);
procedure  print_addrmsg (text:IN STRING,
                      addr: IN STD_LOGIC_VECTOR);
function address_trans (ARG:IN std_logic_vector) return NATURAL ;
function std_memdata_to_bit ( s: IN std_logic_vector
                              )return bit_VECTOR;
                         --RESULT size need to be 1 bit more that size s !!!
function bit_memdata_to_std (   s: IN bit_vector
                             )return std_logic_vector,
                         --result size need to be 1 bit less that s !!!
function std_memdata_to_biginteger ( ARG: IN std_logic_vector
                              )return biginteger;
```

```
                        --RESULT sign=1 means X value
function std_memdata_to_integer (ARG: IN std_logic_vector
                              )return integer;
                        --RESULT sign=1 means X value
function biginteger_memdata_to_std ( data: IN biginteger;
                              Constant size : IN NATURAL
                             )return std_logic_vector;
                        --result when  <0 need to be all        X !!!
function integer_memdata_to_std (    --result when  <0 need  be all  bit=  X !!!
                             data: IN integer;
                              Constant size : IN NATURAL
                             ) return std_logic_vector ;
end MEM_PAC;
-----------------------------------------------------------------------
PACKAGE BODY  mem_pac is
PROCEDURE print_msg (text:IN STRING; data:IN std_logic_vector;
                     addr: IN STD_LOGIC_VECTOR)is
 VARIABLE msg:line;
 BEGIN
   Write (msg,String'("**CHIP MSG **"));
    Write (msg,String'("Time:"));Write (msg, Now);
    Write (msg, text );Write (msg,String'("  data ="));
    hwrite (msg,data); Write (msg,String'("  addr ="));
    hwrite (msg, "00"& addr );--for HWRITE need to
      -- have vector with an odd ( multiple of 4)length
    Writeline(output,msg);
 END;
PROCEDURE print_addrmsg (text:IN STRING;
                     addr: IN STD_LOGIC_VECTOR)is
 VARIABLE msg:line;
 BEGIN
   Write (msg,String'("***CHIP MSG **"));
    Write (msg,String'("Time:"));Write (msg, Now);
    Write (msg, text );   Write (msg,String'("  addr ="));
    hwrite (msg, "00"& addr );--for HWRITE need to
      -- have vector with an odd ( multiple of 4)length
    Writeline(output,msg);
 END;
-----------------------------------------------------------------------
 function address_trans (ARG:IN std_logic_vector
                            ) return NATURAL is
    constant ARG_LEFT: INTEGER := ARG'LENGTH-1;
    alias XXARG:  std_logic_vector(ARG_LEFT DOWNTO 0) is ARG;
    variable XARG: std_logic_vector (ARG_LEFT DOWNTO 0);
    variable RESULT: NATURAL := 0;
    Variable uonce : BOOLEAN := TRUE;
    Variable xonce : BOOLEAN := TRUE;
   variable TMP:STD_ULOGIC:='0';
  begin
    for I in XARG'RANGE loop
      RESULT := RESULT+RESULT;
      TMP:=XXARG(I);
      if TMP = '1'  then
```

```
        RESULT := RESULT + 1;
      elsif (TMP = 'U'and MEM_WARNINGS_ON  and uonce)THEN
                uonce := FALSE;
                assert FALSE
          report "Address vector contains a U - it is being mapped to:0 "
                        severity WARNING;
      elsif (TMP = 'X'and MEM_WARNINGS_ON and xonce) then
                xonce := FALSE,
                assert false
          report "Address vector contains an X - it is being mapped to:0 "
                        severity WARNING;
      end if;
    end loop,
    return RESULT;
  end  address_trans;
-----------------------------
function std_memdata_to_integer
( ARG. IN std_logic_vector ) return integer
  --RESULT <0 means X value of data word
                                  is
    constant ARG_LEFT: INTEGER  = ARG'LENGTH-1,
    alias XXARG:  std_logic_vector(ARG_LEFT DOWNTO 0) is ARG;
    variable XARG: std_logic_vector (ARG_LEFT DOWNTO 0);
    variable RESULT· integer := 0;
    Variable uonce : BOOLEAN := TRUE;
    Variable xonce : BOOLEAN := TRUE;
   variable TMP:STD_ULOGIC ='0',
  begin
    for I in XARG'RANGE loop
      RESULT := RESULT+RESULT;
      TMP =XXARG(I);
      if TMP = '1'  then
        RESULT := RESULT + 1,
      elsif (TMP = 'U'and MEM_WARNINGS_ON  and uonce)THEN
                uonce := FALSE;
                assert FALSE
                  report "Data vector contains a U   " &
                   "- it is being mapped to all:X "
                        severity WARNING;
                 RESULT :=-RESULT;
                 EXIT;
      elsif (TMP = 'X'and MEM_WARNINGS_ON and xonce) then
                xonce := FALSE;
                assert false
                   report "Data vector contains an X -  mapped to all:X "
                        severity WARNING;
                 RESULT .=-RESULT;
                 EXIT;
      end if;
    end loop,
    return RESULT,
  end ;  -- std_memdata_to_integer;
-----------------------------------------------------------------
```

```
function std_memdata_to_biginteger (
                                    ARG: IN std_logic_vector
                                    ) return biginteger
                                   --RESULT <0 means X value of data word
                                    is
    constant ARG_LEFT· INTEGER := ARG'LENGTH-1,
    variable REC_RESULT: biginteger ; --·= 0,--?
  begin
      REC_RESULT.lowbit.=std_memdata_to_integer( ARG( 31 downto 0)),
      REC_RESULT.higbit:=std_memdata_to_integer(ARG( ARG left- 32 downto 32));
       return REC_RESULT,
  end ;  -- std_memdata_to_biginteger,
 function biginteger_memdata_to_std (
   --result when  <0 need  be all  bit=  X '''
                              data: IN biginteger,
                               Constant size : IN NATURAL
                              ) return std_logic_vector is
        variable vect : std_logic_vector(size - 1 downto 0),
        variable temp : biginteger := data,
     begin
       if temp lowbit <0 OR temp.higbit <0 then
          vect:=(OTHERS=>'X'),
       else
         vect (30 downto 0)·=integer_memdata_to_std( temp lowbit,31),
         vect (size -1 downto 31):=integer_memdata_to_std( temp higbit,size-31),
       end if, ·
        return vect,
     end,
function integer_memdata_to_std
(  data: IN integer; Constant size : IN NATURAL
                              ) return std_logic_vector is
        variable vect . std_logic_vector(size - 1 downto 0),
        variable temp . integer  = data,
     begin
       if temp <0 then
          vect:=(OTHERS=>'X'),
       else
         for i in 0 to size - 1 loop
           if ( (temp mod 2) = 0 ) then
              vect(i) .= '0',
           else
              vect(i) := '1',
           end if;
           temp := temp / 2;
         end loop;
        end if,
        return vect;
        assert ( temp = 0 )
           report "size of returned vector to small to represent  number"
           severity ERROR,
     end,
 ------------------------------------------------------------
  FUNCTION std_memdata_to_bit (  s. IN std_logic_vector
```

```
                              )RETURN bit_VECTOR
                        --it size need to be 1 bit more that size of s !!!
                                 is
alias sv:  std_logic_vector(s'LENGTH-1 DOWNTO 0) is s;
VARIABLE result:  bit_vector(s'LENGTH   DOWNTO 0) ;
BEGIN
  result(s'LENGTH)   :='0',--means no X in  the any of  std  bits
  FOR i IN sv'RANGE LOOP --
   CASE sv(i) IS
     WHEN '0'|'L'=> result(i):='0';
     WHEN '1'|'H'=> result(i):='1';
     WHEN 'X'|'W'|'U'=> result(s'LENGTH ) ='1',
                        result(i):='0',
                        EXIT;
     WHEN OTHERS => result(i):='0';
   END CASE;
  END LOOP,
  RETURN result;
 END;
----------------------
 FUNCTION  bit_memdata_to_std (
                              s  IN bit_vector
                              )RETURN std_logic_vector
                             --result size need to be 1 bit less that s !!!
                                is
alias sv:  bit_vector(s'LENGTH-1 DOWNTO 0) is s,
 VARIABLE result: std_logic_vector(s LENGTH-2 DOWNTO 0) ,
BEGIN
 IF sv(s'LENGTH-1)='1' THEN
  result:=(OTHERS=>'X'),
 ELSE
  FOR i IN result'RANGE LOOP
    CASE sv(i) IS
      WHEN  '1'   => result(i):='1',
      WHEN OTHERS => result(i):='0';
    END CASE;
  END LOOP;
 END IF,
 RETURN result,
END;
END ; -- OF mem_pac
```

7.5. VERILOG-модель блока памяти

VERILOG-модель алогична VHDL-модели, но в ней есть описание другого способа отображения выходных задержек — в тексте блока specify закомментирован вариант реализации выходных задержек средствами встроенных системных операторов.

7.5.1. Интерфейс микросхемы

```
     `timescale 1ps / 1ps
module  dual_port_ramv (  AL ,  AR ,RWL,  RWR,   OEL_n, OER_n ,
       IOL ,   IOR , CLKL,  CLKR ,  CEOL_n,
       CE1L ,  CEOR_n,  CE1R       );
    parameter           Device_Type   = 83; //может быть f83 or 67 mhz
    parameter           data_length   = 36;
    parameter           addr_length   = 18;
    parameter           data_num      = 8192* 128; //32;
      // timing check,mem funct and debug  control parameters
    parameter           debug         = 0 ;//1;
                              //0 отключает отладочную печать
    parameter          CollisioCheckOn= 1;   //0 for eliminate this check
    parameter          OutXon         = 1,
   //0 если не ставить выход в X  после периода thold - удержания и до tsetup
    parameter           XOn           = 1;   //при коллизиях выдать x;
    parameter           tREG_DEL      = 100;
//ports order not the same as in the datasheet//////////////////////
      input   [addr_length-1 :0 ]   AL   ;
      input  [addr_length-1 :0 ]   AR  ;
      input                        RWL  ;
      input                        RWR  ;
      input                       OEL_n ;
      input                       OER_n ;
      inout [data_length-1  :0]    IOL  ;
      inout [data_length-1  :0]    IOR  ;
      input                        CLKL ;
      input                        CLKR ;
      input                       CEOL_n,
      input                        CE1L ;
      input                       CEOR_n;
      input                        CE1R ;
//////////////////////////////////////////////////////////////////
```

7.5.2.Тело модуля

```
// ПОЛЕЗНЫЕ КОНСТАНТЫ--------------------------------
reg [data_length-1  :0  ]   HIZ ;initial HIZ  = {data_length{1'bz}};
reg [data_length-1  :0  ]   HIX ;initial HIX  = {data_length{1'bx}};
reg [addr_length-1  :0  ]   HIZA;initial HIZA = {addr_length{1'bz}};
reg [addr_length-1  :0  ]   HIXA;initial HIXA = {addr_length{1'bx}};
reg [data_length-1 : 0]sdram [0 : data_num-1] ;
reg [addr_length-1 : 0]  al_reg,ar_reg ;
reg [data_length-1 : 0] iol_in_reg,ior_in_reg ;
//reg [data_length-1 : 0] iol_out_reg,ior_out_reg ;
reg rwl_reg,rwr_reg ;
reg cs1l_reg,cs2l_reg,cs1r_reg,cs2r_reg ;
initial begin
  cs1l_reg=0;cs2l_reg=0;cs1r_reg=0;cs2r_reg=0;
 end
reg [data_length-1 : 0]datal_reg,datar_reg ;      //= [OTHERS=>'Z'];
```

```
initial begin
datal_reg=HIX;datar_reg=HIX,
end
wire rwl_w,rwr_w,
reg [data_length-1 · 0]iol_int,ior_int,
reg   oeld_n, oerd_n,
wire  cel_n,cer_n,wire csl,csr ,
wire outl_dis,
assign  outl_dis= !((cs2l_reg==1'b1) &&( oeld_n==1'b0)&&( rwl_reg==1'b0) );
wire  outr_dis=!((cs2r_reg==1'b1) &&( oerd_n==1'b0)&& (rwr_reg==1'b0 ));
assign csl= ( CEOL_n==1'b0 && CE1L== 1'b1)? 1'b1 : 1'b0;
assign csr= (CEOR_n==1 b0  && CE1R== 1'b1)? 1'b1   1'b0,
assign cel_n =! csl,
assign cer_n =! csr,
wire iol_in_check=csl && !RWL,
wire ior_in_check=csr && !RWR,
reg   corr_l_reg,corr_r_reg,  //-- for data corruption check
reg [addr_length-1   0]  acorr_l_reg,acorr_r_reg ;
```

7.5.3. Задание и контроль временных параметров

```
//условные выражения в блоке specify  не проходят в MODELSIM, и этот блок следует
//заменить на эквивалентный при использовании такой системы моделирования.
  specify
      specparam
      // ЗНАЧЕНИЯ ПАРАМЕТРОВ
        tCYC2= ( Device_Type   == 83)? 12_000·15_000,//12_000,
        tCH2 = ( Device_Type   == 83)?  4_000 6_000 , // min
        tCL2 =( Device_Type    == 83)?  4_000 6_000  ,  // min
        tSA  =( Device_Type    == 83)?  2_500. 2_500, // min
        tHA  = ( Device_Type   == 83)?  500  ·500,  // min
        tSC  = ( Device_Type   == 83)? 2_500 · 2_500, // min
        tHC =  ( Device_Type   == 83)? 500   : 2_500 , //
        tSW =( Device_Type     == 83)?  2_500  2_500,// min
        tHW = ( Device_Type    == 83)?  500  .500, // min
        tSD = ( Device_Type    == 83)? 2_500 : 2_500, // min =
        tHD = ( Device_Type    == 83)?  500  :500, // min = 0 ns
//OUT DEL
 //OUT DEL
        tOE = ( Device_Type   == 83)?    6_000.6_500 , // max
        tOLZ =( Device_Type   == 83)?    1000 :1000,  // min
        tOHZ  = ( Device_Type   == 83)?  3_000.3_000  , // max
        tCD2 =  ( Device_Type   == 83)?  6_000.6_500  ,// max =
        tDC = ( Device_Type   == 83)?    2_000: 2_000,  // min
        tCKHZ  =( Device_Type   == 83)?  5_500 6_000,  // max
        tCKLZ = ( Device_Type   == 83)?  100  :100,
                     //may be need =tHD for eliminate viol msg then read
                     // min =0 in the datasheet !!!!!!
        tCCS = ( Device_Type   == 83)?  5_000 .6_000 ;  // min
/*  закомментирован вариант реализации выходных задержек
     средствами встроенных системных операторов
     //Output Delay
```

```
    (CLKL *> IOL) = (tCD2,tCD2,tCKHZ,tKQ,tCKHZ ,tCKLZ),      // (0>1,1>0,0>Z,Z>1,1>Z,Z>0)
    (OEL_n *> IOL) = (tOE,tOE,tOHZ,tOEQ ,tOHZ,tCKLZ),        // (0>1,1>0,0>Z Z>1 1>Z,Z>0)
  //  tolz, tcklz=toe??
*/
//ПРОВЕРКИ ------------------------------
//clk checks------
   $period(posedge CLKL &&& csl, tCYC2 ),
   $period(posedge CLKR &&& csr, tCYC2 ),
   $width (posedge CLKL &&& csl, tCH2  ),
   $width (posedge CLKR &&& csr, tCH2  ),
   $width (negedge CLKL &&& csl, tCL2  ),
   $width (negedge CLKR &&& csr, tCL2  ),
// setup-hold checks-----
   $setuphold(posedge CLKL &&& csl, AL, tSA, tHA),      //address set-up
   $setuphold(posedge CLKR &&& csr, AR, tSA, tHA),
   $setuphold(posedge CLKL &&& csl, RWL, tSW, tHW),    //RW
   $setuphold(posedge CLKR &&& csr, RWR, tSW, tHW),//
   $setuphold(posedge CLKL &&&  iol_in_check, IOL, tSD, tHD),//DATA
   $setuphold(posedge CLKR &&&  ior_in_check, IOR, tSD, tHD),//
   $setuphold(posedge CLKL , csl, tSC, tHC),           //CHIP SELECT
   $setuphold(posedge CLKR , csr, tSC, tHC),            //
   //$setuphold(posedge CLKL, adv_, tSAV, tHAV),
endspecify
//НИЖЕ ПРОВЕРКА  ЗАДАНИЯ ТИПА УСТРОЙСТВА
integer dev_type,
 initial
begin
   $write( \n*******************************************************\n )
   $write(    dual_port_ramv                                      \n ),
   $write(    Rev  01    march 2001                                 \n ),
   $write( *******************************************************\n ),
   if(( Device_Type   != 83)&&( Device_Type   != 67))begin
       $write( \n*******************************************************\n ),
       $write(   Warning  WRONG DEVICE TYPE !!, assumed 67 !!!          \n ),
       $write( \n*******************************************************\n )
   end
end
////-*******************************************************//-
```

7.5.4.ФУНКЦИОНАЛЬНАЯ ЧАСТЬ

```
//ЗАЩЕЛКИВАНИЕ СИГНАЛОВ ЛЕВОГО ПОРТА/////////////////
//LEFT PORT CONTROL/////////////////////-
always @( posedge CLKL)//chip select
begin
 if (CLKL==1 b1) begin
    cs1l_reg<= #(tREG_DEL)csl  ,
   // cs2l_reg<=cs1l_reg,
    if ((cs1l_reg==1 b1) &&( cs2l_reg==1 b0 )) begin
      #(tCKLZ), cs2l_reg<=1 b1    , //tDC,
           ////??not hZ !!//model approximation, !!
    end
```

```
    else if( (   cs1l_reg==1'b0) && (cs2l_reg==1'b1 )) begin
      #(tCKHZ) ;  cs2l_reg<=1'b0   ;
    end
    else begin
       cs2l_reg<=cs1l_reg,
    end // if,
 end // if,
end // PROCESS,
always @( posedge CLKL)//data in
begin
   if (CLKL==1'b1 ) begin
      iol_in_reg<=#(tREG_DEL)IOL   ;
   end //if;
end //PROCESS;
always @( posedge CLKL)//addr in
begin
 if (CLKL==1'b1 ) begin
    al_reg<=#(tCCS)AL   ;// tREG_DEL;
  end //if;
end //PROCESS,
assign rwl_w=  ( RWL==1'b0 && csl==1'b1)?1'b1:1'b0;
    //1:means read,0:write,
always @( posedge CLKL)// w/r
begin
  if (CLKL==1'b1  ) begin
    if (rwl_w==1'b1 )
       rwl_reg<=#(tCKLZ )rwl_w ; //tDC;  ;//??not hZ !!
    else
       rwl_reg<=#(tCKHZ)rwl_w ;
    //--DATA corruption check
     if (CollisioCheckOn==1 ) begin //--after read -write
       if ( rwl_reg==0 &&  cs1l_reg==1 && rwl_w==1 && oeld_n==0  )begin //--after read
next write
         corr_l_reg<=1,acorr_l_reg<= AL,
       end
       else begin
         corr_l_reg<=0,
       end // IF,
       if( corr_l_reg==1 && (rwl_w==0 || (rwl_w==1 && acorr_l_reg !=AL)) ) begin
        $display(  "time =%t :Possible Data CORRUPTION when WRITE after Read  to Left mem
port addr=%h", $time, al_reg);
       end // IF,
     end
  end //if,
end //PROCESS;
//ЗАЩЕЛКИВАНИЕ СИГНАЛОВ ПРАВОГО ПОРТА ОПУЩЕНО
//  при восстановлении замените  в именах описания левого порта l на r  ///////
//и скопируйте//
//ЧТЕНИЕ-ЗАПИСЬ**********************************
  reg [data_length-1 : 0]data_vx,data_vy ;
  reg clk_collision ;
  time tdelta_p, posr_time, posl_time;
  reg left_collision,right_collision,
```

