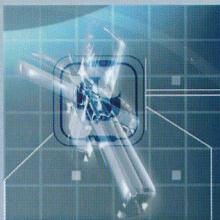
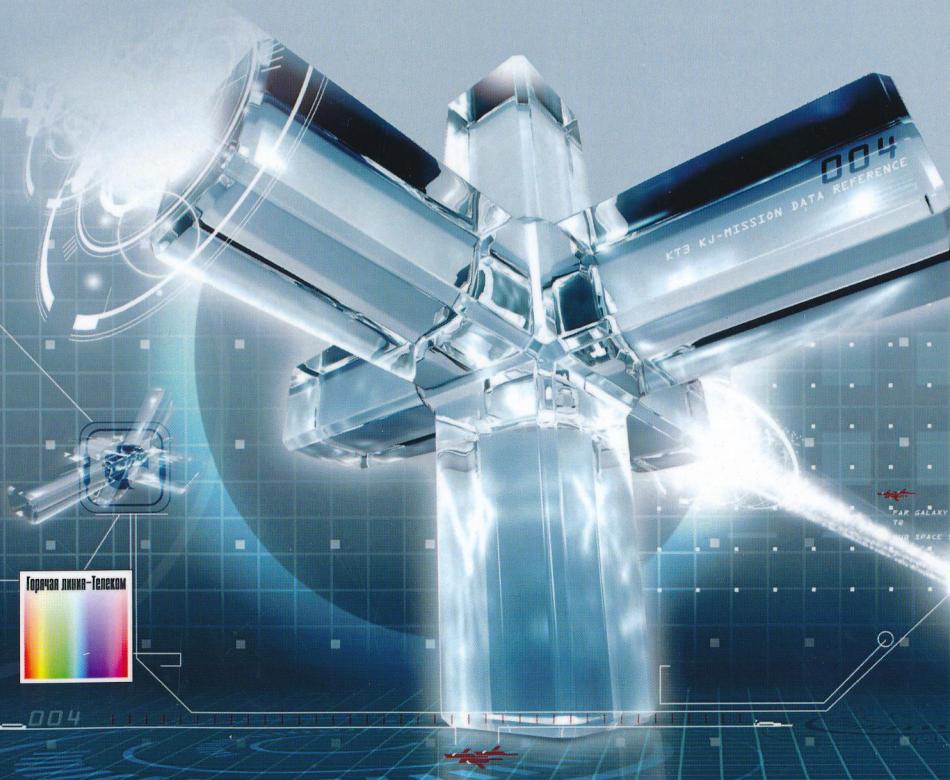


В. В. СОЛОВЬЕВ



ОСНОВЫ ЯЗЫКА ПРОЕКТИРОВАНИЯ ЦИФРОВОЙ АППАРАТУРЫ

VERILOG



В. В. Соловьев

**Основы языка
проектирования
цифровой аппаратуры**

VERILOG

Москва
Горячая линия - Телеком
2014

УДК 681.3
ББК 32.852.3
С60

Р е ц е н з е н т: профессор, доктор технических наук А. А. Баркалов, профессор Факультета Электроники, Информатики и Телекоммуникации Зеленогурского Университета (Зелена Гура, Польша).

Соловьев В. В.

С60 Основы языка проектирования цифровой аппаратуры Verilog. – М.: Горячая линия – Телеком, 2014. – 206 с.: ил.

ISBN 978-5-9912-0353-1.

Рассмотрен популярный язык проектирования цифровой аппаратуры Verilog. В книге достаточно полно описаны основные синтаксические элементы и конструкции языка с точки зрения их практического использования. Каждая конструкция языка сопровождается примером. Изложение материала не привязано к определенной элементной базе или конкретному программному средству проектирования, поэтому материал книги может использоваться при разработке проектов как на заказных СБИС и БМК, так и на ПЛИС. Популярность языку Verilog придает простота синтаксиса, во многом совпадающего с языком программирования С, а также большие возможности при описании цифровых устройств и систем, как для синтеза, так и для моделирования, от уровня транзисторов до сложных иерархических структур. Язык Verilog предоставляет также возможности для своего расширения. Для этого служит механизм определения пользовательских примитивов UDP и язык программирования интерфейса PLI.

Для разработчиков цифровых устройств и систем самостоятельно изучающих язык Verilog, будет полезна преподавателям, аспирантам и студентам соответствующих специальностей вузов.

ББК 32.852.3

Адрес издательства в Интернет WWW.TECHBOOK.RU

ISBN 978-5-9912-0353-1

© В. В. Соловьев, 2014
© Издательство «Горячая линия–Телеком», 2014

Посвящаю моей жене

Введение

Начало нашего века знаменуется бурным развитием и внедрением электронных изделий в повседневную жизнь, а также во все сферы человеческой деятельности (представьте ребенка без электронной игрушки, подростка без смартфона и самолет без бортового компьютера). Однако все эти интересные вещи, прежде чем изготовить, необходимо спроектировать. Поэтому огромная армия инженеров трудится над созданием новых и более совершенных электронных устройств. Канули в лету времена, когда принципиальные схемы сложных устройств чертились вручную или с помощью графических редакторов. На смену им пришли языки проектирования.

Сейчас разработка какого-либо электронного проекта во многом напоминает разработку программ: написание кода программы (написание кода проекта), компиляция программы (компиляция проекта), отладка программы на эталонных примерах (моделирование проекта с помощью специальной программы, называемой симулятором), создание исполняемого кода программы (реализация проекта в микросхеме), отладка программы на реальных примерах (физическое моделирование проекта на макетных платах), передача программы заказчику (реализация проекта в виде электронного изделия), сопровождение программы (сопровождение проекта).

В настоящее время наблюдается бурное развитие языков описания аппаратуры (Hardware Description Language — HDL) высокого уровня. Одним из таких языков является язык Verilog. Язык Verilog родился в среде разработчиков цифровых систем, поэтому быстро завоевал популярность у инженеров-практиков. Вскоре язык Verilog стал главным конкурентом языка VHDL. Язык VHDL был разработан по заказу Министерства обороны США, которое всячески способствовало его широкому распространению. На сегодняшний день некоторые программные средства проектирования работают следующим образом: проекты, написанные на языке VHDL, транслируют в язык Verilog, а затем используют компилятор языка Verilog, поскольку язык Verilog более близок к аппаратуре, чем язык VHDL. Отсюда возникает вопрос, не лучше ли сразу создавать проекты на языке Verilog, поскольку любая трансляция требует дополнительных накладных

расходов с точки зрения быстродействия, потребляемой мощности и площади на кристалле, занимаемой проектом.

Однако широкому распространению языка Verilog среди русскоязычных проектировщиков в значительной степени препятствует отсутствие книг по языку Verilog. Те немногочисленные статьи, которые иногда появляются в периодических журналах [1–3] и книга Полякова [4] не могут закрыть указанную проблему. Изучение же языка по стандартам [5–7] весьма затруднительно, поскольку стандарты языка Verilog, прежде всего, предназначены не для пользователей языка, а для разработчиков компиляторов языка. В то же время в мире издано большое количество книг, посвященных языку Verilog, как для начинающих пользователей [8, 9], так и для опытных специалистов [10, 11]. Отметим также, что язык Verilog является объектом серьезных научных исследований [12–21].

В данной книге достаточно полно описываются основные синтаксические элементы и конструкции языка Verilog с точки зрения их практического использования. Каждая конструкция сопровождается примером. Поскольку язык Verilog предназначен как для описания проекта, так и для моделирования проекта, то не все конструкции языка Verilog могут быть реализованы аппаратно, т. е. синтезированы. Если конструкция синтезируется, то в книге приводится ее реализация на уровне регистровых передач (Register Transfer Level — RTL). Кроме того, непосредственно в тексте книги предлагается большое количество заданий, выполнение которых способствует более глубокому изучению языка. Некоторые задания посвящены изучению особенностей используемого читателем компилятора. Выполнение таких заданий позволяет приобрести практические знания о возможностях конкретного компилятора. В книге также много замечаний. Они заостряют внимание читателя на отдельных свойствах языка Verilog, его реализации в компиляторах и др. Все замечания следует внимательно прочитать и переосмыслить.

Изложение материала данной книги не привязывается к определенной элементной базе или конкретному программному средству проектирования. Другими словами, материал книги может использоваться при разработке проектов как на заказных СБИС и БМК, так и на ПЛИС. Для демонстрации результатов реализации примеров используется пакет Quartus II версии 12.1 фирмы Altera, однако читатель может использовать любой другой пакет.

Книга имеет следующую структуру. В самом начале, в главе 1, дается быстрое введение в язык Verilog, где на простом примере показан процесс разработки проектов на основе языка Verilog, от описания проекта до его моделирования, а также показаны возможности языка Verilog при разработке сложных иерархических проектов. Затем

приводятся базовые элементы языка. Таким образом, прочтение только первой главы уже позволяет читателям разрабатывать простые проекты.

Главными конструкторскими единицами языка Verilog являются модули, им посвящена глава 2. Чтобы в очередной раз «не изобретать велосипед», можно воспользоваться уже готовыми решениями, такими, как примитивы и библиотечные модули, которые описываются в главе 3. Каждый язык имеет свои типы данных, с которыми он оперирует. Типы данных языка Verilog имеют свои специфические особенности и рассматриваются в главе 4. Использование языка проектирования невозможно без знаний операций, описываемых в главе 5. Особую роль в языке Verilog играет оператор непрерывного назначения, рассматриваемый в главе 6. Функционирование проекта в языке Verilog описывается в виде совокупности взаимодействующих между собой процессов, которые представляют процедурные блоки, рассматриваемые в главе 7. При моделировании важнейшую роль играет время функционирования отдельных частей проекта и всего проекта в целом. Операторы управления процедурным временем представлены в главе 8.

И вот все готово для описания алгоритма функционирования проекта во времени. Для этого служат операторы процедурного назначения, рассматриваемые в главе 9, и операторы процедурного программирования, описываемые в главе 10. Важную роль в языке Verilog также играют атрибуты, представленные в главе 11. С помощью атрибутов пользователь может передать указания (и параметры) компилятору как выполнять компиляцию отдельных фрагментов кода. Блоки генерации, рассматриваемые в главе 12, позволяют сократить описание повторяющихся или незначительно отличающихся фрагментов кода. Задачи и функции языка Verilog, описываемые в главе 13, во многом подобны подпрограммам и функциям языков программирования. Блок спецификаций, рассматриваемый в главе 14, позволяет определять значения временных параметров, которые могут использоваться при работе симулятора и временного анализатора. Системные задачи и функции приводятся в главе 15, а директивы компилятора — в главе 16. При разработке сложных проектов различными проектировщиками для согласования местоположений отдельных частей проекта служат конфигурации и конфигурационные блоки, рассматриваемые в главе 17. Синтезируемые конструкции языка Verilog приводятся в главе 18.

Настоящая книга, в первую очередь, предназначена разработчикам цифровых устройств и систем для самостоятельного изучения языка Verilog. Поскольку изложение основных элементов и конструкций языка достаточно полное, книга может также использоваться в

качестве справочника языка Verilog. Кроме того, материал книги может быть использован преподавателями для чтения лекций и проведения практических занятий, а также, безусловно, книга предназначена для студентов соответствующих специальностей в качестве учебного пособия при подготовке к практическим занятиям, экзаменам, при написании курсовых и дипломных работ.

Автор выражает искреннюю благодарность сотрудникам фирмы Атомтех (г. Минск, Республика Беларусь), которые прослушали курс лекций на основе рукописи данной книги, за многочисленные вопросы и замечания, способствовавшие улучшению материала книги.

Г л а в а 1

Предварительное знакомство с языком Verilog

1.1. История языка Verilog

Язык Verilog разработали Phil Moorby и Prabhu Goel зимой 1983/1984 года для фирмы Automated Integrated Design System (в 1985 г. была переименована в Gateway Design Automation). Первоначально язык Verilog предназначался исключительно для моделирования цифровых устройств. В 1884 г. фирма Gateway начала продажу программы моделирования (симулятора) под названием Verilog XL. С течением времени симулятор Verilog XL становится все более популярным среди разработчиков цифровых систем.

В 1987 г. фирма Synopsys начала использовать язык Verilog в качестве языка для спецификации при синтезе проектов цифровых систем. С этого момента язык Verilog стал использоваться для задач логического синтеза.

В 1989 г. фирму Gateway купила фирма Cadence — крупный производитель систем автоматизации проектирования (САПР) электронной техники. В соответствии со своей политикой фирма Cadence разделила язык Verilog и симулятор Verilog XL на два независимых продукта. В результате несколько фирм купили права на использование языка Verilog.

В 1990 г. фирма Cadence дала разрешение на публичное использование языка Verilog. С этого момента любая фирма могла использовать язык Verilog без специального лицензионного соглашения. Главной целью такого шага было повышение конкурентоспособности языка Verilog с языком VHDL. В этом же году была создана группа под названием Open Verilog International (OVI) для контроля спецификаций языка Verilog, поскольку каждый разработчик компилятора языка Verilog мог по-своему интерпретировать язык Verilog. Группа OVI выполнила несколько улучшений языка Verilog и в 1993 г. объявила о новой спецификации языка.

В 1995 г. был принят первый стандарт языка Verilog — IEEE 1364-1995. С этого момента язык Verilog, подобно языку VHDL, стал полноправным открытым языком проектирования.

В 2001 г. появился очередной стандарт языка Verilog — IEEE 1364-2001, который определил ряд существенных улучшений языка Verilog и был назван Verilog-2001.

В 2000 г. была начата работа под руководством группы Accelera (консорциум фирм, занимающихся проектированием цифровых систем и разработкой САПР) над языком System Verilog.

В 2005 г. появилось сразу два стандарта, касающихся языка Verilog, Verilog-2005 (IEEE 1364-2005) и System Verilog-2005 (IEEE 1800-2005). В стандарте Verilog-2005 получили отражения небольшие изменения стандарта Verilog-2001.

В последующем идеи языка Verilog получили свое развитие в языке System Verilog, для которого были разработаны стандарты IEEE 1800-2009 и IEEE 1800-2012.

В настоящее время большинство производителей программного обеспечения для проектирования и моделирования цифровой аппаратуры поддерживают стандарт языка Verilog-2001 (Aldec, Altera, Axiom Design Automation, Cadence Design Systems, Dolphin Integration, Fintronic, Frontline, Huada Empyrean Software, Mentor Graphics, Simucad Design Automation, Sugawara Systems, SynaptiCAD, Synopsys, Tachyon Design Automation, WinterLogic, Xilinx и др.). Поэтому в данной книге рассматривается версия языка Verilog-2001 стандарта IEEE 1364-2001. Для демонстрации примеров описания цифровых устройств используется пакет Quartus II версии 12.1 фирмы Altera.

1.2. Первый проект на языке Verilog

1.2.1. Описание проекта

Таблица 1.1

Таблица истинности однобитового сумматора

Входы			Выходы	
cin	a	b	cout	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Традиционно, при изучении языков программирования первая программа выводит на экран символьную строку «Hello, world!». Безусловно, язык Verilog также позволяет выводить строку символов на экран, но все же Verilog это язык проектирования, а не программирования. Поэтому в качестве первого проекта на языке Verilog рассмотрим описание однобитового сумматора. Функционирование однобитового сумматора можно представить в виде таблицы истинности, приведенной в табл. 1.1, где a и b — единичные биты слов A и B соответственно; cin — перенос из предыдущего разряда; s — бит суммы; cout — перенос в следующий разряд.

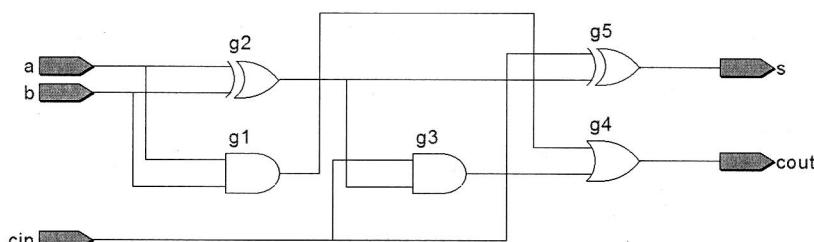


Рис. 1.1. Схема однобитового сумматора на вентильном уровне

На основании табл. 1.1 можно записать следующие логические уравнения выходных функций:

$$\begin{aligned} s &= \overline{\text{cin}} \cdot \overline{a} \cdot b + \overline{\text{cin}} \cdot a \cdot \overline{b} + \text{cin} \cdot \overline{a} \cdot \overline{b} + \text{cin} \cdot a \cdot b; \\ \text{cout} &= \overline{\text{cin}} \cdot a \cdot b + \text{cin} \cdot \overline{a} \cdot b + \text{cin} \cdot a \cdot \overline{b} + \text{cin} \cdot a \cdot b, \end{aligned} \quad (1)$$

которые после минимизации и эквивалентных логических преобразований можно представить в следующем виде (сомневающиеся могут обратиться к любому учебнику по основам электроники и вычислительной техники):

$$\begin{aligned} s &= a \oplus b \oplus \text{cin}; \\ \text{cout} &= a \cdot b + (a \oplus b) \cdot \text{cin}, \end{aligned} \quad (2)$$

где знак \oplus означает логическую операцию Исключающее ИЛИ (XOR). Преимуществом уравнений (2) по сравнению с уравнениями (1) является то, что они могут быть реализованы на вентилях с двумя входами (рис. 1.1).

Описание нашего первого примера однобитового сумматора на языке Verilog представлено в листинге 1.1.

Листинг 1.1. Структурное описание однобитового сумматора на рис. 1.1.

```

*****
* код проекта однобитового сумматора, выполненный в стиле описания
* структурной модели на вентильном уровне
***** /
```

```

module add_1_1
    (input cin, a, b,          // описание входных портов
     output s, cout);         // описание выходных портов
    /* объявления */
    wire g1_o, g2_o, g3_o;    // промежуточные переменные
    /* операторы */
    and g1 (g1_o, a, b);     // экземпляр первого вентиля AND
    xor g2 (g2_o, a, b);     // экземпляр первого вентиля XOR

```

```

and g3 (g3_o, g2_o, cin); // экземпляр второго вентиля AND
or g4 (cout, g1_o, g3_o); // экземпляр вентиля OR
xor g5 (s, g2_o, cin); // экземпляр второго вентиля XOR
endmodule

```

Основной единицей языка Verilog является модуль, который начинается ключевым словом **module** и заканчивается словом **endmodule**. За ключевым словом **module** следует название проекта *add_1_1*, за которым в круглых скобках следует список портов модуля: входных (**input**) и выходных (**output**). Описание портов модуля в языке Verilog всегда заканчивается точкой с запятой («;»).

Если для описания проекта требуются дополнительные переменные, они должны быть объявлены перед их первым использованием. В нашем примере это цепи (**wire**), соответствующие выходам вентиляй: *g1_o*, *g2_o* и *g3_o*. Собственно описание нашего проекта состоит из описаний экземпляров каждого вентиля на рис. 1.1. Поскольку логические вентиля относятся к примитивам языка Verilog, они могут использоваться без предварительного описания или объявления.

Описание каждого отдельного вентиля состоит из названия типа вентиля (**and**, **or**, **xor**), названия экземпляра (*g1*, *g2*, *g3*, *g4*, *g5*), а также списка портов конкретного экземпляра. Список портов представляет собой список сигналов, назначаемых портам соответствующего экземпляра. Для вентилях характерным является то, что выход вентиля в списке портов всегда является первым элементом, за которым следуют входы вентиля. Такая организация портов вентиляй вполне логична, поскольку вентиля могут иметь произвольное число входов и только один выход. Для передачи сигналов между вентилями используются объявленные ранее дополнительные переменные *g1_o*, *g2_o* и *g3_o*.

Замечание. Скалярные (однобитовые) внутренние соединения в языке Verilog можно не объявлять, компилятор их создает автоматически.

Согласно приведенному замечанию строку в листинге 1.1 с объявлением переменных *g1_o*, *g2_o* и *g3_o* можно опустить.

Задания.

- Выполните на языке Verilog описание однобитового сумматора по уравнениям (1).
- Выполните минимизацию уравнений (1) (например, с помощью карт Карно) и по полученным уравнениям опишите однобитовый сумматор на языке Verilog.

Листинг 1.1 представляет собой структурное описание проекта на вентильном уровне. Подобным образом можно описать любой проект,

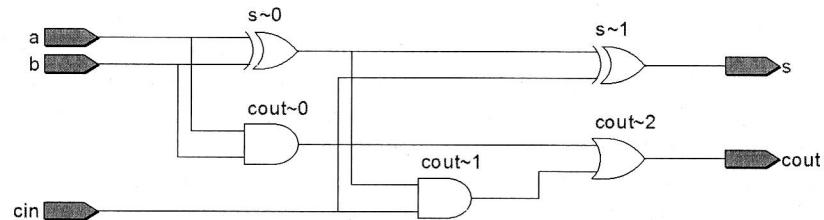


Рис. 1.2. Реализация модуля *add_1_2* из листинга 1.2: пример описания однобитового сумматора с помощью логических уравнений

причем не обязательно на вентильном уровне. Фактически структурное описание полностью соответствует графическому представлению проекта и здесь не видно преимуществ использования языка проектирования, по сравнению с графическим редактором. Поэтому подобный стиль описания проектов в языках проектирования используется редко.

Этот же проект можно описать с помощью логических уравнений (2).

Листинг 1.2. Описание однобитового сумматора с помощью параллельных операторов назначения

```

/***********************
код проекта однобитового сумматора, выполненный в стиле описания логических уравнений
*****/
module add_1_2
  (input cin, a, b,
   output s, cout);
  assign s = a & b ^ cin; // описание функции суммы
  assign cout = a & b | (a & b) & cin; // описание функции переноса
endmodule

```

В листинге 1.2 для описания проекта однобитового сумматора использованы операторы непрерывного назначения **assign**. Данные операторы всегда выполняются параллельно независимо от их места в коде проекта. Здесь справа от знака равенства описаны выражения, использующие логические операции, обозначаемые следующими знаками: «|» — ИЛИ (OR), «&» — И (AND), «^» — Исключающее ИЛИ (XOR). Отметим также, что отрицание обозначается знаком «~». Реализация модуля *add_1_2* показана на рис. 1.2. Отметим, что схемы на рис. 1.1 и 1.2 полностью совпадли, хотя были представлены различными стилями описания.

Замечание. Стиль описания проекта, приведенный в листинге 1.2, наиболее часто используется для описания комбинационных схем на основании логических (булевых) уравнений.

Язык Verilog также позволяет описывать проекты на уровне поведения, т. е. алгоритма функционирования, с помощью процедурных операторов. Чтобы описать наш проект однобитового сумматора на уровне поведения, выявим некоторые закономерности в его функционировании. Например, функция переноса `cout` будет равна единице, если на входах сумматора одновременно появляется не менее двух единиц. В этом случае описание однобитового сумматора можно представить следующим образом:

Листинг 1.3. Описание однобитового сумматора с помощью процедурных операторов

```
*****  
код проекта однобитового сумматора, выполненный в стиле описания поведения  
*****
```

```
module add_1_3
  (input cin, a, b, // описание входных портов
   output reg s, cout); // описание выходных портов
  always @ (cin, a, b) begin
    if (a & b | cin & a | cin & b) cout = 1; // описание функции суммы
    else cout = 0;
    s = a ^ b ^ cin; // описание функции переноса
  end
endmodule
```

В листинге 1.3 использован процедурный оператор `if-else`, а также процедурный оператор блокирующего назначения `«=»`. Кроме того, для выходных функций `s` и `cout` был использован тип переменных `reg`. Дело в том, что согласно правилам языка Verilog переменные, записываемые слева от знака равенства в процедурных операторах назначения, должны иметь тип `reg`. Реализация модуля `add_1_3` показана на рис. 1.3.

После того как некоторый компонент проекта описан, он может быть использован в проекте многократно. Например, однобитовый сумматор может использоваться для построения 4-битового сумматора.

Листинг 1.4. Описание 4-битового сумматора

```
*****  
код проекта 4-битового последовательного сумматора, выполненный путем создания четырех экземпляров однобитового сумматора  
*****
```

```
module add_4
  (input CIN, input wire [3:0] A, B, // описание входов
   output wire [3:0] S, output COUT); // описание выходов
```

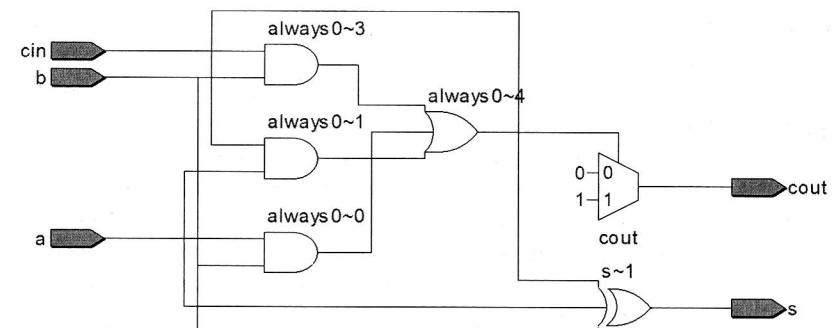


Рис. 1.3. Реализация модуля `add_1_3` из листинга 1.3: пример описания однобитового сумматора с помощью процедурных операторов

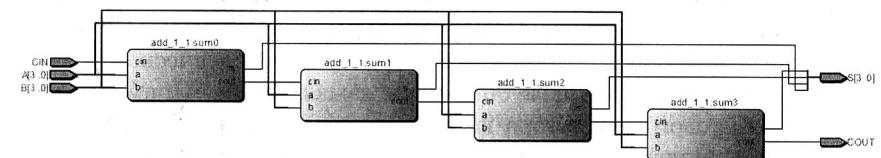


Рис. 1.4. Реализация модуля `add_4` из листинга 1.4: иерархическая структура последовательного сумматора на 4 бита, построенная из четырех экземпляров однобитовых сумматоров

```
wire [2:0] C; // описание промежуточных // переменных
```

```
/* описание четырех экземпляров однобитовых сумматоров */
add_1_1 sum0(CIN,A[0],B[0],S[0],C[0]);
add_1_1 sum1(C[0],A[1],B[1],S[1],C[1]);
add_1_1 sum2(C[1],A[2],B[2],S[2],C[2]);
add_1_1 sum3(C[2],A[3],B[3],S[3],COUT);
endmodule
```

В листинге 1.4 использовано четыре экземпляра модуля однобитового сумматора `add_1_1` с именами `sum0`, `sum1`, `sum2` и `sum3`. Для передачи значения сигнала переноса между разрядами здесь используются промежуточные переменные `C[0]`, `C[1]`, `C[2]` типа `wire`. Реализация модуля `add_1_4` показана на рис. 1.4.

Подобным образом на языке Verilog описываются достаточно сложные иерархические проекты.

В общем случае язык Verilog позволяет выполнять описание проектов на следующих уровнях:

- транзисторов;
- вентилей;
- логических уравнений;
- регистровых передач (register transfer level — RTL);

- поведенческом (behavioral);
- структурном (системном).

Описание проектов на уровне транзисторов используется редко. Например, такой стиль описания может быть использован при проектировании новых библиотечных элементов для заказных и полузаказных СВИС. Пример описания проекта на уровне вентилей приведен в листинге 1.1. Для описания проекта на уровне логических уравнений достаточно представить проект в виде системы логических уравнений. При этом могут использоваться промежуточные переменные. Пример описания проекта на уровне логических уравнений приведен в листинге 1.2. Описание на уровне регистровых передач (RTL) отличается от описания проектов на уровне вентилей только тем, что вместо вентилей используются функциональные элементы уровня регистровых передач: регистры, счетчики, сумматоры, дешифраторы, мультиплексоры и др. Для описания проектов на поведенческом уровне используются процедурные операторы языка Verilog. Пример описания однобитового сумматора на поведенческом уровне приведен в листинге 1.3. Описание проекта на системном уровне подобно описанию на уровне регистровых передач, только вместо функциональных элементов уровня регистровых передач используются функциональные блоки системного уровня (процессоры, памяти, шины, устройства управления, устройства ввода-вывода и др.).

1.2.2. Моделирование проекта

После того как проект описан, его следует проверить (верифицировать), т. е. выполнить моделирование проекта. Ранее, до появления языка Verilog, для моделирования использовались графические редакторы, специальные языки, отличные от языков описания аппаратуры, и др. Однако, поскольку язык Verilog изначально был предназначен для моделирования, то он же используется и для моделирования проекта.