```
  time  time_l_read, time_r_read,time_l_write,time_r_write,
  initial begin
    time_l_read=0; time_r_read=0; time_l_write=0, time_r_write=0;
  end
  reg [addr_length-1 : 0] addr_l_write, addr_r_write,addr_l_read,addr_r_read;
  reg start_check_col; initial  start_check_col=1'b0,
////////////////////////////////////////////////////////////////
 //rwx_p:PROCESS(CLKL,CLKR)
always @(posedge CLKL ) begin
//ЛЕВЫЙ ПОРТ//////////////////////////-
    if ( ( rwl_reg   == 1'b0 ) && ( csll_reg==1'b1) ) begin
        //READ LEFT////////-
           time_l_read=$time;
           addr_l_read=al_reg;
           data_vx  = sdram [al_reg],
           datal_reg  <=   data_vx;
           if (debug ) begin
       $display("t %0t DEBMSG: R L port,data =%h,addr %h",
              $time,sdram[al_reg],al_reg);
           end //if;
    end
    else if (  (  rwl_reg   == 1'b1)   && ( csll_reg==1'b1 )) begin
   //WRITE LEFT////-
        addr_l_write=al_reg;
        time_l_write=$time,
        start_check_col=1'b1;
           sdram[ al_reg]=iol_in_reg;
          if (debug ) begin
      $display('t %0t  MSG: W  L  port,data =%h,addr =%h",
           $time,sdram[al_reg],al_reg);
          end //if;
   end //if,
end //always,
 //ПРАВЫЙ ПОРТ//////////////////////////////-
always @(posedge CLKR ) begin
   if ((  rwr_reg   == 1'b0)  &&( csir_reg==1'b1)) begin
         //-READ RIGHT////
         time_r_read=$time;addr_r_read=ar_reg,
          data_vy  = sdram [ar_reg]; datar_reg  <=data_vy,
         // start_check_col=1'b1;
         if (debug ) begin
          $display("t %0t   MSG  READ from R port,data =%h,addr = %h",
              $time,sdram[ar_reg],ar_reg);
        end //if,
   end
   else if (  (  rwr_reg   == 1'b1 ) && (csir_reg==1'b1)) begin
      //WRITE RIGHT////-
     addr_r_write=ar_reg; time_r_write=$time,
     start_check_col=1'b1,
        sdram[ar_reg]=ior_in_reg;
        if (debug ) begin
          $display("t %0t  MSG: W to RIGHT  port,data =%h,addr = %h",
              $time,sdram[ar_reg],ar_reg),
```

```
      end //if;
  end //if,
end // always
//КОЛЛИЗИИ
always @( posedge CLKL or posedge CLKR ) begin
  if (CollisioCheckOn &&  (start_check_col==1'b1 ) ) begin
    //simultaneous  WRITE, we set mem word to X!!!
    if ( ($time-time_r_write)<tCCS && ($time-time_l_write)<tCCS &&
           (addr_r_write ==addr_l_write) ) begin
        if( XOn && (iol_in_reg != ior_in_reg) ) begin
          sdram[al_reg]=HIX;
          $display("t =%t, : Collision when W to BOUTH  ports,
           X' val was written to addr =%h",$time,al_reg),
        end //if,
    end //if,
    // sim write to right -read from left
   if (( ($time-time_r_write)<tCCS) && (( $time -time_l_read)<tCCS )
               && (addr_r_write ==addr_l_read) ) begin
         if( XOn ) begin
            data_vx  = HIX,
            datal_reg  <=   data_vx,
         end //if,
   $display("t =%t, : Collis when R  LEFT mem port,OUT  DATA =X,addr =%h',
            $time,al_reg),
   //  sim write to left -read from right
   end
   else if (   ($time-time_l_write)<tCCS && ($time -time_r_read)<tCCS
                 && (addr_l_write ==addr_r_read) ) begin
         if( XOn ) begin
            data_vx  = HIX,
            datar_reg  <=   data_vx,
          end //if,
       $display('t =%t, Collis when R RIGHT mem port,OUT DATA =X,addr =%h",
               $time,ar_reg),
   end //if,
 end //if;
end //PROCESS,
////ВЫХОДНЫЕ ДРАЙВЕРЫ шины данных
//with && without X betwin hold and setup time!!!!!!!!!!/////////-
always @( datar_reg) begin
      if(( datar_reg '= ior_int) && OutXon  ) begin
        #(tDC) ior_int=HIX,
        #(tCD2-tDC)ior_int= datar_reg ,
      end
      else begin
         #(tCD2)ior_int= datar_reg ,
      end //if,
end // process,
always @(  datal_reg) begin
      if( (datal_reg '=iol_int)&& OutXon  ) begin
        #(tDC) iol_int=HIX,
        #(tCD2-tDC)iol_int= datal_reg ;
      end
```

```
      else begin
         #(tCD2)iol_int= datal_reg ,
      end //if,
end //process,
// OE DISTRIBUTED DELAY//////////////////////
//assign oeld_n =  (OEL_n ==1'b1)? #(tOHZ)1'b1  #(tOLZ) 1'b0 , //1'b0 after tOE
//assign oerd_n = (OER_n ==1'b1)? #(tOHZ)1'b1. #(tOLZ) 1 b0 , //1 b0 after tOE,
always @(OEL_n) begin
if (OEL_n ==1'b1)
    oeld_n <=#(tOHZ)1 b1,
else
   oeld_n <=#(tOLZ) 1'b0 ;
end
always @(OER_n) begin
if (OER_n ==1'b1)
    oerd_n <=#(tOHZ)1'b1;
else
   oerd_n <=#(tOLZ) 1'b0 ,
end
////-OUTPUT GATe WITHOUT DELAY model/////////////////////////////////-
assign IOL=((cs2l_reg==1'b1) &&( oeld_n==1 b0)&&( rwl_reg==1'b0) )?
              iol_int: HIZ;
assign IOR=((cs2r_reg==1'b1) &&( oerd_n==1 b0)&& (rwr_reg==1'b0 ))?
             ior_int.HIZ,
endmodule  // OF dual_port_ramv
```

7.6. Тестирующая программа

7.6.1. Переменные и константы

Тестирующая программа рассчитана на возможность совместного моделирования VHDL и VERILOG-моделей памяти (системы ACTIVE–HDL и MODELSIM).

Тест содержит 4 банка памяти — 2 VHDL и 2 VERILOG.

В тесте используются процедуры записи в память и чтения из памяти (представлены только описания процедур левого порта — процедуры, тестирующие правый порт можно восстановить по аналогии с предыдущими примерами — букву l замените на r.

Сам тест предполагает после проверки записи- чтения в ряд ячеек воссоздания условий коллизий и т. п.

```
//--** File name   tb_dual_port_ramv v          **
//-------------------------------------------------------------
// Параметры управления тестом
// Если задать  tDelta = 0; то тест не нарушает временные
//параметры подаваемых на память сигналов
// иначе надо задать  tDelta  = 100,(100 ps)
//  для замены типа кристаллов - отличия по быстродействию
// ( 67 или 83 mhz  )
// надо заменить `define DeviceType
// Для смены соотношения сдвигов тактом наадо изменить
```

```
// define tDel_clk_x2y 0   например define tDel_clk_x2y 400
// ( 4000  вызывает коллизии  , так как  tCCS =5000)
// для отключения выдачи сообщений надо   define debug 0
 timescale 1ps / 1ps
  define debug   0
  define DeviceType   83
  define tDel_clk_x2y    4000
  define data_length 36
  define addr_length  19 // for 2 bank tb config  //18
//left bit of addr used for bank selection!!!
 // define data_num 131072
module tb_dual_port_ramv,
//// testbench for checking  the model of dual port SDRAM
// Inputs of the bank mems-----------------------------
wire clkl, clkr
reg rwl, rwr, oel_n, oer_n, ce0l_n, ce1l, ce0r_n, ce1r,
//  I/Os for chip
reg [ addr_length-1 0] al, ar,//big common addr, used in the tasks
wire [ addr_length-2 0] al_bank=al[ addr_length-2 0],
wire [ addr_length-2 0] ar_bank=ar[ addr_length-2 0],
////individual CHIP SELECT signals,
// used in the tasks see beside instanses
wire [ data_length-1 0] iol_w, ior_w,
//buses for VHDL first-distributed out del model 2 bank inst vhdl
wire [ data_length-1 0] iol_w2, ior_w2
// buses for verilog first model  distributed del model 2 bank inst verilog
wire [ data_length-1 0] iol_w3, ior_w3,
// buses for verilog second model, OUT del by SPECIFY 2 bank inst verilog
//Outputs-----------------------------
//ВСТАВКА ПАРАМЕТРОВ ЗАДЕРЖЕК из файла
 include  dual_port_ramv_timing_data v
//////////////// TIMES ///////////////////////
 reg [8*4:1] DeviceType , //range of device MAX frequency
 reg [31 0]  tDelta,  // INSERT VIOLATION
 reg [31 0]  tCLK_XY_del,  // INSERT clkr delay to clkl
 integer dev_type,
// TIMING PARAM VARIABLES--------------------------------
time tCYC2, // min =
time  tCH2  ,// // min
time  tCL2 ,//  // min
time  tSA ,// // min =
time  tHA  ,//  // min = 0 ns
//OUT DEL
time  tCD2  //  // max =
time  tDC  //    // min
time  tCKLZ  ,//  0,  // min
time  tCKHZ , //  // min
time  tOE //    // min
time  tOLZ ,// 0,  // min
time  tOHZ  // 3_000,  // min
```

```
time  tCCS ;// 5_000;  // min
///////////////INITIAL FOR DEVICE TYPE TIME PARAM////////
initial begin
tDelta=0;
if( `DeviceType == 83)
    dev_type= 1;
else if ( `DeviceType ==67)
    dev_type= 2;
else
    $display("TB ERROR-  defined WRONG DEVICE TYPE"),
tCYC2 =  tCYC2_arr[dev_type] , //12_000; // min =
tCH2 =  tCH2_arr [dev_type] ;//   // min
tCL2=  tCL2_arr[dev_type]  ,//  // min
tSA=  tSA_arr[dev_type] ;//  // min =
tHA = tHA_arr[dev_type] ;//   // min = 0 ns
//OUT DEL
tCD2= tCD2_arr  [dev_type] ;//   // max =
tDC=  tDC_arr  [dev_type] ;//    // min
tCKLZ=   tCKLZ_arr[dev_type]  ;//   // min
tCKHZ=   tCKHZ_arr [dev_type] ; //  // min
tOE=   tOE_arr [dev_type] ;//    // min
tOLZ=    tOLZ_arr  [dev_type] ;//   // min
tOHZ=     tOHZ_arr[dev_type] ;//   // min
tCCS=  tCCS_arr[dev_type] ;//   // min
end
/////////////END OF INITIALISATION OF  TIME PARAM//////////
// полезные константы--------------------------------
reg [`data_length-1  :0  ]   HIZ ;initial HIZ  = {`data_length{1'bz}},
reg [`data_length-1  :0  ]   HIX ;initial HIX  = {`data_length{1'bx}},
reg [`addr_length-1  :0  ]   HIZA;initial HIZA = {`addr_length{1'bz}};
reg [`addr_length-1  :0  ]   HIXA;initial HIXA = {`addr_length{1'bx}},
// common cycle var
integer i,j,k;
//////////////////////////////////////////////////////
// ТАКТОВЫЙ ГЕНЕРАТОР-CLOCK GEN*******
  reg pclk;
//pclk_gen
 initial begin
      pclk =0;
      forever begin
        #(tCL2 -tDelta);
         pclk <= ~pclk;
        #(tCYC2-tCL2-tDelta);
         pclk <= ~pclk;
      end
  end
reg [31:0] tDIF;
initial tDIF=`tDel_clk_x2y ;// initial val clock to clock delay
assign  clkl=pclk;
assign  #tDIF clkr=pclk;
//end clk gen*********************************************
```

7.6.2. ПРОЦЕДУРЫ ЗАПИСИ-ЧТЕНИЯ

```
    ************************************
reg [`data_length-1:0] iol, ior;
integer test_number;
initial
 begin
  iol      =HIZ; ior      =HIZ;
  ar       =HIZA; al       =HIZA;
  test_number=0;
 end
integer error;
// TEST HEADER MSG TASK-------------------------------------
task TEST_HEAD_MSG;
  input [31*8-1:0] test_name; //-------------
  input [31*8-1:0] expected_res;
  begin
   test_number= test_number+1;
   $display ("\nTime =%0t.START Test_number %0d.",$time,test_number);
   $display (" TEST_NAME %0s ",test_name);
   $display ("Expected results: %0s \n",expected_res);
  end
  endtask
//-----ПРОЦЕДУРЫ ЗАПИСИ В ЛЕВЫЙ ПОРТ -WRITE TASKS-------
//запись кажые 2 такта
task LEFT_PORT_WRITE input [`addr_length-1:0] ADDR;
 input [`data_length-1:0] DATA,
 integer i;
begin
 @(posedge clkl),
#(tCYC2 - (tSA - tDelta)); //tSAetup
 oel_n =1;              // output disable
 ce0l_n =0, ce1l =1;      //chip enable
 rwl =0,               //write
 al       = ADDR;
 iol      = DATA;
@(posedge clkl);
 if (`debug == 1)
 $display ("Time =%0t.Ver:LEFT port WRITE : addr=%h,data=%h,busval=%h",
 $time, al,iol,iol_w );
 #(tHA - tDelta); //thold #500;
 rwl  =1,
 al       =HIZA; iol      =HIZ;
 ce0l_n =1, ce1l =0,      //chip disable
//@(posedge clkl);
//delay for eliminate CEL_n delayed to 2 clk=1 for next posible read!
end
endtask
//процедура  записи группы слов на каждом такте
task L_PORT_FAST_WRITE;//--
 input [31.0] start_num,
 input [31:0] num;
```

```
reg [`addr_length-1:0]  ADDR;
reg [`data_length-1:0]  DATA;
integer n;
integer i;
begin
@(posedge clkl);
$display ("\nTime =%0t. START TASK Left port fast write \n", $time),
#(tCYC2 - (tSA - tDelta)); //tsetup
oel_n =1;                // output disable
ce0l_n =0; ce1l =1;        //chip enable
rwl =0;                  //write
ADDR=start_num;
/  ADDR={17'b0,1'bx};
DATA=start_num;
al        = ADDR;
iol       = DATA;
for (i=start_num; i<=start_num+num-2;i=i+1)begin
  @(posedge clkl);
  if (`debug == 1)
     $display ("T =%0t.Ver:Lport WRITE : addr=%h,data=%h, bus==%h",
   .$time, al,iol,iol_w );
  #(tHA - tDelta); //thold #500;
  al        =HIXA;
  iol       =HIX;
  ce0l_n =1; ce1l =0;        //chip disable
  #(tCYC2 -(tHA -tDelta)- (tSA - tDelta)); //tsetup
   ce0l_n =0; ce1l =1;        //chip enable
   ADDR=ADDR+1; DATA=DATA+1;
   al        = ADDR; iol       = DATA;
end              // END FOR
@(posedge clkl);
if (`debug == 1)
     $display ("T =%0t.Ver:L port WRITE LAST WORD : addr=%h,data=%h.",
         $time, al,iol );
#(tHA - tDelta); //thold #500;
al        =HIZA;iol       =HIZ;
ce0l_n =1; ce1l =0;        //chip disable
end
endtask
//--------------------READ -TASKS------------------READ
//Процедуры чтения
// чтение  одного слова
task LEFT_PORT_READ;
input [`addr_length-1:0] ADDR;
input [`data_length-1:0] DATA;
begin
@(posedge clkl);
#(tCYC2 - (tSA  - tDelta)); //tsetap address tS
ce0l_n =0; ce1l =1;        //chip enable
oel_n =0,                  // output enable
rwl =1,                    //read
al        = ADDR;
```

```
@(posedge clkl);
 #(tHA -  tDelta); // thold address tHA
 rwl  =1; al       =HIZA;
 ce0l_n =1; ce1l =0;       //chip disable  ??
 @(posedge clkl);
 @(posedge clkl);
//@(posedge clkl),
//#(tCD2+100);
if(DATA === iol_w)begin
     if (`debug == 1)
       $display ("Time =%0t.Ver:Left port read: addr=%h,data=%h.",
         $time, ADDR,iol_w );
end
else begin
    error =1;
    $display ("T =%0t.ERR.Ver:L port read: addr=%h,data=%h, exp.data=%h ",
       $time, ADDR,iol_w,DATA );
end
end
endtask
//чтение группы слов на каждом такте
// с возможной вставкой одной записи и
// отключением выхода памяти
task L_PORT_READ_WRITE;//-----------------FAST READ -WRITE
 // READ with possibility of WRITE  and out disable
 input [31:0] start_num;  // начальный адрес
 input [31:0] num;        //число слов > 2
 input [31:0] w_num;     //  номер ячейки в которую пойдет запись 1 слова
                          // если мы этого не хотим-задаем  этот номер > num
 input [31.0] oe_num;     // номер ячейки для которой oe отключает выход
                          //   памяти  на 1 такт
                         // когда мы этого не желаем  задаем его > num
 input [31:0] tSETUP_OE; // задержка начала отключения  OE   по отношению к clk
 input [31:0] tHOLD_OE;
 input comp,              //когда параметр ==1  не производится
                           // сравнение выхода с эталоном
 reg [`addr_length-1.0]  ADDR;
 reg [`data_length-1:0]  DATA;
 integer n;
 time toex_posedge,integer i,
 reg oe_controlled; reg write_was;
 reg [31:0] write_num;  reg[31:0] oe_dis_num;
begin
// тело опущено
//////////////////////////// end of tasks ////////////////////////
//- ---------------------------------------------------------------------------
always @( error)
if (error == 1)
   begin
     $display ("\nTime =%0t.TEST STOPPED - ERROR.Check messages above.",$time);
     repeat(10) @(posedge clkl);
     $stop;
```

7.6.3. Подача тестовых векторов

```
//////////НИЖЕ ЗАДАНИЕ ТЕСТА test sequence ////////////////////////////////////
integer  test_num;    // number of test case, convinent to include in waveform
reg [8*20:0] test_name;// name of test case, convinent to include in waveform
//integer  i, j;
reg [2:0] b_lng;
initial
 begin  #1;
  //$dumpvars;
  //$recordvars;
  error =0; i =0; j =0;
  rwl =1; rwr =1;  oel_n=1; oer_n =1;
  ce0l_n =1; ce1l = 0; ce0r_n =1; ce1r =0;
 // SIMPLE TESTS 1-4 WITH CE and OE don't READING CASES----------------
 TEST_HEAD_MSG (" #1 Left port fast write","TO addr 1-10 write data 1-10");
 L_PORT_FAST_WRITE(1,10 );//!!
// L_PORT_FAST_WRITE(HIXA,10 );
 TEST_HEAD_MSG (" #2 Left port fast READ"," read data 1-10 from addr 1-10");
L_PORT_FAST_READ(1,10,4,8,3*tSA,tHA,1 );
  TEST_HEAD_MSG (" #3 RIGHT port fast write","TO addr 11-20 write data 11-20");
 R_PORT_FAST_WRITE(11,10 );
 TEST_HEAD_MSG (" #4. RIGHT port fast READ","read  from addr 11-20,cs and OE ");
// test_num =101; //-----------
// test_name="RIGHT port fast READ";
// $display ("\nTime =%0t. Test #101. RIGHT port fast READ \n", $time);
 R_PORT_FAST_READ(11,10,14,18,3*tSA,tHA,1 );
//-------------------TWO -BANK TESTS- 5-9---------------------
 TEST_HEAD_MSG (" #5  Left port fast write with bank 1 to 2 changing");
 L_PORT_FAST_WRITE(18'h3FFFE,10 );
 TEST_HEAD_MSG (" #6 Left port fast READ with bank 1 to 2 changing " );
L_PORT_FAST_READ(18'h3FFFE,10,0,0,0,0,1 );
 TEST_HEAD_MSG (" #7  Right port fast write with bank 1 to 2 changing");
 R_PORT_FAST_WRITE(18'h3FFFA,10 );
 TEST_HEAD_MSG (" #8 Right port fast READ with bank 1 to 2 changing ");
 R_PORT_FAST_READ(18'h3FFFA,10,0,0,0,0,1 );
// --чтение-запись вперемежку с отдельного порта -----
//READ TO WRITE TO READ -------------------
 test_num =1001; //-------------
 test_name="Left port READ-to-write";
 $display ("\nTime =%0t. Test #1001. Left port READ -WRITE\n", $time);
 $display ("after READ 2 words EXPECTED  NO OP cycle  WRITING 1 word=3 to addr 3");
 // set word 3 to 0
 LEFT_PORT_WRITE(3,0);
 L_PORT_READ_WRITE (1,10,3,20,tSA,tHA,1 );
//check writing 3 to addr 3
 L_PORT_FAST_READ(1,10,0,0,0,0,1 );
 test_num =1002; //-------------
 test_name="Left port READ-to-write";
 $display ("\nTime =%0t. Test #1002. Left port READ -WRITE\n", $time);
 $display ("после чтения 2  ожидается запись 1 слова по адресу   3 ");
  LEFT_PORT_WRITE(3,0);//сначала 0 в ячейку с адресом 3
  L_PORT_READ_WRITE (1,10,3,3,3*tSA,tHA,1 );//чтение с прерыванием на запись
```