Модуль для моделирования однобитового сумматора представлен в листинге 1.5.

Листинг 1.5. Модуль для моделирования однобитового сумматора

```
'timescale 1ns/1ps          // определение единицы времени
                                // моделирования
module test_add_1();         // начало модуля
    reg tcin, ta, tb;        // объявление переменных
    wire ts, tcout;          // для входных векторов
                            // объявление переменных
                            // для выходных функций
```

```
add_1_1 sum(tcin,ta,tb,ts,tcout); // создание экземпляра модуля
initial begin: test           // процедурный оператор
                            // с именем test
    integer i;                // объявление локальной
                            // переменной
    $display("Результаты моделирования однобитового сумматора:");
    $timeformat(-9,1,"ns",8);   // определение формата времени
    $monitor($time,": cin=%b a=%b b=%b s=%b cout=%b",
             tcin,ta,tb,ts,tcout);
    for (i=0; i<8; i=i+1) begin // начало оператора цикла
        #10;                   // оператор задержки
                                // на 10 единиц времени
        {tcin,ta,tb} = i;       // формирование входного вектора
    end                      // окончание оператора цикла
    end                      // окончание процедурного
                            // оператора test
endmodule                  // окончание модуля
```

Отметим некоторые особенности приведенного описания. В первой линии кода с помощью системной директивы **'timescale** определяется значение временных единиц 1ns (одна наносекунда) и точность измерения времени 1ps (одна пикосекунда). Поскольку модуль для моделирования с именем **test_add_1** извне не получает никаких сигналов, а также не формирует сигналы во внешнюю среду, то он портов не имеет. Затем объявляются переменные для формирования входных векторов **tcin**, **ta**, **tb** и получения результатов суммирования **ts**, **tcout**. В следующем пункте создается модель проекта в виде экземпляра модуля **add_1_1** с именем **sum**.

Эмуляция модели осуществляется с помощью процедурного блока **initial** с именем **test**, который выполняется только один раз. Здесь объявляется локальная переменная **i** целого типа и вызываются системные задачи: **\$display** для вывода строки символов; **\$timeformat** для определения формата представления времени и **\$monitor** для наблюдения за указанными переменными и вывода их значений в случае изменения хотя бы одной из переменных (кроме значения времени). Затем следует процедурный оператор типа **for**, в теле которого выполняется задержка на 10 единиц времени и формируется значение входных векторов.

Поскольку в языке Verilog процедурные блоки и экземпляры модулей выполняются параллельно, то сформированное в блоке **initial** значение входного вектора (переменные **tcin**, **ta** и **tb**) автоматически будет передано на вход экземпляра **sum**. Время работы экземпляра **sum** не определено, поэтому принимается равным 0.0. Таким образом, значения выходных переменных **ts** и **tcont** будут сформированы

```
# Результаты моделирования однобитового сумматора:
# 0: cin=x a=x b=x s=x cout=x
# 10: cin=0 a=0 b=0 s=0 cout=0
# 20: cin=0 a=0 b=1 s=1 cout=0
# 30: cin=0 a=1 b=0 s=1 cout=0
# 40: cin=0 a=1 b=1 s=0 cout=1
# 50: cin=1 a=0 b=0 s=1 cout=0
# 60: cin=1 a=0 b=1 s=0 cout=1
# 70: cin=1 a=1 b=0 s=0 cout=1
# 80: cin=1 a=1 b=1 s=1 cout=1
```

Рис. 1.5. Результаты моделирования однобитового сумматора

без задержки, а системная задача **\$monitor** выведет их значения и значения входных переменных на экран.

Задание. Сравните листинги с описанием однобитового сумматора с листингом 1.5. Как вы считаете, что занимает у проектировщика больше времени описание проекта или моделирование проекта?

Результаты моделирования однобитового сумматора приведены на рис. 1.5.

Анализируя результаты моделирования, разработчик может сделать выводы о правильности функционирования проекта и в случае необходимости внести в описание проекта соответствующие изменения.

Замечание. Анализ результатов моделирования также может выполняться с помощью языка Verilog, а на экран при этом будут выводиться только результаты такого анализа.

1.3. Базовые элементы языка Verilog

1.3.1. Ключевые слова

Предварительное знакомство с ключевыми словами является хорошей практикой при изучении языков программирования и проектирования. Знание ключевых слов необходимо хотя бы потому, что их нельзя использовать в качестве идентификаторов. В табл. 1.2 и 1.3 приводятся ключевые слова стандартов Verilog-1995 и Verilog-2001, реализованные в пакете Quartus II фирмы Altera. Если вы используете другую интегрированную среду для разработки своих проектов, то вам необходимо ознакомиться с ключевыми словами реализованной версии языка Verilog. Заметим, что внимательное прочтение ключевых слов нового языка дает некоторое представление о языке, а также вызывает определенный интерес и вопросы, ответы на которые можно найти при последующем изучении языка.

Задание. Познакомьтесь с ключевыми словами компилятора

Ключевые слова языка Verilog версии 1995 года				
always	endprimitive	medium	realtime	tranif0
and	endspecify	module	reg	tranif1
assign	endtable	nand	release	tri
begin	endtask	negedge	repeat	tri0
buf	event	nmos	rnmos	tril
bufif0	for	nor	rpmos	triand
bufif1	force	not	rtran	trior
case	forever	notif0	rtranif0	trireg
casex	fork	notif1	rtranif1	vectored
casez	function	or	scalared	wait
cmos	highz0	output	small	wand
deassign	highz1	parameter	specify	weak0
default	if	pmos	specparam	weak1
defparam	ifnone	posedge	strong0	while
disable	initial	primitive	strong1	wire
edge	inout	pull0	supply0	wor
else	input	pull1	supply1	xnor
end	integer	pullup	table	xor
endcase	join	pulldown	task	
endmodule	large	rcmos	time	
endfunction	macromodule	real	tran	

Таблица 1.3
Ключевые слова языка Verilog версии 2001 года (дополнение к версии 1995 года)

automatic	inmdir	pulsestyle_ondetect
cell	include	pulsestyle_onevent
config	instance	signed
endconfig	liblist	showcancelled
endgenerate	library	unsigned
generate	localparam	use
genvar	noshowcancelled	

языка Verilog используемой вами интегрированной среды проектирования.

1.3.2. Идентификаторы

Идентификаторы языка Verilog используются в качестве имен переменных, модулей, функций, экземпляров модулей, экземпляров примитивов и др. Идентификаторы могут содержать большие (A-Z) и малые (a-z) буквы латинского алфавита, цифры (0-9), знак подчеркивания («_»), а также знак доллара («\$»). Другие знаки ASCII могут использоваться в идентификаторах с предшествующим знаком

обратного слеша («\»). Начинаться идентификаторы должны с буквы или знака подчеркивания, а заканчиваться «белым» знаком. Длина идентификаторов не должна превышать 1024 символов. Язык Verilog различает в идентификаторах прописные и строчные буквы, поэтому abc, Abc, ABC — это разные идентификаторы.

Примеры правильных идентификаторов:

add_1, g1, g_2, g_2_1, _adder, CLK, clk, XOR.

Примеры неправильных идентификаторов:

1_add, 2012_project, \$RESET, хор.

Здесь идентификатор XOR, записанный заглавными буквами, правильный, а хор — неправильный, поскольку он совпадает с ключевым словом xor (все ключевые слова записываются строчными буквами).

1.3.3. Белые знаки

Белые знаки (*white space*) используются для форматирования описания проекта, чтобы придать коду более читабельную форму.

Замечание. Описание проекта на языке Verilog называется кодом.

К белым знакам в языке Verilog относятся: пробел, знак табуляции, знак новой линии (*carriage return*) и знак перевода страницы (*formfeed*). Практически белые знаки можно вставлять в любом месте кода. Однако строка символов, заключенная в кавычки, не должна содержать знак новой линии.

1.3.4. Комментарии

В языке Verilog используются две формы комментариев: для одной строки и блочные. Комментарий одной строки начинается с двух знаков слеша («//») и распространяется до конца строки. Блочный комментарий (так же, как в языке С [22]) начинается с двух знаков «/*» и заканчивается двумя знаками «*/». Блочный комментарий может охватывать несколько строк кода. Однако запрещено вложение блочных комментариев.

Предостережение. Будьте внимательны при выключении части кода с помощью блочного комментария. Если выключаемая часть кода также содержит блочный комментарий, то будет выключена часть кода до первого встретившегося знака «*/».

1.4. Сигналы, сети, драйверы

В общем случае в языке Verilog нет такого понятия, как сигнал (несмотря на то что язык Verilog используется для проектирования цифровых систем, где сигнал является ключевым понятием). Вместо

сигналов в языке Verilog используется понятие сети (*nets*). Сети используются для соединения портов (выводов) компонентов проекта. Основными компонентами проекта являются модули, в качестве частей проекта могут также выступать примитивы, функции и задачи, которые также могут иметь порты.

Сеть характеризуется двумя параметрами: логическим значением и логической мощностью.

Логическое значение определяет состояние сигнала сети.

Сигналы в сеть поступают от источников сигналов, называемых драйверами (*drivers*). Поэтому второй параметр сети определяет логическую мощность (*strength*) драйвера, от которого в сеть поступает сигнал. Одна сеть может иметь несколько драйверов. Отсюда возникает проблема определения значения сигнала в сети в зависимости от логической мощности драйверов.

Обычно логическая мощность драйвера или сети определяется для единичного (высокого) и нулевого (низкого) значений логических сигналов.

1.4.1. Логические значения

В языке Verilog каждый бит (единичный сигнал) может иметь четыре значения:

0 — логический ноль или «ложь»;

1 — логическая единица или «правда»;

x — неизвестное значение (т. е. может быть как 0 так и 1) или неопределенное значение (*don't care*);

z — состояние высокой импеданции, третье или плавающее (*floating*) состояние.

Когда значение z появляется на входе или в выражении, то результат будет таким же, как при значении x. Таким образом, значение z отличается от значения x только для выходных сигналов.

В языке Verilog допускаются еще два логических значения, которые используются программным обеспечением только для внутренней симуляции и не могут использоваться в описании проекта:

L — частично неизвестное значение: либо 0, либо z, но не 1;

H — частично неизвестное значение: либо 1, либо z, но не 0.

Замечание. Моделирование проекта по его описанию на языке Verilog с помощью специальной программы называется симуляцией (*simulation*), а программа, выполняющая моделирование — симулятором (*simulator*).

Как можно заметить, число различных значений, которые могут принимать сигналы в языке Verilog не велико и при решении различных задач моделирования может быть недостаточным. Поэтому

многие разработчики компиляторов языка Verilog увеличивают количество возможных значений сигналов.

Задание. Узнайте, какие логические значения может принимать одиночный сигнал языка Verilog в интегрированной среде, которую вы используете.

1.4.2. Логическая мощность (сила) сигналов

Логические значения сигналов могут иметь 8 уровней мощностей: четыре определяются источником (драйвером) сигнала; три определяются емкостью и один является состоянием высокой импеданции (т. е. не имеет мощности). Уровни логической мощности представлены в табл. 1.4. Здесь уровень мощности с наибольшим номером имеет большую мощность.

Цепь с несколькими источниками сигналов может иметь комбинацию мощностей, которая представляется в виде восьмеричных чисел и логического значения, например: 65x, где 6 и 5 — уровни логической мощности источников сигналов; x — логическое значение x.

Таблица 1.4

Уровни мощности источников сигналов

Уровень мощности	Название мощности	Ключевые слова	Мнемонические обозначения
7	supply drive (управление питанием)	supply0 supply1	Su0 Su1
6	strong drive (сильное управление)	strong0 strong1	St0 St1
5	pull drive (управление подтягиванием)	pull0 pull1	Pu0 Pu1
4	large capacitive (большая емкость)	rlarge	La0 La1
3	weak drive (слабое управление)	weak0 weak1	We0 We1
2	medium capacitive (средняя емкость)	medium	Me0 Me1
1	small capacitive (малая емкость)	small	Sm0 Sm1
0	high impedance (высокая импедансия)	highz0 highz1	HiZ0 HiZ1

1.5. Числа

1.5.1. Представление целых чисел

Для представления чисел в языке Verilog используют следующий формат:

size'base value

где *size* — размер, десятичное число, определяющее количество битов в представлении числа; '*base* — база, символ, определяющий систему счисления, в которой представляется число; *value* — значение числа. Для представления системы счисления ('*base*) используются следующие символы:

- 'b или 'B — для двоичных чисел;
- 'd или 'D — для десятичных чисел;
- 'h или 'H — для шестнадцатиричных чисел;
- 'o или 'O — для восьмиричных чисел.

Отметим, что при задании любого числа в языке Verilog указывается число битов (*size*), в котором это число будет записано. Исключение составляют использование однобитовых значений в некоторых конструкциях языка. Каждый такой случай будет в книге специально оговариваться при рассмотрении соответствующей конструкции.

В общем случае размер (*size*) и символ базы (*base*) в представлении числа являются необязательными. По умолчанию обычно принимается число в десятичной системе счисления (*base='D*), записываемое в 32 разрядах (*size = 32*). Конкретные значения умалчиваемых значений размера и базы зависят от реализации компилятора.

Задание. Узнайте умалчиваемые значения размера (*size*) и базы (*base*) в представлении чисел для используемого вами компилятора.

Примеры чисел:

8'b01011011 // двоичное число в 8 разрядах
16'0F2E // 16-е число в 16 разрядах
157 // десятичное число в 32 разрядах (по умолчанию)

Если двоичное представление значения числа превышает заданный размер числа (*size*), то старшие значения в двоичном представлении числа отбрасываются. В случае, когда значение числа занимает меньше разрядов, чем указано в размере числа, то число располагается в младших разрядах, а в старшие разряды записываются нули. Исключения составляют случаи, когда наибольшим значащим значением числа является z или x, в последнем случае старшие разряды заполняются значением соответственно z или x.

Примеры поразрядного представления целых чисел приведены в табл. 1.5.

Замечания.

1. Вместо значения z в представлении чисел можно использовать знак вопроса (?).

Примеры представления целых чисел

Таблица 1.5

Число	Значение	Примечания
10	0...01010	32 разряда
'07	0...00111	32 разряда
1'b1	1	
1'b0	0	
8'Hc5	11000101	
6'hF0	110000	
6'hA	1010	
8'b0	0	
8'b1	1	
8'bx	xxxxxxxx	Усечение старших разрядов
8'b1x	0000001x	Заполнение нулями старших разрядов
8'b0x	0000000x	
8'hx	xxxxxxxx	
8'hzx	zzzzxxxx	
8'hz1	zzzz0001	
8'bx1	xxxxxxxx1	
8'bx0	xxxxxxxx0	
8'hz	zzzzzzzz	
8'h0z	0000zzzz	

2. Для улучшения читабельности в представлении значений больших чисел может использоваться знак подчеркивания («_»), например:

16'b0001_1011_1010_1100 // двоичное 16-разрядное число

При компиляции знак подчеркивания игнорируется, он также не может быть первым значением числа.

3. Число со знаком «минус» представляется в дополнительном коде (дополнения до 2).

1.5.2. Представление действительных чисел

Для представления действительных чисел в языке Verilog имеется два формата: с десятичной точкой и экспоненциальное представление.

Представление с десятичной точкой имеет следующий формат:

value_i.value_f

где *value_i* — целая часть числа; *value_f* — дробная часть числа.

Экспоненциальное представление действительных чисел имеет следующий формат:

baseeexponent или *baseEexponent*,

где *base* — база числа; *exponent* — значение экспоненты.

Замечания.

1. В представлении с десятичной точкой целые числа должны быть с обеих сторон от десятичной точки.

2. В экспоненциальном представлении не должно быть пробелов до и после букв «е» или «Е».

Примеры представления действительных чисел:

0.5

3e4 = 3·10⁴ = 3000

5.8E-3 = 5.8·10⁻³ = 0,0058

1.6. Параллелизм языка Verilog

В отличие от языков программирования язык Verilog допускает параллельное выполнение своих конструкций во время моделирования. К таким конструкциям относятся:

- экземпляры модулей;
- экземпляры примитивов;
- операторы непрерывного назначения (*assign*);
- процедурные блоки (*initial*, *always*).

Замечание. Параллельное выполнение отдельных конструкций языка Verilog может также учитываться синтезатором.

Параллелизм языков проектирования полностью соответствует физической природе электронных схем: при подаче напряжения все компоненты схемы работают одновременно и параллельно, а по цепям схемы одновременно и параллельно передаются электрические сигналы. В языке Verilog экземпляры примитивов и модулей соответствуют компонентам схемы, операторы назначения — проводным соединениям, а процедурные блоки — функциональным блокам цифровых устройств.

Кроме того, параллелизм языка Verilog весьма удобен при моделировании устройств и был использован в листинге 1.5 при описании модуля для моделирования проекта однобитового сумматора.

Г л а в а 2

Модули

2.1. Определение модулей

Как было указано ранее, модуль является основной конструкторской единицей языка Verilog. Все модели цифровых устройств, модели отдельных частей цифровой системы и сама цифровая система представляются в виде модулей.

Формат (нотация) определения модуля имеет следующий вид:

```
module module_name
#(parameter_declaration, parameter_declaration,...)
  (port_declaration port_name, port_name,...,
   port_declaration port_name, port_name,...);
  module items
endmodule
```

В данном формате порты модуля объявляются непосредственно в списке портов модуля, т. е. в круглых скобках. Старый формат допускает объявление портов модуля в теле модуля. Старый формат определения модуля имеет следующий вид:

```
module module_name (port_name, port_name,...);
  port_declaration port_name, port_name,...;
  port_declaration port_name, port_name,...;
  module items
endmodule
```

Определение модуля всегда начинается с ключевого слова **module** и заканчивается ключевым словом **endmodule**.

Замечание. Вместо ключевого слова **module** может использоваться **macromodule** (для обозначения крупных частей системы).

За ключевым словом **module** следует имя модуля *module_name*, определяемое пользователем. За именем модуля в круглых скобках

Модули

следует список параметров модуля. Списку параметров модуля предшествует знак «#» (для его отличия от списка портов модуля). За списком параметров также в круглых скобках следует список портов модуля. Список портов модуля заканчивается точкой с запятой («;»), после чего следуют элементы модуля (*module items*).

Список параметров является необязательным элементом объявления модуля. Если модуль не имеет параметров, то сразу за именем модуля следует список портов модуля.

Список портов также является необязательным элементом модуля. Дело в том, что в процессе проектирования на языке Verilog модули могут использоваться для различных целей, например модули могут описывать модели отдельных частей цифровой системы, саму цифровую систему, модель окружающей среды цифровой системы и др. Отметим, что модуль самого верхнего уровня иерархии в модели взаимодействия цифрового устройства или системы и окружающей среды не имеет портов.

Задание. Приведите примеры цифровых устройств и систем, которые не имеют входов, выходов, внешних выводов.

2.2. Элементы модулей

В качестве элементов модулей (*module items*) могут выступать следующие конструкции языка Verilog:

- объявление типов данных;
- объявление параметров;
- экземпляры (*instances*) модулей;
- экземпляры примитивов;
- блоки генерации кода (*generate blocks*);
- процедурные блоки (*procedural blocks*);
- непрерывные назначения (*continuous assignments*);
- определение задач (*tasks*);
- определение функций;
- блоки спецификаций (*specify blocks*).

В коде проекта элементы модуля могут следовать в произвольном порядке. Единственное исключение — объявление типов данных и параметров должны встречаться раньше их первого использования.

В языке Verilog имеется два основных стиля написания модулей: **поведенческий** (*behavioral*) и **структурный** (*structural*). Поведенческий стиль описания еще называют функциональным или описанием на уровне регистровых передач (RTL). С помощью поведенческого стиля описывается функционирование проекта. Структурный стиль описывает экземпляры модулей и примитивов, а также соединения между ними. Для передачи сигналов между компонентами структуры могут использоваться промежуточные переменные. С помощью

структурного стиля логические, структурные и принципиальные схемы описываются подобно тому, как это делается в графических редакторах. Допускается также смешанный стиль описания модулей: поведенческо-структурный.

Пример использования структурного стиля для описания однобитового сумматора был приведен в листинге 1.1. Поведенческое (функциональное) описание этого же сумматора с помощью оператора непрерывного назначения **assign** на основании логических уравнений приведено в листинге 1.2, а с помощью процедурных операторов — в листинге 1.3. В качестве альтернативы приведем старый стиль объявления модулей, когда порты модуля объявляются в теле модуля.

Листинг 2.1. Описание однобитового сумматора со старым стилем объявления портов

```
module add_1_4 (cin,a, b, s, cout); // объявление заголовка модуля
input cin,a, b; // объявление входных портов
output s, cout; // объявление выходных портов
assign s = a & b & cin;
assign cout = a & b | (a & b) & cin;
endmodule
```

Здесь изменилось только описание портов модуля.

Задание. Сравните описание однобитового сумматора на листингах 1.1 и 2.1. Какой стиль описания вам больше нравится?

2.3. Объявления портов

Нотация объявления портов имеет следующий вид:

port_direction data_type signed range port_name, port_name, ...;

где **port_direction** — направление передачи сигналов; **data_type** — тип передаваемых данных; **signed** — ключевое слово; **range** — спецификация диапазона битового поля; **port_name** — имя порта.

Старая нотация объявления портов имеет следующий вид:

port_direction signed range port_name, port_name, ...;
data_type_declarations

Направление передачи сигналов (**port_direction**) может иметь следующее значение:

input — для входных портов;

output — для выходных портов;

inout — для двунаправленных портов.

В качестве типа передаваемых данных (**data_type**) может использоваться любой тип данных языка Verilog, за исключением типа **real**.

Значение направления передачи сигналов (**port_direction**) ограничивает типы данных, передаваемых через порт. Так, на входные и

дву направленные порты могут быть поданы только сетевые (**net**) типы данных. На выходных портах модуля могут быть сформированы любые типы данных языка Verilog, за исключением типа **real**.

Замечание. Чтобы передать через порт некоторое значение типа **real**, его необходимо предварительно преобразовать с помощью системных задач **\$realtobits** и **\$bitstoreal**.

Ключевое слово **signed** указывает, что значения, передаваемые через порт, интерпретируются как число со знаком (знаковое значение — *signed value*) в дополнительном коде (дополнение до 2-х). Если либо порт, либо тип данных внутреннего сигнала, подсоединеного к порту, объявляются как знаковые значения, то они оба являются знаковыми.

Спецификация диапазона (**range**) битового поля порта имеет следующий формат:

[msb:lsb]

где **msb** — номер наибольшего значащего бита; **lsb** — номер наименьшего значащего бита.

Если спецификация диапазона не задана, то порт имеет ширину одного бита, а передаваемые сигналы считаются скалярными (**scalar**); в противном случае передаваемые сигналы считаются векторными (**vector**).

В качестве номеров наибольшего (**msb**) и наименьшего (**lsb**) битов могут выступать числа, константы, выражения, а также обращение к константной функции (функция, которая возвращает значение константы). Спецификация диапазона допускает как возрастающий (**big-endian**), так и убывающий (**little-endian**) порядок перечисления битов (т. е. **[0:7]** и **[8:1]** — разрешенные записи диапазона).

Минимальный размер битов порта ограничен значением 256 битов. Однако большинство производителей компиляторов языка Verilog допускают число битов порта до 1 миллиона.

Задание. Узнайте максимальный размер диапазона битов порта, который допускает используемый вами компилятор языка Verilog.

Размеры диапазона типа данных и типа порта должны иметь одинаковые значения. В случае их несовпадения некоторые компиляторы принимают размер типа данных без сообщения об ошибке.

Предупреждение. Будьте внимательны с размерами типов данных и типов портов.

Задание. Узнайте, выдает ли используемый вами компилятор сообщение об ошибке в случае не совпадения размеров порта и данных.

Ниже приведены примеры объявления портов:

input a, b, sel; // три скалярных порта

```

input signed [15:0] a,b; /* два 16-битовых порта, которые передают данные в дополнительном коде; порядок битов — little endian */
output signed [31:0] result; /* то же, но 32-битовый порт */
output reg signed [32:1] sum; /* 32-битовый порт, подсоединенные к порту внутренние сигналы имеют знаковый тип данных reg, порядок битов — little-endian */
input [0:15] data_bus; /* двунаправленный 16-битовый порт с порядком битов — big-endian */
input [15:12] addr; /* четыре разряда вектора addr используются в качестве входов (msb и lsb могут иметь любые значения) */
parameter WORD = 32;
input [WORD-1:0] addr; /* в объявлении порта могут использоваться константные выражения */
parameter SIZE=4096;
input [log2(SIZE)-1:0] addr; /* а также вызываться константные функции */

```

Задание. Узнайте стандартные константные функции используемого вами компилятора.

2.4. Экземпляры модулей

Прежде чем использовать модуль в качестве части цифровой системы, следует создать экземпляр (*instance*) модуля. Экземпляр модуля представляет некоторый аналог вызова функции в языках программирования. Но если в языках программирования вызов функции соответствует обращению к части кода программы, то создание экземпляра модуля соответствует установке на плате экземпляра микросхемы определенного номинала (модуля).

Подобно тому, как после установки микросхемы на плате к ее выводам необходимо подсоединить линии с передаваемыми сигналами, при создании экземпляра модуля следует указать сигналы, подсоединяемые к портам модуля.

Имеется два способа указания подсоединения сигналов к портам экземпляра модуля: по местоположению и по имени порта.

Формат создания экземпляра модуля, когда сигналы передаются по местоположению, имеет вид

```
module_name instance_name (signal, signal, ...);
```

где *module_name* — имя модуля; *instance_name* — имя экземпляра модуля; (*signal*, *signal*, ...) — список подсоединяемых сигналов.

В случае, когда к некоторому порту экземпляра модуля сигнал не подсоединяется, в этом месте подряд ставятся две запятые. Этот прием называется пропуск сигнала. Если требуется пропустить несколько сигналов, то в списке сигналов ставится несколько запятых.

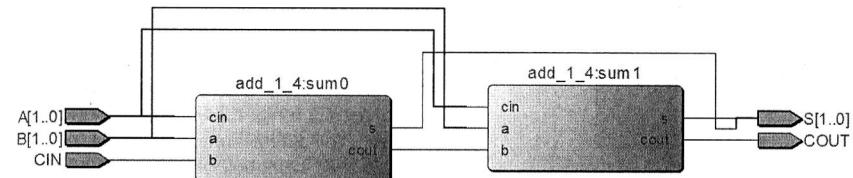


Рис. 2.1. Реализация модуля add_2 2-битового сумматора из листинга 2.2, построенного из двух однобитовых сумматоров

Например, проект 2-битового сумматора из двух экземпляров однобитного сумматора может выглядеть следующим образом:

Листинг 2.2. Описание 2-битового сумматора из двух однобитовых сумматоров

```

module add_2 (input wire [1:0] A, B, input CIN,
           output wire [1:0] S, output COUT);
    wire C0; /* перенос из нулевого разряда */
    add_1_4 sum0 (A[0], B[0], CIN, S[0], C0);
    add_1_4 sum1 (A[1], B[1], C0, S[1], COUT);
endmodule

```

Здесь *add_1_4* — имя модуля однобитового сумматора описанного в листинге 2.1; *sum0* и *sum1* — имена двух экземпляров модуля. Отметим, что в данном проекте для передачи сигнала переноса между двумя экземплярами однобитового сумматора использовалась промежуточная цепь *C0*. Реализация листинга 2.2 приведена на рис. 2.1.

Формат создания экземпляра модуля, когда сигналы передаются по именам портов, имеет вид

```
module_name instance_name (.port_name (signal),
                           .port_name (signal), ...);
```

где *port_name* — имя порта, которому передается соответствующий сигнал.

Замечание. В данном формате каждому имени порта предшествует точка («.»).

Если к некоторому порту экземпляра модуля сигнал не подсоединяется, то имя этого порта не указывается в списке портов экземпляра модуля.

Передача сигналов по имени порта требует больше символов кода, однако освобождает разработчика от заполнения или знания порядка следования портов в объявлении модуля. Кроме того, данный способ уменьшает количество ошибок при написании кода проекта. Поэтому он используется при создании экземпляров модулей с большим количеством портов.