```
 //проверка - что по адресу 3  на самом деле ? 3?
 L_PORT_FAST_READ(1,10,0,0,0,0,1 );
  test_num =1003; //-------------
 test_name="Right port READ-to-write";
 $display ("\nTime =%0t. Test #1003. Right port READ -WRITE\n", $time);
 $display ("after READ 2 words EXPECTED  NO OPERATION cycle  WRITING 1 word");
// set word 3 to 0
 RIGHT_PORT_WRITE(3,0);
 R_PORT_READ_WRITE (1,10,3,20,tSA,tHA,1 );
//check writing to addr 3
 R_PORT_FAST_READ(1,10,0,0,0,0,1 ),
 test_num =1004; //------------
 test_name="Right port READ-to-write";
 $display ("\nTime =%0t. Test #1004. Right port READ -WRITE\n", $time);
 $display ("after READ 2 words EXPECTED OE CONTROLLED  WRITING 1 word ");
// set word 3 to 0
 RIGHT_PORT_WRITE(3,0),
 R_PORT_READ_WRITE (1,10,3,3,3*tSA,tHA,1 );
//check writing to addr 3
 R_PORT_FAST_READ(1,10,0,0,0,0,1 );
//-----------------------------------------------------
 test_num =105, //-------------
 test_name="BANK SELECT READ";
 $display ("\nTime =%0t. Test #105.BANK SELECT READ  \n", $time);
 $display ("EXPECTED CHANGE DATA OUT STREAM FROM BANK1 TO BANK 2");
 L_PORT_FAST_READ(18'h3FFFE,10,20,20,0,0,1 );
 test_num =106; //-------------
 test_name="LPORT:R-to-W-to-R,OE=0";
 $display ("\nTime =%0t. Test #106. LPORT:R-to-W-to-R,OE=0\n", $time),
 $display ("EXPECTED NO OPERATION cycle");
 fork
 begin  L_PORT_FAST_READ(1,3,20,20,0,0,0 );// 5 clk duration
          repeat (1)@(posedge clkl);
        L_PORT_FAST_READ(1,3,20,20,0,0,0 );
 end
 begin repeat(3) @(posedge clkl); L_PORT_FAST_WRITE(3,1 );
         repeat(5) @(posedge clkl);
 end
 join
 test_num =107; //-------------
 test_name="LPORT:R-to-W-to-R,OE controlled";
 $display ("\nTime =%0t. Test #106. LPORT:R-to-W-to-R,OE controlled\n", $time);
 fork
   begin    L_PORT_FAST_READ(1,3,20,20,0,0,0 ),//5 clk
           repeat (1)@(posedge clkl);
           L_PORT_FAST_READ(1,3,20,20,0,0,0 );
   end
   begin   repeat(5) @(posedge clkl),  L_PORT_FAST_WRITE(1,1 ),
           repeat(5) @(posedge clkl);
   end
 join
// Параллельная работа портов
 test_num =102; //-------------
```

```
test_name="CONCURENT Left-RIGHT ports fast write ,
$display ("\nTime =%0t  Test #102   Lw-Rw CONCURENT\n , $time),
$display ("EXPECTED CONFLICT MSG AND X VAL TO THE MEM 10 words'),
fork
  L_PORT_FAST_WRITE(1,10 ),
  R_PORT_FAST_WRITE(1,10 ),
join
test_num =1011, //-------------
test_name="RIGHT port fast READ ,
$display ("\nTime =%0t  Test #1011  RIGHT port fast READ \n", $time),
$display ("EXPECTED  X VAL from THE MEM 10 words,but  comparator disabled "),
R_PORT_FAST_READ(1,10,20,20,0,0,0 ),
test_num =103, //-------------
test_name=" Left-read RIGHT-write CONCUR  ,
$display ("\nTime =%0t  Test #103   Lr,Rw CONCURENT  \n , $time),
$display ('CONFLICT MSG AND X VAL from  MEM 10 words, comparator disabled'),
fork
  L_PORT_FAST_READ(1,10,20,20,0,0,0 ),
  R_PORT_FAST_WRITE(1,10 ),
join
test_num =104; //-------------
test_name=" Left-write RIGHT-read CONCUR ,
$display ("\nTime =%0t. Test #104   Lw,Rr CONCURENT  \n", $time),
$display (" NO conflict  because when clk collision, waddr/= raddr ),
fork
R_PORT_FAST_READ(1,10,20,20,0,0,1 ),
L_PORT_FAST_WRITE(1,10 ),
join
// end concurent tests
test_num =2; //-------------
test_name='Left port write ,
$display ("\nTime =%0t  Test #2  Left port write \n , $time),
for (i=0, i<=13;i=i+1)    // 131071
  begin LEFT_PORT_WRITE(i,i),   end
repeat(20) @(posedge clk1),
 test_num =3;//------------------
test_name="Left port read",
$display ("\nTime =%0t. Test #3  Left port read   \n', $time),
for (i=0; i<=13;i=i+1)     //131071
  begin LEFT_PORT_READ (i,i), end
test_num =4, //-------------------
test_name='Left port write/read sequent',
$display ("\nTime =%0t. Test #4  Left port write/read sequent   \n , $time),
for (i=0, i<=10,i=i+1)
  begin
   LEFT_PORT_WRITE(i,i); LEFT_PORT_READ (i,i),
  end
test_num =5;
test_name="right port write";
$display ("\nTime =%0t  Test #5  RIGHT port write   \n", $time),
for (i=11; i<=22;i=i+1)
  begin RIGHT_PORT_WRITE(i,i); end
repeat(20) @(posedge clk1);
```

```
 test_num =55; //-------------
 test_name="right port fast write",
 $display ("\nTime =%0t. Test #55  Right port fast write \n", $time),
 R_PORT_FAST_WRITE(1,10 );
test_num =6;
 test_name="right port read";
$display ("\nTime =%0t. Test #6  RIGHT port read . \n , $time);
for (i=11; i<=22;i=i+1)
   begin
    RIGHT_PORT_READ (i,i),
   end
 test_num =7;
  test_name=" RIGHT port write/read sequent ;
 $display ("\nTime =%0t. Test #7  RIGHT port write/read sequent . \n', $time);
for (i=23; i<=35;i=i+1)
   begin
    RIGHT_PORT_WRITE(i,i), RIGHT_PORT_READ (i,i),
   end
/*
test_num =8;
  test_name="Concurrent write/read from ports ",
 $display ("\nTime =%0t. Test #8. Concurrent write/read from ports . \n", $time);
 for (i=23; i<=35,i=i+1)
   begin
   fork
    LEFT_PORT_WRITE  (i,i),
    RIGHT_PORT_WRITE (i,i);
    LEFT_PORT_WRITE  (i,i);
   join
   fork
    RIGHT_PORT_READ (i,i);
    LEFT_PORT_READ  (i,i),
    oin
   end
*/
 test_num =9;
  test_name=" Left port write and  RIGHT port read sequent";
 $display ("\nTime =%0t Test #9.Left port write and  RIGHT port read seq \n",
                $time),
 for (i=50; i<=80;i=i+1)
   begin
    LEFT_PORT_WRITE(i,i), RIGHT_PORT_READ (i,i);
   end
test_num =10;
   test_name="RIGHT port write and  LEFT port read sequent";
 $display ("\nTime =%0t Test #10 RIGHT port write and LEFT port read seq.\n",
 $time);
 for (i=81; i<=100;i=i+1)
   begin
    RIGHT_PORT_WRITE(i,i); LEFT_PORT_READ (i,i),
   end
 test_num =11;
 $display ("\nTime =%0tTest #11.RIGHT port write fast and LEFT port read \n",
```

```
    $time),
 repeat (20) @(posedge clkl),     // delay to change clkl
 fork
   for (i=101, i<=200,i=i+1)    RIGHT_PORT_WRITE(i,i),
   for (j=101, j<=200,j=j+1)    LEFT_PORT_READ  (j,j),
 join
 repeat (5) @(posedge clkl),    // delay to change delay betwin clk
// clkl leader
// start collision--------------------
  tDIF=1000,
 repeat (5) @(posedge clkr)     // delay to change delay betwin clk
 test_num =13,
 $display ( \nTime =%0t  Test #13  CONCURENT  ports write with collision   \n , $time),
 fork
   for (i=201, i<=210,i=i+1)    RIGHT_PORT_WRITE(i,i),
   for (j=201, j<=210,j=j+1)    LEFT_PORT_WRITE(j,j),
 join
 repeat (5) @(posedge clkl),
test_num =14,
 $display ( \nTime =%0t Test #14 COnC RP- wr  and  LP  read with collision  \n ,
                $time)
 fork
   for (i=211, i<=220,i=i+1)    RIGHT_PORT_WRITE(i,i),
   for (j=211, j<=220,j=j+1)    LEFT_PORT_READ(j,j)
 join
 repeat (5) @(posedge clkl),
test_num =15,
 $display ( \nT =%0t Test #15  CONC RP read and LP  write with collision  \n ,
  $time),
 fork
   for (i=211, i<=230 i=i+1)    RIGHT_PORT_READ(i,i),
   for (j=211, j<=230,j=j+1)    LEFT_PORT_WRITE(j,j),
 join
//////////////////////////////////////////
 if(error)begin
    $display ( \nTime =%0t  ERROR  Test FAILED  See messages above \n ,
          $time),
end
else begin
$display ( \n************************************************* ),
 $display ( \nTime =%0t    Test PASSED \n , $time),
 $display ( ************************************************************** ),
// $finish,
$stop,
end
end
//////////////////////////// test sequence finish ////////////////
//-------------------------------------------------------------------------------
// --НИЖЕ КОНКРЕТИЗАЦИИ ,блоков ПАМЯТИ ------------------------------
assign iol_w=iol,assign ior_w=ior,
assign iol_w2=iol,assign ior_w2=ior,
assign iol_w3=iol,assign ior_w3=ior,
//chip selection by using   msf addr bit
```

```
wire addr_msfl= al[`addr_length-1 ],
wire addr_msfr= ar[`addr_length-1 ],
wire ce0l_n_bank1,ce0l_n_bank2,
assign ce0l_n_bank1=(addr_msfl==0)?ce0l_n:1,
assign ce0l_n_bank2=(addr_msfl==1)?ce0l_n 1,
wire ce0r_n_bank1,ce0r_n_bank2,
assign ce0r_n_bank1=(addr_msfr==0)?ce0r_n 1;
assign ce0r_n_bank2=(addr_msfr==1)?ce0r_n.1,
/// vhdl  модели банков памяти
// банк 1- модель VHDL
 \dual_port_ramv_vh bank1 (
     al_bank,        ar_bank,//  ar  ,
     rwl,            rwr, // rwr  ,
     oel_n,          oer_n, // oer_n ,
     iol_w,          ior_w, //  ior  ,
     clkl ,          clkr,//  clkr ,
     ce0l_n_bank1,   ce1l,//  ce1l ,
     ce0r_n_bank1,   ce1r //  ce1r
     );
/// vhdl - банк 2 модель VHDL
 \dual_port_ramv_vh bank2 (
     al_bank,          ar_bank,//  ar  ,
     rwl,              rwr, // rwr  ,
     oel_n,            oer_n, // oer_n ,
     iol_w,            ior_w, //  ior  ,
     clkl,             clkr,//  clkr ,
     ce0l_n_bank2      ce1l,//  ce1l ,
     ce0r_n_bank2,     ce1r //  ce1r
 ),
/// verilog модели
//банк 1
 dual_port_ramv bank1_ver (
     .AL(al_bank),     AR(ar_bank),//  ar  ,
     .RWL(rwl),       .RWR(rwr), // rwr  ,
      OEL_n(oel_n),    OER_n(oer_n), // oer_n ,
     .IOL(iol_w2),    .IOR(ior_w2), //  ior  ,
      CLKL(clkl),      CLKR(clkr),//  clkr ,
     .CE0L_n(ce0l_n_bank1),.CE1L(ce1l),//  ce1l ,
     .CE0R_n(ce0r_n_bank1),   CE1R(ce1r) //  ce1r
 );
//банк 2
dual_port_ramv bank2_ver (
     .AL( al_bank),      .AR(ar_bank),//  ar  ,
     .RWL(rwl),           RWR(rwr), // rwr  ,
     .OEL_n(oel_n),      .OER_n(oer_n), // oer_n ,
      IOL(iol_w2),        IOR(ior_w2), //  ior  ,
      CLKL(clkl),        .CLKR(clkr),//  clkr ,
     .CE0L_n(ce0l_n_bank2),  CE1L(ce1l),//  ce1l ,
     .CE0R_n(ce0r_n_bank2),  CE1R(ce1r) //  ce1r
 );
endmodule
```

7.6.4. Временные параметры сигналов теста

```
Файл с временными параметрами теста dual_port_ramv_timing_data v
// dual_port_ramv_timing_data,
//////////////////////////////////////-
integer ii,
reg[31 0]   tCYC2_arr[1 2] , initial begin
                tCYC2_arr[1]  =  12_000, // min
                tCYC2_arr[2]  =  15_000_,
              end
reg[31 0]   tCH2_arr [1 2],initial begin
                tCH2_arr [1] =  4_000 , // min
                tCH2_arr [2]=   6_000,
              end
integer   tCL2_arr[1 2] ,initial begin
                tCL2_arr [1]  = 4_000 , // min
                tCL2_arr[2] =   6_000 , //  min
              end
integer   tR_arr[1 2] ,initial begin
                tR_arr [1]  = 3_000   // min
                tR_arr[2] =   3_000 , //  min
              end
integer   tF_arr[1 2] ,initial begin
                tF_arr [1]  = 3_000 , // min
                tF_arr[2] =   3_000 ,//  min
              end
integer   tSA_arr[1 2],initial begin
                 tSA_arr[1]  = 2_500, // min =
                 tSA_arr[2]  = 2_500,
              end
integer   tHA_arr[1 2],initial begin
                 tHA_arr[1]  =  500, // min = 0 ns
                 tHA_arr[2] =   500,
              end
integer   tSC_arr[1 2],initial begin
                 tSC_arr[1]  = 2_500, // min =
                 tSC_arr[2]  = 2_500,
              end
integer   tHC_arr[1 2],initial begin
                 tHC_arr[1]  =  500, // min = 0 ns
                 tHC_arr[2] =   500,
              end
integer   tSW_arr[1 2],initial begin
                 tSW_arr[1]  = 2_500, // min =
                 tSW_arr[2]  = 2_500,
              end
integer   tHW_arr[1 2],initial begin
                 tHW_arr[1]  =  500, // min = 0 ns
                 tHW_arr[2] =   500,
              end
integer   tSD_arr[1 2],initial begin
                 tSD_arr[1]  = 2_500, // min =
                 tSD_arr[2]  = 2_500,
              end
```