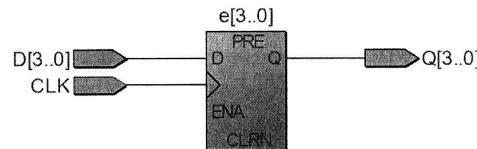


Рис. 2.2. Реализация модулей *reg_4.1* и *reg_4.2* 4-битового регистра из листингов 2.3 и 2.4

В качестве примера рассмотрим проект 4-х битового регистра на основе примитива D-триггера **dff**, порядок следования портов которого нам неизвестен, однако известны имена портов: *d* — вход данных; *q* — выход; *clk* — вход сигнала синхронизации.

Листинг 2.3. Описание 4-битового регистра на основе примитива **dff** D-триггера

```
module reg_4.1 (input wire [3:0] D,
    input wire CLK,
    output wire [3:0] Q);
    dff e0 (.clk(CLK), .d(D[0]), .q(Q[0]));
    dff e1 (.clk(CLK), .d(D[1]), .q(Q[1]));
    dff e2 (.clk(CLK), .d(D[2]), .q(Q[2]));
    dff e3 (.clk(CLK), .d(D[3]), .q(Q[3]));
endmodule
```

В данном примере мы использовали только три порта примитива **dff**: *clk*, *d* и *q*, причем информация о порядке портов, а также названия других портов нам могут быть неизвестны. Реализация листинга 2.3 приведена на рис. 2.2.

Замечание. В общем случае создаваемые экземпляры модулей могут перечисляться через запятую без повторения имени модуля. Примером может служить листинг 2.4.

Листинг 2.4. Альтернативное описание 4-битового регистра

```
module reg_4.2 (input wire [3:0] D,
    input wire CLK,
    output wire [3:0] Q);
    dff e0 (.clk(CLK), .d(D[0]), .q(Q[0])),
    e1 (.clk(CLK), .d(D[1]), .q(Q[1])),
    e2 (.clk(CLK), .d(D[2]), .q(Q[2])),
    e3 (.clk(CLK), .d(D[3]), .q(Q[3]));
endmodule
```

Замечание. Имя экземпляра модуля является необязательным, т. е. можно создавать элементы проекта без указания имен экземпляров модулей, как в примере из листинга 2.5.

Листинг 2.5. Пример создания четырех инверторов без указания имен экземпляров.

```
module wor_wand (input a,b,
```

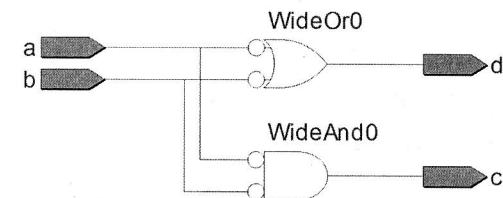


Рис. 2.3. Реализация модуля *wor_wand* из листинга 2.5: пример соединения выходов инверторов с помощью цепей *wor* и *wand*

```
output wand c,
output wor d;
not (c,a), (c,b); // сеть типа wand
not (d,a), (d,b); // сеть типа wor
endmodule
```

Реализация листинга 2.5 приведена на рис. 2.3.

Задание. Опишите два проекта 8-битового сумматора на основе однобитового сумматора, в одном из которых сигналы экземплярам модулей передаются по местоположению (порядку следования), а в другом — по именам портов.

2.5. Параметры

Использование параметров в объявлении модулей и при создании экземпляров модулей позволяет в ряде случаев значительно сократить код проекта, повысить его читабельность, а также уменьшить количество ошибок при написании кода.

В объявлении модуля список параметров модуля начинается со знака «#» и предшествует списку портов модуля (смотри подраздел 2.1). Например, описание параметризированного модуля сумматора может иметь следующий вид.

Листинг 2.6. Проект параметризированного сумматора, где параметр *N* определяет ширину входных слов и результата

```
module add_N #(parameter N = 4)// определение параметра N с
// умалчивающим значением 4
(input [N-1: 0] a, b, input cin, // определение ширины
// входных слов
output [N-1: 0] s, output cout); // определение ширины
// выходных слов
assign {cout, s}= a + b + cin;
endmodule
```

В приведенном примере параметр *N* используется для задания ширины входных слов *a* и *b*, а также суммы *s*. В объявлении модуля можно указывать умалчивающие значения параметров, которые применяются тогда, когда при создании экземпляра модулей данный параметр не переопределяется. Для нашего примера умалчивающее

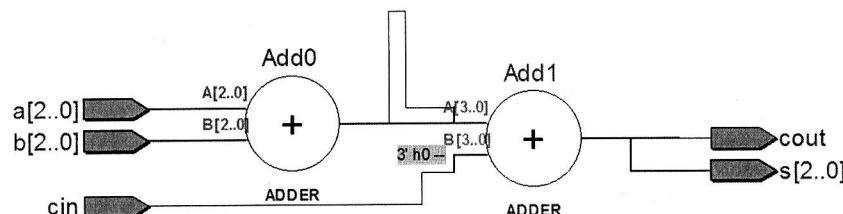


Рис. 2.4. Реализация модуля `add_N` параметризованного сумматора из листинга 2.6: пример использования арифметических функций и операции конкатенации в левой части оператора `assign`

значение параметра N равно 4. Реализация листинга 2.6 приведена на рис. 2.4.

В данном примере также использованы следующие операторы: непрерывного назначения `assign`, конкатенации (склеивания битов и битовых полей) «{ }», а также арифметической суммы «+». Оператор конкатенации в левой стороне оператора `assign` необходим потому, что в результате суммирования может появиться сигнал переноса из старшего разряда суммы.

В общем случае модуль может иметь несколько параметров. При этом в объявлении модуля приводится список параметров, как указано в разделе 2.1.

Создание экземпляра параметризованного модуля также имеет два формата:

```
module_name # (value, value, ...) instance_name (signal, ...);
  module_name # (.parameter_name (value), .parameter_name(value),
  ...) instance_name (signal, ...);
```

где `value` — передаваемое значение параметра для конкретного экземпляра модуля; `parameter_name` — имя параметра.

Как можно заметить, форматы передачи параметров экземплярам модулей практически полностью совпадают с форматами передачи сигналов портам экземпляров модулей. Отличие заключается только в наличии знака «#» перед списком передаваемых значений.

Примеры создания экземпляров модулей:

```
/* Пример создания экземпляра сумматора на 32 бита */
wire [31:0] A, B, S;
wire CIN, COUT;
add_N # (32) sum_32_v1 (A, B, CIN, S, COUT); // передача параметра
                                                // по местоположению
add_N #(N(32)) sum_32_v2 (A,B,CIN,S,COUT); // передача параметра
                                                // по имени
```

В качестве еще одного примера рассмотрим вариант описания устройства вычитания на 16 битов, приведенное в листинге 2.7. Для вы-

полнения операции вычитания здесь используется следующий принцип двоичной арифметики: $A - B = A + \bar{B} + 1$, где \bar{B} означает инверсию всех разрядов двоичного слова B .

Листинг 2.7. Описание вычитающего устройства на 16 битов

```
module sub_16 (input [15:0] A, B,
  output [15:0] D, output COUT);
  wire [15:0] nB = ~B; // инверсия разрядов слова B
  add_N # (16) S_16 (A,nB,1'b1,D,COUT); // экземпляр сумматора
endmodule
```

Задание. Опишите параметризованный модуль вычитания, используя арифметическую операцию вычитания.

2.6. Неявная передача значений параметров

В языке Verilog также возможна неявная передача значений параметров экземплярам модулей. Для этого используется оператор `defparam`, который имеет следующий формат:

```
defparam hierarchy_path.parameter_name =value;
```

где `hierarchy_path` — иерархическое имя экземпляра модуля; `parameter_name` и `value` — имя и значение параметра соответственно.

Преимуществом использования оператора `defparam` является то, что он может записываться в любом месте кода и переопределять параметры любого экземпляра любого модуля.

Предупреждение. Будьте внимательны при использовании оператора `defparam`, поскольку его применение снижает читабельность кода.

Пример использования оператора `defparam`:

```
add_N S_64(Cin, A, B, S, Cout); // создание экземпляра модуля
// add_N с именем S_64
defparam S_64.N = 64; // задание значения параметру N
// экземпляра S_64
```

2.7. Массивы экземпляров модулей

Кроме создания единичных экземпляров модулей язык Verilog позволяет создавать массивы экземпляров модулей (*array of instances*). В этом случае формат создания экземпляров модулей имеет следующий вид:

```
module_name instance_name instance_array_range (signal, signal,...);
  module_name instance_name instance_array_range
  (.port_name (signal), .port_name (signal), ... );
```

где `module_name`, `instance_name`, `port_name` и `signal` — соответственно имя модуля, имя сигнала, имя порта и передаваемый сигнал; `instance_array_range` — диапазон массива экземпляров.

Диапазон массива экземпляров имеет следующий формат:

[left_index : right_index],

где *left_index* и *right_index* – левый и правый индексы диапазона массива. Число создаваемых экземпляров модулей определяется так:

$\text{abs}(\text{left_index} - \text{right_index}) + 1$,

где *abs* – функция, возвращающая абсолютное значение.

Если в массиве битовая ширина порта модуля равняется ширине подсоединеного к порту сигнала, то сигнал подсоединяется к каждому экземпляру модуля. Отметим, что данное свойство используется для присоединения управляющих сигналов ко всем экземплярам модуля.

Если битовая ширина порта модуля отличается от ширины подсоединеного к порту сигнала, то порт каждого экземпляра подсоединяется к части сигнала, начиная с наиболее правого индекса экземпляра, подсоединеного к наиболее правой части вектора. Данный процесс распространяется к левой части вектора. Для подсоединения всех экземпляров суммарное число битов портов экземпляров должно совпадать с шириной вектора (размер сигнала и размер порта должны умножаться).

Множественные экземпляры модулей могут также создаваться с помощью блока генерации кода (*generate block*).

Замечание. Механизм массивов экземпляров модулей поддерживается не всеми компиляторами языка Verilog.

Задание. Узнайте, поддерживается ли механизм массивов экземпляров модулей используемым вами компилятором.

Примеры создания массивов экземпляров модулей:

Листинг 2.8. 8-битовый буфер из восьми однобитовых примитивов *bufif0*

```
module buf_8_1 (input [1:8] in, oe,
    output [1:8] out);
    bufif0 b[1:8] (out, in, oe); // создание 8 экземпляров bufif0
endmodule
```

Последний пример равносителен следующему описанию:

```
module buf_8_2 (input [1:8] in, oe,
    output [1:8] out);
    bufif0 (out[1], in[1], oe[1]),
        (out[2], in[2], oe[2]),
        (out[3], in[3], oe[3]),
        (out[4], in[4], oe[4]),
        (out[5], in[5], oe[5]),
        (out[6], in[6], oe[6]),
```

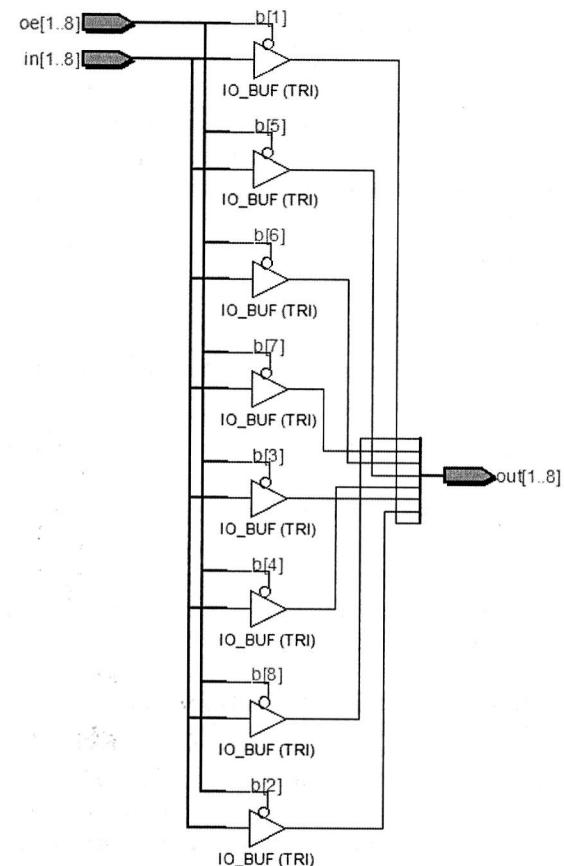


Рис. 2.5. Реализация модуля *buf_8_1* из листинга 2.8: пример построения 8-битового буфера из восьми однобитовых примитивов *bufif0*

```
(out[7], in[7], oe[7]),
(out[8], in[8], oe[8]);
```

endmodule

Реализация листинга 2.8 приведена на рис. 2.5.

Задание. Опишите 8-битовый буфер из восьми однобитовых примитивов *bufif0*, который управляемся одним сигналом разрешения ОЕ.

Еще несколько примеров:

```
/* Пример восьми 8-битовых буферов с тремя состояниями; каждый экземпляр подсоединенится к 64-битовому вектору; общая шина управления en подсоединеняется ко всем экземплярам */
```

```
module tribuf64bit (output wire [63:0] out,
    input wire [63:0] in,
```

```

    input wire enable);
tribuf8bit i [7:0] (out, in, enable);
endmodule
/* Пример 8-битового буфера с тремя состояниями, простроено-
го из восьми однобитовых примитивов bufif1 */
module tribuf8bit (output wire [7:0] y,
    input wire [7:0] a,
    input wire en);
bufif1 u [7:0] (y, a, en);
endmodule

Однако при описании регистров использование механизма созда-
ния массива экземпляров модулей не требуется.

/* Пример 8-битового регистра из восьми однобитовых примити-
вов dff */
module reg_dff_8 (output [7:0] Q,
    input [7:0] D,
    input clk, rst);
DFF reg_8 (.d(D), .clk(clk), .clrn(rst), .q(Q));
endmodule

```

2.8. Иерархия модулей и иерархия имен

Обычно проекты большинства цифровых устройств и цифровых систем представляют собой совокупность модулей. Использование экземпляра одного модуля в пределах другого модуля приводит к созданию иерархии модулей. Модуль, находящийся на вершине дерева иерархии, называется **главным модулем** или **модулем высшего уровня** (*top-level module, highest-level module*). В каждом проекте только один модуль является главным. Компилятор пакета Quartus II автоматически создает графическое изображение дерева иерархии модулей проекта. Примером иерархического проекта может служить 2-битовый сумматор из листинга 2.2.

В языке Verilog число уровней иерархии модулей и взаимосвязи модулей в иерархическом дереве никак не ограничены. Возникают вопросы:

- как обратиться к требуемому модулю из определенной точки в дереве иерархии модулей?
- могут ли различные модули иметь одинаковые имена в различных ветвях дерева иерархии?

Это не праздные вопросы. Ответ на первый вопрос требуется, когда некоторый узел (например, счетчик или компаратор) часто используется в различных частях проекта. Ответ на второй вопрос

становится актуальным, когда необходимо объединить части проекта, разрабатываемые разными проектировщиками.

Ответы на поставленные вопросы дает **иерархическое имя модуля**. В языке Verilog местоположение модуля в иерархии модулей определяется иерархическим именем или иерархическим путем (*hierarchy path*). Различают полный (*full*) и относительный (*relative*) иерархические пути.

Полное имя (полный путь) модуля содержит имя модуля верхнего уровня и любое число имен экземпляров модулей вниз к объекту ссылки. Различные имена экземпляров модулей в иерархическом пути отделяются точкой.

Относительное имя (относительный путь) содержит имя экземпляра модуля в текущем модуле, за которым следует любое число имен экземпляров модулей вниз к объекту ссылки.

Таким образом, модули с одинаковыми именами, но находящиеся в разных ветвях иерархии модулей — это разные модули, поскольку они имеют различные иерархические имена. Далее, в любой точке проекта можно создать экземпляр любого модуля проекта. Для этого достаточно знать его полное иерархическое имя. Чтобы сократить длинные иерархические имена, можно использовать относительные имена модулей.

2.9. Области иерархии и области действия имен

В языке Verilog имеется четыре основных типа областей действия имен:

- область действия глобальных имен;
- область действия определений модулей, функций, задач и именованных блоков;
- область действия блоков спецификации;
- область действия атрибутов.

Глобальные имена являются видимыми во всех областях действия имен. К глобальным именам относятся:

- имена модулей;
- имена примитивов;
- имена определений конфигурации;
- имена макротекстов, созданные системной задачей ‘**define**’.

Замечание. Имена макротекстов становятся видимыми от места их определения в исходном коде. В предшествующей части исходного кода макроимена не видны.

Границы областей действия определений создают новый уровень иерархии. К таким областям относятся:

- определение модулей;

- определение функций;
- определение задач;
- именованные блоки (*named blocks*) **begin-end** и **fork-join**.

Кроме того, свои области действия также определяют блоки спецификации (*specify blocks*) и атрибуты.

Идентификационное имя, определенное в пределах области действия имени, является уникальным по отношению к этой области действия и не может быть переопределено в пределах этой области действия. Ссылки к идентификационным именам в пределах области действия будут вначалеискаться в локальной области действия, а затем вверх по дереву иерархии областей действия до границы модуля.

Г л а в а 3

Примитивы и библиотечные модули

3.1. Где можно найти готовое решение

Прежде чем приступить к разработке устройства или цифровой системы, неплохо ознакомиться с первичными элементами (примитивами) языка Verilog, с первичными элементами используемого компилятора, набор которых обычно значительно шире примитивов языка Verilog, с библиотеками стандартных функциональных узлов, предоставляемых используемой интегрированной средой, а также с открытыми для свободного использования и приобретаемыми за деньги библиотеками готовых проектов на языке Verilog и др.

Этого, конечно, можно не делать, но представьте свое разочарование, когда после многих усилий с вашей стороны окажется, что разрабатываемый вами узел является общедоступным. Кроме того, он значительно превосходит разработанный вами проект по таким параметрам, как стоимости реализации (площадь, занимаемая на кристалле), быстродействие (задержка сигнала на критическом пути или максимальная частота функционирования) и потребляемая мощность.

Все общедоступные компоненты проекта на языке Verilog можно разделить на следующие группы:

- примитивы языка Verilog;
- примитивы компилятора языка Verilog;
- библиотеки примитивов используемой интегрированной среды (в нашем случае пакет Quartus II);
- библиотеки IP-блоков (Intellectual Properties) используемой интегрированной среды;
- общедоступные библиотеки IP-блоков;
- библиотеки IP-блоков других производителей.

Базовые примитивы языка Verilog достаточно бедны, т. е. стандарт языка Verilog представляет очень ограниченный набор примитивов. Положительными качествами языка Verilog является то, что они

могут быть использованы в любом месте проекта без предварительного определения или объявления. Имена примитивов всегда видны во всем дереве иерархии проекта.

Библиотеки примитивов интегрированной среды по использованию аналогичны базовым примитивам языка Verilog: для их применения не требуется никаких усилий со стороны пользователя.

Обычно, кроме библиотеки примитивов, интегрированные программные средства проектирования, подобные пакету Quartus II, предоставляют большой набор уже готовых проектных решений стандартных функциональных блоков, которые спроектированы специалистами высокой квалификации и тщательно протестированы. В документации фирмы Altera можно встретить рекомендацию не пытаться разрабатывать подобные блоки самостоятельно, поскольку лучше все равно не получится. Автор данной книги усомнился в приведенном утверждении и разработал методику проектирования компараторов, которая позволяет строить компараторы большого размера (свыше тысячи битов одного входного слова), а также создавать компараторы по стоимости реализации и быстродействию превосходящие библиотечные решения [23–26].

3.2. Примитивы языка Verilog

Встроенные в языке Verilog примитивы можно разделить на две группы: вентильные и переключательные. Группа вентильных примитивов, кроме собственно вентилей, включает также буферы. Описание вентильных примитивов приведено в табл. 3.1.

Группа переключательных примитивов представляет транзисторы разного типа. Может показаться странным, что примитивами языка проектирования высокого уровня, которым является язык Verilog, являются транзисторы. Однако не следует забывать, что язык Verilog первоначально создавался как язык моделирования, и он должен иметь возможность моделирования элементов цифровой системы на всех уровнях: от транзисторного до структурного. Описание транзисторных примитивов приведено в табл. 3.2.

Замечание. Не все компиляторы языка Verilog поддерживают переключательные примитивы, например в ПЛИС переключательные примитивы не используются.

Таблица 3.1

Вентильные примитивы языка Verilog

Имена примитивов	Описание свойств
and, nand, or, nor, xor, xnor	Один выход, один или более входов
buf, not	Один или более выходов, один вход
bufif0, bufif1, notif0, notif1	Один выход, один вход, один управляющий вход
pullup, pulldown	Один выход, один или более входов

Таблица 3.2
Переключательные примитивы языка Verilog

Имена примитивов	Описание свойств
cmos, rcmos	Один или более выходов, вход управления n, вход управления p
tran, rtran	Два двунаправленных вывода
tranif0, tranif1, rtranif0, rtranif1	Два двунаправленных вывода, один управляющий вход

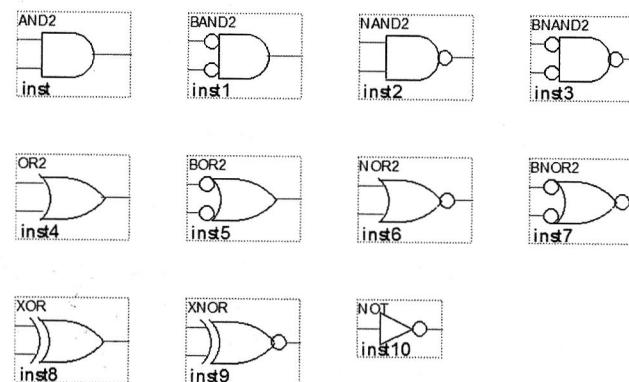


Рис. 3.1. Логические примитивы пакета Quartus II

Задание. Узнайте, поддерживает ли используемый вами компилятор переключательные примитивы.

В пакете Quartus II примитивы pullup и pulldown, а также все переключательные примитивы не поддерживаются. Однако в то же время имеется достаточное количество логических и буферных примитивов, приведенных на рис. 3.1 и 3.2 соответственно.

Форматы для создания экземпляров вентильных и переключательных примитивов имеют следующий вид:

```
gate-type (drive_strength) # (delay) instance_name
[instance_array_range] (terminal, terminal, ...);
switch_type # (delay) instance_name
[instance_array_range] (terminal, terminal, ...);
```

В приведенных форматах конструкции *delay*, *strength*, *instance_name* и *instance_array_range* являются необязательными.

Замечание. Порты в определении экземпляра примитива называются терминалами (*terminals*).

Конструкция *delay* представляет задержку прохождения сигналов через примитив. По умолчанию задержка равна 0. Для представления задержек могут использоваться целые и действительные числа, которые определяют количество единиц времени задержки (размер

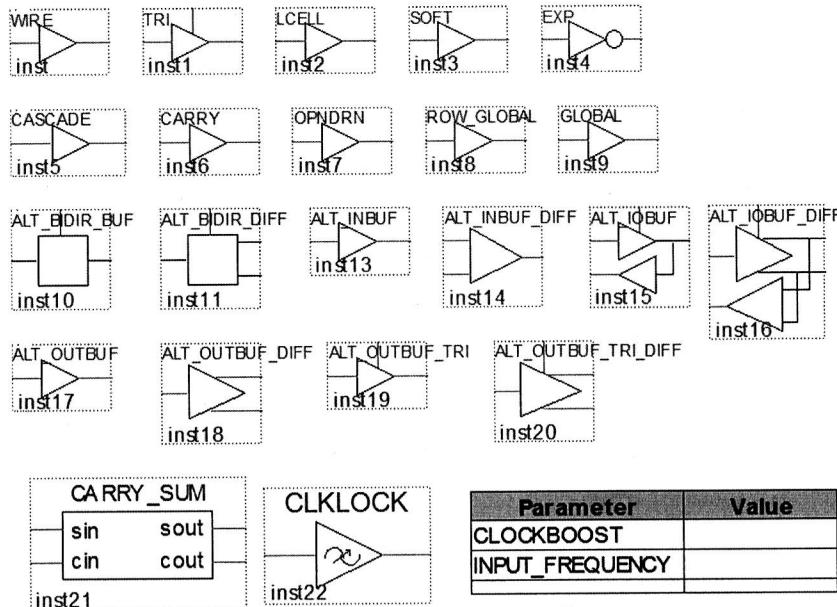


Рис. 3.2. Буферные примитивы пакета Quartus II

одной единицы времени определяется системной директивой ‘timescale’).

Задержка может задаваться одним, двумя или тремя значениями в виде:

```
# del          // определение задержки одним значением
# (del10, del01) // определение задержки двумя значениями
# (del10, del01, del2) // определение задержки тремя значениями
```

где *del* — значение задержки для всех переключений выходов; *del10* — значение задержки при переключении выходов с 1 в 0; *del01* — значение задержки при переключении выходов с 0 в 1; *del2* — значение задержки при переключении выходов в высокоимпедансное состояние.

Каждое из значений *del*, *del10*, *del01* и *del2* могут представляться либо одним числом, либо иметь следующий формат:

min : *typ* : *max*

где *min* — минимальное, *max* — максимальное, *typ* — типичное значение задержки.

Примеры определения задержек:

#3 — задержка в 3 временные единицы для всех переключений выходов;

#2:3:4 — минимальная задержка 2, максимальная 4 и типичная 3 временных единицы для всех переключений выходов;

#(5,6) — задержка выходов при переключении с 1 в 0 равна 5, а при переключении с 0 в 1 равна 6 временным единицам;

#(2:3:4, 3:4:5, 4:5:6) — задержка выходов при переключении с 1 в 0 определяется форматом 2:3:4, при переключении с 0 в 1 определяется форматом 3:4:5, а при переключении в высокоимпедансное состояние — форматом 4:5:6.

Конструкция *strength*, определяющая мощность источника сигнала, может иметь следующие форматы:

(*strength1*, *strength0*) // при переключении выхода с 1 в 0

(*strength0*, *strength1*) // при переключении выхода с 0 в 1

где *strength0* и *strength1* могут быть любым из ключевых слов, которые приведены в табл. 1.4 и представляют уровни мощности источников логических сигналов.

Мощность источника сигнала может определяться только для вентильных примитивов. Переключательные примитивы передают уровень мощности входного сигнала на выход.

Примеры определения мощности источника сигнала:

(*supply1*, *pull0*) — определяет мощность источника сигнала при переключении из 1 в 0 следующим образом: *supply1* — для единичного значения и *pull0* — для нулевого значения;

(*supply0*, *pull1*) — определяет мощность источника сигнала при переключении из 0 в 1 следующим образом: *supply0* — для нулевого значения и *pull1* — для единичного значения;

Имя экземпляра примитива (*instance_name*) может использоваться для обращения к определенному примитиву в средствах отладки, при конфигурировании во временных диаграммах и др.

Конструкция диапазона массива экземпляров *instance_array_range* позволяет одновременно определять несколько экземпляров примитивов, каждый из которых подсоединяется к определенному биту вектора. Диапазон имеет следующий формат:

[*left_index* : *right_index*]

где *left_index* и *right_index* — левый и правый индексы диапазона.

Экземпляры примитива начинают подсоединяться от наиболее правого индекса примитива к наиболее правым битам вектора. Каждому экземпляру множества примитивов присваивается свой индекс экземпляра. Примитивы начинают последовательно подсоединяться к вектору, начиная от правого индекса к левому, а также к битам вектора начиная от правого бита к левому. Ширина битового вектора должна равняться числу элементов массива экземпляров примитива.

Если вместо вектора используется скаляр (единственный сигнал), то он подсоединяется ко всем экземплярам массива.

Замечание. Конструкция *instance_array_range* поддерживается не всеми компиляторами языка Verilog.

Задание. Проверьте, поддерживает ли используемый вами компилятор конструкцию *instance_array_range*.

Замечание. Если ваш компилятор не поддерживает конструкцию (*instance_array_range*), не отчаивайтесь. Несколько экземпляров примитива могут также создаваться с помощью блока генерации (*generate block*).

Примеры создания экземпляров примитивов:

```
and i1 (out, in1, in2); // 2-входовой вентиль И с 0-й задержкой
and # 5 (O, i1, i2, i3, i4); /* 4-входовой вентиль И с задержкой
    сигналов 5 временных единиц на переходах
    по всем выходам */
not # (2,3) u7 (out, in); /* инвертор, для которого определены две
    различные задержки: при переключении с
    1 в 0 — 2 единицы времени, а с 0 в 1 — 3*/
buf (pull0,strong1) (y, a); /* буфер с различной мощностью сигналов
    для нулевого (pull0) и единичного (strong1)
    значений */
wire [31:0] y, a; // определение 32-битовых векторов (шин)
buf # 2.7 i [31:0] (y, a); /* создание массива 32 буферных примитивов
    i0, i1, ... i31, подсоединеных к шинам
    y и a; задержка для каждого элемента
    составляет 2,7 временных единиц */
```

3.3. Примитивы, определяемые пользователем

Если предопределенных примитивов в языке Verilog недостаточно, пользователь может создать свои собственные примитивы. Для этого в языке Verilog предусмотрен специальный механизм, называемый определением пользовательских примитивов (*user defined primitives* — UDPs). Согласно механизму UDP формат для определения примитива имеет следующий вид:

```
primitive primitive_name
  (output reg = logic_value terminal_declaration,
   input terminal_declarations);
  table
    table_entry;
    .....
    table_entry;
  endtable
endprimitive
```

где **primitive**, **output**, **reg**, **input**, **table**, **endtable**, **endprimitive** — ключевые слова; *primitive_name* — имя примитива; *logic_value* — логическое значение; *terminal_declaration* — определение выходного терминала (порта); *terminal_declarations* — определения входных терминалов (портов); *table_entry* — строка таблицы истинности.

Старый стиль определения примитива имеет следующий вид:

```
primitive primitive_name (output, input, input,...);
  output terminal_declaration;
  input terminal_declarations;
  reg output_terminal;
  initial output_terminal = logic_value;
  table
    table_entry;
    .....
    table_entry;
  endtable
endprimitive
```

Замечание. Не все компиляторы языка Verilog поддерживают механизм определения пользовательских примитивов.

Задание. Узнайте, поддерживает ли используемый вами компилятор механизм определения пользовательских примитивов.

Все терминалы (порты) в определении примитива должны иметь ширину в один бит. Для примитива допускается только один выход, который должен быть первым в списке параметров. Максимальное число входов для комбинационных примитивов — 10, последовательностных — 9. Функционирование примитива задается с помощью таблиц истинности (*table_entry*). Для определения значений в таблице истинности могут использоваться логические уровни 0, 1 и X, при этом значение Z трактуется как X.

Конструкция *reg = logic_value* является необязательной, она используется для определения начального состояния (после включения питания) выхода последовательностных примитивов.

Одна строка таблицы истинности определяет одно значение выхода для одного набора значений входов. Формат строки таблицы истинности для комбинационных примитивов имеет вид

input_logic_values : output_logic_value;

и для последовательностных примитивов

input_logic_values : previous_state : output_logic_value;

где *input_logic_values* — набор входных значений; *output_logic_value* — значение выхода; *previous_state* — для последовательностных примитивов значение предыдущего состояния выхода.

Входные значения в строке таблицы истинности отделяются белыми знаками, причем они должны следовать в том же порядке, в котором они следуют в определении входных портов. Если какая-либо комбинация входных значений в таблице истинности не определена, то для этой комбинации значение выхода равно X.

Только один из входных сигналов (выполняющий роль синхросигнала для последовательностных примитивов) может иметь переходные значения входного сигнала. Если такой входной терминал имеется, то он должен определить переходные значения для всех строк таблицы истинности.

Если переходное значение определяется для одного входа, примитив становится чувствительным к переходам на всех входах. Следовательно, все другие входы должны иметь строки таблицы истинности для покрытия переходов. Если какой-либо переход не будет найден, то для данного перехода примитив на выходе будет иметь значение X. Уровни чувствительности входов имеют преимущество над переходными значениями в таблице истинности.

Для определения значений сигналов в таблице истинности могут использоваться следующие символы:

0 — логический ноль на входе или выходе;

1 — логическая единица на входе или выходе;

x или X — неизвестное значение на входе или выходе;

- — не известное значение выхода (для последовательностных примитивов);

? — неопределенное значение (*don't care*); для входа это может быть 0,1 или X;

b или B — неопределенное значение (*don't care*); для входа это может быть 0 или 1;

(VW) — изменение значения (переход) входного сигнала с V на W, например (01) представляет переход с 0 на 1;

g или R — возрастающий переход входа, то же самое, что (01);

f или F — падающий переход входа, тоже самое, что (10);

r или P — позитивный (положительный) переход входа: (01), (0X) или (X1).

n или N — негативный (отрицательный) переход входа: (10), (1X) или (X0);

* — любой переход входа (любое изменение значения входного сигнала), то же самое, что (??).

Примеры пользовательских примитивов представлены в листингах 3.1 и 3.2.

Листинг 3.1. Пример комбинационного примитива, заданного пользователем

```
/* пример двухвходового мультиплексора */
primitive my_mux (y, a, b, sel);
  output y;
  input sel, a, b;
  table
    // a b sel : y
    0 ? 0 : 0;      // выбирается a, значение b неопределено
    1 ? 0 : 1;      // выбирается a, значение b неопределено
    ? 0 1 : 0;      // выбирается b, значение a неопределено
    ? 1 1 : 1;      // выбирается b, значение a неопределено
  endtable
endprimitive
```

Листинг 3.2. Пример последовательностного примитива, заданного пользователем:

```
primitive my_dff (output reg q = 0,
  input d, clk, rst);
  table
    // d clk rst : state : q
    ? ? 0 : ? : 0; //брос по низкому уровню rst
    0 R 1 : ? : 0; //на возрастающем фронте состояния
                     // выхода принимает значение входа
    1 R 1 : ? : 1; //на возрастающем фронте состояния
                     // выхода принимает значение входа
    ? N 1 : ? : -; //игнорирование падающего фронта
    * ? 1 : ? : -; //игнорирование всех фронтов на входе d
    ? ? P : ? : -; //игнорирование положительного фронта
                     // на rst
    0(0X)1 : 0 : -; //устранение дребезга
    1(0X)1 : 1 : -; //устранение дребезга
  endtable
endprimitive
```

Вид примитивов my_mux и my_dff на уровне регистровых передач приведен на рис. 3.3 и 3.4 соответственно.

В листинге 3.3 приведен пример использования пользовательских примитивов.

Листинг 3.3. Использование пользовательских примитивов

```
module user_primitives (input a, b, c, clk, clrn,
  output y);
  wire r;           //промежуточная переменная
  my_mux g1(r,a,b,c); //пользовательский мультиплексор
  my_dff g2(y,r,clk,clrn); //пользовательский триггер
endmodule
```

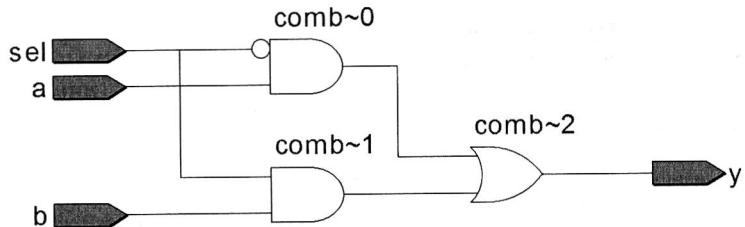


Рис. 3.3. Реализация пользовательского примитива my_mux из листинга 3.1

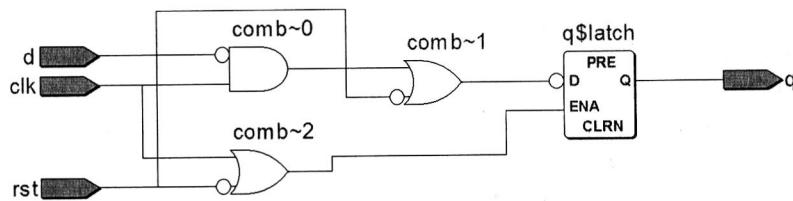


Рис. 3.4. Реализация пользовательского примитива my_dff из листинга 3.2

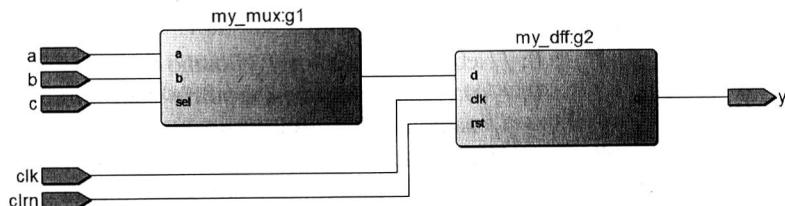


Рис. 3.5. Реализация модуля user_primitives из листинга 3.3

Вид проекта из листинга 3.3 на уровне регистровых передач приведен на рис. 3.5.

Г л а в а 4

Типы данных

4.1. Два класса типов данных

В языке Verilog имеется два класса основных типов данных: сети (*net*) и переменные (*reg*).

Сетевые типы данных (*net*) используются для выполнения соединений между частями проекта. Сети отражают значение и уровень мощности источников сигналов, а также значение емкости, однако сети не имеют своего собственного значения. В то же время сети выполняют функцию определения значения сигнала в случае, когда имеется несколько источников сигналов.

Типы данных переменные (*reg*) используются для временного хранения данных. Определенные значения переменным могут назначаться в процедурных блоках *initial* и *always*, а также в теле задач и функций. Поэтому, если, например, значение выходного порта назначается внутри процедурного блока *always*, то он должен иметь тип *reg*, даже если реализуется комбинационная схема.

Замечание. Здесь «задача» — это конструктивная единица языка Verilog, подробно будет рассматриваться позже.

Переменные могут хранить только логические значения, они не могут запоминать логическую мощность. Переменные не инициализируются по умолчанию. В начале симуляции они имеют значение **X**, поэтому для использования переменных им должно быть присвоено определенное значение.

Правила использования типов данных сети и переменные:

- 1) когда сигнал управляется выходом модуля, выходом примитива или непрерывным назначением (*assign*), используйте сетевые типы данных (*net*);
- 2) когда сигналу значение назначается в процедурном блоке (*initial* или *always*), используйте типы данных из класса переменных (*reg*).

4.2. Сетевые типы данных

Сетевые типы данных служат для соединения структурных компонент проекта. Типы данных *net* используются в следующих случаях:

- когда сигнал управляется выходом экземпляра модуля или примитива;
- когда сигнал подсоединяется к входному или двунаправленному порту модуля, в котором он объявляется;
- когда сигнал находится в левой части оператора непрерывного назначения (**assign**).

Объявления данных типа *net* (сети) имеют три формата:

```
net_type signed [range] #(delay) net_name [array],...;  
net_type (drive_strength) signed [range] #(delay) net_name=  
continuous_assignment;
```

```
trireg (capacitance_strength) signed [range] #(delay, decay_time)  
net_name [array],...;
```

Первый формат используется при объявлении списка и массивов цепей одного типа. Второй формат используется при объявлении цепи вместе с неявным оператором непрерывного назначения. Третий формат используется для объявления цепей типа **trireg**.

В данных форматах конструкции *signed*, *[range]*, *delay*, *[array]*, *(strength)*, *decay_time* являются необязательными (*optional*).

Здесь *net_type* является одним из следующих ключевых слов:

wire — соединение типа цепь, значение сигнала определяется по правилам КМОП (CMOS)-технологии;

wor — соединение выходов по ИЛИ (OR), значение сигнала определяется по правилам эмиттерно-связанной логики ЭСЛ (ECL);

wand — соединение выходов по И (AND), значение сигнала определяется по правилам логики открытого коллектора (open-collector);

supply0 — логическая константа 0, логическая мощность питания;

supply1 — логическая константа 1, логическая мощность питания;

tri0 — в высокоимпедансном состоянии подтягивание к низкому уровню;

tri1 — в высокоимпедансном состоянии подтягивание к высокому уровню;

tri — то же самое, как **wire**;

trior — то же самое, как **wor**;

triand — то же самое, как **wand**;

trireg — в высокоимпедансном состоянии удерживает последнее значение, логическая мощность сигнала емкостная.

Ключевое слово **signed** указывает, что значение интерпретируется как целое число со знаком в дополнительном коде (дополнение до 2-х). Если либо порт, либо подсоединяемая к порту сеть объявлены как знаковые, то оба они будут знаковыми.

Конструкция *[range]* определяет диапазон битов сигнала в формате *[msb:lsb]*. Если в объявлении конструкция диапазона отсутствует, то ширина цепи по умолчанию принимается равной 1 биту. Параметры *msb* и *lsb*, определяющие наиболее значащий и наименее значащий биты диапазона, могут быть числами, константами, выражениями или вызовами константных функций. Допускается указание границ диапазона как по возрастанию (*big-endian*), так и по убыванию (*little-endian*) индексов. Максимальная ширина диапазона ограничена значением $2^{16} = 65\,536$ битов. Однако многие компиляторы допускают ширину диапазона до 1 миллиона битов.

Задание. Узнайте, какую максимальную ширину диапазона сетевых переменных допускает используемый вами компилятор.

Конструкция *delay* (задержка) может задаваться только для сетевых типов данных. Синтаксис конструкции *delay* соответствует синтаксису простых задержек (рассмотрен в разделе 3.2).

Конструкция *[array]* предназначена для объявления массивов и имеет следующий формат:

```
[first_address:last_address] [first_address:last_address]...,
```

где каждые квадратные скобки определяют одну размерность массива.

Массив может иметь любое число размерностей. Каждая размерность массива определяется диапазоном адресов. Элементы *first_address* и *last_address* могут быть числами, константами, выражениями или вызовами константных функций. Значения адресов могут задаваться как по возрастанию, так и по убыванию. Максимальный диапазон одной размерности ограничен значением $2^{24} = 16\,777\,216$, но многие компиляторы не ограничивают максимальное значение диапазона.

Задание. Узнайте максимальный размер диапазона массивов для вашего компилятора.

Конструкция *(strength)* определяет уровень логической мощности сигнала и имеет следующий формат:

```
(strength0, strength1)
```

где *strength0* и *strength1* — числа от 0 до 7, определяющие логическую мощность сигнала при значениях сигнала 0 и 1 соответственно (логическая мощность сигналов была рассмотрена в разд. 1.4.2).

Замечание. Во втором формате конструкция (*drive_strength*) определяет логическую мощность источника сигнала, а в третьем формате конструкция (*capacitance_strength*) определяет емкостную мощность цепи.

Конструкция *decay_time* определяет количество времени, в течение которого сеть *triereg* будет помнить (удерживать) значение сигнала после того, как все источники сигналов будут отключены, т. е. их выходы переведены в высокоимпедансное состояние. По истечении этого времени сигнал примет значение **X**.

Конструкция *decay_time* имеет следующий формат:

(*rise_delay*, *fall_delay*, *decay_time*).

где *rise_delay* — задержка нарастающего фронта сигнала; *fall_delay* — задержка падающего фронта сигнала; *decay_time* — время затухания сигнала. Параметры *rise_delay*, *fall_delay* и *decay_time* представляют собой положительные целые десятичные числа, выражющие временной интервал количеством временных единиц (значение временной единицы определяется с помощью системной директивы ‘timescale’).

Кроме того, непосредственно сразу за ключевым словом, определяющим тип сети (*net_type*) может следовать ключевое слово **vector-ed** или **scaler-ed**, которые указывают на векторный или скалярный характер сигнала. Если тип данных определен как векторный, то компилятор разрешает доступ к битам вектора.

Примеры объявлений сетей:

```
wire a, b, c;          /* три 1-битовые скалярные сети */
tri1 [7:0] date_bus;  /* 8-битовая сеть, в третьем состоянии
                        подтягивается к верхнему уровню */

wire signed [1:8] result; /* 8-битовая знаковая сеть */
wire [7:0] Q[0:15][0:256]; /* 2-мерный массив 8-битовых цепей */
wire #(2.4, 1.8) carry; /* сеть с заданными задержками воз-
                        растания (2,4) и понижения (1,8)
                        уровня сигнала */

wire [0:15] (strong1, pull0) sum = a + b; /* 16-битовая сеть с
                                            заданной логической мощностью
                                            источника и неявным оператором
                                            непрерывного назначения */

triereg (small) #(0,0,35) ran_bit; /* цепь с логической мощностью
                                         малой емкости и временем хранения
                                         сигнала 35 временных единиц */
```

4.3. Значение сигнала сети

Значение сигнала сети определяется с помощью следующих типов цепей: **wire (tri)**, **wand (triand)**, **wor (trior)**, **tri0** и **tri1**. Когда

Таблица 4.1
Значение сигнала в различных цепях для двух источников сигналов

Значения источников		Значение сигнала в сети				
a	b	wire (tri)	wand (triand)	wor (trior)	tri0	tri1
0	0	0	0	0	0	0
0	1	X	0	1	X	X
0	Z	0	0	X	0	0
0	X	X	0	X	X	X
1	1	1	1	1	1	1
1	Z	1	X	1	1	1
1	X	X	X	1	X	X
Z	Z	Z	Z	X	0	1
Z	X	X	X	X	X	X
X	X	X	X	X	X	X

соединяются два источника сигналов, то результирующее значение сигнала в цепи зависит от значений на выходах источников сигналов, а также от типа соединяющей цепи. Результаты соединения двух источников сигналов с помощью цепей различных типов приведены в табл. 4.1.

Необходимость в определении значения сигнала в цепи возникает тогда, когда цепь имеет несколько источников сигнала. В качестве примера рассмотрим соединения выходов буферов и инверторов с помощью различных типов цепей, представленных в листинге 4.1.

Листинг 4.1. Соединения различных типов цепей

```
module net_connects(
    input [1:14] x,
    input [1:6] c,
    output wire y1,
    output wor y2,
    output wand y3,
    output tri0 y4,
    output tri1 y5,
    output triereg y6,
    output supply0 y7,
    output supply1 y8);
not //g1(y1,x[1]), // запрещено соединение
//g2(y1,x[2]), // выходов типа wire
g3(y2,x[3]),
g4(y2,x[4]),
g5(y3,x[5]),
g6(y3,x[6]);
bufif1 g7(y4,x[7],c[1]),
g8(y4,x[8],c[2]);
```

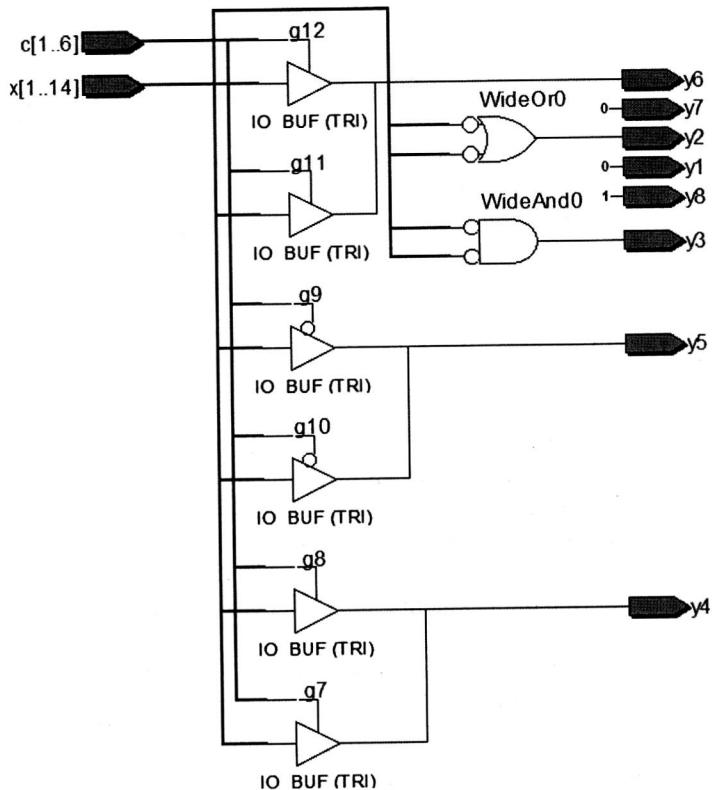


Рис. 4.1. Реализация примеров соединений выходов вентилей с помощью цепей различного типа из листинга 4.1: y1 — wire; y2 — wor; y3 — wand; y4 — tri0; y5 — tri1; y6 — trireg; y7 — supply0; y8 — supply1

```

bufif0 g9(y5,x[9],c[3]),
        g10(y5,x[10],c[4]);
bufif1 g11(y6,x[11],c[5]),
        g12(y6,x[12],c[6]);
endmodule

```

Реализация листинга 4.1 показана на рис. 4.1.

Задание. Проанализируйте листинг 4.1 и рис. 4.1, объясните результаты синтеза.

4.4. Типы данных переменные

Типы данных переменные (**reg**) используются для запоминания значений в процедурных блоках. Переменные запоминают только логические значения, они не могут запоминать значение логической мощности сигналов. Когда сигнал находится в левой части процедур-

ного оператора назначения **assign**, то всегда должен использоваться тип данных **reg**. В старых версиях языка Verilog переменные назывались регистрами (**registers**).

Объявления типов данных **reg** имеют следующие форматы:

```

variable_type signed [range] variable_name, variable_name, ... ;
variable_type signed [range] variable_name = initial_value, ... ;
variable_type signed [range] variable_name [array], ... ;

```

Первый формат используется для объявления списка переменных одного типа. Второй формат используется при объявлении переменных вместе с их начальными значениями. Третий формат используется для объявления массивов переменных.

В данных форматах конструкции **signed**, **[range]**, **[array]** и **initial_value** являются необязательными.

Конструкция **variable_type** определяет тип переменной и может задаваться одним из следующих ключевых слов:

reg — переменная, занимающая в памяти заданное число битов, со знаком (знаковая), если задана конструкция **signed**, или без знака (беззнаковая);

integer — 32-битная переменная со знаком;

time — 64-битная беззнаковая переменная;

real — переменная с плавающей точкой двойной точности;

realtime — то же самое, что **real**.

Конструкция **signed** может использоваться только с переменными типа **reg**; указывает, что значение интерпретируется как целое число со знаком в дополнительном коде (дополнения до 2).

Конструкция **[range]** определяет диапазон битов для переменных типа **reg**. В случае отсутствия конструкции **[range]** переменная **reg** трактуется как 1-битовая переменная. Формат диапазона точно такой же, как в определении сетевых данных. Максимальный размер диапазона для переменных **reg** равен $2^{16} = 65\,536$ битов. Некоторые компиляторы имеют максимальный размер диапазона равный 1 миллиону битов.

Конструкция для задания массивов **[array]** имеет точно такой же формат, как и для сетевых типов данных. Одномерный массив переменных типа **reg** трактуется как память (**memory**).

Конструкция **initial_value** устанавливает начальное значение переменной. Начальное значение переменной устанавливается в нулевом моменте времени симуляции точно так же, как если бы значение переменной назначалось в процедуре **initial**. Если начальное значение не задано, то переменные типа **reg**, **integer** и **time** имеют значение **X**, а переменные типа **real** и **realtime** — **0.0**.

Кроме того, сразу за ключевым словом **reg** могут следовать ключевые слова **vectored** или **scalared**, которые имеют точно такое же значение, как для сетевых типов данных.

Примеры объявления переменных:

```
reg a, b, c;           // три скалярные однобитовые переменные
reg signed [7:0] d1, d2; // две 8-битовые переменные со знаком
reg [7:0] Q[0:3][0:15]; // 2-мерный массив 8-битовых переменных
integer i, j;          // две знаковые переменные целого типа
real r1, r2;            // две действительные переменные двойной
                        // точности
reg clock=0, reset=1;   // две переменные типа reg с начальными
                        // значениями
```

4.5. Другие типы данных

Кроме рассмотренных типов данных, в языке Verilog имеются следующие типы данных: **parameter**, **localparam**, **specsparam**, **genvar** и **event**. Эти типы данных являются не основными и выполняют вспомогательные функции.

4.5.1. Параметры

Тип данных **parameter** позволяет задавать для модулей определенные значения констант для чисел, времени и задержек (*delays*), а также задавать строку символов. Для каждого экземпляра модуля значение параметров может переопределяться (задаваться заново).

Форматы типа данных **parameter**:

```
parameter signed [range] constant_name =value, ...;
parameter constant_type constant_name =value, ....
```

В этих форматах конструкции **signed**, **[range]** и **constant_type** являются необязательными. Конструкции **signed** и **[range]** имеют прежние значения, они определяют числа со знаком и битовый диапазон соответственно. Если при определении константы не задан диапазон **[range]**, то в памяти для представления константы будет выделено минимальное количество битов, достаточное для представления инициализирующего значения.

Конструкция **constant_type** определяется следующими ключевыми словами: **integer**, **time**, **real** и **realtame**. Константа, объявленная с указанным типом данных, будет иметь те же самые свойства, что и переменная этого типа. Если тип данных для константы не определен, то по умолчанию константа будет приведена к типу данных последнего присвоенного значения.

Примеры определения параметров:

```
parameter integer period = 10; // константа целого числа
```

```
parameter [2:0] s1 = 3'b001, // три 3-битовые константы, данный
                      // стиль используется при задании
                      // кодов внутренних состояний
                      // конечных автоматов
```

4.5.2. Локальные параметры

Тип данных **localparam** позволяет задавать значение локальных параметров, действующих только внутри модуля. Локальные параметры не могут переопределяться при создании экземпляра модуля.

Форматы типа данных **localparam**:

```
localparam signed [range] constant_name =value, ...;
localparam constant_type constant_name =value, ....
```

В этих форматах конструкции **signed**, **[range]** и **constant_type** являются необязательными и имеют прежние значения.

Пример:

```
localparam signed offset = -5; /* безразмерная знаковая константа по умолчанию будет занимать ширину значения инициализации */
```

4.5.3. Параметры блока спецификации

Тип данных **specsparam** позволяет задавать значение констант для блока спецификации (будет рассмотрен в главе 14). Тип данных **specsparam** может объявляться в области действия модуля или в области действия блока спецификации. Значение констант может быть переопределено с помощью файлов SDF или PLI.

Формат типа данных **specsparam**:

```
specsparam constant_name =value, ....
```

Пример:

```
specsparam delay_1 = 10; // задание константе delay_1 значения 10
t_2 = 3 : 4 : 6 ;           // задание t_2 значения задержки в виде
                           // строки символов
```

4.5.4. Переменные генерации

Тип данных **genvar** определяет переменную, которая используется исключительно внутри цикла генерации кода (*generate loop*) в блоках генерации (блоки генерации рассматриваются в главе 12). Эта переменная используется компилятором для создания повторяющихся частей кода и не может быть использована во время симуляции.

Формат типа данных **genvar**:

```
genvar event_name, ....
```

Пример:

```
genvar i; // объявление переменной генерации i
```

4.5.5. Тип данных событие

Тип данных **event** (событие) определяет флаги, которые могут использоваться для синхронизации параллельных процессов в пределах модуля.

Формат типа данных **event**:

```
event event_name, ....
```

Пример:

```
event sync_a, sync_b ; // объявление переменных типа события
                      // sync_a и sync_b
```

4.5.6. Строки

Строки в языке Verilog, в отличие от языков программирования, не относятся к основным типам данных. Как правило, строки не поддерживаются синтезаторами, а используются для вывода текстовых сообщений при моделировании. Строки также используются в качестве аргументов в некоторых системных задачах и функциях.

Строчкой в языке Verilog является последовательность символов, заключенная в кавычки («...») и располагающаяся в одной строке кода. Строки, используемые в качестве операндов в выражениях и назначениях, должны трактоваться как беззнаковые целые константы, представленные последовательностью 8-битовых значений знаков в коде ASCII.

В языке Verilog строка объявляется как переменная типа **reg**, размер которой равен числу символов, умноженному на 8, например:

```
reg [12*8:1] str;
initial begin
    str = «Hello, world!»;
end
```

Все действия со строками осуществляются с помощью операторов языка Verilog. Если строка меньше размера переменной, то она слева заполняется нулями. Если строка больше размера переменной, то символы слева отбрасываются.

Пример вывода строки символов представлен в листинге 4.2.

Листинг 4.2. Тестовый пример для работы со строками

```
module str_test;
reg [14*8:1] str;
initial begin
    str = "Hello, world!";
    $display("%s is stored as %h", str, str);
    str = {str,"!!!"};
    $display("%s is stored as %h", str, str);
end
```

```
end
endmodule
```

Задание. Выполните пример из листинга 4.2 для проверки, поддерживает ли ваш компилятор работу со строками.

4.6. Выбор битов и битовых полей

Выбор или обращение к одному биту вектора имеет следующий формат:

```
vector_name [bit_number]
```

где **vector_name** — имя вектора, **bit_number** номер выбираемого бита.

Номер выбираемого бита (**bit_number**) может быть целым числом, константой, сетью, переменной или выражением.

Замечание. Битовым полем называется непрерывная последовательность битов некоторого вектора.