```
integer   tHD_arr[1:2];initial begin
                tHD_arr[1]  =  500; // min = 0 ns
                tHD_arr[2] =   500;
              end
//ВЫХОДНЫЕ ЗАДЕРЖКИ
integer    tOE_arr [1:2];initial begin
                tOE_arr[1]  =   6_000;  // max
                tOE_arr[2] =    6_500;
              end
integer    tOLZ_arr  [1:2];initial begin
                tOLZ_arr [1]  =  1000;  // min
                tOLZ_arr [2] =   1000;
              end
integer     tOHZ_arr[1:2];initial begin
                tOHZ_arr[1]   =  3_000;  // max
                tOHZ_arr [2]  =  3_000;
               end
integer   tCD2_arr  [1:2];initial begin
                tCD2_arr [1] =   6_000; // max =
                tCD2_arr [2]=    6_500;
              end
integer   tDC_arr  [1:2];initial begin
                tDC_arr [1] =  2_000;   // min
                tDC_arr [2] =  2_000 ;
              end
integer    tCKHZ_arr [1:2];initial begin
                tCKHZ_arr[1]   =  5_500;  // max
                tCKHZ_arr[2] =    6_000;
              end
integer    tCKLZ_arr[1:2] ;initial begin
                tCKLZ_arr[1]  =  100;    // min
                tCKLZ_arr [2] =  100 ;
              end
integer     tCCS_arr[1:2];initial begin
                tCCS_arr[1]   =  5_000;  // min
                tCCS_arr [2]  =  6_000;
               end
```

Вопросы к главе 7

1. Управление левым портом не отличается от правого, нельзя ли выделить описание в отдельный модуль и дважды конкретизировать?

Ответ: можно, попробуйте реализовать этот модуль, особенно это эффективно для моделей четырехпортовых памятей.

2. Почему в VERILOG-тесте описаны разные процедуры чтения из левого и правого портов, нельзя ли описать одну?

Так как старый VERILOG не позволяет параллельно запускать несколько копий процедуры (моделирование конфликтов в памяти), это было сделано умышленно.

Если у Вас используется версия 2000, то попробуйте реализовать одну процедуру.

Приложение 1
Краткий справочник по языку VHDL

Ниже в упрощенном виде перечислены основные конструкции языка VHDL с примерами их реализации. При описании языка его синтаксические понятия обозначены по-русски, например, имя, выражение и т. п. Символ ::= (это есть) отделяет левую и правую части правил. В квадратных скобках указаны необязательные компоненты синтаксических конструкций. В фигурных скобках { } заключается список альтернативных конструкций. Слово «или» разделяет их альтернативные элементы, многоточие разделяет граничные значения элементов, запятая — сами элементы.

Конструкции, которыми дополнена версия VHDL-93, отмечены символом '*' в начале строки.

1. Основы VHDL

1. Алфавит VHDL

буквы::=A...Z, a...z
цифры::=0,...9
специальные символы::= & ' " * + - , . : ; # < = > | ( ) / _
символ пробела::=

2. Ограничители

Символ	*Наименование и пояснения*
&	амперсанд — операция сцепление строк
'	апостроф — для записи символьных литералов и атрибутивных имен
"	кавычки — для записи строковых литералов
*	операция умножения
**	операция возведения в степень
+	операции сложения и вычитания
,	запятая — разделяет элементы списка
.	точка — элемент вещественных констант и составных имен
:	двоеточие — элемент описаний имен (сигналов, меток, портов и т. п.)
;	точка с запятой — символ конца оператора
#	диез — элемент записи литералов в 16-ричной, 8-ричной, двоичной системах счисления
<	операции отношения: меньше
=	равно
>	больше
!	(или |) для разделения элементов оператора выбора (case)
( )	круглые скобки
=>	стрелка — для ключевых параметров и элементов выбора
\	* косая черта — ограничитель расширенных имен

<=	оператор назначения сигнала, а также операция меньше-равно
:=	оператор присваивание переменной
/=	операции отношения: не равно
>=	больше-равно
<>	бокс — для указания неограниченных массивов
--	символ начала комментария
_	подчеркивание (в именах и литералах)

Для разделения лексических элементов помимо ограничителей могут использоваться пробелы, символы управления форматом (возврат, табуляция и т. п.) символ конца строки.

3. Идентификаторы

Последовательность букв, цифр и знака _ (подчеркивание), начинающаяся буквы.

Примеры: VA_SIA, E2_E4

Заглавные и строчные буквы в идентификаторах не различаются.

Расширенные

*Расширенные идентификаторы — последовательность графических символов кода ASCII, заключенных в обратные косые черточки. В расширенных идентификаторах различаются большие и малые буквы.

Пример: \ VASIA*liza\ .

4. Литералы

Вид литерала	*Примеры и пояснения*
число	
вещественное	128.3 1.284E+2 — вещественное 128.3
целое	128 12_8 — целое 128
в 16-ой системе	16#7F# — целое 127 в 16-ой системе
в 2-ой	2#111_1111# — целое 127 в 2-ой системе
символ	'0' 'A' — символы 0 и A
строка	"xo, xo" "0101" — строки
	X"AF" — шестнадцатиричный — "10111111"
битово-строковые	B"10111111" — двоичный
	O"77" — восьмеричный — "111111"

5. Комментарии

начало комментария — символ --, конец комментария — конец строки

Пример: -- это комментарий

6. Имена

Виды имен	*Примеры*
зарезервированное	and, or, entity и т. д., смотри ниже их список
простое имя-идентификатор	BB E2_E4
индексируемое	AA(i) — i-ый элемент массива AA
составное	X.Z — элемент Z структуры X
сечение	R (0 to 7) — сечение массива R
атрибутивное	S'DELAYED — атрибут DELAYED сигнала S

Список зарезервированных (ключевых) имен VHDL
(символом отмечены введенные в VHDL-93)*

abs	component	guarded	next	range	subtype
access	configuration	if	nor	record	then
after	constant	*impure	not	register	to
alias	disconnect	in	null	*reject	transport
all	downto	*inertial	of	rem	type
and	else	inout	on	report	*unaffected
architecture	elsif	is	open	return	units
array	end	label	or	*rol	until
assert	entity	library	others	*ror	use
attribute	exit	linkage	out	select	variable
begin	file	*literal	package	severity	wait
block	for	loop	port	signal	when
body	function	map	*postponed	*sla	while
buffer	generate	mod	procedure	*sll	with
bus	generic	nand	process	*sra	*xnor
case	*group	new	*pure	*srl	xor
				shared	

Запрещается использовать зарезервированные слова как объявленные идентификаторы. Например, неверно variable entity:bit;

7. Типы и виды (классы) данных (объектов) VHDL (классификация по форме описаний и свойствам)

Скалярные типы

Перечислимый	Числовые (целый и вещественный)	Физический

В том числе предопределенные — определенные в пакете STANDARD скалярные типы

BIT	INTEGER	TIME
BOOLEAN	POSITIVE	
CHARACTER	NATURAL	
SEVERITY_LEVEL	REAL	
FILE_OPEN_KIND		
FILE_OPEN_STATUS		

Кроме того, VHDL имеет ссылочный тип(access).

Составные (структурные) типы (агрегаты)

массив (array)	запись (record)	файл (file)

В том числе определенные в пакетах STANDARD и TEXTIO структурные типы

BIT_VECTOR	TEXT
STRING	

Классы (виды) данных (объектов) включают: константы (constant), переменные (variable), сигналы (signal).

8. Типы данных — группировка по операциям

Ниже приведены предопределенные (пакет STANDARD) типы данных VHD и их описатели, сгруппированные по допустимым операциям над ними с указан ем диапазона значений:

Логический тип	*Арифметический (числовой) тип*	*Символьный тип*	*Физический тип*	*Прочие типы*
BOOLEAN (булевский FALSE, TRUE)	REAL (вещественный)	CHARACTER (символ ASCII)	TIME (единицы: fs-фемтосек., ps-пикосек., ns-наносек., us-микросек., ms-милисек., sec-секунды, min-минуты, hr-часы)	FILE (файловый) SEVERITY_LEVEL уровень серьезности нарушений ACCESS (указатель)
BIT (битовый '0','1')	INTEGER (целый)	STRING (строка символов, одномерный массив символов)		
BIT_VECTOR (битовый вектор - одномерный массив битов)	POSITIVE (целые >= 1)			
	NATURAL (целые >=0)			

9. Операции (в группе — приоритет одинаков, группы следует сверху вниз в по рядке уменьшения приоритета)

Тип	Состав	Пояснения
Разные	**, abs, not	Степень, абсолютное значение, логическое НЕ
Мультипликативные	*, /, mod, rem	A=(A/B)*B+(A rem B) A=B*N+(A mod B)
Знаковые	+, -	одноместные +, -
Аддитивные	+, -, &	сложение, вычитание, сцепление (конкатенация)
Отношения	=, /=, <, <=, >, >=	равно, не равно, меньше, меньше-равно, больше, больше-равно
* Сдвиги и ротации	sll, srl, sla, sra, rol, ror	влево, вправо, влево арифм., вправо арифм., циклический влево, циклический вправо
Логические	and, or, nand, nor, xor, xnor	И, ИЛИ, И-НЕ, ИЛИ-НЕ, исключающее ИЛИ (XOR)

Операции отношения и логические определены над скалярными типами дан ных и одномерными массивами (логические над Bit и Boolean).

Внимание! AND и OR имеют одинаковый приоритет.

Пример	*Результат*	*Пояснения*
2**3	8	возведение в степень
abs (-25)	25	абсолютная величина
not B"101"	B"010"	поразрядная инверсия
2*3	6	умножение

3/2	1	при делении целых остаток отбрасывается
7 mod 2	1	
7 rem 2	1	остаток
-7 rem 2	- 1	остаток
B"101"&B"10"	B"10110"	конкатенация (сцепление)
B"101">B"100"	TRUE	отношение
'M'<'A'	FALSE	
B"101" and B"100"	B"100"	длины операндов битовых векторов должны совпадать
B"101" or B"010"	B"111"	логические операции
B"101" nand B"010"	B"111"	определены над битовыми
B"101" nor B"010"	B"000"	и булевскими векторами
B"101" xor B"011"	B"110"	равной длины
B"101" xnor B"011"	B"001"	
TRUE or FLASE	TRUE	
* B"101" sll 2	B"100"	* — операции VHDL 93,
* B"101" srl 2	B"001"	отсутствовавшие в VHDL-87
* B"101" sla 2	B"100"	
* B"101" sra 2	B"100"	
* B"101" rol 2	B"110"	
* B"101" ror 2	B"011"	

Примеры неверных выражений: B"101" AND B'1', B"101"+1

Операторы объявлений языка VHDL

10. Объявления типов

Синтаксис	*Примеры*
объявление типа	
type имя is указание типа;	type A is range 0 to 100; -- объявлен тип A
объявление подтипа	-- его подтип B
subtype имя is [функция разрешения] имя типа [ограничение диапазона или индексное];	subtype B is A range 10 to 20; subtype X is Fun_Res BIT_VECTOR (0 to 10);
перечислимый тип	
type имя is (список значений);	type A4 is('x', '0', '1', 'z'); -- номер позиции перечисли- -- мого значения 'X' равен 0
целый (диапазонный) тип	
type имя is range диапазон;	type FF is range -2 to 6 ; -- диапазон типа FF от -2 до 6
физический (масштабный)	
type имя is range диапазон units последовательность единиц измерения, начиная с базовой end units;	type DISTANCE is range 0 to 1 E16 units A; -- ангстрем nt=10A; -- далее другие единицы

	end units;
<u>индексируемый тип (массив)</u> type имя is array (диапазон) of указание подтипа; указание подтипа::=[имя функции] имя типа [ограничение диапазона или индексное]	type TAB is array (6 to 7,10 downto 2)of A4; -- TAB двумерный массив -- типа A4 type M is array (positive range<>)of integer; -- M неограниченный массив
<u>тип запись (структура)</u> type имя is record объявления элементов end record[имя];	type ZAP is record B:character; C: BIT; end record;
<u>неполностью объявленный тип</u> type имя;	type BB;
<u>файловый тип</u> type имя is file of тип;	type F1 is file of ZAP;
<u>тип указатель (ссылочный)</u> type имя is access указание подтипа;	type PTR is access BB;

11. Операторы объявлений VHDL

Синтаксис	*Примеры*
<u>объявление констант</u> constant список имен: указание подтипа [:=значение];	constant PI: REAL:=3.14; constant B: A4:='X'; constant D: INTEGER;

<u>объявление переменных</u>
[shared]variable список имен:
указание подтипа
[:=начальное значение];
	variable PP:REAL:=-5.1;
	variable CC:BIT_VECTOR(0 to 7);
	variable MASS:M(1 to 6);

<u>объявление файлов</u> file имя:указание подтипа указание_направленности is логическое имя файла;	file DATA:F1 open read_mode is "XOXO"; --входной файл DATA с -- логическим именем --XOXO
<u>объявление сигналов</u> signal список имен: указание подтипа [register или BUS] [:=начальное значение];	signal S1, S2: BIT; signal AAA: BIT_VECTOR (4 downto 0):="11111"; signal ZZ: Bres register; --тип Bres - с функцией разрешения
<u>объявление атрибутов</u> attribute имя: тип;	attribute KONTAKT_NO: POSITIVe;
<u>объявление алиаса</u>	

(дополнительного имени) alias идентификатор: подтип is имя;	variable RN:BIT_VECTOR (0 to 20); -- нулевой разряд-имя SIGN alias SIGN:BIT is RN(0);

<u>объявление компонент</u>
component имя
[generic(параметры настройки);]
[port(описания локальных портов);]
end component[имя];
	component INER
	generic(TZ:TIME:=10 ns);
	port(X1, X2:IN BIT; Y:OUT BIT);
	end component;

<u>12. Операторы объявления подпрограмм</u>
(имя_П - имя подрограммы)

<u>объявление процедуры</u> procedure имя_П [(список формальных параметров)]; имя_П:=идентификатор или "зарезервированное слово — операция"	procedure RR1(X:IN STRING; signal Y:IN INTEGER; signal Z:OUT REAL);
<u>объявление функций</u> function имя_П[(список формальных параметров)] return тип;	function FAN (V:IN BIT_VECTOR) return BIT; function "+"(A, B: A4) return A4;

<u>Описание пакетов</u>

Синтаксис объявлений пакетов опущен — смотрите примеры пакетов STANDARD и TEXTIO ниже

<u>13. Описание объекта проекта</u>

entity имя объекта проекта is [generic(параметры настройки);] [port(список портов);] [объявления типов ,подтипов, констант, сигналов, файлов, подпрограмм, атрибутов, алиасов, спецификации атрибутов, спецификации разъединений, описания тел подпрограмм, связывающие указания] [begin параллельные операторы утверждений, пассивного вызова процедур, пассивных процессов] end[entity][имя];	-- интерфейс N-разрядной -- схемы И-НЕ с задержкой 20 ns; entity INE_N is generic(TZ:TIME:=20 ns; N:NATURAL); port(X:IN BIT_VECTOR (1 to N); Y:OUT BIT); end INE_N;

<u>14. Описание архитектуры объекта проекта</u>

architecture имя of имя объекта проекта is [объявления подпрограмм, типов, подтипов, констант, сигналов,	-- архитектура N-разрядной -- схемы И-НЕ architecture P_INEN of INE_N is

файлов, алиасов, компонент, атрибутов, спецификации атрибутов, спецификации конфигураций, спецификации разъединений, связывающие указания]
begin
[параллельные операторы]
end[architecture] [имя];

```
begin
  process(X)
    variable A:BIT;
     begin
      A:='0';
      for i in X'RANGE
     loop
       if X(i)='0' then
       A:='1';
       exit;
       end if;
     end loop;
    Y<=A after TZ;
  end process;
end P_INEN;
```

15. Объявление конфигурации

[library список имен библиотек;]
[указание использования;]
configuration имя of
имя объекта проекта is
{связывающие указания или
спецификации атрибутов}
конфигурация блока
end [имя];

```
library WORK, LIB1; use WORK.all,
LIB1.all;
  configuration C1 of
   WORK.PROEKT1 is
     for BLOC1
       for K1: R use
         entity LIB1.U1(A2)
           generic map (3 ns);
     end for;
   end for;
end C1;
```

конфигурация блока ::=
for{ имя архитектуры
или метка оператора блока
или метка оператора генерации}
[(спецификация индексов)]
связывающее указание
{конфигурация блока или
конфигурация компоненты}
end for;
связывающее указание::= use элемент указания
элемент указания::=
entity имя библиотеки.имя объекта
(имя архитектуры)
[карта параметров настройки]
[карта портов];
конфигурация компоненты::=
for спецификация компоненты
[связывающее указание]
[конфигурация блока]
end for;

Пример конфигурации компоненты со связывающим указанием:

```
for all:IOPORT
use entity A(BEH)
port map (X, Y, Z);
end for;
```

16. Спецификация

Спецификация ассоциирует дополнительную информацию с ранее выполненными VHDL- описаниями.

спецификация атрибута

```
attribute {имя атрибута
    of {список объектов или
        others или
        all} :класс is выражение;
класс::=entity или procedure или
procedure или function или package или
type или subtype или constant или signal
или variable или component или label
```

```
-- атрибут емкость контакта
attribute CAPACITANCE
of KONTAKT :signal is 15 PF;
-- атрибут номер контакта
attribute PIN_NO of
CIN : signal is 10;
```

спецификация конфигурации

```
for {метка экземпляра или
others или
all}: имя компоненты
use
аспект объекта проекта
[generic map (список сопоставления
параметров настройки)]
[port map(список сопоставления портов)];
аспект объекта проекта ::=
entity имя объекта[(имя архитектуры)]
```

спецификация конфигурации связывает информацию с меткой, соответствующей индивидуальному экземпляру компонент данного типа

```
for C1 : SUM use
    entity WORK.ADD(PA);

    for all: INER use
        entity LIB.LA3(X);

for E1, E2 : RECT use
    entity XY
    generic map (10 ns)
    port map (X1, X2, X3);
```

спецификация разъединения

```
disconnect
{others или all или список имен
охраняемых сигналов: тип}
after задержка;
```

```
disconnect X, Y:BIT after 10 ns;
disconnect others after 50 ns;
```

17. Фрагмент проекта

```
Фрагмент проекта ::=
[указание контекста]

библиотечный модуль
библиотечный модуль::=
объявление объекта или
объявление конфигурации или
объявление пакета или
описание архитектуры или
тело пакета
```

```
library X, Y; use X.all, Y.D;
entity Z is port
(A, B: in MM;C:inout EE);
end Z;
-- Из библиотеки X доступно все;
-- из Y - модуль D.
-- Библиотеки WORK и STD
-- не нуждаются
-- в указании видимости.
```

указание контекста::=
library список имен или
use список составных имен;

18. Последовательные операторы

Синтаксис	*Пояснения и примеры*
оператор присваивания переменной левая часть:=выражение; левая часть::=имя или агрегат	--Изменяет значение переменной Y := X+1; A(i):=D(j+1); B(1 to 3) := C(5 downto 3);
оператор назначения сигнала левая часть<=[reject] [inertial или transport]диаграмма; диаграмма::=список элементов диаграммы элемент диаграммы::=выражение [after время] или null [after время]	--Изменяет планируемые -выходные --формы сигнала X<= M after 10 ns; X<='0'after 10 ns, '1' after 20 ns; X<=TRANSPORT M after N;
оператор ожидания wait [on список сигналов] [until условие][for задержка]; условие::=булево выражение	--Вызывает приостановку процесса wait on A, B until C='1'; wait until X > Y; wait for 15 ns;
оператор утверждения assert условие [report сообщение] [severity NOTE или WARNING или ERROR или FAILURE]; сообщение::="строка"	--Проверяет истинность условия и ------формирует сообщение, -- если условие ложно assert X<0 report "X больше 0" severity WARNING;
оператор сообщения report строка;	--выдача сообщения report "VASIA"; report "AA" & integer'image(20)

условный оператор
if условие then операторы
[elsif условие then операторы]
[else операторы]
end if;
	-- Выбирает для выполнения одну
	-- из альтернатив
	if A>B then X:=Y; Z:=K;
	else M:=K;
	end if;

оператор выбора
 	
case выражение is
when выбор_1 => операторы_1
when выбор_2 => операторы_2
when выбор_i => операторы_i
end case;
выбор ::= значение или диапазон или простое имя или others
	-- Выбирает для выполнения
	--одну из альтернатив
	case Y is
	when 2|3 => X:=0;
	when 5 => X:=X+1; L:=F;
	when 6 => Z:=X; C:=M+N;
	when 7 to 9 => M:=N;
	when others => C:=D;
	end case;

оператор цикла (3 варианта) [имя:] loop последовательность операторов	-- Выполняет тело цикла 0 или -- более раз. Ниже i:=1; -- примеры эквивалентны C1:loop X(i):=X(i+1);i:=i+1;

end loop [имя];	exit C1 when i>10;
[имя:] for индекс in	end loop C1;
интервал loop	C2:for i in 1 to 10 loop
последовательность операторов	X(i):=X(i+1);
end loop [метка];	end loop C2;
[имя:] while условие loop	i:=1;
последовательность операторов	C3:while i<=10 loop
end loop [имя];	X(i):=X(i+1); i:=i+1;
	end loop C3;
оператор выхода	-- Служит для выхода из цикла
exit [метка цикла] [when условие];	exit when A>B;
оператор перехода	-- Служит для завершения одной из
next [метка цикла] [when условие];	-- итераций цикла
пустой оператор	-- Не вызывает никаких действий
null;	
оператор возврата	-- Завершает исполнение функции
	-- или процедуры.
return [выражение];	return A+B;
оператор вызова подпрограммы	-- Обеспечивает вызов процедуры
	-- или функции
имя процедуры [(список параметров)];	PR1(X, Y, Z);--PR1- процедура
имя функции [(список параметров)]	X:=FUNC(E+1);--FUNK- функция

19. Параллельные операторы VHDL

Синтаксис	*Пояснения и примеры*
оператор процесса	
[имя:][postponed] process	P1:process (X1, X2)
[(список чувствительности)]	variable Z: BIT;
[объявления]	begin
begin	Z:=X1 and X2;
[последовательные операторы]	wait for 10 ns;
end process[postponed] [имя];	Y<=Z;
	end process P1;
оператор блока	-- Определяет внутренний блок,
	-- представляющий часть проекта
[имя]:block[(охранное выражение)]	-- D-триггер срабатывает
	-- по фронту C
[параметры настройки][порты]	-- описан в форме охраняемого
	-- блока
[объявления]	B1:block (C='1' and not C'STABLE')
begin	begin
параллельные операторы	Q<=guarded D after 10 ns;
end block [имя];	end block B1;
параллельный оператор назначения сигнала	
[имя:][postponed]условное назначение	A<=B after 10 ns;

или выборочное назначение;	-- инерциальная задержка
условное назначение::=	C<= transport '1' after 60 ns;
левая часть<=[guarded][transport]	-- транспортная задержка
условная диаграмма;	-- ниже условное назначение
условная диаграмма::=	A<='1' when X>Y else
[диаграмма when условие else]	'0' when X=Y else
диаграмма	'Z' after 4 ns;
выборочное назначение::=	-- ниже выборочное назначение
with выражение select	with i select
левая часть<=[guarded][transport]	D<="1000" when 10,
[диаграмма when выбор,]	"0100" when 20,
диаграмма when выбор;	"0011" when others;

<u>параллельный оператор утверждения</u>

[имя:] [postponed] assert условие	
[report сообщение]	assert X>0 OR (X<-5)
[severity NOTE или WARNING или ERROR	report " X вне пределов"
или FAILURE];	severity WARNING;

<u>оператор параллельного вызова процедуры</u>

[метки:] [postponed]	
имя [(фактические параметры)];	PR1(X, Y, Z);

<u>оператор конкретизации компонента</u>

имя:имя компонента[generic map	D1:INER port map(X1, X2, Y);
фактические значения параметров)]	D2:ILI generic map(10 ns)
[port map(фактические значения)];	port map(A=>X,B=>2,C=>1);

<u>оператор генерации (2 варианта)</u>

метка: if условие generate	M1:for i in range 1 to 10
параллельные операторы	generate
end generate [метка];	--10 компонент INER
метка: for параметр in диапазон	B: INER(X(i),Y(i),Z(i));
generate	end generate M1;
параллельные операторы	
end generate [метка];	

2. Основные различия версий VHDL-93 и VHDL-87

1. Откладываемый процесс (postponed)

Это процесс, который может запускаться любым событием в одном из нескольких дельта-циклов модели, но только после достижения системой устойчивого состояния, т. е. после последнего дельта-цикла в данный момент модельного времени.

Синтаксис — слово postponed, употребленное перед одним из четырех типов параллельных операторов:

postponed оператор утверждения assert или
оператор процесса process или
оператор параллельного вызова процедуры или
оператор параллельного назначения сигнала

Примеры: postponed A<=X or Y; -- отложенное назначение A.
postponed process N, K -- отложенный процесс
begin c:=d; end postponed process;

2. Разделяемые переменные (shared)

Допускается описание переменных в блоках, т. е. вне процессов и подпрограмм. Тем самым обеспечивается доступ к этим переменным из нескольких процессов. В связи с этим возможен недетерминизм в модели. Использование подобных переменных целесообразно лишь в моделях системного уровня для поддержки объектно-ориентированного стиля программирования и с применением контроля доступа к разделяемым переменным из нескольких процессов.

Пример: shared variable X: integer:=25;

3. Непосредственная конкретизация (прямое создание экземпляров компонент)

Для тех пользователей, которые находят трехступенчатый механизм, состоящий из:

1) описания компонент,
2) конкретизации компонент,
3) связывания посредством конфигурации,

излишне «многословным» при структурном описании систем, VHDL-93 допускает непосредственную конкретизацию. Прямым использованием идентификаторов объектов проекта и архитектур в операторах конкретизации можно задавать всю необходимую информацию в конкретизациях компонент.

Пример:
X:entity WORK.F(F_A) port map (A1,A2,B1,B2);

Пояснение — в операторе конкретизации по имени X — указан объект проекта F, помещенный в библиотеку проекта WORK вместе с его архитектурой F_A, и карта его портов.

4. Группы (group)

Понятие группы используется для выделения подмножеств объектов. Конструкция предназначена не для моделирования, а для спецификации систем при автоматическом синтезе.

5. Оператор сообщений(report)

6. Оператор параллельного назначения сигнала-несколько изменен синтаксис и добавлено ключевое слово UNAFFECTED для описания случая отсутствия транзакций на выход

Изменения в предопределенном окружении (стандартных пакетах) VHDL

1. Дополнения в составе операций

Они включают операцию исключающего ИЛИ-НЕ (XNOR) в группе логических операций и новую группу сдвиговых влево (l) или вправо (r) операций — логического sll, srl, арифметического sla, sra и циклического rol, ror сдвига. Первый аргумент — это сдвигаемая величина (одномерный массив bit или boolean), второй — направление и величина сдвига (integer).

Пример: B"1011" sll 2 равно B"0110"
B"1011" sla 1 равно B"1110"
B"1011" rol 1 равно B"0111".

2. Новые предопределенные атрибуты

Возможность преобразования значений скалярного типа в строковое представление и наоборот обеспечивается атрибутами 'IMAGE и 'VALUE.

Атрибут DRIVING_VALUE обеспечивает доступ к значению, назначенному заданным процессом в сигнал (оно может отличаться от эффективного значения этого сигнала).

Атрибут FOREIGN обеспечивает связь с подпрограммами на других языках, а атрибут IMPURE позволяет использовать функции с побочным эффектом.

Атрибуты 'ASCENDING, 'BEHAVIOR, 'STRUCTURE, 'PATH_NAME, INSTANT_NAME, 'SIMPLE_NAME обеспечивают дополнительные возможности.

Незначительные изменения и уточнения

1. Механизм задержки с резекцией

Механизм задержки расширен возможностью указания максимальной длины входного импульса, который не изменяет выходной сигнал — более продолжительные сигналы проходят на выход даже еслит длина их меньше, чем задержка, указанная в операторе назначения (в VHDL-87 это было невозможно!). Используется ключевое слово reject — резекция.

Пример:

S<= reject 3 ns transport X after 7 ns;

Импульс X длительностью менее 3 ns не проходит через S. Задержка же более продолжительного сигнала в S равна 7 ns.

2. Расширенные идентификаторы

Синтаксис идентификаторов обогащен т. н. «расширенными идентификаторами». Эти идентификаторы заключаются в скобки из символов \ и могут включать любые графические символы, допускаемые в качестве лексических элементов языка, включая символ пробела. Это расширение упрощает связь с уже существующими библиотеками компонент и пакетов на других языках.

3. Алиасы для разных объектов

В VHDL допускаются альтернативные имена (alias) не только для объектов типа константа, переменных и сигналов, но и для всех других элементов, определяемых объявленными именами, например, для объектов проекта.

4. Оператор генерации (generate) с объявлениями

Оператор генерации в VHDL-93 определен, как область объявлений, и может дополнительно включать собственные объявления, в том числе объявления компонент, конкретизируемых в этом операторе.

5. Оператор report для выдачи сообщений

В ряде случаев при отслеживании состояния модели необходимо получать сообщения без тестирования определенных условий, что в VHDL-87 делается с помощью оператора assert, в котором проверка условия заменена константой FALSE. В VHDL-93 введен специальный оператор report для этих же целей.

6. Симметричные операторные скобки

Определенное упрощение синтаксиса языка достигается единообразным стилем скобочных форм. Закрывающая скобка содержит повторение открывающей. Помимо симметричных скобок типа if - end if, process - end process и т. п. допускаются (но не обязательны) дополнения после end в следующих случаях: entity - end entity, architecture - end architecture, configuration - end configuration, package - end package, package body - end package body.

7. Открытие и закрытие файлов

Файлы могут открываться и закрываться — процедуры FILE_OPEN, FILE_CLOSE.

3. Синтезабельное подмножество языка VHDL

Синтезабельное подмножество VHDL определено стандартом IEEE P1076.6 .

Список основных синтезабельных ключевых слов и конструкций приведен ниже.

Ключевые слова VHDL

alias, all, and, architecture, abs, array, attribute, begin, block, body
buffer, case, component, character, constant, downto, else, elsif,
end, entity, exit, for, function, generate, generic, if, in, inout, is,
library, loop, map, package, mod, nand, nor, next, null, of, on, open,
or, others, out, then, port, procedure, process, range, record, rem, return, select,
signal, subtype, to, type, until, use, variable, wait, when, while, with, xor

Типы и виды данных (объектов)

Скаляры и вектора
Std_logic, Std_ulogic, Boolean, bit, integer,
Std_logic_vector, Std_ulogic_vector, bit_vector,
а также типы SIGNED и UNSIGNED, предопределенные в пакетах NUMERIC_BIT, NUMERIC_STD.

Описания (объявления)

Описание объекта проекта	entity,architecture
Описание параметров	generic
Описание портов	in,out,inout
Описание сигналов	signal
Описание констант	constant
Описание переменных	variable
Описание типа	type,subtype
Описание функций	function
Описание процедур	procedure

Операторы

Процесса — process end process

Конкретизации компонентов — допускается как позиционное так и поименованное (ключевое) соответствие портов сигналам

Назначение сигнала — <=

Присваивание переменной	:=
Условный	if, then, elsif, else, end if
Цикла, выхода, перехода	for, while, loop, end loop
Выбора	case, end case
Процедуры	

Дополнительные ограничения и пояснения

Задержки. Система синтеза обычно игнорирует задержки

Неопределенные и высокоимпедансные значения сигналов. Запрещается использовать значения X и Z в выражениях — только при описании тристабильных буферов разрешается присваивать значение Z.

Начальные значения сигналов. Запрещается указывать их в объявлениях сигналов (в реальной схеме при включении питания и до поступления сигнала сброса значение триггеров не определено).

Часть других конструкций языка, как и задержки, просто игнорируется системами синтеза, а иные вызывают появление предупреждающих сообщений и сообщений об ошибках.

4. Предопределенное окружение языка VHDL

4.1. Пакет STANDARD

Пакет STANDARD определяет ряд типов, подтипов и функций. Предполагается, что в начале каждого модуля проекта присутствует неявное описание контекста, ссылающееся на этот пакет. Пакет STANDARD не может быть изменен пользователем.

```
package STANDARD is
- предопределенные перечислимые типы:
type boolean is (false,true);
type bit is ('0', '1');
type character is (
nul, soh, stx, etx, eot, enq, ack, bel,
bs, ht, lf, vt, ff, cr, so, si,
dle, dc1, dc2, dc3, dc4, nak, syn, etb,
can, em, sub, esc, fsp, gsp, rsp, usp,

' ', '!', '"', '#', '$', '%', '&', '''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',

'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
```

```
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', del,

c128, c129, c130, c131, c132, c133, c134, c135,
c136, c137, c138, c139, c140, c141, c142, c143,
c144, c145, c146, c147, c148, c149, c150, c151,
c152, c153, c154, c155, c156, c157, c158, c159,

-- the character code for 160 is there (NBSP),
-- but prints as no char

' ', 'Ў', 'ў', '?', '¤', '?', '¦', §",
'Ё', '©', 'Є', '«', '¬', '', '®', 'Ї',
'°', '+', 'І', 'і', 'ґ', 'µ', '¶', '·',
'ё', '№', 'є', '»', 'ј', 'Ѕ', 'ѕ', 'ї',
--ниже настройка на кириллицу

'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З',
'И', 'Й', 'К', 'Л', 'М', 'Н', 'О', 'П',
'Р', 'С', 'Т', 'У', 'Ф', 'Х', 'Ц', 'Ч',
'Ш', 'Щ', 'Ъ', 'Ы', 'Ь', 'Э', 'Ю', 'Я',

'а', 'б', 'в', 'г', 'д', 'е', 'ж', 'з',
'и', 'й', 'к', 'л', 'м', 'н', 'о', 'п',
'р', 'с', 'т', 'у', 'ф', 'х', 'ц', 'ч',
'ш', 'щ', 'ъ', 'ы', 'ь', 'э', 'ю', 'я' );

type severity_level is (note, warning, error, failure);

-- предопределенные перечислимые типы:

type integer is range -2147483648 to 2147483647; -- определен реализацией;

type real is range -1.0E308 to 1.0E308; -- определен реализацией;

--предопределенный тип TIME;

type time is range -2147483647 to 2147483647
units
fs;
ps = 1000 fs;
ns = 1000 ps;
us = 1000 ns;
ms = 1000 us;
sec = 1000 ms;
min = 60 sec;
hr = 60 min;
end units;
subtype delay_length is time range 0 fs to time'high;
-- функция, возвращающая текущее время моделирования;

impure function now return delay_length;
- предопределенные целые подтипы;
```

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
-- предопределенные индексируемые типы;

type string is array (positive range <>) of character; --номера позиций с 1 !!!
type bit_vector is array (natural range <>) of bit;-- номера позиций с 0 !!!
type file_open_kind is (
read_mode,
write_mode,
append_mode);
type file_open_status is (
open_ok,
status_error,
name_error,
mode_error);
attribute foreign : string;
end standard;
```

Примечание: ASCII мнемоники для символов FS, GS, и US представлены в типе CHARACTER, соответственно, символами FSP, GSP, RSP, USP во избежание конфликта с единицами типа TIME.

4.2. Пакет TEXTIO

```
package TEXTIO is
  type LINE is access string;
  type TEXT is file of string;
  type SIDE is (right, left);
subtype WIDTH is natural;

    -- changed for vhdl92 syntax:
file input : TEXT open read_mode is "STD_INPUT";
file output : TEXT open write_mode is "STD_OUTPUT";
    -- changed for vhdl92 syntax (and now a built-in):
procedure READLINE(file f: TEXT; L: out LINE);
procedure READ(L:inout LINE; VALUE: out bit; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out bit);
procedure READ(L:inout LINE; VALUE: out bit_vector; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out bit_vector);
procedure READ(L:inout LINE; VALUE: out BOOLEAN; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out character; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out character);
procedure READ(L:inout LINE; VALUE: out integer; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out integer);
procedure READ(L:inout LINE; VALUE: out real; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out real);
procedure READ(L:inout LINE; VALUE: out string; GOOD : out BOOLEAN);
procedure READ(L:inout LINE; VALUE: out string);
procedure READ(L:inout LINE; VALUE: out time; GOOD : out BOOLEAN);
```

```
procedure READ(L:inout LINE; VALUE: out time);

    -- changed for vhdl92 syntax (and now a built-in):
procedure WRITELINE(file f : TEXT; L : inout LINE);
procedure WRITE(L : inout LINE; VALUE : in bit;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in bit_vector;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in BOOLEAN;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in character;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in integer;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in real;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0;
    DIGITS: in NATURAL := 0);
procedure WRITE(L : inout LINE; VALUE : in string;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0);
procedure WRITE(L : inout LINE; VALUE : in time;
    JUSTIFIED: in SIDE := right;
    FIELD: in WIDTH := 0;
    UNIT: in TIME := ns);
end;
```

4.3. Предопределенные атрибуты

Атрибут определяет некоторую характеристику именованного понятия. Ряд атрибутов относится к предопределенным и может одним из следующих видов: типом, диапазоном, значением, сигналом или функцией. Остальные атрибуты определяются пользователем и являются константами.

Ряд предопределенных атрибутов сигналов уже был рассмотрен в разделе 2. Ниже приведена часть остальных предопределенных атрибутов.

Атрибты массивов на примере массива AX и строки CX:

```
type TT is: array (10 downto 0) of integer; variable AX:TT;
variable CX: string (1 to 7);
```

AX'left равно 10, CX'left равно 1
AX'right равно 0, CX'right равно 7
AX'low равно 0, CX'low равно 1
AX'high равно 10, CX'high равно 7
AX'range равно 10 downto 1, CX'range равно 1 to 7
AX'length равно 11, CX'length равно 7.

Атрибуты типов на примере типа EE:
Type EE is range 10 to 70;

EE'left равно 10	EE'right равно 70
EE'low равно 10	EE'high равно 70
EE'image(11) равно "11"	EE'value("11") равно 11
EE'Pos (11) равно 2	EE'suc(11) равно 12

5. Многозначная логика — IEEE пакеты и функции преобразования типов

Используемые обозначения.

Русскими словами обозначены понятия — например МАССИВ.

В символы [] заключена необязательная конструкция.

Символы / и | означают металингвистическое ИЛИ.

Например to | downto означает возможность употребления либо того, либо другого ключевого слова.

Сокращение — эквивалент.

b — BIT
u/l — STD_ULOGIC/STD_LOGIC
bv — BIT_VECTOR
uv — STD_ULOGIC_VECTOR
lv — STD_LOGIC_VECTOR
un — UNSIGNED
sg — SIGNED
na — NATURAL (целое > 0)
In — INTEGER
sm — SMALL_INT это подтип (subtype INTEGER range 0 to 1)

Обозначения групп типов данных:
STD_ULV_T ::= u/l |uv | lv --наименование общее для трех типов

Обозначения групп операций:
ARITH_OP ::=+ | - | * | / | rem | mod -- АРИФМЕТИЧЕСКИЕ
RELAT_OP ::= < | > | <= | >= | = | /= -- ОТНОШЕНИЯ

1. Пакет IEEE STD_LOGIC_1164

1.1. Логические значения

'U'	Неинициализированное
'X'/'W'	Сильная/Слабая неопределенность
'0'/'L'	Сильный/слабый 0
'1'/'H'	Сильная/Слабая 1
'Z'	Высокий импеданс
'~'	Безразличное

1.2. Предопределенные типы

Скалярные типы

STD_ULOGIC — базовый тип

Подтипы: STD_LOGIC — разрешенный STD_ULOGIC
X01 — разрешенный X,0,1
X01Z — разрешенный X,0,1,Z
UX01 — разрешенный U,X,0,1
UX01Z — разрешенный U,X,0,1,Z .

Векторные типы
STD_ULOGIC_VECTOR(na to| downto na) — массив STD_ULOGIC
STD_LOGIC_VECTOR(na to | downto na) — массив STD_LOGIC

1.3. Переопределенные операции

Операция	*Левый операнд*	*Символ*	*Правый операнд*
Поразрядное И	STD_ULV_T	and	STD_ULV_T
Поразрядное ИЛИ	STD_ULV_T	or	STD_ULV_T
Поразрядное Исключающее ИЛИ	STD_ULV_T	xor	STD_ULV_T
Поразрядное НЕ		not	STD_ULV_T

Внимание! Арифметические операции над std_logic векторами не допускаются!

1.4. Функции преобразования типов данных

xmap — необязательный аргумент, управляющий способом преобразования значений, отличных от 1 и 0. Без него, например X, обычно преобразуется в 0.

Из	*В*	*Функция*	*Пример, пояснение*
u/l	b	TO_BIT(apr[,xmap])	TO_BIT('X') дает '0'
uv,lv	bv	TO_BITVECTOR(apr,[xmap])	TO_BITVECTOR("0ZU1")="0001"
b	ul,lv	TO_STDULOGIC(apr)	TO_STDULOGIC("0101")="0101"
bv,ul	lv	TO_STDLOGICVECTOR(apr)	
bv,lv	uv	TO_STDULOGICVECTOR(apr)	

1.5. Функции над сигналами

Фронт RISING_EDGE (ИМЯ СИГНАЛА) — фронт сигнала
Срез FALLING_EDGE (ИМЯ СИГНАЛА) — срез
Есть ли Xl IS_X (ИМЯ ОБЬЕКТА) — есть ли значение X в обьекте?
Пример: Is_x("0X1") дает TRUE.

2. Пакет IEEE NUMERIC_STD

Нужен при арифметических операциях над std_logic векторами.
Переопределяет арифметические операции и операции отношения.
Предопределенные типы — массивы STD_LOGIC.
UNSIGNED(na to | downto na) array of STD_LOGIC -- беззнаковый вектор (число)
SIGNED(na to | downto na) array of STD_LOGIC -- левый разряд знаковый

2.1. Переопределенные операции

Левый операнд	*Операция*	*Правый операнд*	*Результат*
	abs	sg	sg
	—	sg	sg
un	ARITH_OP	un	un
sg	ARITH_OP	sg	sg

un	ARITH_OP	na	un
sg	ARITH_OP	In	sg
un	RELAT_OP	un	bool
sg	RELAT_OP	sg	bool
un	RELAT_OP	na	bool
sg	RELAT_OP	In	bool

Из приведенного видно что применение этого пакета позволяет например складывать данные типа unsigned как числа без знака и т. д.

2.2 Функции преобразования типов

Из	*В*	*Функция*
un,lv	sg	SIGNED(apr)
sg,lv	un	UNSIGNED(apr)
sg,un	lv	STD_LOGIC_VECTOR(apr)
un,sg	In	TO_INTEGER(apr)
lna	un	TO_UNSIGNED(apr)
In	sg	TO_SIGNED(apr)

Пример: to_integer("0101") дает целое 5.

2.3. Предопределенные функции сдвигов

Функция	*Тип результата*
SHIFT_LEFT(un,na)	un -- сдвиг влево
SHIFT_RIGHT(un, na)	un
SHIFT_LEFT(sg,na)	sg
SHIFT_RIGHT(sg,na)	sg
ROTATE_LEFT(un,na)	un
ROTATE_RIGHT(un, na)	un
ROTATE_LEFT(sg, na)	sg
ROTATE_RIGHT(sg,na)	sg
RESIZE(sg, na)	sg
RESIZE(un, na)	un

3. Пакет IEEE NUMERIC_BIT

Позволяет производить арифметические операции над двоичными векторами.
Переопределяет арифметические операции и операции отношения.
Предопределенные типы — массивы BIT.
UNSIGNED(na to | downto na)array of bit
SIGNED(na to | downto na)array of bit

3.1. Переопределенные операции

Левый операнд	*Операция*	*Правый операнд*	*Результат*
	abs	sg	sg
	—	sg	sg
un	ARITH_OP	un	un
sg	ARITH_OP	sg	sg
un	ARITH_OP	na	un
sg	ARITH_OP	In	sg

un	RELAT_OP	un	bool
sg	RELAT_OP	sg	bool
un	RELAT_OP	na	bool
sg	RELAT_OP	In	bool

Имена функций сдвига те же, что в пакете NUMERIC_STD.

3.2. Функции преобразования типов

Из	*В*	*Функция*
un,bv	sg	SIGNED(apr)
sg,bv	un	UNSIGNED(apr)
sg,un	bv	STD_LOGIC_VECTOR(apr)
un,sg	In	TO_INTEGER(apr)
Ina	un	TO_UNSIGNED(apr)
In	sg	TO_SIGNED(apr)

4. Пакет STD_LOGIC_ARITH

Включает арифметические операции, операции отношения и сдвига.
Предопределенные типы — массивы STD_LOGIC.
UNSIGNED(na to | downto na)
SIGNED(na to | downto na)
SMALL_INT Integer значения 0 или 1
Переопределенные операции-описание опущено

Функции преобразования типов

Из	*В*	*Функция*
un,lv	sg	SIGNED(apr)
sg,lv	un	UNSIGNED(apr)
sg,un	lv	STD_LOGIC_VECTOR(apr)
un,sg	In	CONV_INTEGER(apr)
In,un,sg,u	un	CONV_UNSIGNED(apr,размер)
In,un,sg,u	sg	CONV_SIGNED(apr, размер)
In,un,sg,u	lv	CONV_STD_LOGIC_VECTOR(apr, размер)

Пример: CONV_STD_LOGIC_VECTOR(5, 4) дает вектор "0101".

5. Пакеты STD_LOGIC_UNSIGNED и STD_LOGIC_UNSIGNED

Unsigned обеспечивает возможность отношений и арифметических операций над std_logic_vector как над беззнаковыми числами, а signed — как над числами со знаком (левый разряд) в дополнительном коде.