Выбор битового поля имеет следующие три формата:

```
vector_name [bit_number :bit_number];
vector_name [starting_bit_number +: port_select_width];
vector_name [starting_bit_number -: port_select_width],
```

где **vector_name** и **bit_number** имеют прежнее значение; **starting_bit_number** — номер начального бита; **port_select_width** — ширина выбираемой части битов.

Первый формат практически совпадает с нотацией диапазона порта, здесь указываются два крайних номера битов, определяющих битовое поле. Второй и третий форматы позволяют определить битовое поле по номеру начального бита и ширине битового поля. При этом во втором формате номера битов поля по отношению к начальному биту увеличиваются, а в третьем формате — уменьшаются.

Номера битов (**bit_number** или **starting_bit_number**) в приведенных форматах должны быть числами или константами. Ширина выбираемой части вектора (**port_select_width**) может быть числом, константой или вызовом константной функции. При выборе битового поля порядок номеров битов должен соответствовать порядку битов в объявлении вектора, т. е. если наименьший значащий бит имеет наименьший номер в объявлении, то наименьший значащий бит в выбираемой части также должен иметь наименьший номер.

Примеры:

```
wire [7:0] data; // цепь на 8 битов (шина из 8 линий)
data [7:3]; /* примеры правильного выбора битового поля */
data [7:-5];
data [3:7]; /* примеры неправильного выбора битового поля */
data [3:+5];
```

Задание. Проверьте правильность или неправильность приведенных выше примеров с помощью компилятора.

4.7. Выбор элементов массива и битовых полей элементов массива

Формат выбора элемента массива:

`array_name [index][index] ...`

где `array_name` — имя массива; `index` — номер индекса, определяющий выбираемый элемент.

Примеры выбора элементов массива:

```
wire [7:0] my_array [0:31] [0:63]; /* объявление двухмерного
массива 8-битовых цепей */
```

```
my_array [0] [0]; // выбор первого элемента массива my_array
my_array [31] [63]; // выбор последнего элемента массива my_array
```

Формат выбора бита элемента массива:

`array_name [index][index]...[bit_number]`

где `array_name` и `index` имеют прежние значения; `bit_number` — номер выбираемого бита.

Примеры выбора битов элементов массива:

```
my_array [0] [0] [0]; // выбор нулевого бита в первом
// элементе массива my_array
my_array [31] [63] [7]; // выбор седьмого бита в последнем
// элементе массива my_array
```

Формат выбора битового поля элемента массива:

`array_name [index] [index]...[part_select]`

где `[part_select]` — выбор части поля битового вектора, может иметь один из трех рассмотренных ранее форматов выбора битового поля, например:

```
my_array [0] [0] [7:3] // выбор пяти старших битов первого
// элемента массива my_array
my_array [0] [0] [7:-5] // то же самое
```

Замечание. В каждый момент времени только один элемент массива может быть прочитан или записан.

4.8. Объявление памяти

Память в языке Verilog представляется с помощью одномерных массивов переменных типа `reg`. Такие массивы могут использоваться для моделирования памяти типа ROM, RAM или регистровых

файлов. В общем случае массивы в языке Verilog могут быть многомерными, но они не могут быть реализованы в типичных блоках памяти реального физического устройства.

Примеры объявления памяти:

```
reg [7:0] mem_1x8; // одно 8-битовое слово в памяти
reg mem_8x1 [7:0]; // память из 8 1-битовых элементов
reg [7:0] mem_1024x8 [0:1023]; // память из 1024 8-битовых слов
reg [7:0] mem_5x8 [0:4]; // память из 5 8-битовых слов
```

Пример использования памяти:

```
reg [7:0] mema [0:255]; // память из 256 8-битовых слов
mem[1] = 0; // назначение элементу памяти mema
// с индексом 1 значения 8'b00000000
```

Однако следует различать регистры и память из 1-битовых слов, например:

```
reg [7:0] regb; // 8-битовый регистр (не память)
reg memb [7:0]; // память из 8 1-битовых слов
```

Замечание. Для чтения и записи элементов массивов, которые представляют собой память (одномерный массив переменных типа `red`), можно использовать следующие системные задачи: `$readmemb`, `$readmemb`, `$sreadmemb` и `$sreadmemb`.

Г л а в а 5

Операции

5.1. Операции языка Verilog

Синтаксис большинства операций языка Verilog совпадает с синтаксисом языка программирования С [22]. Операции используются в выражениях для определения действий, выполняемых над операндами. Другими словами, аргументами операций являются операнды. Операндами большинства операций могут быть: числа, сети, переменные, отдельные биты (сигналы), векторы, части векторов (битовые поля), вызовы функций.

Замечание. Некоторые операции запрещены для действительных чисел (чисел с плавающей запятой).

В логических операциях, возвращающих значения «истина» или «ложь», 1-битовое значение 1 означает истину, 0 — ложь и **X** — неопределенное значение.

Операции языка Verilog принято делить на следующие группы:

- побитовые операции (bitwise);
- операции редукции (reduction);
- логические операции;
- операции отношения (relational);
- операции идентичности (identity);
- арифметические операции;
- разносторонние (miscellaneous) операции.

5.2. Побитовые операции

Побитовые операции языка Verilog приведены в табл. 5.1. Эти операции играют основную роль в описании проектов с помощью булевых функций и булевых выражений. При этом логические операции булевой алгебры над векторами и битовыми полями выполняются побитно.

Результаты побитовых операций приведены в табл. 5.2. При вычислении битовых значений величина **Z** принимается равной **X**.

Операции

63

Таблица 5.1

Побитовые операции

Символ	Пример	Описание
\sim	$\sim m$	Инверсия каждого бита вектора m , унарная операция
$\&$	$m \& n$	Логическая операция И (AND) каждого бита вектора m с каждым битом вектора n
$ $	$m n$	Логическая операция ИЛИ (OR) каждого бита вектора m с каждым битом вектора n
\wedge	$m \wedge n$	Логическая операция Исключающее ИЛИ (XOR) каждого бита вектора m с каждым битом вектора n
$\sim\wedge$ или $\wedge\sim$	$m \sim\wedge n$	Инверсия операции Исключающее ИЛИ (XNOR) каждого бита вектора m с каждым битом вектора n
\ll	$m \ll n$	Сдвиг влево на n позиций содержимого вектора m , значение слева заполняются нулями
\gg	$m \gg n$	Сдвиг вправо на n позиций содержимого вектора m , значение справа заполняются нулями

Таблица 5.2

Результаты побитовых операций

Операнды		Результаты выражений				
a	b	$\sim a$	$a \& b$	$a b$	$a \wedge b$	$a \sim\wedge b$
0	0	1	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	0
1	1	0	1	1	0	1
0	X	1	0	X	X	X
X	0	X	0	X	X	X
1	X	0	X	1	X	X
X	1	X	X	1	X	X
X	X	X	X	X	X	X

В качестве примера использования побитовых операций сдвига рассмотрим устройство умножения некоторой величины на 10, описание которого приведено в листинге 5.1.

Листинг 5.1. Пример использования операций сдвига для умножения на 10

```
// Используемая формула: A*10 = A*(2 + 8) = A*2 + A*8
module mult_on_10 (input [7:0] A, // исходная величина
                    output [11:0] Y); // результат больше на 4 бита
    wire [8:0] Ax2 = A << 1; // A*2, неявное использование
                                // оператора назначения
    wire [10:0] Ax8 = A << 3; // A*8, неявное использование
                                // оператора назначения
    assign Y = Ax2 + Ax8; // Y = (A << 1) + (A << 3),
                          // явное использование оператора
                          // назначения
endmodule
```

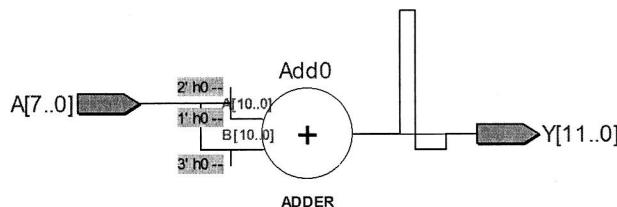


Рис. 5.1. Реализация модуля *mult_on_10* из листинга 5.1, выполняющего умножение числа *A* на 10 с помощью операций сдвига

Результат синтеза примера из листинга 5.1 приведен на рис. 5.1.

Задание. Опишите параметризованный модуль умножения произвольного целого числа на число 10, в котором в качестве параметра выступает число битов исходного числа.

5.3. Операции редукции

Операции редукции, иначе называемые операциями сокращения, по мнемонике напоминают побитовые операции, однако их действие кардинально отличается от побитовых операций. Побитовая операция поочередно применяется к каждому биту двух векторов и результатом является вектор. Операция редукции вначале применяется к первым двум битам вектора. Следующими operandами операции редукции являются результат предыдущего применения операции и следующий бит вектора. Данный процесс последовательно применяется ко всем битам вектора.

Результатом операции редукции является один бит. Другими словами, операции редукции редуцируют (сокращают) битовый вектор до одного бита, последовательно применяя соответствующую логическую операцию к каждому биту вектора.

Операции редукции языка Verilog приведены в табл. 5.3, а примеры их применения — в табл. 5.4.

Пример использования операций редукции приведен в листинге 5.2, где определяется, является ли число единиц битового вектора четным и имеют ли все биты значение 1.

Листинг 5.2. Пример использования операций редукции

```
module inputs_test(
    input [7:0] in,      // входной битовый вектор
    output parity,      // истина, если число единиц четно
    all_ones);          // истина, если все биты имеют значение 1
    assign parity = & in;
    assign all_ones = & in;
endmodule
```

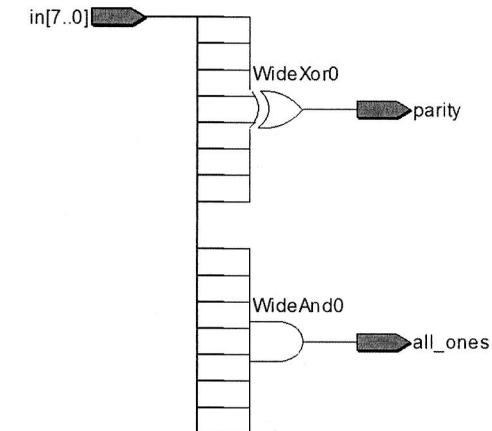
Таблица 5.3
Операции редукции

Символ	Пример	Описание
&	& m	Редуцирует биты вектора <i>m</i> по И (AND)
$\sim\&$	$\sim\& m$	Редуцирует биты вектора <i>m</i> по И-НЕ (NAND)
	m	Редуцирует биты вектора <i>m</i> по ИЛИ (OR)
$\sim $	$\sim m$	Редуцирует биты вектора <i>m</i> по ИЛИ-НЕ (NOR)
\wedge	$\wedge m$	Редуцирует биты вектора <i>m</i> по Исключающее ИЛИ (XOR)
$\sim\wedge$ или $\wedge\sim$	$\sim\wedge m$	Редуцирует биты вектора <i>m</i> по инверсии Исключающее ИЛИ (XNOR)

Таблица 5.4
Примеры применения редукции

Операнд	Результаты редукции		
a	$\& a$	$ a$	$\wedge a$
00000000	0	0	0
11111111	1	1	0
10101010	0	1	1
110011zz	0	1	x
1111111x	x	1	x

Рис. 5.2. Реализация модуля *inputs_test* из листинга 5.2: пример использования операций редукции для определения во входном слове четного числа единиц и функции, когда все биты имеют значение 1



Результаты синтеза примера из листинга 5.2 даны на рис. 5.2.

В листинге 5.3 приведен еще один пример использования побитовой операции и операции редукции для реализации в компараторе функции равенства.

Листинг 5.3. Реализация функции равенства

```
module comp_eq (
    input [7:0] a,b,      // входные векторы
    output eq);          // функция равенства
    wire [7:0] im;       // вспомогательная переменная
    assign im = a & b;   // нахождение битов, в которых a и b не равны
    assign eq = ~| im;   // если в векторе im имеется хотя бы одна
                        // единица, то a и b не равны
endmodule
```

Результаты синтеза примера из листинга 5.3 даны на рис. 5.3.

5.4. Логические операции

Логические операции используются в логических выражениях, например, операторов *if*, *for*, *while*. Результатом логических опера-

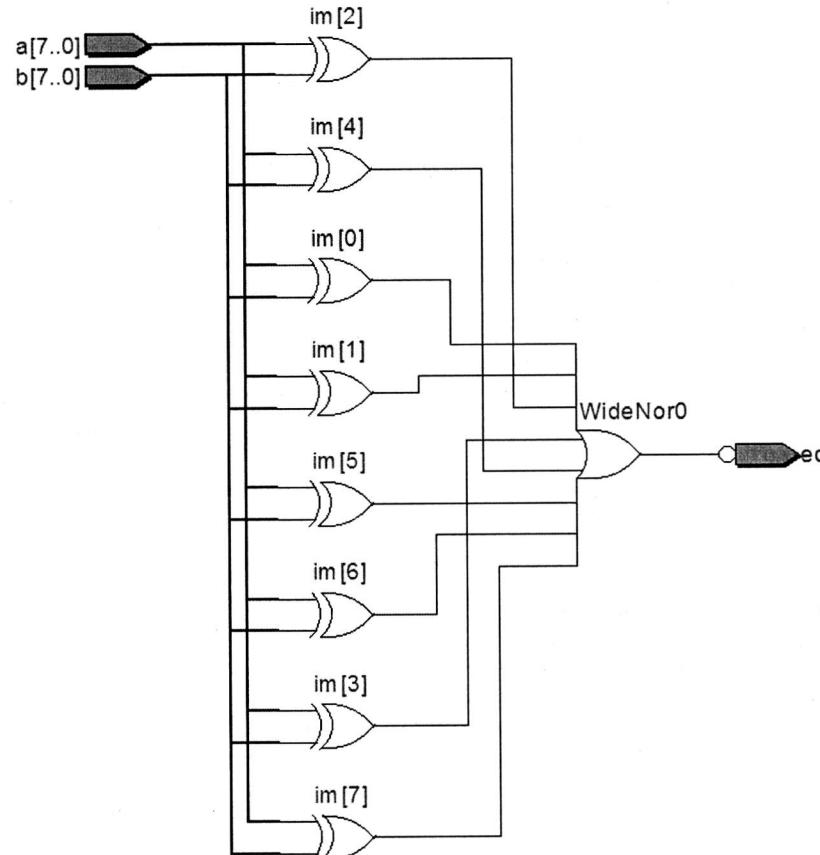


Рис. 5.3. Реализация модуля *compr_eq* из листинга 5.3: пример использования побитовой операции редукции NOR для реализации в компараторе функции равенства

ций является 1- битовое значение, причем 1 означает истину, а 0 и X — ложь. Логические операции представлены в табл. 5.5.

Таблица 5.5

Логические операции

Символ	Пример	Описание
!	! m	истина, если m ложно
&&	m && n	истина, если m и n истинны
	m n	истина, если либо m, либо n истина

В листинге 5.4 представлен пример устройства управления микроконтроллера, которое проверяет, что данная команда является вычитанием.

Листинг 5.4. Пример фрагмента устройства управления микроконтроллера

```
module if_subtraction
```

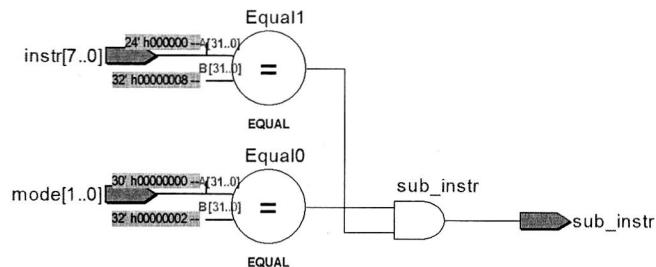


Рис. 5.4. Реализация модуля *if_subtraction* из листинга 5.4: пример использования логических операций при реализации устройства управления микроконтроллера

```
(input [7:0] instr,
input [1:0] mode,
output sub_instr);
parameter IMMEDIATE=2'b00, DIRECT=2'b01; /* локальные
параметры, определяющие режим работы микроконтроллера */
parameter SUBA_imm=8'h80, SUBA_dir=8'h90, /* локальные
параметры, определяющие код операции */
SUBB_imm=8'hc0, SUBB_dir=8'hd0;
assign sub_instr = (((mode == IMMEDIATE) &&
((instr == SUBA_imm) ||
(instr == SUBB_imm))) ||
((mode == DIRECT) &&
((instr == SUBA_dir) ||
(instr == SUBB_dir))));
```

endmodule

Результаты синтеза примера из листинга 5.4 даны на рис. 5.4.

5.5. Операции отношения

Операции отношения часто являются операндами логических операций, они проверяют отношения (больше, меньше, равно и др.) между значениями различных типов данных. Результатом операций отношения является 1-битовое значение, указывающее на истинность или ложность данного отношения. Операции отношения представлены в табл. 5.6.

Таблица 5.6

Операции отношения

Символ	Пример	Описание
==	m == n	Истина, если m равно n
!=	m != n	Истина, если m не равно n
<	m < n	Истина, если m меньше n
>	m > n	Истина, если m больше n
<=	m <= n	Истина, если m меньше или равно n
>=	m >= n	Истина, если m больше или равно n

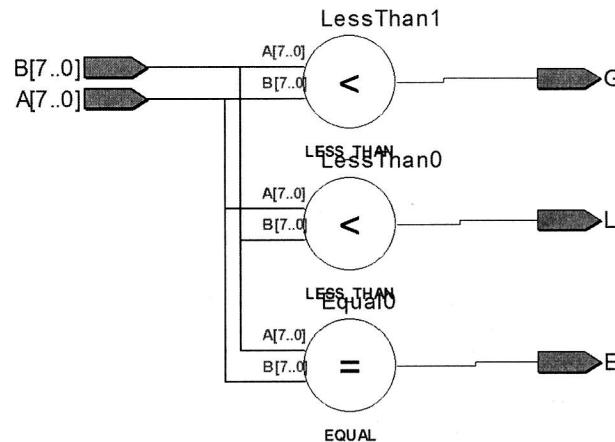


Рис. 5.5. Реализация модуля *comparator* из листинга 5.5: пример использования операций отношения для построения параметризованного компаратора

Пример использования операций отношения дан в листинге 5.5.

Листинг 5.5. Параметризованный компаратор

```

module comparator
#(parameter SIZE = 8)
(input [SIZE-1:0] A,B,
output L, E, G);
assign L = (A <B);
assign E = (A == B);
assign G = (A >B);
endmodule
  
```

Результаты синтеза примера из листинга 5.5 даны на рис. 5.5.

Замечание. Если хотя бы один operand в операциях отношения имеет в своих битах значение **X** или **Z**, то результатом операции отношения будет значение **X**.

Другими словами, операции отношения позволяют сравнивать только значения operandов, а в случае появления хотя бы в одном из битов любого operand'a значений **X** или **Z** результат операции отношения будет равен **X**. Возникает вопрос: как сравнивать operandы, отдельные биты которых могут принимать значения **X** или **Z**. Для этого служат операции идентичности.

5.6. Операции идентичности

Операции идентичности в языке Verilog представлены двумя операциями: **==** и **!=**. Отличие этих операций от операций отношения равенства (**==**) и неравенства (**!=**) заключается в том, что они

позволяют проверять, кроме 0 и 1, логические значения битов **X** и **Z**. Другими словами, операции идентичности позволяют проверять все четыре возможные логические значения каждого бита operandов: 0, 1, **X** и **Z**.

Замечание. Не все компиляторы поддерживают операции идентичности при синтезе проекта.

Задание. Проверьте, поддерживает ли используемый вами компилятор операции идентичности при синтезе проекта.

5.7. Арифметические операции

Арифметические операции языка Verilog приведены в табл. 5.7.

Таблица 5.7

Арифметические операции		
Символ	Пример	Описание
+	m + n	Сложение m с n
-	m - n	Вычитание n из m
-	-m	Отрицание m (представление в дополнительном коде)
*	m * n	Умножение m на n
/	m / n	Деление m на n
%	m % n	Остаток от деления m на n
**	m ** n	Возведение m в степень n
<<<	m <<< n	Сдвиг m влево на n раз, освободившиеся биты справа заполняются нулями
>>>	m >>> n	Сдвиг m вправо на n раз, если m знаковая величина, то освободившиеся слева биты заполняются значением знакового разряда, в противном случае — нулями

Пример использования арифметических операций:

```
assign {cout, s} = a + b + cin;
```

Замечание. Не все арифметические операции поддерживаются программными средствами при синтезе проекта.

Задание. Проверьте, какие из арифметических операций поддерживаются в используемом вами компиляторе при синтезе проекта.

5.8. Разносторонние операции

Различные операции языка Verilog, которые не попали в рассмотренные выше группы, называются разносторонними (*miscellaneous*). Разносторонние операции представлены в табл. 5.8.

С помощью условной операции можно легко реализовать шинный мультиплексор 2-1 (листинг 5.6).

Листинг 5.6. Реализация мультиплексора с помощью условной операции

```

module mux_2_1(
  input [7:0] a, b,      // входные шины
  
```

Разносторонние операции

Таблица 5.8

Символ	Пример	Описание
? :	sel ? m : n	Условная операция, если sel истина, возвращает m, иначе возвращает n
{}	{m,n}	Операция конкатенации, соединяет m с n, образуя большой вектор
{ { } }	{n{m}}	Операция повторения, соединяет m само с собой n раз
->	-> m	Операция наступления события, переключает значение m, которое имеет тип данных event

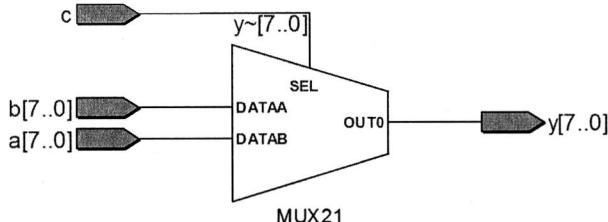


Рис. 5.6. Реализация модуля *mix_2.1* из листинга 5.6: пример использования условной операции для построения шинного мультиплексора

```

input c,           // сигнал выбора входной шины
output [7:0] y); // выходная шина
assign y = c ? a: b;
endmodule

```

Реализация примера из листинга 5.6 приведена на рис. 5.6.

В качестве еще одного примера рассмотрим использование условного оператора для реализации простого арифметико-логического устройства (АЛУ).

Листинг 5.7. Пример использования условных операторов

```

module mini_ALU (input [7:0] a,b, // аргументы
                  input [2:0] op, // код операции
                  output [7:0] result); // результат
parameter ADD=3'h0, SUB=3'h1, // коды выполняемых операций
          AND=3'h2, OR=3'h3, XOR=3'h4;
assign result = ((op == ADD) ? a+b : (
                           (op == SUB) ? a - b : (
                           (op == AND) ? a & b : (
                           (op == OR) ? a | b : (
                           (op == XOR) ? a ^ b : (a))))));
endmodule

```

Реализация примера из листинга 5.7 приведена на рис. 5.7.

Задание. Опишите АЛУ для реализации всех бинарных ло-

Операции

гических операций, всех бинарных арифметических операций, всех унарных логических операций, всех унарных арифметических операций, АЛУ простого микроконтроллера.

Некоторые примеры использования оператора конкатенации:

```

wire [3:0] a, b;           // векторы на 4 бита
wire [7:0] c, d;           // векторы на 8 битов
wire [11:0] e, f;          // векторы на 12 битов
assign c = {a,b};          // результат 8 битов
assign e = {b,a,b};         // результат 12 битов
assign f = {3{a}};          // результат 12 битов: {a,a,a}
assign b = {4{e == f}};     // результат 4 бита: { (e == f),
                           // {(e == f), (e == f), (e == f)} }
assign f = {a,d};          // результат 12 битов
assign e = {2{1'b1,a,1,b0}}; // результат 12 битов: { 1,a,0,1,a,0 }
assign {a,b} = d;           // результат 8 битов
assign {a,b,f} = {e,d} + 1; // результат 20 битов,
                           // но может быть переполнение!

```

5.9. Выполнение операций

Перед выполнением операций все операнды расширяются (выравниваются) до размера наибольшего вектора в операторе (с обеих сторон от знака назначения). Беззнаковые значения расширяются нулями, а знаковые — значением знакового разряда. Операции конкатенации и повторения выполняются до выравнивания операндов.

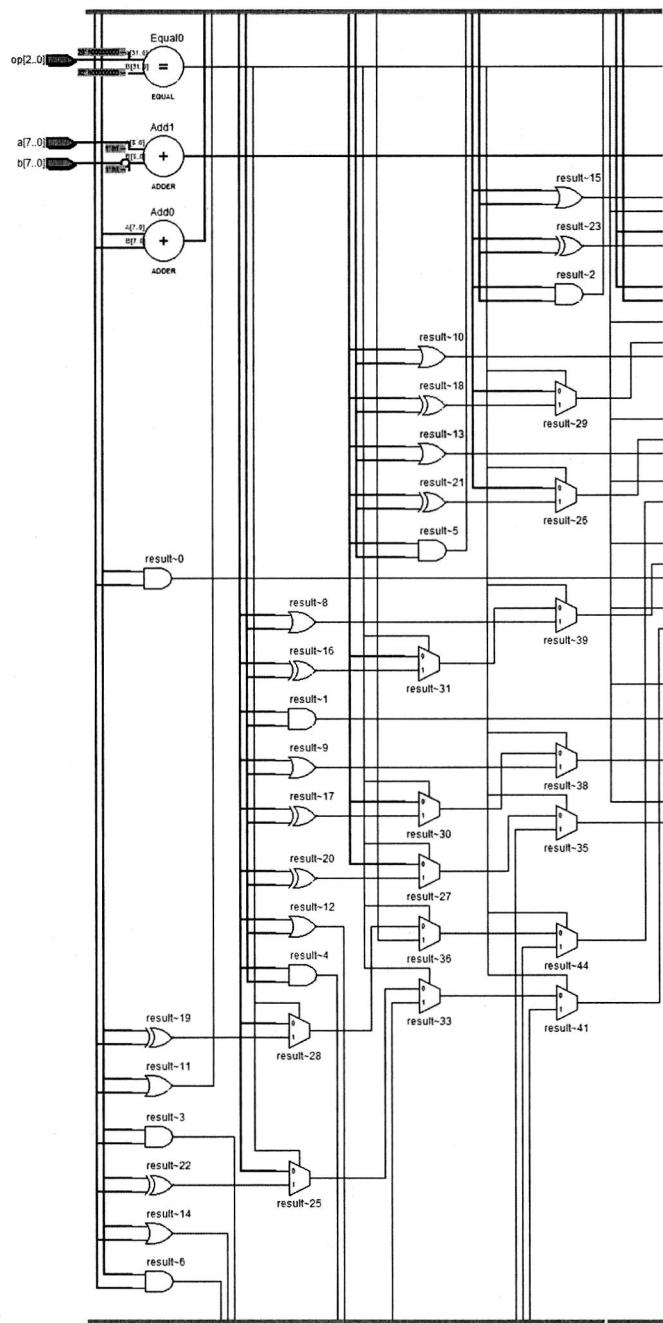
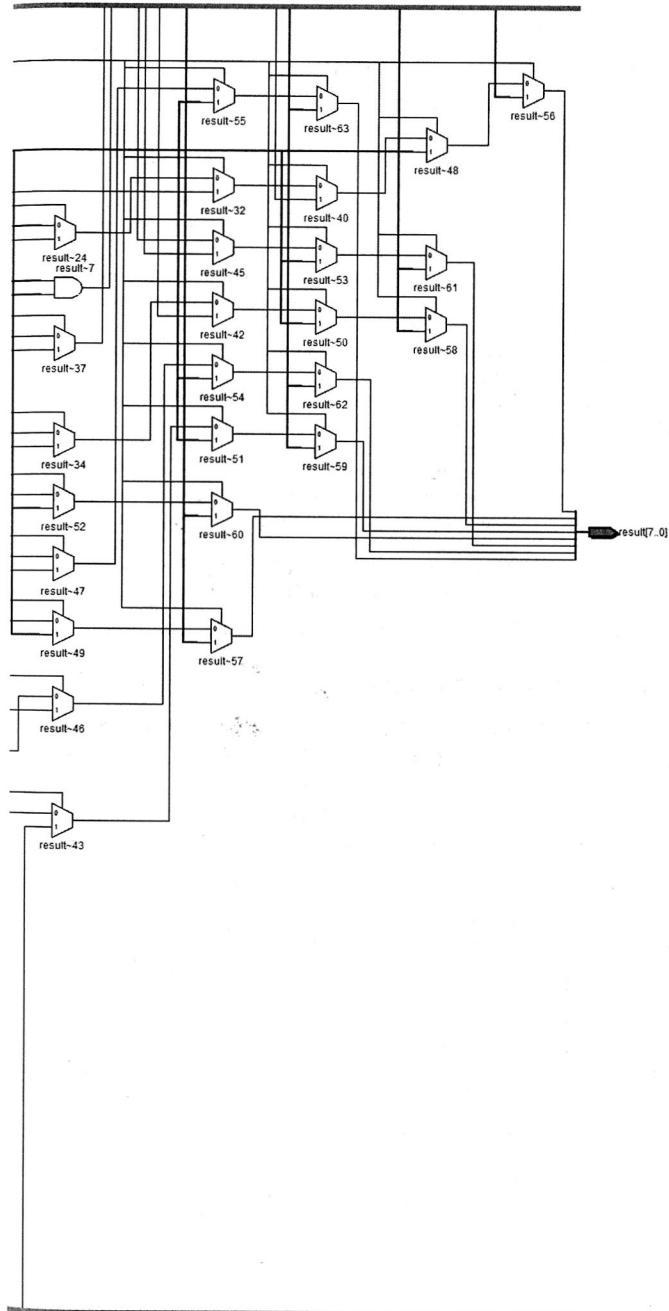
Арифметические операции выполняются по следующим правилам:

- если любой операнд является действительным числом, то выполняется арифметика чисел с плавающей точкой;
- если любой операнд является беззнаковым значением, то выполняется беззнаковая арифметика;
- если все операнды являются знаковыми значениями, то выполняется знаковая арифметика.

Замечание. Операнд может быть приведен к знаковому или беззнаковому виду с помощью системных функций `$signed` и `$unsigned`.

5.10. Приоритет операций

Выражения языка Verilog вычисляются в порядке приоритета операций. Для изменения порядка вычислений используются круглые скобки. Приоритеты операций языка Verilog представлены в табл. 5.9, где приоритет групп операций понижается с увеличением номера группы операций.

Рис. 5.7. Реализация модуля *mini_ALU* из листинга 5.7: пример

Вложенного использования условных операций при описании простого АЛУ

Таблица 5.9

Приоритет операций

Номер группы операций	Символы	Примечания
1	!, ~, +, -	Унарные операции логического отрицания, побитовой инверсии, арифметические унарные знаки плюс и минус
2	{ }, { { } }	Операции конкатенации и повторения
3	()	Круглые скобки, определяющие порядок выполнения операций
4	**	Арифметическая операция возведения в степень
5	*, /, %	Арифметические операции умножения, деления и остаток от деления
6	+,-	Арифметические бинарные операции сложения и вычитания
7	<<, >>, <<<, >>>	Операции логического и арифметического сдвигов
8	<, <=, >, >=	Операции отношения
9	==, !=, ===, !==	Операции равенства и идентичности
10	&, ~&	Операции AND и NAND, побитовые и редукции
11	^, ~^	Операции XOR и XNOR, побитовые и редукции
12	, ~	Операции OR и NOR, побитовые и редукции
13	&&	Логическая операция AND
14		Логическая операция OR
15	? :	Условная операция

5.11. Размеры битовых выражений

Размеры битовых выражений языка Verilog, получаемых в зависимости от используемых операций, приведены в табл. 5.10.

Таблица 5.10

Размеры битовых выражений

Выражение	Размер	Операция
i op j	max(L(i), L(j))	Бинарные операции +, -, /, *, %, &, , ^, ~, &
op i	L(i)	Унарные операции +, -, ~
i op j	1 bit	==, !=, ==, !=, &&, , «, =, », =, &, ~&, , ~ , ^, ~^!
i op j	L(i)	», «, **, », ><, «
i ? j : k	max(L(j), L(k))	Условная операция
{i,...,j}	L(i)+...+L(j)	Операция конкатенации
{i{...,k}}	i*(L(j)+...+L(k))	Операция повторения

Оператор непрерывного назначения
assign

6.1. Присваивание значений в языке Verilog

В языке Verilog нет привычных операторов присваивания, как в языках программирования. Дело в том, что присваивание значений основным типам данных (сетям *net* и переменным *reg*) имеет различную природу. Присваивание значений переменным (*reg*) подобно оператору присваивания в языках программирования. Однако сети (*net*) предназначены для соединения элементов проекта и ассоциируются с электрическими цепями, соединяющими выводы электронных элементов.

Прежде всего следует отметить, что в электронных схемах цепи постоянны и непрерывны, т. е. они не изменяются во время работы цифрового устройства или системы. Присваивание некоторого значения цепи означает задание значения сигнала в цепи, определяемого источником сигнала (драйвером). Причем у одной сети таких источников сигналов может быть несколько и они могут функционировать независимо и одновременно.

Оператор непрерывного назначения **assign** определяет значение одного из источников сигналов сети. Поскольку цепи в электронных схемах неизменны и постоянно подсоединенны к источникам сигналов, оператор **assign** называется непрерывным (*continuous*). Окончательное значение сигнала в сети определяется типом цепи, т. е. каким образом источники сигналов подсоединенны к сети по OR, по AND, с открытым коллектором и др. (соединение цепей различного типа было рассмотрено в разделе 4.3).

Поскольку все сигналы в электронной схеме передаются по цепям одновременно, то и все операторы непрерывного назначения **assign** выполняются параллельно. Кроме того, в языке Verilog также параллельно выполняются:

- экземпляры модулей;
- экземпляры примитивов;
- процедурные блоки.

6.2. Форматы оператора непрерывного назначения

Оператор непрерывного назначения **assign** имеет два формата: явный и неявный. Явный формат оператора **assign** имеет вид:

```
assign # (delay) net_name = expression;
```

при этом предполагается, что цепь была ранее объявлена с помощью следующего формата:

```
net_type [size] net_name;
```

Неявный формат оператора **assign** используется при объявлении имени цепи. Он, фактически, повторяет предыдущие два формата без ключевого слова **assign**:

```
net_type (strength) [size] # (delay) net_name = expression;
```

В данных форматах конструкции **[size]**, **# (delay)** и **(strength)** являются необязательными. Здесь **net_type** — один из сетевых типов данных, за исключением типа **trireg**; **[size]** — размер вектора левой части в формате **[msb : lsb]**; **delay** — временная задержка выполнения операции назначения (по умолчанию имеет значение 0), имеет точно такой же формат, как у примитивов (рассмотрено ранее в подразделе 3.2); **strength** — логическая мощность сигнала (рассмотрена ранее в подразделе 1.4.2), по умолчанию имеет значение **(strong1, strong0)**.

Имя цепи **net_name** в левой части оператора **assign** должно быть ранее объявленным именем цепи или именем порта модуля. В последнем случае дополнительное объявление имени порта не требуется. Если необъявленное имя цепи **net_name** не является именем порта, то по умолчанию подразумевается однобитовая цепь типа **wire**.

Выражение **expression** в правой части оператора непрерывного назначения **assign** может включать любые типы данных, любые операции и вызовы функций.

Оператор непрерывного назначения **assign** функционирует следующим образом. При любом изменении любого из значений в выражении **expression** с правой стороны, заново вычисляется выражение **expression** и полученное значение по истечении времени, определяемого задержкой **delay**, присваивается величине слева от знака **«=»**.

Примеры использования оператора **assign**:

```
// явное непрерывное назначение
wire [31:0] mux_out;
assign mux_out = sel ? a : b; // мультиплексор шин a и b на 32 бита
// неявное непрерывное назначение
tri [0:15] # 2.7 buf_out = en ? in : 16'bz; // 16-битовый буфер с тремя
// состояниями и задержкой в 2,7 временных единиц
```

```
// неявное непрерывное назначение с указанием логической
// мощности
wire (strong1, pull0) [63:0] ALU_out = ALU_function(opcode, a, b);
// вызов функции
```

6.3. Использование оператора непрерывного назначения

Прежде всего отметим два важных свойства оператора **assign**:

- параллельность — все операторы **assign** в одном модуле выполняются параллельно независимо от их местоположения;
- чувствительность к событиям, происходящим с сигналами справа от знака равенства, т. е. всякий раз, когда происходит изменение значения цепи или переменных в правой части, заново вычисляется выражение и полученное значение назначается цепи в левой части оператора **assign**.

Оператор **assign** широко используется для описания комбинационных схем с помощью уравнений булевой алгебры. Примером может служить однобитовый сумматор, описанный в листинге 1.2 по уравнениям (1). Здесь не потребовались объявления дополнительных цепей, поскольку все элементы выражений и цепи слева от знака **«=»** являются портами модуля. В качестве альтернативы рассмотрим описание однобитового сумматора с промежуточной цепью **a_xor_b** в листинге 6.1.

Листинг 6.1. Пример однобитового полного сумматора с неявным использованием оператора непрерывного назначения

```
module sum_1_1
  (input a, b, cin,
   output s, cout);
  wire a_xor_b = a & b;      // неявное использование оператора
                             // непрерывного назначения
                             // одновременно с объявлением цепи
  assign s = a_xor_b & cin;  // явное использование оператора
                             // непрерывного назначения
  assign cout = (a & b) | ((a_xor_b) & cin);
endmodule
```

Реализация примера из листинга 6.1 приведена на рис. 6.1.

Отметим, что использование оператора **assign** позволяет описывать комбинационные устройства без знания структуры устройства. Отпадает также необходимость в каком-либо проектировании (преобразованиях логических уравнений, минимизации, факторизации, декомпозиции и др.). Достаточно знать логические уравнения, описывающие функционирование комбинационной схемы, а все остальное

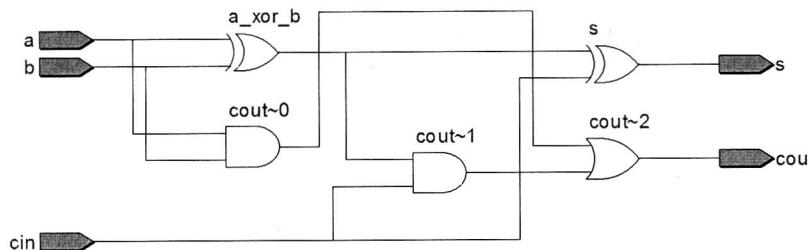


Рис. 6.1. Реализация модуля sum_1_1 из листинга 6.1

компилятор сделает автоматически. Это освобождает разработчиков от выполнения формальных достаточно сложных преобразований, связанных с логическим синтезом и позволяет сосредоточиться на алгоритме функционирования цифрового устройства или системы. Кроме того, подобный стиль проектирования значительно снижает вероятность появления ошибок в результате ручного проектирования и увеличивает читабельность кода проекта.

В левой стороне оператора **assign** может находиться операция конкатенации, как показано в листинге 6.2.

Листинг 6.2. Пример использования операции конкатенации в левой части оператора **assign**

```
module sum_1_2
  (input a, b, cin,
   output s, cout);
  assign { cout, s} = a + b + cin;
endmodule
```

Результат синтеза примера из листинга 6.2 приведен на рис. 6.2.

После вычисления выражения и выполнения всех преобразований левой части, в общем случае слева и справа от знака «`=`» в операторе **assign** будут находиться битовые векторы. В случае их одинакового размера значения битов правого вектора будут назначаться битам левого вектора, начиная с наименее значащего бита. Если число битов

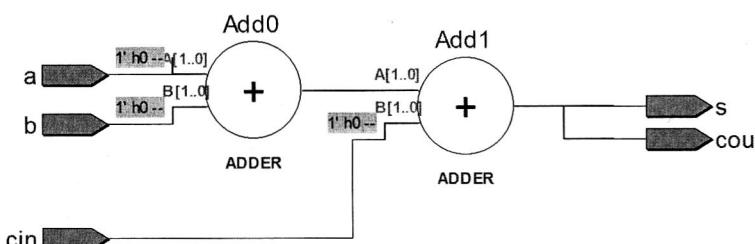


Рис. 6.2. Реализация модуля sum_1_2 из листинга 6.2

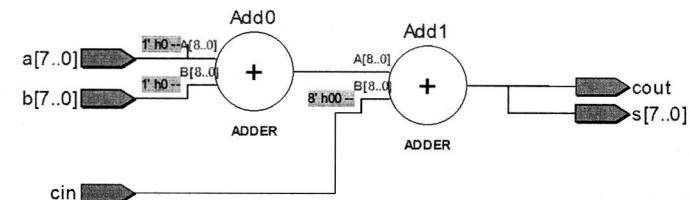


Рис. 6.3. Реализация модуля sum_8 из листинга 6.3: пример 8-битового сумматора с использованием арифметических операций

правой стороны больше, чем число битов левой стороны, то старшие биты будут отброшены (утеряны). Если число битов левой стороны больше, чем число битов правой стороны, то старшие биты левой стороны заполняются нулями. В качестве примера в листинге 6.3 показано описание 8-битового сумматора.

Листинг 6.3. Пример 8-битового полного сумматора

```
module sum_8 (input [7:0] a, b,
  input cin,
  output [7:0] s,
  output cout);
  assign { cout, s} = a + b + cin;
endmodule
```

Результат синтеза примера из листинга 6.3 приведен на рис.6.3.

В следующем примере для описания АЛУ оператор **assign** используется многократно (листинг 6.4). В данном примере АЛУ выполняет операции сложения и вычитания над аргументами *a* и *b*. При этом формируется значение сигнала переноса *c*, а также формируются значения двух флагов: больше *gt* и нулевого значения результата *z*.

Листинг 6.4. Пример простого АЛУ

```
module simple_ALU(
  input [7:0] a, b,           // аргументы
  input op,                  // код операции
  output [7:0] r,             // результат
  output gt, c, z);          // формируемые флаги
  assign { c,r} = op ? (a + b) : (a - b);
  assign gt = (a > b);
  assign z = (r == 0);
endmodule
```

Результат синтеза примера из листинга 6.4 приведен на рис. 6.4.

Таким образом с помощью оператора непрерывного назначения **assign** можно описывать достаточно сложные комбинационные схемы.

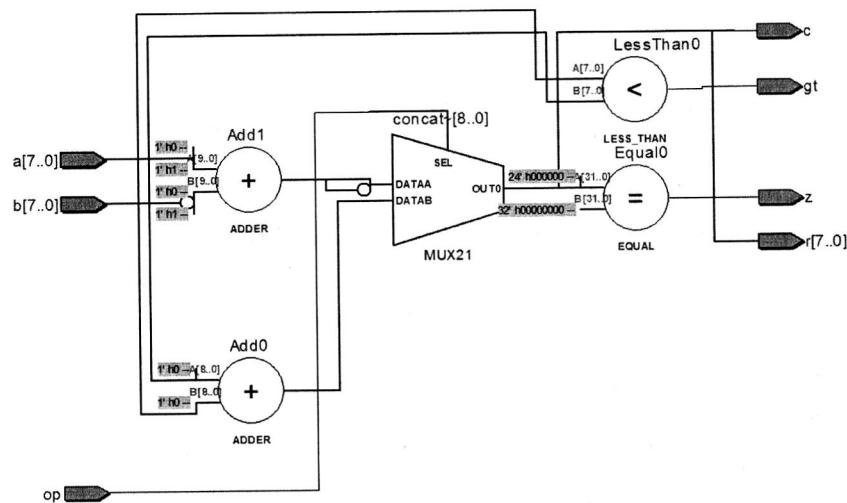


Рис. 6.4. Реализация модуля *simple_ALU* из листинга 6.4: пример многократного использования оператора **assign** с различными операциями при описании комбинационной схемы

Задание. Приведите примеры различных способов использования оператора **assign** при описании проектов.

Процедурные операторы и блоки

7.1. Процедурные операторы *initial* и *always*, процедурные блоки

Подобно тому, как языки программирования позволяют описывать алгоритмы, язык Verilog позволяет описывать функционирование цифровых устройств, т. е. создавать функциональные модели проектов. Одним из главных свойств языка Verilog является возможность учета реального времени функционирования каждого элемента, что позволяет создавать модели систем реального времени.

В языке Verilog имеется только два процедурных оператора: **initial** и **always**, с помощью которых образуются процедурные блоки. Процедурный блок представляет собой группу операторов, заключенных в операторные скобки.

Операторы внутри процедурного блока типа **initial** выполняются только один раз. Блоки типа **initial** обычно используются для задания начальных значений переменным перед началом симуляции. Операторы внутри процедурного блока типа **always** выполняются непрерывно в виде бесконечного цикла. В случае последовательного выполнения после выполнения последнего оператора в группе, выполнение начинается с первого оператора группы. Блоки типа **always** обычно используются для описания функционирования цифрового устройства.

Замечание. Если операторы **initial** и **always** используются без операторных скобок, то говорят об операторах **initial** и **always**; если операторы **initial** и **always** используются вместе с операторными скобками, то говорят о блоках **initial** и **always**.

Назначения в процедурных блоках выполняют назначения значений переменным, которые удерживают эти значения до тех пор, пока им не будут присвоены другие значения. С помощью процедурных блоков могут описываться как комбинационные, так и последовательностные схемы.

7.2. Операторные скобки begin-and и fork-join

В языке Verilog имеются два типа операторных скобок, определяемых ключевыми словами **begin-and** и **fork-join**.

Примеры процедурных блоков:

```
initial
  begin // процедурный блок initial с процедурными скобками
    // begin-and
    /* здесь может располагаться группа операторов */
  end
always
  fork // процедурный блок always с процедурными скобками
    // fork-join
    /* здесь может располагаться группа операторов */
  join
```

Операторы, заключенная в операторные скобки **begin-and**, выполняются последовательно так, как они записаны в исходном коде проекта. Время начала выполнения каждого оператора в группе **begin-and** соотносится с временем выполнения предыдущего оператора. Операторы, заключенные в операторные скобки **fork-join**, выполняются параллельно. Время начала выполнения каждого оператора в группе **fork-join** соотносится с временем выполнения всей группы операторов.

Если группа операторов состоит только из одного оператора, то операторные скобки не нужны и операторы **initial** и **always** могут использоваться без операторных скобок **begin-and** или **fork-join**.

Замечание. Обычно операторные скобки **fork-join** не поддерживаются синтезаторами.

Задание. Проверьте, поддерживает ли используемый вами компилятор операторные скобки **fork-join** при синтезе проектов.

7.3. Именованные процедурные блоки

Процедурным блокам могут назначаться имена. Если некоторому процедурному блоку приписано имя, то такой блок называется **именованным блоком** (*named block*).

Примеры именованных процедурных блоков:

```
initial
  begin: block_1 // именованный процедурный блок initial
    // с именем block_1
    /* здесь может располагаться группа операторов */
  end
```

always

```
fork: block_2 // именованный процедурный блок always
  // с именем block_2
  /* здесь может располагаться группа операторов */
join
```

Именованные блоки, в отличие от блоков без имени, обладают рядом дополнительных свойств:

- имена именованных блоков могут использоваться в иерархическом пути при поиске конкретного элемента в иерархии проекта;
- именованные блоки могут иметь объявления собственных локальных переменных и параметров;
- выполнение операторов именованного блока может быть прервано с помощью процедурного оператора **disable** и др.

7.4. Формат процедурных блоков

В общем случае процедурные блоки могут иметь следующий формат:

```
type_of_block @ (sensitivity_list)
  statement_group : group_name
    local_variable_declarations
    time_control procedural_statements
  end_of_statement_group
```

где конструкции *sensitivity_list*, *statement_group*, *group_name*, *local_variable_declarations*, *time_control* и *end_of_statement_group* являются необязательными.

Конструкция *type_of_block* определяет тип процедурного блока. Она может принимать одно из следующих ключевых слов: **initial** или **always**.

Конструкция *@ (sensitivity_list)* называется **списком чувствительности**. Она позволяет контролировать различные временные события (изменение значения одного или нескольких сигналов, приход возрастающего или падающего фронта сигнала и др.) для управления выполнением операторов процедурного блока. Подробно формат списка чувствительности будет рассмотрен позже.

Конструкции *statement_group* и *end_of_statement_group* — это одна из операторных скобок **begin-and** или **fork-join**. Если процедурный блок содержит только один оператор, то процедурные скобки могут опускаться. В именованных процедурных блоках за ключевыми словами **begin** или **fork** после двоеточия записывается имя (*group_name*) процедурного блока.

Именованные процедурные блоки могут также содержать объявления локальных переменных и параметров (*local_variable_declar-*

tions), область действия которых ограничена размером процедурного блока.

За локальными объявлениями следуют операторы процедурного блока. Конструкция `time_control` используется для управления временем выполнения следующего за данной конструкцией оператора. Конструкция `time_control` может ставиться перед каждым оператором процедурного блока.

Примеры процедурных блоков:

```
/* блок initial, выполняется один раз, последовательно задает начальные значения переменной test_in с интервалом в 5 временных единиц; присваивание блоку имени test_loop позволяет объявить внутри блока локальную переменную i
*/
initial
  begin: test_loop          // присваивание блоку имени test_loop
    integer i;               // объявление локальной переменной i
    for (i=0; i<=15; i=i+1) // процедурный оператор цикла for
      #5 test_in = i;        // присваивание значения каждые
                             // 5 временных единиц
  end
/* блок initial, также выполняется один раз, однако процедурные скобки fork-join вызывают одновременное выполнение операторов внутри блока; в результате времена выполнения операторов являются абсолютными по отношению к времени начала выполнения данного блока
*/
initial
  fork
    bus = 16'h0000;           // присваивание выполняется с задержкой
                             // 0 временных единиц
                             // от начала выполнения блока initial
    #10 bus = 16'hC2AF;      // то же, с задержкой 10 временных ед.
    #25 bus = 16'hD29A;      // то же, с задержкой 25 временных ед.
  join
/* пример процедурного блока always, выполнение которого управляетя изменением значения любого из входных сигналов
*/
always @ (a or b or cin)
  begin
    s = a + b + cin;         // в данном случае процедурные скобки
                             // можно опустить
  end
```

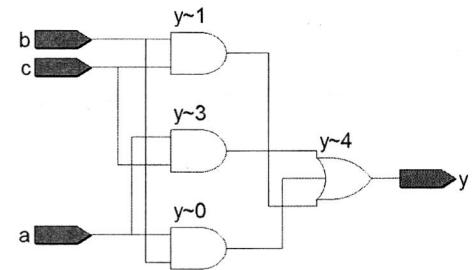


Рис. 7.1. Реализация модуля `maj_3` из листинга 7.1: пример использования процедурного блока `always` при реализации комбинационной схемы мажоритарного контроля

```
/* пример процедурного блока always, выполнение которого управляется возрастающим фронтом синхросигнала clk
*/

```

```
always @ (posedge clk)
  q = data;
```

В качестве еще одного примера рассмотрим описание комбинационной схемы мажоритарного контроля (большинства) для трех входов, приведенное в листинге 7.1.

Листинг 7.1. Схема мажоритарного контроля для трех входов

```
module maj_3 (
  input a, b, c,           // описание входов
  output reg y);          // описание выхода
  always @ (a, b, c)      // процедурный блок со списком
                           // чувствительности
    begin                  // начало процедурного блока
      y = (a & b) | (b & c) | (a & c); // оператор назначения
    end                    // конец процедурного блока
  endmodule
```

Блок `always`, описывающий поведение этой схемы, использует оператор управления событиями `@`, который включает список переменных (`a, b, c`), рассматриваемый как список чувствительности блока `always`. Говорят, что блок `always` является чувствительным к переменным `a, b` и `c`. Когда случается событие (изменение значения) на любой из этих переменных, то начинается процесс выполнения блока `always`. В результате будет вычислено булево выражение в правой части оператора назначения и его значение присвоено переменной `y`. Эта переменная удерживает свое значение до тех пор, пока в следующий момент времени не случится событие на входах `a, b` или `c`. В приведенном примере процедурный блок `always` включает только один оператор, поэтому операторные скобки `begin-end` можно опустить. Результат синтеза примера из листинга 7.1 приведен на рис. 7.1.

Г л а в а 8

Управление процедурным временем

В языке Verilog предусмотрено три способа управления процедурным временем, т. е. временем выполнения процедурных операторов, это оператор задержки **#**, чувствительности **@** и ожидания **wait**.

8.1. Оператор задержки **#**

Оператор задержки **#** в процедурных блоках имеют следующий формат:

```
# delay
```

где **delay** — величина, определяющая количество единиц времени, на которое задерживается выполнение следующего оператора (параметры единиц времени задаются с помощью системной директивы **'timescale'**). В качестве задержки может выступать число, переменная или выражение.

Примеры задержек:

```
# 5          // задержка на 5 временных единиц  
# 3.7       // задержка на 3,7 временных единиц
```

8.2. Оператор чувствительности **@**

Оператор чувствительности **@** позволяет задержать выполнение следующего оператора (группу операторов) до тех пор, пока не произойдет хотя бы одно из проверяемых событий. Такими событиями могут быть изменение уровня сигнала или приход фронта сигнала. Оператор чувствительности может иметь следующие форматы:

```
@ (edge signal or edge signal or ...)  
@ (edge signal, edge signal, ...)  
@ (*)
```

где в круглых скобках записывается список чувствительности, **edge** — фронт сигнала, может быть одним из следующих ключевых слов: **posedge** (возрастающий фронт) или **negedge** (падающий фронт);

or — ключевое слово; **signal** — сигнал, изменение значения которого проверяется.

Конструкция **edge** перед любым из сигналов может отсутствовать. Если перед сигналом записано одно из ключевых слов **posedge** или **negedge**, то проверяется соответствующий фронт сигнала. Если конструкция **edge** перед сигналом отсутствует, то проверяется любое изменение уровня сигнала.

В качестве сигналов в списке чувствительности могут выступать сети или переменные любого размера. Сигналы в списке чувствительности могут перечисляться через запятую. Вместо запятой может использоваться ключевое слово **or**. Если в списке чувствительности только один сигнал, то круглые скобки можно опустить. Знак звездочки (*****) в круглых скобках равносителен тому, что в список чувствительности включены все читаемые оператором (или группой операторов) сигналы, т. е. все сигналы, которые для данного оператора являются входными.

Примеры оператора чувствительности:

```
@ (a, b, c)      // проверяется изменение уровня сигналов a, b и c  
@ (a or b or c) // то же самое  
@ (posedge clk1, negedge clk2) // проверяется возрастающий  
                                // фронт сигнала clk1 и падающий фронт сигнала  
                                // clk2  
@ (*)             // проверяется изменение уровня всех входных  
                                // сигналов
```

8.3. Оператор ожидания **wait**

Оператор ожидания **wait** имеет следующий формат:

```
wait (expression).
```

Данный оператор задерживает выполнение следующего оператора до тех пор, пока значение выражения **expression** истинно. В качестве выражения **expression** может выступать любое выражение.

Примеры использования оператора **wait**:

```
wire [7 : 0] addr;      // адресная шина  
wait (addr != 8'hFA)   // проверка значения на адреснойшине  
data = mem [addr];    // подключение ячейки памяти mem  
                      // с адресом addr кшине data
```

Обычно синтезаторы не поддерживают оператор задержки **#** и оператор ожидания **wait**. Однако список чувствительности в процедурном блоке **always** при синтезе должен поддерживаться всеми компиляторами. Поэтому далее остановимся более подробно на рассмотрении списка чувствительности и той роли, которую он играет при синтезе комбинационных и последовательностных схем.

8.4. Список чувствительности

Список чувствительности служит для приостановления (задержки) выполнения процедурного оператора до тех пор, пока не наступит определенное событие, связанное с изменением значений входных сигналов.

Замечание. Здесь под входными сигналами понимаются такие сигналы, которые являются входными для данного процедурного оператора или процедурного блока.

Чаще всего список чувствительности используется в процедурном операторе **always**, поэтому в наиболее общем случае список чувствительности имеет следующий вид:

```
always @ (signal, signal, ...) statement;
```

где **always** — процедурный оператор, в котором используется список чувствительности; @ — специальный знак, который предшествует списку чувствительности; **signal**, **signal**, ... — список сигналов в круглых скобках, изменение значения любого из которых возобновит выполнение процедурного оператора; **statement** — один или группа процедурных операторов, выполнение которых приостанавливается данным списком чувствительности.

Замечание. Если процедурных операторов несколько, то они должны быть взяты в процедурные скобки (**begin-end** или **fork-join**).

Действие приведенного формата списка чувствительности следующее: выполнение оператора **statement** (или группы операторов) будет приостановлено до тех пор, пока какой-либо из сигналов, перечисленных в списке чувствительности, не изменит своего состояния.