Пример использования функции преобразования conv_integer ниже.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
use work.STD_LOGIC_arith.all;use work.STD_LOGIC_unsigned.all;
architecture beh of pol5 is
s ignal i : integer; signal a,b,c : std_logic_vector(0 to 10):=(others=>'0')
begin
c<=unsigned(b)+1;
    i<= conv_integer(a);-- unsigned
    b<=conv_std_logic_vector(i,5);-- arith
end beh;
```

Приложение 2
VERILOG — краткий справочник

В данном справочнике в основном представлено подмножество языка, используемое при функциональном описании систем и тестов. Опущены вопросы схемотехнического уровня описаний,встроенных и пользовательских примитивов,интерфейса с другими языками — PLI, конфигурирование, генерация и т. п. Специально отмечены новые свойства версии стандарта IEEE 1364—2001 (ниже VERILOG-2000), который полностью включает свойства предыдущей версии VERILOG—1995.

Принятые обозначения:

... — повторяющиеся элементы синтаксической конструкции;

[] — в квадратных скобках заключены необязательные элементы синтаксической конструкции (исключение — случаи использования [] в самих синтаксических конструкциях);

или — разделяет альтернативные варианты конструкции, как и символ |.

1. Лексические элементы

1.1. Символы

Символы кода ASCII и невидимые знаки, именуемые ниже как пустое_место: пробел, табуляция, конец строки, перевод каретки, EOF — «конец файла».

1.2. Комментарии и атрибуты

1.2.1. Комментарий

// начало одной строки комментариев, заканчивающейся концом строки.
/* начало многострочного комментария, заканчивающегося */

Пример: f=1;//ХОХО-ПАРНИША

1.2.2. Атрибут (VERILOG-2000) — текст в скобках (* *)

Специальный комментарий для систем синтеза, отражающий свойства модулей, операторов или портов.

Пример: (* full_case *) case (k)1:a=b,2: a=c; endcase // атрибут оператора case

1.3. Символы операций и скобок

	VERILOG-1995 и дополнительные VERILOG-2000						
Скобки	( )	[ ]	{ }	? :		(* *)	V-2000
Операции							
Арифметические	+	-	*	/	%	**	V-2000
Логические побитовые	&	|	^	~^	~		

Отношения	<	>	<=	>=				
Редукции	&	~&	|	~|	^	~^		
Логические	&&	||	!					
Сравнения	==	!=						
Идентичности	===	!==						
Сдвига	<<	>>					<<<	>>> V-2000
Разные	->						@*	V-2000
Присваивание	=	<=						

1.4. Имена

1.4.1. Простые

Имена(идентификаторы) начинаются с букв a-z A-Z или символа _ или символа $ и могут включать помимо них цифры 0-9.

Примеры: reset_n, clr, e2_e4, _F, A___B_

Большие и малые буквы в именах различаются.

Пример разных имен: VaS и Vas.

1.4.2. Системные

Имена, начинающиеся с $ зарезервированы за системными процедурами и функциями:

Пример: $time.

1.4.3. Расширенные имена

Начинаются со знака обратной косой черты, могут включать любые символы и заканчиваются пустым местом.

Пример: \f+d-pad1.

1.4.4. Ключевые слова (зарезервированные имена)

always	endmodule	large	reg	tranif0
and	endprimitive	macromodule	release	tranif1
assign	endspecify	nand	repeat	tri
attribute	endtable	negedge	rnmos	tri0
begin	endtask	nmos	rpmos	tri1
buf	event	nor	rtran	triand
bufif0	for	not	rtranif0	trior
bufif1	force	notif0	rtranif1	trireg
case	forever	notif1	scalared	unsigned
casex	fork	or	signed	vectored
casez	function	output	small	wait
cmos	highz0	parameter	specify	wand
deassign	highz1	pmos	specparam	weak0
default	if	posedge	strength	weak1
defparam	ifnone	primitive	strong0	while
disable	initial	pull0	strong1	wire
edge	inout	pull1	supply0	wor
else	input	pulldown	supply1	xnor
end	integer	pullup	table	xor
endcase	join	rcmos	task	
endfunction	medium	real	time	
	module	realtime	tran	

1.4.4.1. Новые(дополнительные) ключевые слова VERILOG-2000

Блок конфигурации — config , endconfig , design, instance, cell, use, liblist.

Блок генерации — generate, endgenerate, genvar, localparam.

Прочие — automatic, cell, noshowcancelled, pulsestyle_onevent, pulsestyle_ondetect, showcancelled.

Новые системные функции и процедуры — $ferror, $fgetc, $fgets, $fflush, $fread, $fscanf, $fseek, $fsscanf, $ftel, $rewind, $sformat, $swrite, $swriteb, $swriteh, $swriteo, $ungetc.

Директивы компиляции — `ifndef, `elsif, `undef,`line.

1.4.4.2. Ключевые слова — синтезабельное подмножество VERILOG-95 (более точно см. в описании конкретной системы синтеза)

and	always	assign	begin	buf	bufif0	bufif1	case
casez	casex	default	defparam	disable	else	end	endcase
endfunction		endmodule	endtask	function	for	if	inout
input	integer	module	macromodule		nand	negedge	nor
not	notif0	notif1	or	output	parameter	posedge	
reg	endtask	supply0	supply1	task	tri	tri0	tri1
wand	wire	while	wor	xnor	xor		

1.4.4.3. Ключевые слова — несинтезабельное подмножество VERILOG-95 (более точно см. в описании конкретной системы синтеза)

cmos	deassign	disable	edge	endprimitive		endspecify	endtable
event	force	forever	fork	highz0	highz1		
ifnone	join	large	medium	nmos			
pmos	primitive		pull0				
pull1	pulldown		pullup	rcmos	real	realtime	release
repeat	rnmos	rpmos	rtran	rtranif0	rtranif1	scalare	
small	specify	specparam	strong0	strong1	table	task	
time	tran	tranif0	tranif1	triand	trior	trireg	
vectored		wait	weak0	weak1			

1.4.4.4. Ключевые слова — верификабельное подмножество VERILOG-95

always	assign	begin	case	casex	default	defparam
else	end	endcase	endfunction	endmodule	function	if
initial	inout	input	module	negedge	or	output
parameter	posedge	reg	tri	tri0	tri1	wire

1.4.5. Составные (иерархические) имена

Они организуют доступ к переменной, соединению, процедуре или функции из любого места описания проекта. Доступ может быть организован с помощью иерархического имени, определяющего путь к обьекту. Полный путь начинается с имени старшего модуля проекта и содержит имена конкретизаций модулей, содержащих обьект; относительный путь начинается с имени конкретизации данного модуля.

Пример: ALU.carry.bl1.cin

1.5. Логические значения

1.5.1. Алфавит

Значения четырехзначного алфавита :1'b0,1'b1,1'bz,1'bx.
Символ ? — это еще один способ представления логического значения Z.

Обозначение	Пояснения
0	Ноль, низкий уровень, ложь
1	Единица, высокий уровень, истинно
z или Z	Состояние высокого импенданса (третье состояние)
x или X	Неопределенное, неизвестное, неинициализированое

1.5.2. Сила (вес) сигнала

Verilog-сигналы (данные вида соединение) могут принимать значение восьми сил(весов): 4 управляющих, 3 емкостных и силу третьего состояния (сила высокого импенданса).

Уровень силы	Название силы	Ключевое слово		Мнемоника	
7	(питание) Supply Drive	supply0	supply1	Su0	Su1
6	(сильный сигнал -с выхода элемента) Strong Drive	strong0	strong1	St0	St1
5	(привязака шины к источнику) Pull Drive	pull0	pull1	Pu0	Pu1
4	(большая емкость соединения) Large Capacitive	large		La0	La1
3	(слабый сигнал — с выхода элемента) Weak Drive	weak0	weak1	We0	We1
2	(средняя емкость) Med. Capacitive	medium		Me0	Me1
1	(малая емкость соединения) Small Capacitive	small		Sm0	Sm1
0	(третье состояние) High Impedance	highz0	highz1	HiZ0	HiZ1

2. Литералы

2.1. Целые

Синтаксис: размер'основание код или размер'sоснование код

Размер — необязателен, задает разрядность числа. Если разрядность не указана, то по умолчанию число 32-разрядное.

'основание — необязателен. Определяет основание системы счисления (d — десятичное, b — двоичное, o — восьмеричное, h — шестнадцатеричное). По умолчанию основание десятичное.

s — признак знакового литерала (sb, sd, so, sk).

2.1.1. Целые литералы без указания числа разрядов (размера), но с системой кодировки (основанием)

Примеры:

17255, 'd17255 // десятичная система
'sd441 // десятичная система со знаком
'b10x1 // двоичная система
'sb1011 // двоичная система со знаком
'hAZX1 // шестнадцатиричная система
'sh1019 // шестнадцатиричная система со знаком

2.1.2. Целые литералы с указанием количества разрядов (размера) и системы кодировки (основанием)

Примеры изображения числа 57:
Двоичная система 8'b0011_1001 // символ _ используется для наглядности
Шестнадцатиричная 8'h39 // в векторе 8 разрядов
Восьмиричная 8'o071

Размер числа считается справа налево (от младших разрядов -lsb до старших- msb). Для беззнаковых литералов действуют правила дополнения до размера.

Когда указанный в описании размер числа меньше, чем фактическое значение, то старшие левые разряды обрезаются (3'hA равно 3'b010).

Когда указанный размер числа больше, чем фактическое значение и самый левый бит имеет значение 0 или 1, то все остальные старшие биты заполняются нулями (6'hA равно 6'b001010).

Когда указанный размер числа больше, чем фактическое значение и самый левый бит имеет значение Z или X, то все остальные старшие биты заполняются соответственно Z или X (6'bX равно 6'bXXXXXX).

Для чисел со знаком — дополнение знаком (6'shA равно 111010).

2.2. Вещественные (реальные — real) литералы

Синтаксис	
Значение значение	Запись с точкой
Основание E Степень Основание e Степень	Запись в экспоненциальной форме

Примеры записи числа ПИ: 3.14 .314E1.

3. Типы и виды данных

Тип определяет множество принимаемых значений данных, вид-свойства и множество операторов над данными. Имеется два основных вида данных: переменные и связи.

3.1. Вид переменная

Может быть разных типов — вектором, целым, вещественным, временем (reg, integer, real, time, realtime).

Ключевое слово описателя типа	Функциональные возможности
reg	Векетор любого размера
integer	32-разрядная переменная со знаком
time	64-разрядная переменная со знаком
real или realtime	Переменная с плавающей запятой

3.2. Вид соединение (цепь, связь)

Ключевое слово описателя подвида	Функциональные возможности
wire или tri	Простой провод (соединение, связь, цепь)
wor или trior	Монтажное ИЛИ
wand или triand	Монтажное И
tri0	Привязка к 0 в третьем состоянии
tri1	Привязка к 1 в третьем состоянии
supply0	Постоянный 0 (сила источника питания)
supply1	Постоянная единица (сила источника питания)
trireg	Сохранение последнего значения сигнала на шине при установлении ее драйверов в третье состояние

Этот вид данных описывает связи схемы (цепи, соединения).

Соединения передают и логические значения и логические веса сигналов.

Этот вид порта подразумевается по умолчанию в случае обьявления имени порта:

как выходного порта без указания типа reg.

как входного или двунаправленного порта.

Подвиды соединений: wire, wor, wand, trio и др.

3.3. Другие виды данных

Вид, тип	Функциональные возможности
parameter	Параметры (константы) служат для хранения целых, реальных чисел, времени, задержек или массивов строк кода ASCII. Параметры могут изменяться при каждой конкретизации модуля
specparam	Определяемая в блоке specify константа, для хранения целых чисел, реальных чисел, времени, задержек или массивов символов ASCII
event	Флаг события. Часто используемый для синхронизирования параллельных процессов в пределах модуля тип
localparam	(VERILOG-2000)-локальная константа блока specify
genvar	(VERILOG-2000)-локальная переменная целого типаблока generate

4. Объявление данных и область видимости

4.1. Объявление данных

Синтаксис
Тип_переменной signed размер *имя_переменной, имя_переменной*, ... ;
Тип_переменной signed размер *индентификатор_массива* размер_массива;
Тип_соединения signed размер *#(задержка) имя, имя1*, ... ;
Тип_соединения(вес) signed размер #(задержка) *имя* = продолжительность ;
trireg (мощностной вес) signed размер #(задержка, разброс времени) *имя, имя*, ... ;
parameter *имя_константы = значение, имя_константы = значение*, ... ;
specparam *имя_константы = значение, иям_константы = значение*, ... ;
event *имя_события, имя_события*, ... ;

Данные могут быть объявлены с описателями signed (необязательный параметр оператора объявления).

Размер (необязательный параметр) — это диапазон в квадратных скобках (от левой границы — старший бит к правой — младший бит) изменения индексов разрядов обьявленного данного.

Границы — должны быть целые числа, параметры целого типа или константное целое выражение. Максимальная длина векторов по крайней мере 65,536 бит (2^{16})..

размер_массива определяется содержанием скобок [*первый_адрес : последний_адрес*].

первый_адрес и *последний_адрес* должены быть целыми числами.
Максимальный размер массива — по крайней мере 16,777,216 слов (2^{24}).

Задержка (необязательный параметр синтаксической конструкции) может быть только на соединениях (связях, цепях).

Синтаксис — #задержка | #(задержка) |
#(з_фронта,з_среза) | #(з_фронта,з_среза,з_переключения) |
#(мин:тип:макс) -для всех переключений |
#(мин_з_фронта :тип_з_фронта:макс_з_фронта, мин_з_среза :тип_з_среза:макс_з_среза) |
#(мин_з_фронта :тип_з_фронта:макс_з_фронта, мин_з_среза :тип_з_среза:макс_з_среза, мин_з_перекл :тип_з_перекл:макс_з_перекл)
Где сокращение мин_з означает МИНимальная Задержка, тип_з — типовая.

Вес-сила сигнала (необязательный параметр) определен как (сила1, сила0) или (сила0, сила1). См. веса-силы сигналов.

Время_хранения (необязательный параметр) определяет интервал времени, в течение которого соединение типа trireg будет хранить старое состояние (после того, как все его драйверы отключатся).

Синтаксис конструкции разброс_времени — задержка фронта, среза, время_хранения.

Примеры описаний данных

Примеры объявлений данных	Описание
wire a1, b1, c1;	3 скалярных соединения
tri1 [9:0] d;	10-разрядное соединение — (провод, шина), привязанная к источнику 1 при переходе в третье состояние
reg [9.0] r;	10-разрядная переменная без знака (вектор)
reg [7:0] RAM [0:2048];	Переменная-одномерный массив векторов (память); 8-разрядный, с 2 К элементов
wire #(5.1,3.6) c;	Соединение (цепь) с разной задержкой фронта и среза
wire (strong1,pull0) s = a+b+c;	Соединение с непрерывным присваиванием в него a+b+c и сильным сигналом 1
trireg (small) #(0,0,5) r_bit;	Соединение (цепь) с малой емкостью и удержанием сигнала 1 в течение 5 единиц времени

reg signed [5:0] R4 ;
reg [7:0] ee[0:4095][0:2];
reg [5:0] R3= 6'b111111
	//переменная R4 со знаковым разрядом
	//переменная ee — двумерный массив 4096*3*8 - VERILOG-2000,
	//переменная R3 — задано начальное значение VERILOG-2000,

integer a,b; real r;	целые переменные a, b вещественное r

4.2. Область видимости данных

VERILOG имеет четыре вида пространств видимости данных

4.2.1. Глобальные имена

Видимы везде. Это имена модулей, примитивов, конфигураций и макросов (`define).

4.2.2. Локальные имена

Видимы в пределах своей области определения (если не использовать иерархических имен). Области: модуль, обьявление функции, обьявление процедуры, поименованная группа (begin-end fork-join).

Имена обьявленные в блоке specify действуют в нем.

Атрибуты (VERILOG-2000) действуют в следующем за их обьявлением обьекте.

5. Операции

Арифметические

Название	Пример	Результат
Сложение	2+3	5
Вычитание	2-3	-1
Умножение	2*3	6
Деление цел.	2/3	0
Модуль	2 % 3	2
Степень (VERILOG-2000)	2**3	8

Унарные арифметические

Название	Пример	Результат
Плюс	+1	1
Минус	-1	-1
Минус	-4'b1011	4'b0101
(4'b0101 — дополнительный код)		

Логические поразрядные

Операция	VERILOG	Результат
Поразрядное И	4'b10x1 & 4'b0010	4'b00x0
Поразрядное ИЛИ	4'b10x1 | 4'b0010	4'b1011
Отрицание	~ 4'b10x1	4'b01x0
Исключающее ИЛИ	4'b10x1 ^ 4'b0010	4'b10x1
Исключающее НЕ ИЛИ	4'b10x1~^4'b0010	4'b01x0

Логические операции

Логическое И	4'b1011 && 4'b0010	1'b1
Логическое ИЛИ	4'b1011 || 4'b0010	1'b1
Логическое отрицание	! 4'b1011	1'b0

Унарные операции редукции

И всех разрядов	&4'b1101	1'b0
ИЛИ всех разрядов	| 4'b1101	1'b1
ИЛИ-НЕ всех разрядов	~| 4'b1101	1'b0
И-НЕ всех разрядов	~&4'b1101	1'b1
Исключающее ИЛИ всех разрядов	^4'b1101	1'b1
Исключающее НЕ ИЛИ всех разрядов	~^4'b1101	1'b0

Операции сравнения

Равенство	4'b1011 == 4'b0010	1'b0
Равенство чисел	5 == 6	1'b0
Неравенство	4'b1011 != 4'b0010	1'b1

Операции отношения

Больше	4'b1011 > 4'b0010	1'b1
Меньше	4'b1011 < 4'b0010	1'b0
Больше-равно	4'b1011 >= 4'b0010	1'b1
Меньше-равно	5 <= 5	1'b1

Операции идентичности

Идентично	4'b1011 === 4'b0010	1'b0
Неидентично	4'b1011 !== 4'b0010	1'b1

Сдвиговые операции

Операция	Пример	Результат
Сдвиг влево	4'b1011 << 2	4'b1100
Сдвиг вправо	4'b1011 >> 2	4'b0010
*Сдвиг влево арифметический	4'sb1011 <<< 2	4'sb1100
*Сдвиг вправо арифметический	4'sb1011 >>> 2	4'sb1110
VERILOG-2000 имеет операции <<< и >>>		

Разные

Конкатенация (сцепление)	{3'b100,2'b11}	5'b10011
Реплика-повтор	{3{2'b01}}	6'b010101

Приоритет операций — как и в обычных языках программирования — наибольший у унарных, приоритет ИЛИ меньше, чем у И, но во избежание ошибок лучше пользоваться скобками для указания порядка вычислений.

Пример: (((a+b)*c)/(m-n))

Приоритетность операций	
Одноместные ! ~ + -	Высший приоритет
* / %	
**	
+ -	
<< >> >>> <<<	
< <= > >=	
== != === !==	
& ~&	
^ ~^	
| ~|	
&&	
||	
? :	Низший приоритет

6. Выражения

6.1. Обычные выражения

Строятся из литералов, имен, операций и скобок.

Пример: (F+H)* (C-D)/2 +M[I] & ~ N[2 : 1]
4'b1011 & 1'b1

6.2. Выражения с полями векторов и массивов

VERILOG не допускает работу с полями переменной длины. Однако его версия VERILOG-2000 позволяет использовать переменные индексы полей фиксированной длины.

Пример: reg [32:0] A;reg [3:0] B; reg [7:0] C; C=A[B*8 +: 8];

Помимо обычных выражений допускаются задержанные.

6.3. Задержанные выражения

Используется в операторах процедурного присваивания и является его частью. Сначала оно вычисляется и через время задержки присваивается.
Пример: S<=#(TD) A+B; D=#(T2)C;
Подробнее вопросы задержек рассмотрены ниже.

6.4. Условные выражения

VERILOG допускает также условные выражения- их значение определяется условием. Например выражение (A>B) ? 3'b100:3'b111 дает результат 3'b100 если A>B истино.

Пример: x=(a>b+1)? c: (v && L)?d-1: 0;// в x присваивается условное выражение.

6.5. Выделение разрядов и полей

Примеры:

reg A[6:70];	A[3]=A[4];// массив A, элемент с индексом 4
reg [0:100]B;	B[7:40]=0;// вектор B, разряды 7:40 устанавливаются в ноль

6.6. Выделение элементов массивов

Примеры:

reg A1[6:70];	A1[3]=A1[4];//одномерный массив однобитовых слов
reg [0:100]B1[0:4096]	B1[7]=0;// одномерный массив 101-битовых слов
reg C1[0:3][0:100]	C1[8][4]=1;//VERILOG-2000 двумерный массив

7. Последовательные операторы (процедурные)

Последовательные операторы могут употребляться в функциях, процедурах, блоках. Группа — см. ниже, это один оператор или несколько операторов в операторных скобках begin end или fork join.

7.1. Оператор ожидания

Вариант	*Пример*
ожидание условия	wait A1==B1;
ожидания интервала времени	#10;
ожидание события	@(A or B);
	@(A,B);// VERILOG-2000
	@negedge(C);

Если символ ; отсутствует, то ожидание становится не оператором, а условием начала выполнения следующего оператора.

Примеры эквивалентов:

#4; A=B; @negedge(C);M=N; #4 A=B; @negedge(C) M=N;

7.2. Оператор присваивания переменной

Неблокирующее(<=) присваивание в переменную

Синтаксис: переменная = ожидание выражение;

Примеры: a <= b;v[2:3] <= #2 d[1:2];{v,n} <= c+d;@(posedge clk)x<=Y; (До оператора @(posedge clk)x<=Y; операторы-предшественники исполнятся мгновенно, но значения переменных a, v, n изменятся только через дельта-интервал модельного времени, а переменной v[2:3] — лишь когда пройдет 2 интервала времени.

Блокирующее (=) присваивание в переменную

Синтаксис: переменная <= ожидание выражение;

Примеры: a = d; v[2:3]=#5 d[1:2];{v,n}= c+d;@(posedge clk)x<=Y; (пока не пройдет 5 интервалов модельного времени оператор {v,n}= c+d; не начнет исполняться).

7.3. Условный оператор

if(выражение) оператор или группа операторов
else оператор или группа операторов
(часть else необязательна)

7.4. Оператор выбора

Обычный case

case (выражение)
выбор1: оператор или группа операторов
выбор2,
выбор3 : оператор или группа операторов
default: *оператор или группа операторов*
endcase

Специальный вид case-casez

Использует значение Z для заполнения не используемых битов (разрядов) в условии выбора.

casez (выражение)
выбор1: оператор или группа операторов
выбор2,
выбор3: оператор или группа операторов
default: *оператор или группа операторов*
endcase

Специальный вид case-casex

Использует Z или X значение для заполнения неиспользуемых разрядов.

casex (выражение) *выбор1: оператор или группа операторов*
выбор2, выбор3 : оператор или группа операторов
default: *оператор или группа операторов*
endcase

7.5. Оператор цикла

Имеет четыре варианта.

1. forever оператор или группа_операторов
Бесконечный цикл выполнения оператора или группы операторов.

2. repeat (*выражение*) оператор или группа операторов
Количество выполнений цикла может быть целым числом, переменной и выражением (значение выражения вычисляется при первом прогоне цикла)

3. while (*выражение*) оператор или группа_операторов
Цикл выполняется до тех пор, пока выражение истинно.

4. for (*начальное_назначение; условие_продолжения; приращение*) оператор и группа_операторов
Выполняет присваивание начального значения параметру цикла.
Выполняет оператор или группу операторов тела цикла, пока *условие_продолжения* истинное.
Выполняет приращение параметра в конце каждого прохода цикла.

7.6. Оператор выхода из группы

disable *имя_группы;*
Прекращает выполнение названной группы операторов.

7.7. Примеры последовательных операторов

Примеры использования процедурных (последовательных) операторов
// тактовый генератор clk с периодом 10, который начинает работать после 100 единиц времени initial begin clk = 0,#100 forever #5 clk = ~clk;end

Примеры использования процедурных (последовательных) операторов

```
//описание последовательностной схемы - два регистра
always @(posedge clk)
  begin word[15:8]= word[7:0];v<=v+2;
  end
```