Обычно рассмотренный формат используется при описании функционирования схем, чувствительных к уровню сигнала. Типичными представителями такого рода схем являются защелки (*latch*) или прозрачные триггеры.

Примеры использования списка чувствительности:

Листинг 8.1. Вариант сумматора со списком чувствительности

```
module adder_sen_list_1
  (input a, b, cin,
   output reg s, cout);
  always @ (a, b, cin)
    {cout,s} = a + b + cin;
endmodule
```

Реализация примера из листинга 8.1 приведена на рис. 8.1.

Листинг 8.2. Комбинационная схема с защелкой на выходе

```
module circ_latch (input a, b, c,
                   output reg f);
```

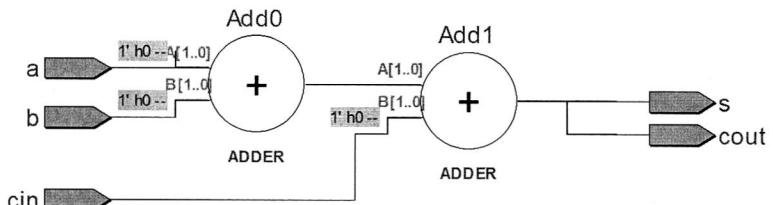


Рис. 8.1. Реализация модуля *adder_sen_list_1* из листинга 8.1: использование процедурного оператора **always** со списком чувствительности при описании однобитового сумматора

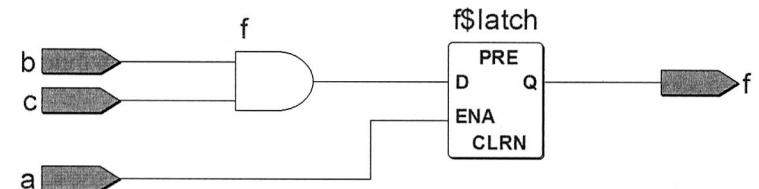


Рис. 8.2. Реализация модуля *circ_latch* из листинга 8.2: установка защелки на выходе в случае ошибки при описании комбинационной схемы

```
always @(a, b, c)
  if (a==1) f = b & c;
endmodule
```

Задание. Проанализируйте листинг 8.2 и попробуйте ответить на вопрос: почему синтезатор на выходе схемы поставит защелку?

Результат синтеза примера из листинга 8.2 приведен на рис. 8.2.

8.5. Список чувствительности в комбинационных схемах

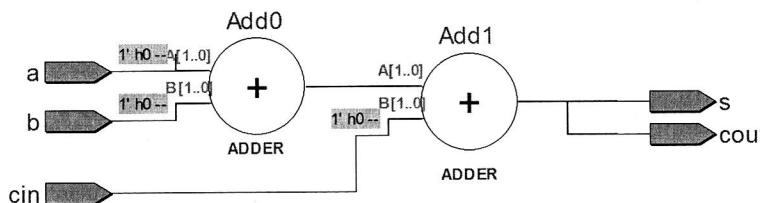
Если при описании комбинационной схемы в списке чувствительности будет отсутствовать хотя бы один из входных сигналов, то синтезатор автоматически на выходах схемы поставит триггеры.

Замечание. Здесь синтезатор — это программа (часть компилятора), которая выполняет синтез схемы по ее описанию на языке Verilog.

Чтобы этого не случилось, при описании комбинационных схем в списке чувствительности должны быть перечислены все входные сигналы. Однако на практике это правило разработчиками часто нарушается, например, при внесении изменений в проект отдельные сигналы могут быть добавлены в выражения, но не в список чувствительности.

Чтобы избежать подобных ошибок в язык Verilog-2001 были добавлены следующие форматы:

@ * и @ (*)

Рис. 8.3. Реализация модуля `adder_sen_list_2` из листинга 8.3

Оба формата равносильны. Звездочка после знака @ (в скобках или без скобок) означает, что список чувствительности включает все входные сигналы данного процедурного оператора. Эти форматы списка чувствительности как раз и используются для описания комбинационных схем.

Примеры использования списка чувствительности в формате @ (*).

Листинг 8.3. Второй вариант сумматора со списком чувствительности

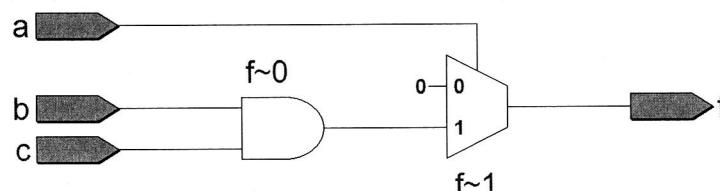
```
module adder_sen_list_2
  (input a, b, cin,
   output reg s, cout);
  always @(*)
    {cout,s} = a + b + cin;
endmodule
```

Реализация примера из листинга 8.3 приведена на рис. 8.3.

Листинг 8.4. Вариант схемы без защелки на выходе, т. е. комбинационной схемы

```
module circ_no_latch (input a, b, c,
                      output reg f);
  always @(*)
    if (a==1) f = b & c;
    else f = 0;
endmodule
```

Реализация примера из листинга 8.4 приведена на рис. 8.4.

Рис. 8.4. Реализация модуля `circ_no_latch` из листинга 8.4: устранение ошибки при описании комбинационной схемы — защелка на выходе отсутствует

Задание. Проанализируйте листинг 8.4 и попробуйте ответить на вопрос: почему синтезатор на выходе схемы защелку не поставил?

8.6. Список чувствительности в последовательностных схемах

Последовательностные (sequential) схемы, как правило, управляются переключением или фронтами сигналов. К таким схемам относятся триггеры, регистры, счетчики, конечные автоматы, контроллеры и др. Для управления временем при описании таких устройств используется следующий формат списка чувствительности:

`always @ (posedge signal, negedge signal, ...)`

Для каждого сигнала в списке чувствительности может быть определен либо возрастающий (posedge), либо падающий (negedge) фронт, но не оба фронта одновременно для одного и того же сигнала. Кроме того, тип фронта сигнала должен быть указан для каждого сигнала в списке. Другими словами, в одном и том же списке сигналов нельзя смешивать управление уровнями и фронтами сигналов.

Задание. Проверьте, допускает ли используемый вами компилятор:

- управление как по возрастающему, так и по убывающему фронту для одного и того же сигнала;
- использование в одном и том же списке чувствительности управления по фронту сигналов и по уровням сигналов для различных сигналов;
- использование в одном и том же списке чувствительности управления по фронту сигналов и по уровням сигналов для одинаковых сигналов.

Пример использования списка чувствительности в последовательностных схемах.

Листинг 8.5. Вариант D-триггера

```
module my_DFF_no_blocking
  (input D, clk,
   output reg Q);
  always @ (posedge clk)
    Q <= D;           // оператор неблокирующего назначения
endmodule
```

Реализация примера из листинга 8.5 приведена на рис. 8.5.

Листинг 8.6. Вариант D-триггера

```
module my_DFF_blocking (input D, clk,
                        output reg Q);
  always @ (posedge clk)
```

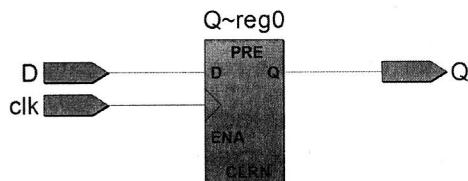


Рис. 8.5. Реализация модуля
my_DFF_no_blocking из листинга 8.5: пример реализации D-триггера с помощью неблокирующего оператора назначения

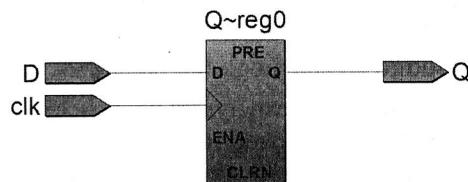


Рис. 8.6. Реализация модуля
my_DFF_blocking из листинга 8.6: пример реализации D-триггера с помощью блокирующего оператора назначения

```

Q = D;           // оператор блокирующего назначения
endmodule

```

Реализация примера из листинга 8.6 приведена на рис. 8.6.

Замечание. Различие между процедурными операторами блокирующего и неблокирующего назначения рассматриваются в следующей главе.

Г л а в а 9

Операторы процедурного назначения

9.1. Общие положения

Как и следует ожидать, основными операторами в процедурных блоках являются операторы назначения. В языках программирования обычно имеется только один оператор назначения, обозначаемый знаком равенства (`==>`). В общем случае операторы процедурного назначения языка Verilog предназначены для присваивания переменным определенных значений. Однако при описании цифровых устройств и систем этого часто бывает недостаточно. Поэтому для более точного описания поведения цифровых устройств в языке Verilog введено сразу несколько операторов процедурного назначения, которые имеют свои особенности, как при моделировании, так и при синтезе.

Прежде всего, в языке Verilog операторы назначения делятся на блокирующие (*blocking*), обозначаемые символом `==>`, и неблокирующие (*non-blocking*), обозначаемые символом `<=>`. Данная особенность операторов назначения может влиять не только на результаты моделирования, но также и на результаты синтеза.

Второй существенный момент — это выполнение операторов процедурного назначения по отношению к операторам управления процедурным временем (*timing control*): задержки `#`, чувствительности `@` и ожидания `wait`. В языке Verilog операторы управления процедурным временем могут ставиться как перед операторами процедурного назначения, так и внутри их: перед выражением в правой части оператора назначения. В последнем случае оператор управления временем называется *внутренней задержкой* (*intra-assignment delay*).

В случае использования операторов процедурного назначения для моделирования проекта следует также учитывать тип операторных скобок: `begin-end` или `fork-join`.

Кроме того, в дополнение к имеющимся возможностям язык Verilog предоставляет две пары операторов `assign-deassign` и `force-release` для выполнения специальных процедурных назначений. Опе-

ратор **assign** в процедурных блоках называется процедурным оператором **непрерывного назначения** и его действие отличается от действия обычного оператора непрерывного назначения **assign**. Оператор **force** позволяет назначать переменным или сетям любые типы данных.

Пользователь языка Verilog должен очень хорошо понимать действие операторов процедурного назначения, как при синтезе, так и при моделировании проекта. Поэтому остановимся более подробно на рассмотрении свойств операторов процедурного назначения языка Verilog.

9.2. Оператор блокирующего назначения «=»

9.2.1. Формат

Процедурный оператор блокирующего назначения имеет следующий формат:

```
variable = expression;
```

где **expression** — некоторое выражение; **variable** — переменная (типа **reg**), которой будет присвоено значение вычисленного выражения.

Замечание. В левой части процедурных операторов назначения можно использовать только переменные типа **reg**, даже в случае реализации комбинационных схем.

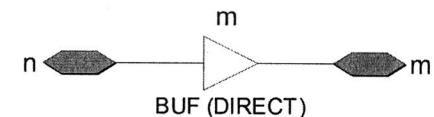
Действие блокирующего процедурного оператора назначения следующее. Когда симулятор встречает в коде данный оператор, то выполняется вычисление выражения **expression** и полученное значение присваивается переменной **variable**. В группе **begin-end** последовательно выполняемых операторов выполнение следующего оператора блокируется до тех пор, пока не завершится выполнение процедуры присваивания. Отсюда следует название данного процедурного оператора назначения — **блокирующий (blocking)**.

В примере из листинга 9.1 первое назначение передает значение сигнала с вывода **n** на вывод **m**, при этом блокируется выполнение второго назначения. Затем второе назначение передает значение сигнала с вывода **m** на вывод **n**. Благодаря процедурным скобкам **begin-end** подобные действия повторяются в виде бесконечного цикла.

Листинг 9.1. Первое назначение изменяет **m**, затем второе назначение изменяет **n**

```
module blocking_in_out ( inout reg n,m);
  always begin
    m = n;
    n = m;
  end
endmodule
```

Рис. 9.1. Реализация модуля **blocking_in_out** из листинга 9.1: пример назначения сигналов в блоке **always** с помощью блокирующего оператора назначения



Реализуемая синтезатором схема по описанию из листинга 9.1 показана на рис. 9.1.

Замечание. Анализируя рис. 9.1, можно сделать вывод, что в данном примере синтезатор реализовал только первый оператор непрерывного назначения.

Приведем еще несколько примеров модулей с использованием операторов блокирующего назначения.

Листинг 9.2. Инвертирование входов

```
module not_inputs(
  input a,b,c,d,
  output reg w,x,y,z);
  always begin
    w = ~ a;
    x = ~ b;
    y = ~ c;
    z = ~ d;
  end
endmodule
```

Листинг 9.3. Объединение входов с помощью вентилей

```
module gating_outputs(
  input a,b,c,d,
  output reg w,x,y,z);
  always begin
    w = a & b;
    x = b | c;
    y = c ^ d;
    z = d ~& a;
  end
endmodule
```

Листинг 9.4. Последовательное объединение входов и выходов с помощью вентилей

```
module gating_in_out_1(
  input a,b,c,d,
  output reg w,x,y,z);
  always begin
    w = a & b;
```

```

x = w | c;
y = x & d;
z = y ~& a;
end
endmodule

```

Листинг 9.5. Еще один пример последовательного соединения вентилей

```

module gating_in_out_2(
    input a,b,c,d,
    output reg z);
    reg w,x,y;
    always begin
        w = a & b;
        x = w | c;
        y = x & d;
        z = y ~& a;
    end
endmodule

```

Листинг 9.6. Соединение нескольких выходов

```

module connect_outs(
    input a,b,c,d,
    output reg y,z);
    always begin
        y = ! a;
        y = ! b;
        z = ! c;
        z = !d;
    end
endmodule

```

Листинг 9.7. Описание простой комбинационной схемы

```

module comb_circuit(
    input data,
    output reg c, d);
    always @(*)
    begin
        c = data;
        d = ~ c & ~ data;
    end
endmodule

```

Задания.

1. В приведенных выше примерах постарайтесь предугадать результаты синтеза.

2. Выполните компиляцию приведенных выше примеров и просмотрите результаты синтеза на уровне регистровых передач (RTL). Объясните полученные результаты. Совпадли ли результаты с вашими предположениями?

Замечание. Результаты синтеза примеров на листингах 9.2–9.7 приведены в конце данной главы.

9.2.2. Управление временем

Как было отмечено выше, операторы управления временем (#, @, wait) могут находиться как перед оператором назначения, так и перед выражением. В первом случае процедурный оператор блокирующего назначения имеет следующий формат:

timing_control variable = expression;

где *timing_control* — один из операторов управления процедурным временем #, @ или wait; *variable* — переменная типа *reg*; *expression* — некоторое выражение.

Когда симулятор встречает в коде подобную конструкцию, то вычисление значения выражения *expression* задерживается на время, указанное в операторе управления процедурным временем *timing-control*.

В качестве примера рассмотрим следующий фрагмент кода:

```

# 10 x = a;
@(posedge clk) y = b;

```

Когда симулятор встретит первую строку кода, то вначале будет выполнена задержка в течение 10 единиц времени и только затем переменной *x* будет присвоено значение переменной *a*. Когда симулятор встретит вторую строку кода, то будет выполнено ожидание нарастающего фронта сигнала *clk*, после чего переменной *y* будет присвоено значение переменной *b*. Общим свойством данных операторов назначения является то, что перед выполнением присваивания осуществляется либо ожидание (блокирование) на указанное время задержки, либо ожидание наступления определенного события. Другими словами, значение выражения справа от оператора «==» не вычисляется до тех пор, пока не закончится время ожидания.

На листинге 9.8 показана реализация рассмотренного фрагмента кода.

Листинг 9.8. Блокирующие процедурные операторы совместно с операторами управления временем

```

module blocking_st (
    input a, b, clk,
    output reg x, y);
    always begin

```

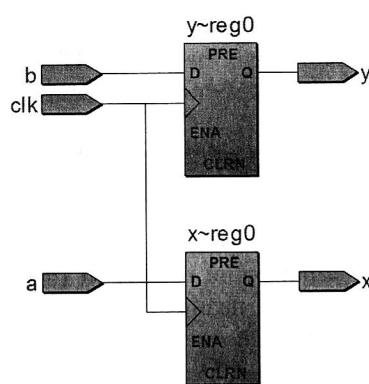


Рис. 9.2. Реализация модуля `blocking_st` из листинга 9.8: пример использования блокирующих операторов назначения вместе с операторами управления временем

следующие присваивания переменной *x* значения входа *a* произойдет не раньше, чем придет возрастающий фронт синхросигнала *clk*. Отсюда в схеме появляется триггер между входом *a* и выходом *x*, управляемый возрастающим фронтом синхросигнала *clk*.

9.2.3. Внутренние задержки

Когда в операторе процедурного назначения оператор управления временем ставится перед выражением, то такая задержка называется *внутренней* (*intra-assignment delay* или *intra-assignment timing control*). Оператор блокирующего назначения в случае использования внутренних задержек имеет следующий формат:

variable = *timing_control expression*;

Данная конструкция работает следующим образом. В момент времени, когда в коде симулятор встречает оператор назначения, вычисляется выражение *expression*, а присваивание вычисленного значения переменной *variable* выполняется после истечения времени, указанного в операторе управления временем *timing_control*. В группе **begin-end** последовательно выполняемых операторов выполнение следующего оператора блокируется до тех пор, пока не завершится выполнение присваивания (по истечении времени задержки).

В качестве примера рассмотрим следующий фрагмент кода:

```
x = # 10 a;
y = @(posedge clk) b;
```

Когда симулятор встретит первый оператор, то вначале будет вычислено значение выражения *a* и это значение будет присвоено неко-

```
# 10 x = a;
@(posedge clk) y = b;
end
endmodule
```

Реализуемая синтезатором схема по описанию из листинга 9.8 показана на рис. 9.2.

Попытаемся объяснить результат работы синтезатора. Прежде всего, синтезатор игнорирует оператор ожидания `#10`. На первый взгляд может показаться, что выход *x* в схеме на рис. 9.2 должен быть соединен простой цепью со входом *a*. Однако следует учесть, что блок операторов `always` выполняется в виде бесконечного цикла. Поэтому второе и все последующие присваивания переменной *x* значения входа *a* произойдет не раньше, чем придет возрастающий фронт синхросигнала *clk*. Отсюда в схеме появляется триггер между входом *a* и выходом *x*, управляемый возрастающим фронтом синхросигнала *clk*.

торой временной переменной. Затем выполняется ожидание в течение 10 единиц времени. В этот момент выполнение процесса, т.е. следующего оператора, будет заблокировано. После окончания времени ожидания переменной *x* будет присвоено значение временной переменной. Действие первого оператора можно выразить следующим фрагментом кода:

```
begin
    aTemp = a;      /* вычисление выражения a и присвоение
                      результата временной переменной aTemp */
    # 10 x = aTemp; /* ожидание 10 единиц времени и присвоение
                      значения aTemp переменной x */
end
```

Когда симулятор встретит второй оператор, то вначале будет вычислено значение выражения *b* и это значение будет присвоено некоторой временной переменной. Затем выполняется ожидание переднего фронта синхросигнала *clk*. После прихода переднего фронта синхросигнала *clk* переменной *y* будет присвоено значение временной переменной. Действие второго оператора можно выразить следующим фрагментом кода:

```
begin
    bTemp = b;      /* вычисление выражения b и присвоение
                      результата временной переменной bTemp */
    @(posedge clk) y = bTemp; /* ожидание переднего фронта
                                сигнала clk и присвоение значения bTemp
                                переменной y */
end
```

На листинге 9.9 показана реализация рассмотренного фрагмента кода.

Листинг 9.9. Блокирующие процедурные операторы с внутренними задержками

```
module blocking_st_intra(
    input a, b, clk,
    output reg x, y);
    always begin
        x = # 10 a;
        y = @(posedge clk) b;
    end
endmodule
```

Реализуемая синтезатором схема по описанию из листинга 9.9 показана на рис. 9.3.

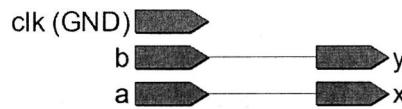


Рис. 9.3. Реализация модуля *blocking_st_intra* из листинга 9.9: пример использования блокирующих операторов назначения с внутренними задержками

Замечание. На основании рис. 9.3 можно сделать вывод, что синтезатор игнорирует внутренние задержки для операторов блокирующего назначения.

9.2.4. Особенности синтеза

Рассмотрим пример использования операторов блокирующего назначения, приведенных в листинге 9.10.

Листинг 9.10. Параллельная реализация триггеров с помощью процедурных операторов блокирующего назначения

```

module blocking_assignments(
    input data, clk,
    output reg x, y, w, z);
    always @(posedge clk)
    begin
        x = data;
        y = x;
        w = y;
        z = w;
    end
endmodule

```

Попытаемся предугадать результат синтеза по описанию, приведенному в листинге 9.10. Прежде всего, можно смело утверждать, что значения переменных *x*, *y*, *w* и *z* будут формироваться на выходах триггеров, которые управляются возрастающим фронтом синхронного сигнала *clk*, поскольку вся группа операторов назначения управляетяется оператором чувствительности @ (*posedge clk*). Первый оператор назначения присваивает значение входа *data* переменной *x*, при этом блокируется выполнение всех последующих операторов назначения. Второй оператор назначения должен присвоить значение переменной *x* переменной *y*, но значение переменной *x* уже известно и оно равно значению входной переменной *data*. Рассуждая подобным образом, можно прийти к выводу, что всем переменным *x*, *y*, *w* и *z* должно присваиваться значение одной и той же входной переменной *data*. Таким образом, схема, описанная в листинге 9.10, реализуется с помощью четырех триггеров, на выходах которых формируется значение переменных *x*, *y*, *w* и *z*, а на входы всех триггеров поступает значение входной переменной *data* (рис. 9.4).

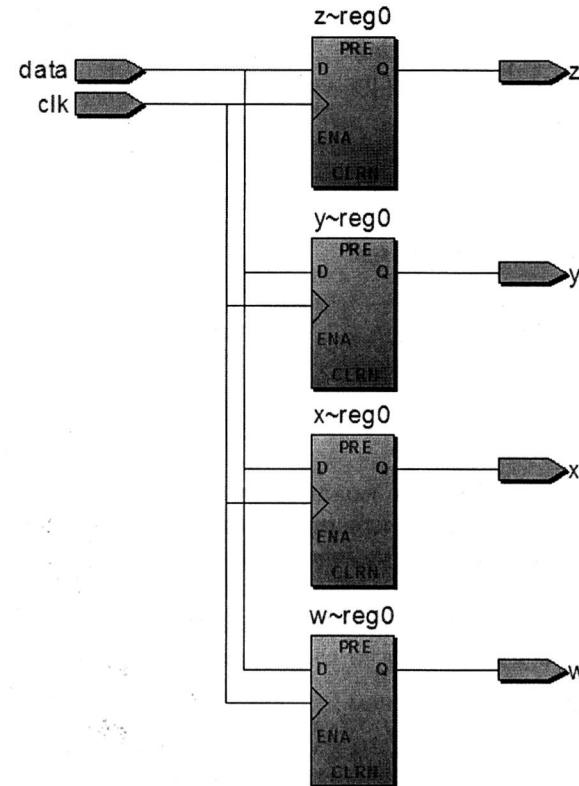


Рис. 9.4. Реализация модуля *blocking_assignments* из листинга 9.10: пример назначения сигналов в блоке *always*, чувствительного к фронту сигнала, с помощью блокирующих операторов назначения

9.3. Оператор неблокирующего назначения «*< =*»

9.3.1. Формат

Процедурный оператор неблокирующего назначения имеет следующий формат:

variable *< =* expression;

где *expression* — некоторое выражение; *variable* — переменная (типа *reg*), которой будет присвоено значение вычисленного выражения.

Действие неблокирующего процедурного оператора назначения следующее. Когда симулятор встречает в коде данный оператор, то выполняется вычисление выражения *expression*, но присваивание полученного значения переменной в левой части откладывается до окончания момента времени симуляции. В группе **begin-end** последовательно выполняемых операторов выполнение следующего оператора

не блокируется, отсюда название: неблокирующий (*non-blocking*) процедурный оператор назначения. Другими словами, следующий оператор будет выполняться без ожидания окончания выполнения присваивания.

Пример использования операторов неблокирующего назначения.

Листинг 9.11. Оба назначения будут вычисляться до изменения *n* и *m*

```
module non_blocking_in_out
  (inout reg n,m);
  always begin
    m <= n;
    n <= m;
  end
endmodule
```

В приведенном примере можно ожидать, что данные с двунаправленного вывода *n* будут передаваться на двунаправленный вывод *m* и одновременно с выводом *m* — на вывод *n*. Однако, поскольку по одной и той же линии физически невозможна одновременная передача данных в двух направлениях, синтезатор отказывается реализовывать представленное описание.

9.3.2. Управление временем

Оператор неблокирующего назначения в случае использования одного из операторов управления временем (*#*, *@*, *wait*) имеет следующий формат:

```
timing_control variable <= expression;
```

где *timing_control* — один из операторов управления процедурным временем *#*, *@* или *wait*; *variable* — переменная типа *reg*; *expression* — некоторое выражение.

Когда симулятор встречает в коде подобную конструкцию, то вычисление значения выражения *expression* задерживается на время, указанное в операторе управления процедурным временем *timing-control*.

В качестве примера рассмотрим следующий фрагмент кода:

```
# 10 x <= a;
@(posedge clk) y <= a & b;
```

Когда симулятор встретит первую линию кода, то вначале будет выполнена задержка в течении 10 единиц времени и только затем переменной *x* будет присвоено значение переменной *a*. При этом не блокируется выполнение следующего оператора. Во второй линии кода осуществляется ожидание нарастающего фронта сигнала *clk*, после

чего переменной *y* будет присвоено значение ранее вычисленного выражения *a & b*.

На листинге 9.12 показана реализация рассмотренного фрагмента кода.

Листинг 9.12. Блокирующие процедурные операторы совместно с операторами управления временем

```
module non_blocking_st(
  input a, b, clk,
  output reg x, y);
  always begin
    # 10 x <= a;
    @(posedge clk) y <= a & b;
  end
endmodule
```

Аналогично рассуждениям, приведенным для примера из листинга 9.8, можно сделать следующие выводы по виду синтезируемой схемы:

- значения переменных *x* и *y* будут формироваться на выходах триггеров, которые управляются возрастающим фронтом сигнала *clk*;
- входу триггера, на выходе которого формируется значение переменной *y*, будет предшествовать комбинационная схема в соответствии с приведенным описанием.

Реализуемая синтезатором схема по описанию из листинга 9.12 показана на рис. 9.5.

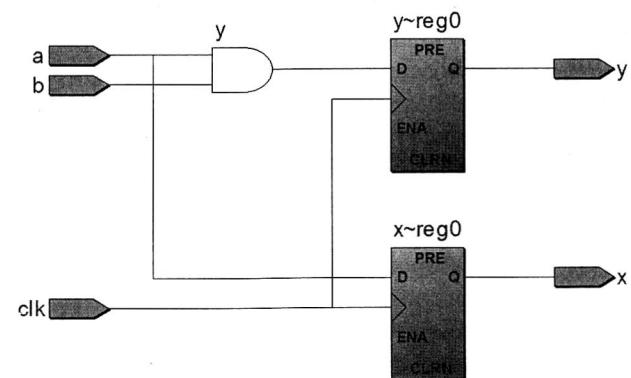


Рис. 9.5. Реализация модуля *non_blocking_st* из листинга 9.12: пример использования неблокирующих операторов назначения вместе с операторами управления временем

9.3.3. Внутренние задержки

Оператор неблокирующего назначения в случае использования внутренних задержек имеет следующий формат:

```
variable < = timing_control expression;
```

Данная конструкция работает следующим образом. В момент времени, когда в коде симулятора встречает оператор назначения, вычисляется выражение *expression*, а присваивание вычисленного значения переменной *variable* выполняется после истечения времени, указанного в операторе управления временем *timing_control*. В группе **begin-end** последовательно выполняемых операторов выполнение следующего оператора не блокируется.

Замечание. С помощью данного формата моделируются задержки передачи данных.

В качестве примера рассмотрим следующий фрагмент кода:

```
x <= # 10 a;
y <= @(posedge clk) a & b;
```

Когда симулятор встретит первый оператор, то вначале будет вычислено значение выражения *a* и это значение будет присвоено некоторой временной переменной. При этом выполнение процесса не блокируется: продолжается выполнение следующего оператора. Затем после истечения 10 единиц времени переменной *x* будет присвоено значение временной переменной.

На листинге 9.13 показана реализация рассмотренного фрагмента кода.