```
// комбинационная схема с сумматором и умножителем
always @(a or b or sel) if (sel==0) y = a + b;else y = a * b;
//ниже то же используя возможности VERILOG-2000
always @* if (sel==0) y = a + b;else y = a * b;
```

```
//ниже фрагмент описания управления процессора и памяти
always @(posedge clk)
 begin
  casez (opcode) //? -синоним Z
   3'b1??: data_bus = accum;
   3'b000: repeat (5) @(posedge clk)
            begin //задержка 5 тактов, потом запись в память
              RAM[address] <= data_bus;address <= address + 1;
            end
   3'b011: begin : load //поименованная группа
            integer i; // i -локальная переменная
            for (i=0; i<2; i=i+1)//две выборки из памяти
              @(negedge clk) data_bus = RAM[i];
            end
   default: $display("неверный код");
 endcase
end
```

8. Процедурные блоки

Процедурные блоки представляют собой параллельные операторы.
Синтаксис:

```
Тип_блока @(список_чувствительности)
  Тип группы: имя_группы
  Обьявление локальных перменных
  Последовательные операторы
Конец_обьявления_группы
```

Тип_блока может быть initial или always

initial — процесс выполнения операторов в этом блоке однократен.

always — процесс выполнения операторор в процедурном блоке может повторяться неоднократно.

список_чувствительности — необязательный параметр.

Синтаксис: элемент_списка1 or элемент_списка2,: // or разделяет элементы или

элемент_списка1, элемент_списка2, : //VERILOG-2000

VERILOG-2000 также позволяет не перечислять имена списка чувствительности символа обобщенного списка чувствительности @*.

Примеры процедурных блоков и групп	Пояснение
initial fork b = 0; #10 b = 1,#20 a = 16'hввАА; join	Блок inital выполняется однократно; fork--join группа выполняет операторы параллельно

always @(a or b or ci)
begin
sum = a + b + ci;
end
	always выполняет операторы многократно
	эквивалентные заголовки блока примера
	always @(a , b , ci)//VERILOG-2000
	always @* //VERILOG-2000

always @(posedge clk) q <= d;	Группа не требуется, когда в блоке всего один оператор

9. Группы операторов

Понятие группы — (составного оператора) используется для обьединения двух или более операторов в один

begin end. — обычная группа(составной оператор)

fork join — параллельная группа. Она содержит две или более параллельных ветвей. Ее исполнение заканчивается с окончанием наиболее медленной из ветвей.

Имя_группы (необязательный параметр) создает возможность обьявлений локальных переменных в группе. Досрочный выход из группы возможен с помощью оператора disable.

Объявление_локальных_переменных — необязательный параметр.

10. Процедуры и функции

10.1. Процедуры (task)

Синтаксис

task *имя*; // В VERILOG-2000 для рекурсивных процедур
// используют ключ automatic. Синтаксис: task automatic *имя*;
обьявления входных-выходных данных
обьявления локальных переменных
оператор или группа операторов
endtask

Процедуры (подпрограммы) могут включать операторы ожидания (#, @, wait).

Пример подпрограммы task	

task read_mem;
input [15.0] a;
input clk;input [31.0]del;
output [31.0] d;
begin
@posedge clk;
#(del)d<=mem[a];
end
endtask
	// Заголовок процедуры
	// Обьявление аргументов
	
	// Чтение из массива

read_mem(PC,tact,25, IR);	// Вызов процедуры

10.2. Функции function

Синтаксис
function [тип или размер] *имя_функции*; //VERILOG-2000 для рекурсивных функций используют ключ automatic *имя*; *Объявление входных параметров* *Объявление локальных параметров* *Последовательный оператор или группа* endfunction

Функция возвращает значение, присвоенное ее имени.
Имеет как минимум один входной параметр (input) и не имеет output, inout.
Тело функции не содержит операторов ожидания (#, @, wait) и параллельных.
[тип или размер] — необязательный параметр, тип-integer или real.
По умолчанию размер возвращаемого значения — 1 бит.

Пример

function [7:0] GetByte; //Заголовок, размер результата 8 бит
input [63:0] Word;
input [3:0] ByteNum;
integer Bit;
reg [7:0] temp; begin
for (Bit=0; Bit<=7; Bit=Bit+1)
temp[Bit] = Word[((ByteNum-1)*8)+Bit];
GetByte = temp;
end
endfunction

this_byte = GetByte(data,4); //Вызов функции

Системные функции:
$time — возвращает текущее модельное время;
$random — возвращает случайное число.

11. Параллельные операторы и блоки

Название, синтаксис *Пример*

1. Блок always (процесс)

```
always @(a or b)begin A:
x= a[2:0]+b+1;y<=#(V) x;
end
```

2. Блок initial (Однократный Процесс)

```
initial begin
x= 1; b=0 ;
end
```

3. Оператор непрерывного присваивания соединению (цепь)

Явный:
assign#(задержка)
имя=выражение;

```
wire [2:0] w; assign w= a & b;
```

Неявный:
подвид (сила) размер
#(задержка) имя=
выражение;

```
wire w1= ~m[1];//неявный
wire (strong0)[7:0]w2= c+d-r;//неявныи
//в шину с сильным нулем
```

4. Оператор конкретизации модуля-компонента

```
k1 M1 (.x1(y1),.p(e2));
k1 M2 (y2,e5);//позиционное соответствие
y2 n2 #(4,6)(c1,c3,v4);//передаются
                        //параметры 4 и 6
```

5. Оператор конкретизации примитива

```
and #(4,5) (y1,x1,x2);
xor #7 (y3,y1,x4,x5,x7);
```

6. Системный оператор мониторинга событий

$monitor(" текст с элементами форматирования ", список_сигналов);
Срабатывает печать при событии в любом сигнале из списка

7. Операторы ветвей группы fork (см. выше)

12. Модуль проекта

12.1. Стиль описания VERILOG-95

Структура модуля

Заголовок

```
  module <имя> (<список_портов>); // список_портов может быть пустым
```

Описания портов

```
  <описание направленности и размера портов>;
  <описание выходных портов вида переменная>;
```

Описания внутренних обьектов // может отсутствовать

```
  <описание внутренних соединений и переменных> ;
  <описание процедур и функций>
```

Тело модуля

```
  <описания параллельных процессов >
  <блок specify>  // может отсутствовать
endmodule
```

Пример заголовка модуля:

```
module M1(clk,rst_n,D,bus,Q);
  input clk,rst_n;                    // 1 битовые входы вида соединение
  input [7:0]D;                       // 8 разрядный вход вида соединение
  inout [7:0] bus;                    // двунаправленный вида соединение
  output signed [7:0] Q; reg[7:0] Q;  // выход Q имеет вид переменная
```

12.2. Стиль VERILOG-2000 (ANSI-C)

```
module имя
  #(объявление_параметров)
  (обьявления_портов);
  тело-модуля
endmodule
```

В частности VERILOG-2000 позволяет совместить описания направления портов, их вида и размера (стиль ANSI-C).

Пример заголовка модуля:

```
module M1(input clk,input rst_n,input [7:0] D,
                inout [7:0 ] bus, output reg signed [7:0]Q );
```

Пример описания внутренних соединений, переменных и функции:

```
wire A,B,C;                  //1 битовые внутренние соединения-сигналы
reg [1:0] R1,R2;             //внутренние двухразрядные переменные-reg
integer I;                   //целая переменная 32 разряда
reg [7:0] mem[0:4095];       //массив 4096 *8
function [8:0] F1;           //обьявлена функция F1
      input x1;input [7:0]x2;reg [8:0]r1;
      begin r1=x2+x1;F1=r1;end
endfunction                  //F1
```

13. Конкретизация — вызов экземпляров модулей

13.1. VERILOG-95

Синтаксис
Вызов с позиционным соответствием сигнал-порт *имя_модуля* имя_конкретизации диапазон (*список_сигналов*);
Вызов с поименованным соответствием (ключевым) сигнал-порт *имя_модуля* имя_конкретизации диапазон_конкретизаций (*.имя_порта(сигнал)*, (*.имя_порта(сигнал)*, ...);
Поименованное (ключевое) переопределение параметров модуля defparam *путь* имя параметра = значение;
Позиционное переопределение параметров модуля *имя_модуля* #(*список значений*) имя_конкретизации (*список_сигналов*);

13.2. VERILOG-2000

VERILOG-2000 допускает более удобную форму поименованной (ключевой) передачи параметров настройки.

Синтаксис:

имя_модуля #(.имя параметра(значение),...) имя_конкретизации (список_сигналов)

Пример: M2 #(.TDEL(15)) K1 (A, ,S_TMP);

Неиспользованные порты обозначаются запятыми в позиционном сопоставлении.

диапазон_конкретизаций — [лев_индекс :прав_индекс] — необязательный аргумент, который указывает количество конкретизаций, каждая из которых объединяется с определенным разрядом.

Пример: vv [7:0] (Y, x1, x2);

Пример

module reg5 (q, d, clock);
output [4:0] q;input [4:0] d;input clock;
 	
dff c1 (q[0], , d[0], clock);
//позиционное соответствие порт-сигнал
dff c2 (.clk(clock),.q(q[1]),.data(d[1]));
//поименованное соответствие порт-сигнал
dff c3 (q[2], ,d[2], clock);
defparam c3.del = 3.2;
//явное переопределение параметров
dff #(4) c4 (q[3], , d[3], clock);
//позиционное переопределение параметров
dff #(.del(6)) c5 (q[4], d[5], clock);
//явное переопределение параметров в стиле VERILOG-2000
endmodule

module dff (q, q_n, data, clk);
output q, q_n;input data, clk;parameter del=1;
always @(posedge clk) q<=#1 data;q_n<=#1 ~data;
endmodule

14. Системные операторы и функции

Их имя начинается со знака $. Системный оператор реализует одно из встроенных системных действий — например вывод. Пользователь может расширять список системных операторов, используя механизм интерфейса PLI. Некоторые из системных операторов приведены ниже.

14.1. Форматируемый ввод-вывод

Коды форматов			
%b	Двоичный формат	%s	Символьный
%o	Восьмеричный	%m	Составное имя
%d	Десятичный	\t	Табуляция
%h	Шестнадцатиричный	\n	Новая строка
%e	Экспоненциальный	\"	Печатать кавычки
%f	Форма с точкой	\\	Обратная косая
%t	Время	%%	Печатать процент
0 перед форматом означает отбрасывание нулей слева			

Печать с новой строки.

$display("текст с символами форматирования", аргумент, аргумент, ...);

а также операторы $displayb,$displayc,$displayh.

Печать не с новой строки.

$write("*текст с символами форматирования* ", *аргумент*, *аргумент*, ...);

а также операторы $writeb,$writec,$writeh.

Печать после окончания переходных процессов

$strobe("*текст с символами форматирования* ", *аргумент, аргумент,* ...);

Работает как $display, но только после окончания всех событий на данном интервале времени.

а также операторы $strobeb,$strobec,$strobeh.

Мониторинг событий в сигналах

$monitor("текст с символами форматирования ", аргумент, аргумент, ...);

Отражает процесс отслеживания событий и печати значений сигналов

а также операторы $monitorb,$monitorc,$monitorh.

14.2. Работа с файлами

Открытие файла

переменная = $fopen("*имя_файла*");
переменная = $fopen("*имя_файла*",тип);

Закрытие файла

$fclose(переменная);

Переменная обьявляется как integer и является многоканальным дескриптором с установкой одного бита для каждого файла.

Действия с нестандартными файлами

$fmonitor(переменная, "текст_с_символами_форматирования",сигнал,сигнал, .),
$fdisplay(переменная, "текст_с_символами_форматирования",сигнал,сигнал, ...);
$fwrite(переменная, "текст_с_символами_форматирования",сигнал,сигнал, ...);
$fstrobe(переменная, "текст_с_символами_форматирования",сигнал,сигнал, ...);

Чтение файлов в массив

$readmemb("*имя_файла*", массив, *начальный_адрес, конечный*);
//чтение двоичных кодов
$readmemh("*имя_файла*", массив, *начальный_адрес, конечный*);
// чтение шестнадцатиричных кодов

Останов

$finish; //конец моделировния
$stop: //приостановка моделирования-можно его продолжить,
// нажав клавишу ВВОД

Прочие системные процедуры и функции:

$fmonitor, $stime, $realtime, $timeformat, $print

Новые системные функции и процедуры VERILOG-2000

$ferror, $fgetc, $fgets, $fflush, $fread, $fscanf, $fseek, $fsscanf, $ftel,$rewind, $sformat, $swrite, $swriteb, $swriteh, $swriteo, $ungetc.

Системные функции и процедуры контроля временных соотношений сигналов см. в следующем разделе.

15. Блок спецификаций временных соотношений-specify (несинтезабельная конструкция)

Синтаксис
specify объявление спецпараметров контроль временных параметров простые задержки путей контакт-контакт задержки путей зависящие от фронтов задержки путей зависящие от состояний endspecify

15.1. Обьявление спецпараметров

specparam *имя_параметра = значение, ...*;

Используются для задания задержек и других параметров.
Значение может быть строкой, целым, вещественным, временем.
VERILOG-2000 допускает также описатель localparam для обьявления внутренних констант.

local param *имя_параметра = значение, ...*;

15.2. Средства проверки временных соотношений

Контроль времени	*Системный оператор*
предустановки	$setup
удержания	$hold
и того и другого	$setuphold
разброса	$skew
восстановления	$recovery
периода	$period
длительности	$width

Синтаксис
$setup(проверяемый_сигнал, проверяющий, время_предустановки, фиксатор); $hold(проверяющий, проверяемый_сигнал, время_предустановки, фиксатор); $setuphold(проверяющий, проверяемый_сигнал, время_предустановки, время_удержания, фиксатор); $skew(проверяющий, проверяемый_сигнал, время_разброса, фиксатор); $recovery(проверяемый_сигнал, проверяющий, время, фиксатор); $period(проверяемый_сигнал, время, фиксатор); $width(проверяемый_сигнал, предел_длительности, порог, фиксатор);

Контролирующие системные операторы можно употреблять только в блоке specify.

Проверяемый и проверяющий сигналы *должны быть портами модуля* (проверяющий — обычно тактовый сигнал).

Предел_длительности и порог — временные параметры

Фиксатор (необязательный параметр) — переменная, используемая как флаг возникновения нарушений.

15.3. Задержки путей распространения сигналов

Задержка простого пути (простая_задержка):

(*имя_входного_порта* полярность:оперция_пути *выходной_порт*) = (задержка);

Задержка пути, зависящая от фронтов сигналов (событийная_задержка):

(событие *входной_порт* операция_пути(*выходной_порт* полярность:источник)) = (задержка);

событие(необязательный параметр) — posedge или negedge. Если оно не указано, предполагается любой перепад.

источник(необязательный параметр).

Задержка, зависящая от состояния схемы:

if (первое_условие) простая_задержка | событийная_задержка

if (следующее_условие) простая_задержка | событийная_задержка

ifnone простая_задержка

В условии могут использоваться только имена входов модуля.

Для вектора следует использовать только его младший разряд.

Для различных задержек того же самого пути необходимо задавать разные условия или события.

ifnone (необязательно).

полярность (необязательное) + или -, указывает будет ли инвертирован входной сигнал.

Операция_пути

*> означает полная_связь

=> означает параллельная_связь.

полная_связь — любой разряд входного сигнала может иметь отдельный путь к любому разряду выхода.

параллельная_связь — разряд в разряд (0 - 0, 1 - 1, ...).

Задержка может состоять из 1, 2, 3, 6 или 12 значений времен переключения.

Время переключения может быть одним числом или тройкой — min:typ:max .

Число задержек	Пояснения
1	Одна задержка для всех переключений выхода
2	Разные задержи фронта и среза выхода
3	Задержки фронта, среза, переключения
6	Задержки фронта, среза, 0->Z, Z->1, 1->Z, Z->0
12	Задержки фронта, среза, 0->Z, Z->1, 1->Z, Z->0, 0->X, X->1, 1->X, X->0, X->Z, Z->X

Примеры	Пояснения
(a => b) = 1.8;	Параллельная связь с одинаковыми задержками переключения 1->0 и 0->1
(a -*> b) = 2:3:4;	Полная связь с разбросом задержек min:typ:max b принимает инверсное значение a
specparam t1 = 3:4:6, t2 = 2:3:4; (a => y) = (t1,t2);	Разные задержки с разбросом для фронта и среза
(a *> y1,y2) = (2,3,4,3,4,3);	6 разных задержек: фронта, среза, 0->Z, Z->1, 1->Z, Z->0
(posedge clk => (qb -: d)) = (2.6, 1.8);	Задержка фронта и среза зависит от фронтов
if (rst && pst) (posedge clk=>(q +: d))=2;	Задержа зависит от состояния и фронтов

16. Синтезабельные конструкции

Список этих конструкции для разных систем синтеза может несколько отличаться.

В новых разработках систем предполагается поддержка нововведений стандарта VERILOG-2000 (эти строки таблицы обозначены звездочкой).

Конструкция	Пояснения
Обьявление модуля	*
Обьявление портов input output inout	*
Соединения типа wire wand wor supply0 supply1	
Переменные типа reg, integer	Длина integer 32 bits
Параметры	Только целые. Может не поддерживаться переопределение параметров; * параметры с указанным размером
Конкретизация модуля	
Встроенные Примитивы and nand or nor xor buf not bufif1 bufif0 notif1 notif0	
Непрерывное присваивание assign	Явная и неявная формы
Непрерывное процедурное присваивание assign	
Функции-function	* рекурсивные со статически ограниченным числом рекурсий
Процедуры-task	
Блоки always	Обязаны иметь список чувствительности, * дополнительно использует ограничители @* и ,
Группы begin--end	

Конструкция	Пояснения
Оператор disable	
Последовательные операторы присваивания = <=	
Операторы if if-else case casex casez	Значения X и Z воспринимаются в смысле только как безразличные
Цикл for	
Циклы while forever	В теле цикла должен быть оператор задержки на такт @(posedge clk) или @(negedge clk))
Операции & ~& | ~| ^ ^~ ~^ == != < > <= => ! && || << >> {} {{}} ? · + - * /	Не поддерживаются операции === и!==, а в некоторых системах также || и && *-VERILOG_2000 — дополнительно <<< >>> и **
Работа с разрядами и полями векторов	В левой части присваиваний границы полей векторов могут быть только константами *

17. Директивы компиляции (перечислены только основные)

В основном это директивы препроцессора ,который обрабатывает текст модели перед компиляцией,частично это директивы управления процессом моделирования.

Их имена начинаются с символа обратной кавычи (`) — не следует путать его с обычной кавычкой ('). Это не операторы — не надо ставить после них символ ;. Область их действя не ограничивается границами модулей, а простирается до другой директивы, отменяющей данную.

Директива отмены директив

`reset_all

Отменяет все предыдущие директивы — восстанавливает значения директив по умолчанию.

Директива масштаба модельного времени

`timescale единица времени / точность задания

единица времени может иметь значения: 1 10 или 100.
единица измеряется в: s — секундах, ms — милисекундах и т. д.(us, ns, ps, fs).

Пример: `timescale 1 ns / 100 ps — задана единица модельного времени -1 наносекунда с точностью одна десятая.

Директива описания макроса

`define *имя_макроса строка*

`define *имя_макроса(аргумент) строка (аргумент)*

строка замещает имя макроса после препроцессорной обработки макровызова
строка-строка текста завершающегося символом конца строки.
имя_макроса в месте его вызова начинается с символа обратной кавычки.
Комментарий не входит в состав строки.

Примеры:

```
`define ONE 1     //обозначение 1
 a<= `ONE;      // после препроцессора будет a<=1;
 `define NOR(dval) nor #(dval)
 `NOR(3) or1 (y,a,b);/  /после получим nor #3 or1(y,a,b)
 `NOR(3:4:5) or2 (o,c,d);
```

Директива отмены макроопределения

`undef имя_макроса

Пример:

`undef ONE

Директива включения файла

`include "имя_файла"

Вставляет в данное место текст другого Verilog-файла. Имя файла может включать путь в каталогах системы.

Директива условной компиляции

```
`ifdef имя_макроса
    строки VERILOG-кода
`else
    строки VERILOG-кода
`endif
```

После препроцессорной обработки остается только одна из альтернативных групп строк

Пример:

Если было объявление 'define RTL, останется wire y = a | b;

```
`ifdef RTL
   wire y = a | b;
`else
   or (y,a,b);
`endif
```

Директива задания типа умалчиваемого соединения default (по умолчанию это тип wire)

`default_тип_соединения тип_соединения

Пример: `default_wand wor

Директива задания значения входа unconnected_drive при отсутствующей связи

Указывает значение, подаваемое на незадействованный вход модуля при его конкретизации.

Пример:

`unconnected_drive pull1 // незадействованные порты подключены к 1

Директива задания режима моделирования задержек delay_mode

```
`delay_mode_zero    //без задержек, даже если в тексте они есть
`delay_mode_unit    //все задержки равны 1
`delay_mode_path
`delay_mode_distributed
```

Директива задания библиотеки модулей uselib

`uselib file=*путь_в_каталоге/ имя_файла.*
`uselib dir=путь /libext=расширение

Пример:

```
`uselib file=/cypress/rtl_lib
   ALU i1 (y1,a,b,op);     //RTL-модель
`uselib dir=/amd/gate_lib libext=.v
   ALU i2 (y2,a,b,op);     //Gate-модель
`uselib //отключение поиска модулей в `uselib
```

Используемые сокращения

ANSI	American National Standardization Institute (американский институт стандартизации)
ASCII	American Standard Code for Interchange Information (стандартный американский код обмена информацией)
ASIC	Application Specific Integrated Circuit (заказная БИС)
CAD	Computer Aided Design (автоматизированное проектирование)
CMOS	Complementary MOS (комплементарный МОП — КМОП)
EDA	Electronics Design Automation (автоматизация проектирования электронной аппаратуры)
EDIF	Electronic Data Interchange Format (стандартный формат обмена данными в САПР конструкторского этапа проектирования электронной аппаратуры)
EOF	End Of File (символ конца файла)
FIFO	First In First Out (очередь)
FPGA	Field Programmable Gate Array (программируемый вентильный массив)
HDL	Hardware Description Language (язык описания аппаратуры)
IEEE	Institute of Electrical Engineers and Electronics (институт электро и электронных инженеров США)
IP	Intellectual Property (интеллектуальная собственность)
LSB	Least Significant Bit (младший значащий разряд)
LUT	Look Up Table (таблица, реализующая логическую функцию — настраиваемый элемент ПЛИС)
MOS	Metal Oxide semiconductor (метал — окисел — полупроводник или МОП)
MSB	Most Significant Bit (старший значащий разряд)

PLA	Programmable Logic Array (программируемый логический массив)
PLI	Programming Language Interface (интерфейс с языками программирования)
RAM	Random Asses Memory (память с произвольным доступом)
RCS	Revision Control System (система ведения проектной базы данных)
ROM	Read Only Memory (постоянное запоминающее устройство — ПЗУ)
RTL	Register Transfer Level (уровень регистровых передач)
SDF	Standard Delay Format (стандартный формат представления данных о задержках схем)
SoC	System on Chip (система, реализованная на одном кристалле)
VHDL	VHSIC Description Language (язык описания сверхбыстродействующих интегральных схем)
VHSIC	Very High Speed Integrated Circuit (сверхбыстродействующая интегральная схема)
БИС	Большая Интегральная Схема (схема высокого уровня интеграции)
БМК	Базовый Матричный Кристалл
МЭИ	Московский Энергетический институт
ПЛИС	Программируемая Логическая Интегральная Схема
ПЛМ	Программируемая Логическая матрица
ППП	Пакет Прикладных Программ
САПР	Система Автоматизации Проектирования
СБИС	Сверх Большая Интегральная Схема (схема сверхВысокого уровня интеграции)
ЭВМ	Электронная Вычислительная Машина

Интернет-ресурсы

Журналы

www.isdmag.com — статьи по САПР и методологии проектирования БИС
www.dacafe.com — новости САПР, конференции, работа
www.eda.com — новости САПР

Ответы на часто задаваемые вопросы, литература, web-ссылки

www.parmita.cjv/verilog/faq — новости VERILOG
www.parmita.cjv/vhdl/faq — новости VHDL
www.angelfire.cjv/in/rajesh52/
www.verilog.net/docs

Пособия, справочники

www.sutherland-hdl.com

Сайты ведущих фирм с методическими материалами и текстами моделей

Компоненты, память, FIFO

www.cypress.com
www.micron.com
www.idt.com
www.intel.com
www.denaly.com
www.samsung.com
www.fmf.org

САПР ПЛИС (FPGA, PLD, CPLD), модели, методики, описания компонент

www.xilinx.com
www.altera.com

Ведущие производители ПО САПР

www.cadence.com
www.synopsys.com
www.mentor.com

Литература

1. IEEE Standard VHDL Language Reference Manual (IEEE std 1076-1993) The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, NewYork, NY 10017-2394, USA, June 1994.

2. IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the VerilogR Hardware Description Language. The Institute of Electrical and Electronics Engineers, Inc.345 East 47th Street, New York, NY 10017-2394, USA. ISBN 1-55937-727-5.

3. Армстронг Дж. Р. Моделирование цифровых систем на языке VHDL. —М.: Мир, 1992. 175 с.

4. Берже Ж. и др. VHDL-92 новые свойства языка описания аппаратуры. —М.: Радио и связь,1995. 216 с.

5. Lipsett R., Chaefer C. F., Ussery C. VHDL: hardware description and design. —M. A.: Kluver Acad. Publisher, 1989. 320 p.

6. ГОСТ Р-50754-95. Язык Описания Аппаратуры Цифровых Систем VHDL. Описание Языка. —М.: Госстандарт России, 244 с. Этот Гост соответствует версии VHDL-87, т. е. IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-1987, VHDL version 7.2.). —N. Y.: IEEE, 1988.

7. VHDL для моделирования,синтеза и формальной верификации аппаратуры/Пер с англ. —М.: Радио и связь, 1995. 360 с.

8. Поляков А. К. Моделирование ЭВМ на языке VHDL. —М.: Моск. энерг. ин-т, 1994. 80 с.

9. Бибило П. Н. Синтез логических схем с использованием языка VHDL. —М.: Солон-Р, 2002. 384 с.

10. Поляков А. К. Моделирование ЭВМ на ЭВМ. —М.: Моск. энерг. ин-т, 1981. 104 с.

11. Разевиг В. Д. Система проектирования цифровых устройств OR-CAD. —М.:, Солон-Р, 2000. 516 с.

12. IEEE standard VHDL 1076.1-1999, Analog and Mixed Signal Extention, -N. Y.: IEEE, 1999.

13. IEEE std 1164.1: Multi_Value Logic System Interface Package.

14. Standard VITAL ASIC Modeling Specification.-NY.: IEEE P1076.4 Feb. 2000.

15. IEEE Std P1364-2001, IEEE Standard Hardware Description Language Based on the VerilogR Hardware Description Language. The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street,- New York, NY 10017-2394, USA.

16. Lee James . VERILOG — QUICKSTART. Kluwer Acad. Publisher, 1999. 324 p.

17. Yatin Trivedy and others Didital Design and Synthesis with VERILOG HDL,Automata Publishing Company,-San Jose,CA, 1993. 376 p.

18. Smith M. Application Specific Integrated Circuits, Addison Wesley Pub. 1997. 1040 p. (ее электронный вариант доступен на сервере www-ee.eng.hawaii.edu/~msmith, на сервере www.dacafee.com и др.)

19. Keating M., Bricard P. REUSE Methodology Manual for System on Chip Design, Kluwer Acad. Pub. 1999. 284 p.

20. Bening L., Foster H. Principle of Verifiable RTL Design. Kluwer Acad Pub. 2000. 300 p.

21. Баричев С. Г., Гончаров В. В., Серов. Основы современной Криптографии. Изд-во Горячая линия-Телеком, 2001. 118 с.

22. Кнышев Д. А., Кузьмин М. О. ПЛИС фирмы XILINX. —М .: изд-во ДОДЕКА, 2001. 238 с.

23. Мальцев П. П., Гарбузов Н. И., Шарапов А. П., Кнышев Д. А. Программируемые логические схемы на КМОП структурах и их применение. —М.: Энергоатомиздат, 1998. 160 с.

24. Потемкин И. С. Функциональные узлы цифровой аппаратуры. —М.: Энергоатомиздат, 1988. 314 с.

25. Tatarnikov Y. Making Verilog Models Compatible with VHDL VITAL Level 0 Models . Integrated System Design Magasine (ISDMAG) May 1999. p. 1—3.

26. Poliakov A., Sokhatski A. Developing VITAL-Compliant VHDL Models. ISDMAG, July 1999, p. 58—60.

27. IEEE Std p1497-1999, Standard for Standard Delay Format (SDF) for the Electronic Design Process. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street,- New York, NY 10017-2394, USA. (ISBN not yet assigned).

28. Sutherland S. Verilog@ HDL Quick Reference Guide — based on Verilog-2001 standard. Sutherland HDL, Inc. 2001. p. 48.

Книги, вышедшие в 2002 г., после того, как данное пособие было сдано в редакцию

29. Угрюмов Е. Цифровая схемотехника. — Спб.: БХВ-Петербург, 2002. 516 с.

30. Грушвицкий Р. И., Мурсаев А. Х. Угрюмов Е. П. Проектирование систем на микросхемах программируемой логики. — Спб.: БХВ-Петербург, 2002. 608 с.

31. Комолов Д. А. и др. Системы автоматизированного проектирования фирмы Альтера Max+PlusII и Quartus II. — М.: ИП РадиоСофт, 2002. 352 с.

Содержание

Предисловие. Языки VHDL и VERILOG 3

Введение. HDL — исторический экскурс и перспективы 6

Глава 1. HDL — взгляд схемотехника и взгляд программиста 15

1.1. HDL — взгляд разработчика аппаратуры 15
1.1.1. Отображаемые аспекты 15
1.1.2. Интерфейс объекта проекта 16
1.1.3. Описание структуры объекта проекта 17
1.1.4. Связь имен компонентов и объекта проекта 19
1.1.5. Поведение объекта проекта 20
1.1.6. Разнообразие стилей описаний архитектур 21

1.2. HDL — взгляд программиста 24
1.2.1. Лексические элементы HDL 25
1.2.2. Данные (объекты): типы и виды 28
1.2.3. Операции и выражения 31
1.2.4. Операторы 34
1.2.5. Механизм расширения языка 39
1.2.6. Область видимости данных 40
1.2.7. Модули и библиотеки проекта 42

Глава 2. Базовые понятия HDL — процессы, задержки, алфавит . . . 44

2.1. Параллельные процессы 44
2.1.1. Параллельные операторы HDL 44
2.1.2. Оператор процесса 45
2.1.3. Краткие формы записи процессов 49
2.1.4. Присваивание с дельта-задержкой 50
2.1.5. Механизм воспроизведения модельного времени 51
Вопросы и упражнения 53

2.2. Задержки сигналов 53
2.2.1. Инерционная и транспортная задержка 53
2.2.2. Резекция и неопределенность коротких сигналов 56

2.3. Векторные операции и компактность описаний систем 57
2.3.1. Векторы 57
2.3.2. Оператор генерации 58

2.4. Алфавит моделирования 59
2.4.1. Четырехзначный алфавит 60
2.4.2. Девятизначный алфавит VHDL 61
2.4.3. X-пессимизм и оптимизм 63

2.5. Описание монтажных И (ИЛИ) и общей шины 64
2.5.1. Общая шина 64
2.5.2. Монтажное И, ИЛИ 65

2.6. Атрибуты объектов и контроль запрещенных ситуаций 66
2.6.1. Контроль запрещенных ситуаций. 66
2.6.2. Атрибуты VHDL-сигналов 66

Глава 3. Способы HDL-описаний простых узлов 69

3.1. Комбинационная схема F 69
3.1.1. Описание интерфейса 69
3.1.2. Процессная форма описания поведения. 70
3.1.3. Потоковое описание поведения 71
3.1.4. Структурное описание 74
3.1.5. Объявление конфигурации 75
3.1.6. Контроль временных соотношений 76
3.1.7. VERILOG-описание, использующее примитивы 77

3.2. Схемы с памятью 78
3.2.1. D-триггер 78
3.2.2. D-триггер со сбросом 79
3.3.3. Схема D-триггера на вентилях ИНЕ 79
3.2.4. D-триггер как примитив VERILOG 80
3.2.5. Модель RS-триггера-защелки 80
3.2.6. Модель T-триггера 81
3.2.7. VHDL — оператор блока в модели триггера типа «защелка» 84

3.3. Модель блока синхронной памяти 84
3.3.1. VHDL-модель. 84
3.3.2. VERILOG-модель 85
3.3.3. VERILOG — модель памяти с учетом задержек и контролем временных параметров сигналов в блоке specify 86
3.3.4. VHDL — модель памяти с общим регистром входных-выходных данных 87

Глава 4. Функциональная верификация HDL-описаний. 88

4.1. Пример верификации описания простого объекта проекта F 89

4.2. Стратегия функциональной верификации 93
4.2.1. Типы тестов 93
4.2.2. Полнота теста 93

4.3. Оценка полноты функциональных тестов 93
4.3.1. Эвристические метрики 93
4.3.2. Программные метрики 94
4.3.3. Автоматно-метрический подход 95
4.3.4. Моделирование неисправностей 95

4.3.5. Мониторинг событий и проверка контрольных соотношений в модели . . . 96

4.4. Компоненты тестирующей программы . . . 97
4.4.1. Тактовый генератор . . . 98
4.4.2. Генератор сигнала сброса . . . 98
4.4.3. Входные векторы . . . 98
4.4.4. Сравнение выходов модели с эталоном (VERILOG) . . . 99

4.5. Быстродействие и расход памяти инструментальной ЭВМ . . . 100
4.5.1. Расход памяти . . . 100
4.5.2. Быстродействие тестирующей программы . . . 100

4.6. Отладка тестирующей программы . . . 102
4.6.1. Порядок отладки . . . 102
4.6.2. Общие рекомендации . . . 104

4.7. Автоматизация построения тестирующих программ . . . 104

4.8. Структурированный тест объекта проекта F . . . 106
4.8.1. Генератор сигналов GEN . . . 106
4.8.2. Регистратор сигналов WRITER . . . 107
4.8.3. Архитектура теста — структурное описание . . . 108

4.9. Модельный эксперимент с самопроверкой . . . 110
4.9.1. VHDL-вариант . . . 110
4.9.2. VERILOG-вариант . . . 111
4.9.3. Модельный эксперимент со сравнением двух моделей F . . . 112

4.10. VHDL-модель и простой тест микросхемы памяти . . . 113
4.10.1. Микросхема К134РУ6 . . . 113
4.10.2. Описание интерфейса микросхемы . . . 114
4.10.3. Архитектура объекта SK134RU6 . . . 115
4.10.4. Модельный эксперимент с микросхемой ОЗУ . . . 116

Глава 5. Синтезабельность HDL-описаний . . . 119

5.1. Общие принципы построения синтезабельных описаний . . . 123
5.1.1. Повторнопригодность проектов . . . 123
5.1.2. Твердые и мягкие макросы . . . 124
5.1.3. Что такое «хороший проект макроса» . . . 124

5.2. Рекомендации по стилю кодирования HDL-описаний . . . 125
5.2.1. Рекомендации общего плана . . . 125
5.2.2. Рекомендуемая структура и примеры имен сигналов . . . 126
5.2.3. Организация базы данных проекта . . . 127

5.3. Что такое «хорошие» модули-макросы . . . 128
5.3.1. Общие рекомендации . . . 128
5.3.2 Дополнительные замечания . . . 130

5.4. RTL-описание 131

5.5. Синтезабельное подмножество HDL 132
5.5.1. Основные синтезабельные конструкции 133
5.5.2. Синтезабельные библиотеки типовых узлов 135
5.5.3. Синтезабельные образы узлов 136

5.6. Синтезабельные описания комбинационных узлов 137
5.6.1. Мультиплексоры 137
5.6.2. Дешифраторы (демультиплексоры) 140
5.6.3. Тристабильный буфер-ключ. 142
5.6.4. n-разрядный компаратор 142
5.6.5. Типичные ошибки в описании комбинационных узлов 143
5.6.6. Результаты синтеза одноразрядного сумматора. 146

5.7. Триггеры и регистры 151
Общая структура описаний 151
5.7.1. D-триггер-асинхронный сброс-установка 153
5.7.2. Триггер-синхронный сброс и установка 153
5.7.3. Регистры с разрешающим входом 154
5.7.4. Защелки 155
5.7.5. Сдвигатели 155
5.7.6. Счетчики 155
5.7.7. Регистровые файлы и блоки памяти. 156
5.7.8. Типичные ошибки в описаниях триггеров и регистров 157
5.7.9. Пример синтеза счетчика 158

5.8. HDL-описания автоматов. 162
5.8.1. Автоматы Мили и Мура 162
5.8.2. VERILOG — описание и тест автомата управления светофором . 162
5.8.3. VHDL-описание и тест автомата управления светофором 164
5.8.4. Синтез VERILOG-описания автомата управления светофором 166

Глава 6. Реализация шифроалгоритма RC4 на ПЛИС. 174

6.1. Шифроалгоритм RC4. 174

6.2. HDL-спецификация алгоритма RC4 176
6.2.1. Verilog 176
6.2.2. VHDL 178

6.3. ПЛИС семейства Virtex 181
6.3.1. Возможности 181
6.3.2. Архитектура семейства Virtex 182

6.4. VHDL-вариант реализации автомата RC4. 186
6.4.1. Блок памяти 186
6.4.2. Распределение микроопераций алгоритма по тактам. 187

6.4.3. VHDL-описание автомата RC4 188
6.4.4. VHDL-тест автомата RC4. 193
6.4.5. Результаты синтеза с памятью на триггерах 195
6.4.6. Результаты синтеза с использованием блочной памяти 198

6.5. VERILOG-описание автомата RC4 200
6.5.1. Описание автомата 200
6.5.2. Тест 203
6.5.3. Результаты синтеза 204

Глава 7. Функциональная модель микросхемы двухпортовой синхронной памяти 206

7.1. Состояние вопроса. 206

7.2. Некоторые свойства моделей RAM 208

7.3. Двухпортовая синхронная память 211

7.4. VHDL-модель блока памяти. 215
7.4.1. Интерфейс 215
7.4.2. Архитектура 216
7.4.3. Пакет со значениями временных параметров. 223
7.4.4. Модуль контроля временных параметров 224
7.4.5. Пакет функций преобразования типов данных 228

7.5. VERILOG-модель блока памяти 232
7.5.1. Интерфейс микросхемы 232
7.5.2. Тело модуля. 233
7.5.3. Задание и контроль временных параметров 234
7.5.4. Функциональная часть 235

7.6. Тестирующая программа 239
7.6.1. Переменные и константы. 239
7.6.2. Процедуры ЗАПИСИ-ЧТЕНИЯ 241
7.6.3. Подача тестовых векторов 245
7.6.4. Временные параметры сигналов теста 251

Приложение 1. Краткий справочник по языку VHDL. 253

1. Основы VHDL 253

2. Основные различия версий VHDL-93 и VHDL-87 264

3. Синтезабельное подмножество языка VHDL 267

4. Предопределенное окружение языка VHDL. 268
4.1. Пакет STANDARD 268
4.2. Пакет TEXTIO 270
4.3. Предопределенные атрибуты 271

5. Многозначная логика — IEEE пакеты и функции преобразования типов. . 272
1. Пакет IEEE STD_LOGIC_1164 272

2. Пакет IEEE NUMERIC_STD 273
3. Пакет IEEE NUMERIC_BIT 274
4. Пакет Synopsys STD_LOGIC_ARITH 275
5. Пакет STD_LOGIC_UNSIGNED 275

Приложение 2. VERILOG — краткий справочник 276

1. Лексические элементы. 276
1.1. Символы 276
1.2. Комментарии и аттрибуты 276
1.3. Символы операций и скобок 276
1.4. Имена 277
1.5. Логические значения 279

2. Литералы 279
2.1. Целые 279
2.2. Вещественные (реальные — real) литералы 280

3. Типы и виды данных 280
3.1. Вид переменная 280
3.2. Вид соединение (цепь, связь). 281
3.3. Другие виды и типы данных 281

4. Объявление данных и область видимости 282
4.1. Обьявление данных 282
4.2. Область видимости обьявлений данных 283

5. Операции 283

6. Выражения 286
6.1. Обычные выражения 286
6.2. Выражения с полями векторов и массивов 286
6.3. Задержанные выражения 286
6.4. Условные выражения 286
6.5. Выделение разрядов и полей 286
6.6. Выделение элементов массивов. 286

7. Последовательные операторы (процедурные). 287
7.1. Оператор ожидания (задержки) 287
7.2. Оператор присваивания переменной 287
7.3. Условный оператор 287
7.4. Оператор выбора 287
7.5. Оператор цикла 288
7.6. Оператор выхода из группы 288
7.7. Примеры последовательных операторов 288

8. Процедурные блоки 289

9. Группы операторов 290

10. Процедуры и функции 290

10.1. Процедуры task 290
10.2. Функции function 291

11. Параллельные операторы и блоки 291

12. Модуль проекта 292
12.1. Стиль описания VERILOG-95 292
12.2. Стиль VERILOG-2000 (ANSI-C) 292

13. Конкретизация — вызов экземпляров модулей 293
13.1. VERILOG-95 293
13.2. VERILOG-2000 293

14. Системные операторы и функции 294
14.1. Форматируемый ввод-вывод 294
14.2. Работа с файлами 295

15. Блок спецификаций временных соотношений-specify (несинтезабельная конструкция) 296
15.1. Обьявление спецпараметров 296
15.2. Средства проверки временных соотношений 296
15.3. Задержки путей распространения сигналов . . . . . 29.

16. Синтезабельные конструкции 298

17. Директивы компиляции (перечислены только основные) 299

Используемые сокращения 302

Интернет-ресурсы 304

Список литературы 305