Листинг 9.13. Неблокирующие процедурные операторы с внутренними задержками

```
module non_blocking_st_intra(
    input a, b, clk,
    output reg x, y);
    always begin
        x <= # 10 a;
        y <= @(posedge clk) a & b;
    end
endmodule
```

Реализуемая синтезатором схема по описанию из листинга 9.13 показана на рис. 9.6.

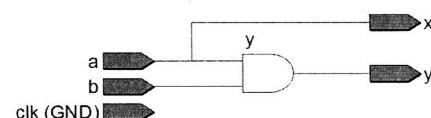


Рис. 9.6. Реализация модуля *non_blocking_st_intra* из листинга 9.13: пример использования неблокирующих операторов назначения с внутренними задержками

Замечание. На основании рис. 9.6 можно сделать вывод, что синтезатор игнорирует внутренние задержки для операторов неблокирующего назначения.

9.3.4. Особенности синтеза

Рассмотрим пример использования операторов неблокирующего назначения из листинга 9.14 (аналог листинга 9.10).

Листинг 9.14. Последовательная реализация триггеров с помощью процедурных операторов неблокирующего назначения

```
module non_blocking_assignments(
    input data, clk,
    output reg x, y, w, z);
    always @(posedge clk)
    begin
        x <= data;
        y <= x;
        w <= y;
        z <= w;
    end
endmodule
```

Попробуем предугадать результат синтеза описания, приведенного в листинге 9.14. Прежде всего, можно утверждать, что значение переменных *x*, *y*, *w* и *z* будут формироваться на выходах триггеров, которые управляются возрастающим фронтом синхросигнала *clk*, поскольку вся группа операторов назначения управляемася оператором чувствительности **@ (posedge clk)**. Первый оператор назначения присваивает значение входа *data* переменной *x*, при этом не блокируется выполнение всех последующих операторов назначения. Второй оператор назначения должен присвоить переменной *y* значение переменной *x*, но значение переменной *x* еще не известно, оно будет известно только в следующем такте после прихода возрастающего фронта синхросигнала *clk*. Поэтому вход триггера, на выходе которого формируется значение переменной *y*, следует подсоединить к выходу триггера, на котором формируется значение переменной *x*. Рассуждая подобным образом, можно прийти к выводу, что триггеры, на выходах которых формируются значения переменных *x*, *y*, *w* и *z*, должны соединяться последовательно, как показано на рис. 9.7.

Подведем некоторые итоги по использованию процедурных операторов назначения. Синтезатор обычно игнорирует операторы управления временем в операторах назначения. Исключение составляет оператор списка чувствительности **@**, который проверяет фронт

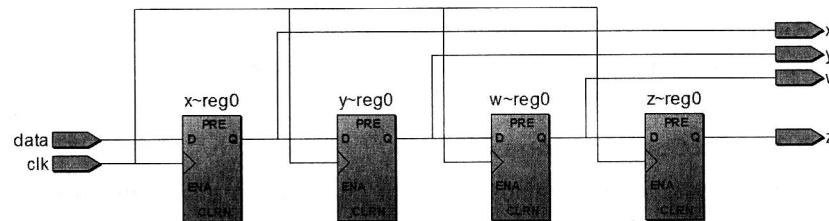


Рис. 9.7. Реализация модуля `non_blocking_assignments` из листинга 9.14: пример назначения сигналов в блоке `always`, чувствительного к фронту сигнала, с помощью неблокирующих операторов назначения (сравните данный пример с рис. 9.4)

сигнала синхронизации. Когда такой оператор управляет процедурным блоком, содержащим блокирующие операторы назначения, то в результате будет синтезирована схема из параллельных элементов памяти. В случае же неблокирующих операторов назначения — будет синтезирована схема из последовательных элементов памяти.

Замечание. Чтобы избежать потенциальных гонок в синтезируемом устройстве, а также для получения достоверных данных при моделировании с нулевым временем задержки, следует оператор блокирующего назначения (`==>`) использовать для описания комбинационных схем, а оператор неблокирующего назначения (`<<=`) использовать для описания последовательностных схем.

9.4. Управление временем в процедурных операторах назначения во время моделирования

Операторы управления временем в операторах назначения (как блокирующих, так и неблокирующих) наиболее часто используются при описании модулей генерации тестовых последовательностей, а также для указания момента присвоения значения некоторой переменной. При этом выполнение процесса может задерживаться (для блокирующих операторов назначения) или не задерживаться (для неблокирующих операторов назначения). Указанные возможности чаще всего применяются на этапе моделирования проекта.

В качестве примера рассмотрим описание однобитового сумматора из листинга 9.15.

Листинг 9.15. Однобитовый сумматор с блокирующими операторами назначения

```
'timescale 1ns/100ps
module add_1_blocking (
    input cin, a, b,
    output reg s, cout);
    always @ (cin, a, b) begin
        s = #5 a & b & cin;
        cout = #3 (a & b) | (b & cin) | (a | cin);
    end
endmodule
```

```
cout = #3 (a & b) | (b & cin) | (a | cin);
end
endmodule
```

Здесь для определения значения суммы и переноса использованы блокирующие операторы назначения с внутренними задержками. Поэтому, когда симулятор встретит в коде первый оператор назначения, то немедленно будет вычислено значение выражения `a & b & cin`, однако присвоение вычисленного выражения переменной `s` будет выполнено только через 5 наносекунд. Поскольку данный оператор назначения является блокирующим, то рассмотрение следующего оператора начнется не раньше, чем через 5 наносекунд. Следовательно, через 5 наносекунд будет вычислено выражение `(a & b) | (b & cin) | (a | cin)`, а присвоение значения переменной `cout` будет выполнено еще через 3 наносекунды, т. е. через 8 наносекунд после того, как случилось изменение значений сигналов на входе.

Неблокирующие операторы назначения позволяют вычислять значение выражения следующего оператора без ожидания окончания передачи значения левой стороне первого оператора. В листинге 9.16 приведено описание однобитового сумматора с неблокирующими операторами назначения.

Листинг 9.16. Однобитовый сумматор с неблокирующими операторами назначения

```
'timescale 1ns/100ps
module add_1_non_blocking (
    input cin, a, b,
    output reg s, cout);
    always @ (cin, a, b) begin
        s <= #5 a & b & cin;
        cout <= #8 (a & b) | (b & cin) | (a | cin);
    end
endmodule
```

Когда симулятор встретит в коде первый оператор назначения, будет также немедленно вычислено выражение правой стороны, присвоение вычисленного значения переменной `s` будет отложено на 5 наносекунд, однако не будет заблокировано выполнение следующего оператора. Другими словами, выражения для переменных `s` и `cout` будут вычислены одновременно, но значение переменной `s` будет присвоено через 5 наносекунд, а переменной `cout` — через 8 наносекунд. Таким образом, функционирование во времени проектов из листингов 9.15 и 9.16 будет одинаковым.

Задание. Проверьте, как изменится функционирование проекта на листинге 9.16, если во втором операторе значение внутренней задержки изменить на 3 наносекунды?

Приведем еще несколько примеров использования операторов управления временем в процедурных операторах назначения при моделировании.

always begin

```
#5 w = ~ a; // w будет присвоено значение !a через 5 ед. времени
#5 x = ~ b; // x будет присвоено значение !b через 10 ед. времени
#5 y = ~ c; // y будет присвоено значение !c через 15 ед. времени
#5 z = ~ d; // z будет присвоено значение !d через 20 ед. времени
end
```

always begin

```
#5 w <= ~ a; // w будет присвоено значение !a через ? ед. времени
#5 x <= ~ b; // x будет присвоено значение !b через ? ед. времени
#5 y <= ~ c; // y будет присвоено значение !c через ? ед. времени
#5 z <= ~ d; // z будет присвоено значение !d через ? ед. времени
end
```

always fork

```
#5 w = ~ a; // w будет присвоено значение !a через ? ед. времени
#5 x = ~ b; // x будет присвоено значение !b через ? ед. времени
#5 y = ~ c; // y будет присвоено значение !c через ? ед. времени
#5 z = ~ d; // z будет присвоено значение !d через ? ед. времени
```

join

always fork

```
#5 w <= ~ a; // w будет присвоено значение !a через ? ед. времени
#5 x <= ~ b; // x будет присвоено значение !b через ? ед. времени
#5 y <= ~ c; // y будет присвоено значение !c через ? ед. времени
#5 z <= ~ d; // z будет присвоено значение !d через ? ед. времени
```

join

Задание. Выполните симуляцию приведенных выше примеров с помощью используемого вами программного средства проектирования и замените знаки вопросов конкретными значениями. Совпадали ли результаты с вашими ожиданиями?

Примеры использования внутренних задержек в процедурных операторах назначения.

always begin

```
w = #5 ~ a; /* переменной w значение !a в момент времени 0
будет присвоено через 5 единиц времени */
x = #5 ~ b; /* переменной x значение !b в момент времени 5
будет присвоено через 10 единиц времени */
```

```
y = #5 ~ c; /* переменной y значение !c в момент времени 10
будет присвоено через 15 единиц времени */
z = #5 ~ d; /* переменной z значение !d в момент времени 15
будет присвоено через 20 единиц времени */
```

end

always begin

```
w <= #5 ~ a; /* w значение !a в момент времени ?
будет присвоено через ? единиц времени */
x <= #5 ~ b; /* x значение !b в момент времени ?
будет присвоено через ? единиц времени */
y <= #5 ~ c; /* y значение !c в момент времени ?
будет присвоено через ? единиц времени */
z <= #5 ~ d; /* z значение !d в момент времени ?
будет присвоено через ? единиц времени */
```

end

always fork

```
w = #5 ~ a; /* w значение !a в момент времени ?
будет присвоено через ? единиц времени */
x = #5 ~ b; /* x значение !b в момент времени ?
будет присвоено через ? единиц времени */
y = #5 ~ c; /* y значение !c в момент времени ?
будет присвоено через ? единиц времени */
z = #5 ~ d; /* z значение !d в момент времени ?
будет присвоено через ? единиц времени */
```

join

always fork

```
w <= #5 ~ a; /* w значение !a в момент времени ?
будет присвоено через ? единиц времени */
x <= #5 ~ b; /* x значение !b в момент времени ?
будет присвоено через ? единиц времени */
y <= #5 ~ c; /* y значение !c в момент времени ?
будет присвоено через ? единиц времени */
z <= #5 ~ d; /* z значение !d в момент времени ?
будет присвоено через ? единиц времени */
```

join

Задание. Выполните симуляцию приведенных выше примеров с помощью используемого вами программного средства проектирования и замените знаки вопросов конкретными значениями. Совпадали ли результаты с вашими ожиданиями?

9.5. Процедурные операторы assign и deassign

Процедурный оператор непрерывного назначения имеет следующий формат:

```
assign variable = expression;
```

Когда симулятор встречает в коде процедурного блока данный оператор, то он отвергает любые другие процедурные назначения переменной *variable* и выполняет присваивание значения выражения *expression* переменной *variable*.

В отличие от обычного оператора непрерывного назначения действие процедурного оператора непрерывного назначения может быть отменено с помощью оператора **deassign**.

Формат оператора **deassign**:

```
deassign variable;
```

Оператор **deassign** отменяет действие ранее встретившегося в коде процедурного оператора непрерывного назначения **assign** в отношении переменной *variable*.

В качестве примера использования операторов **assign** и **deassign** рассмотрим устройство, описанное в листинге 9.17, которое в зависимости от кода операции *op* выполняет либо сложение (*op=0*), либо вычитание (*op=1*) 8-и битовых операндов *a* и *b*. При этом при сложении учитывается перенос из предыдущего разряда *cin* и формируется сигнал переноса в следующий разряд *cout*, а операция вычитания заменяется операцией сложения по формуле $a - b = a + (\sim b) + 1$, где знак \sim обозначает операцию инвертирования всех разрядов операнда *b*.

Листинг 9.17. Устройство сложения и вычитания с операторами **assign** и **deassign**

```
module add_sub_1(
    input [7:0] a, b,           // операнды
    input cin, op,              // перенос из предыдущего разряда
                                // и код операции
    output reg [7:0] s,         // результат
    output reg cout);          // перенос в следующий разряд

always @(*) begin
    if (!op) begin
        deassign s;           // отмена предыдущего назначения
        deassign cout;         // отмена предыдущего назначения
        assign { cout,s } = a+b+cin; // присвоение нового значения
    end
    else begin
        deassign s;           // отмена предыдущего назначения
        deassign cout;         // отмена предыдущего назначения
    end
end
```

```
assign { cout,s } = a+(\sim b)+op; // присвоение нового значения
end
end
endmodule
```

Замечание. Не все компиляторы поддерживают процедурные операторы непрерывного назначения **assign** и **deassign**.

Задание. Проверьте, поддерживает ли используемый вами компилятор процедурные операторы непрерывного назначения **assign** и **deassign**.

9.6. Процедурные операторы force и release

Процедурный оператор **force** имеет следующий формат:

```
force net_or_variable = expression;
```

Данный оператор позволяет в процедурном блоке присвоить значение выражения *expression* любого типа переменной или сети слева от знака равенства ($\ll=$). При этом отвергаются любые другие назначения переменной.

Оператор **release** отменяет действие оператора **force**. Оператор **release** имеет следующий формат:

```
release net_or_variable;
```

В качестве примера использования операторов **force** и **release** рассмотрим устройство из листинга 9.17, в котором вместо операторов **assign** и **deassign** используются операторы **force** и **release**.

Листинг 9.18. Устройство сложения и вычитания с операторами **force** и **release**

```
module add_sub_2(
    input [7:0] a, b,           // операнды
    input cin, op,              // перенос из предыдущего разряда
                                // и код операции
    output reg [7:0] s,         // результат
    output reg cout);          // перенос в следующий разряд

always @(*) begin
    if (!op) begin
        release s;           // отмена предыдущего назначения
        release cout;         // отмена предыдущего назначения
        force { cout,s } = a+b+cin; // присвоение нового значения
    end
    else begin
        release s;           // отмена предыдущего назначения
        release cout;         // отмена предыдущего назначения
    end
end

force { cout,s } = a+(\sim b)+op; // присвоение нового значения
```

```

end
end
endmodule

```

Замечание. Не все компиляторы поддерживают процедурные операторы **force** и **release**.

Задание. Проверьте, поддерживает ли используемый вами компилятор операторы **force** и **release**.

В заключение данной главы приводим результаты синтеза примеров из листингов 9.2–9.7.

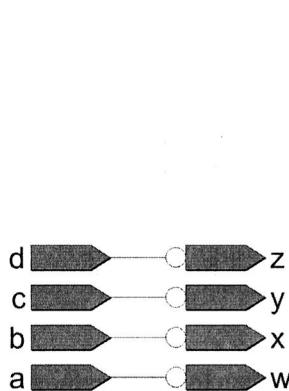


Рис. 9.8. Реализация модуля *not_inputs* из листинга 9.2: пример инвертирования входов

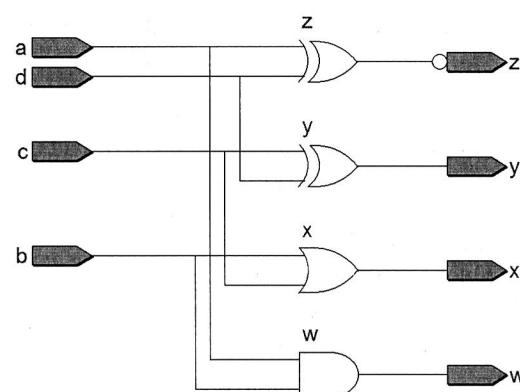


Рис. 9.9. Реализация модуля *gating_outputs* из листинга 9.3: объединение входов с помощью вентилей различного типа

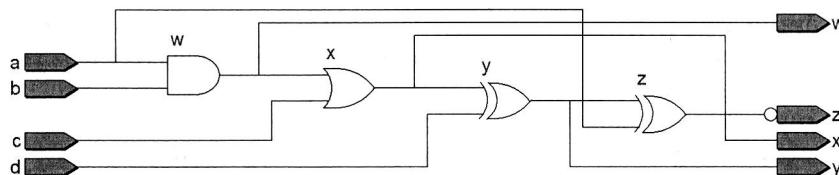


Рис. 9.10. Реализация модуля *gating_in_out_1* из листинга 9.4: последовательное объединение входов и выходов с помощью вентилей

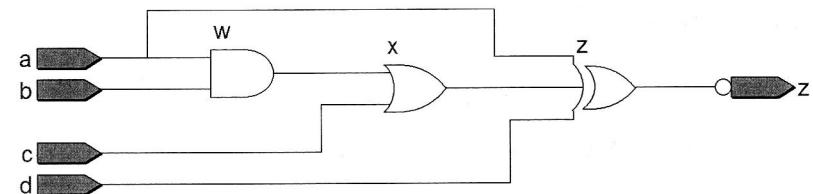


Рис. 9.11. Реализация модуля *gating_in_out_2* из листинга 9.5: последовательное объединение входов с помощью вентилей с использованием промежуточных переменных

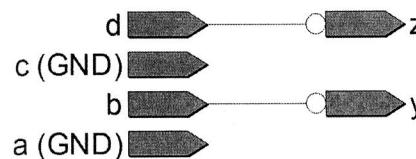


Рис. 9.12. Реализация модуля *connect_outs* из листинга 9.6: соединение выходов инверторов

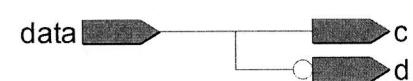


Рис. 9.13. Реализация модуля *comb_circuit* из листинга 9.7: пример описания простой комбинационной схемы

Г л а в а 10

Операторы процедурного программирования

10.1. Общие положения

Операторы процедурного программирования языка Verilog предназначены, в основном, для изменения порядка выполнения процедурных операторов. С помощью этих операторов осуществляются ветвления при последовательном выполнении операторов, образуются циклы, повторения, организуется выполнение отдельных фрагментов кода в случае удовлетворения определенным условиям и др. Большинство операторов процедурного программирования языка Verilog позаимствованы из языков программирования высокого уровня, однако имеются и некоторые особенности.

Язык Verilog включает следующие операторы процедурного программирования: **if-else**; **case**; **casex**; **casexz**; **for**; **while**; **repeat**; **forever**; **disable**.

Замечание. Не все операторы процедурного программирования являются синтезируемыми.

Задание. Узнайте, какие из операторов процедурного программирования для используемого вами компилятора являются синтезируемыми, а какие — нет.

Замечание. В рассматриваемых ниже форматах операторов процедурного программирования везде, вместо оператора **statement** может также находиться группа операторов, взятая в операторные скобки **begin-end** или **fork-join**.

10.2. Оператор **if-else**

Оператор **if-else** позволяет выполнять определенные фрагменты кода при выполнении заданных условий. Формат оператора **if-else**:

```
if (expression) statement
else statement
```

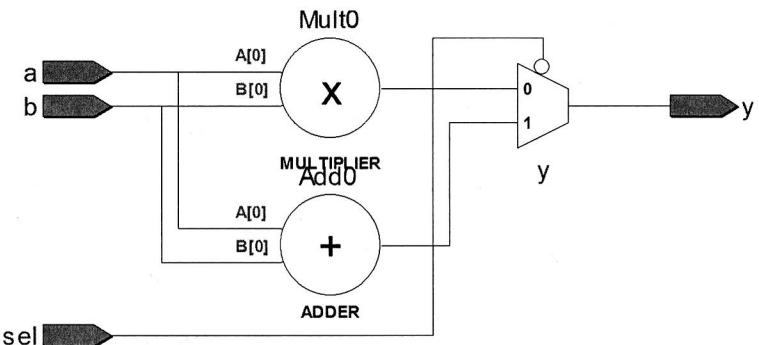


Рис. 10.1. Реализация модуля `adder_mult` из листинга 10.1: пример использования оператора **if** при описании АЛУ, реализующего арифметические операции сложения и умножения

где **if** и **else** — ключевые слова; **expression** — проверяемое выражение; **statement** — выполняемый оператор. Конструкция **else** и следующий за ней оператор могут отсутствовать.

Действие оператора **if-else**. Вначале вычисляется выражение **expression**. Если значение выражения **expression** истина (равно `1'b1`), то выполняется оператор **statement**, следующий за выражением. Если значение выражения **expression** ложь (равно `1'b0`) или неизвестно (равно `X`), то выполняется оператор **statement** после ключевого слова **else**; в случае отсутствия конструкции **else**, выполняется следующий оператор кода.

Примеры использования оператора **if-else**.

Листинг 10.1. Арифметическое устройство, реализующее либо сложение, либо умножение.

```
module adder_mult (input a, b, sel,
                    output reg y);
    always @(*) begin
        if (sel==0) y = a + b;
        else y = a * b;
    end
endmodule
```

Реализация примера из листинга 10.1 приведена на рис. 10.1.

Листинг 10.2. Параметризованный модуль двухходового мультиплексора

```
module mux_N_if #(parameter N=8) // N — ширина слов
    input [N-1:0] A, B,           // входные слова
    output reg [N-1:0] Y,          // выход
    input sel);                  // управляющий вход
    always @(*)                  // список чувствительности
```

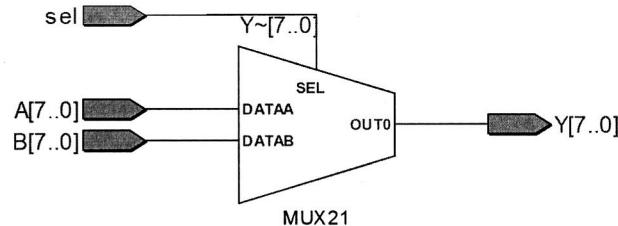


Рис. 10.2. Реализация модуля *mux_N_if* из листинга 10.2: пример реализации шинного мультиплексора с помощью оператора if

```
if (sel==1) Y = B;           // оператор if
else Y = A;
endmodule
```

Листинг 10.3. Использование вместо оператора if условной операции

```
module mux_N_conditional #(parameter N=8)// N — ширина слов
  (input [N-1:0] A, B,           // входные слова
   output [N-1:0] Y,             // выход
   input sel);                 // управляющий вход
  assign Y = (sel)? B: A;
endmodule
```

Реализации примеров из листингов 10.2 и 10.3 приведены на рис. 10.2 и 10.3 соответственно.

Анализ схем на рис. 10.2 и 10.3 показывает, что оператор if практически эквивалентен условной операции.

Листинг 10.4. Описание защелки

```
module latch_1 (output reg q,
  input data, en);
  always @(*)
    if (en==1) q = data;
endmodule
```

Реализация примера из листинга 10.4 приведена на рис. 10.4.

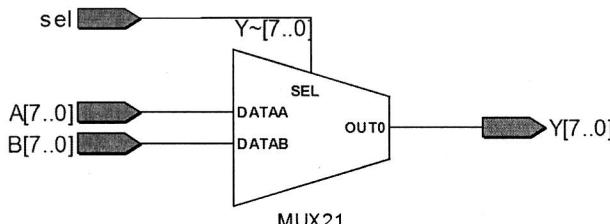
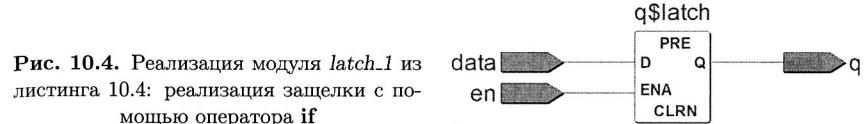


Рис. 10.3. Реализация модуля *mux_N_conditional* из листинга 10.3: пример реализации шинного мультиплексора с помощью условной операции

Рис. 10.4. Реализация модуля *latch_1* из листинга 10.4: реализация защелки с помощью оператора if



Замечание. Отсутствие в операторе if конструкции else при описании комбинационных схем приведет к тому, что синтезатор на выходе схемы установит защелки.

Следует быть внимательными при использовании оператора if для описания комбинационных схем. Дело в том, что если для некоторых входных наборов явно не определены значения выходов комбинационной схемы, то подразумевается, что для этих входных наборов значения выходов не изменятся, т. е. выходы сохранят предыдущие значения. Но сохранение значений выходов требует использование элементов памяти. В результате синтезатор установит на выходе схемы защелки, а схема перестанет быть комбинационной и перейдет в класс схем с памятью (т. е. последовательностных схем). Подобная ситуация случится обязательно, если в операторе if опустить конструкцию else.

Листинг 10.5. Ошибочное описание комбинационной схемы

```
module latch_2 (output reg y,
  input a, b, c, d, e);
  always @(*)
    if (a==1) y = b & c | d & e;
endmodule
```

Результат синтеза схемы из листинга 10.5 показан на рис. 10.5.

Рассмотрим несколько возможных способов использования оператора if-else для реализации комбинационных схем.

Листинг 10.6. Использование конструкции else при реализации комбинационной схемы

```
module comb_circuit_1 (output reg y,
  input a, b, c, d, e);
  always @(*)
    if (a==1) y = b & c | d & e;
    else y = 1;
```

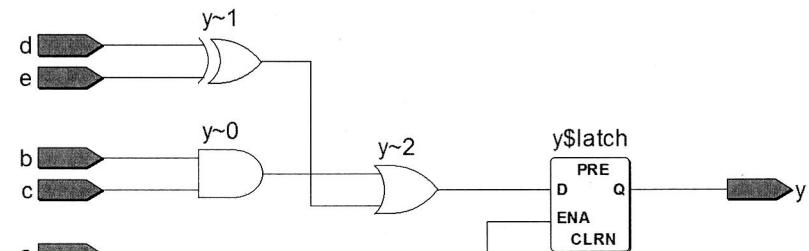


Рис. 10.5. Реализация модуля *latch_2* из листинга 10.5: результат ошибочного описания комбинационной схемы — на выходе схемы устанавливается защелка

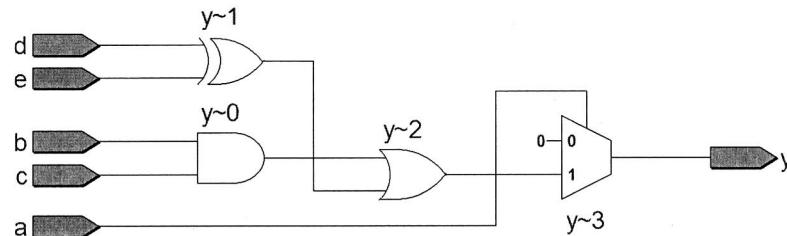


Рис. 10.6. Реализация модулей comb_circuit_1-3 из листингов 10.6–10.8

```
always @(*)
  if (a==1) y = b & c | d & e;
  else y = 0;
endmodule
```

Листинг 10.7. Описание комбинационной схемы с присваиванием начального значения

```
module comb_circuit_2 (output reg y,
  input a, b, c, d, e);
  always @(*) begin
    y = 0; // присваивание начального значения
    if (a==1) y = b & c | d & e;
  end
endmodule
```

Листинг 10.8. Описание комбинационной схемы с присваиванием значения управляющего сигнала

```
module comb_circuit_3 (output reg y,
  input a, b, c, d, e);
  always @(*)
    if (a==1) y = b & c | d & e;
    else y = a; // присваивание значения управляющего входа
endmodule
```

Результат синтеза примеров из листингов 10.6–10.8 одинаков и приведен на рис. 10.6.

Задание. Проанализируйте примеры на листингах 10.6–10.8. Отметьте сильные и слабые стороны каждого из способов описания комбинационной схемы. Сколько еще способов описания данной комбинационной схемы вы можете предложить? Какой из способов является лучшим?

10.3. Оператор case

Оператор **case** позволяет выполнять различные фрагменты кода в зависимости от значений вычисленного выражения. Формат оператора **case**:

```
case (expression)
  case_item: statement;
  ...
  case_item, ..., case_item:statement;
  default: statement;
endcase
```

где **case**, **endcase** и **default** — ключевые слова; **expression** — вычисляемое выражение; **case_item** — константный элемент; **statement** — выполняемый оператор. Конструкция, соответствующая ключевому слову **default** может отсутствовать. В качестве константного элемента **case_item** может выступать константа, константное выражение, вызов константной функции.

Действие оператора case. Вычисляется выражение **expression**. Полученное значение последовательно сравнивается с записанными ниже константными элементами. В случае совпадения значения выражения **expression** с одним из константных элементов **case_item**, выполняется оператор **statement**, непосредственно следующий за данным константным элементом. Если ни один из константных элементов не совпал с вычисленным значением выражения, то выполняется оператор после ключевого слова **default**.

Замечания.

1. Одному оператору может предшествовать несколько константных элементов. Оператор будет выполняться при совпадении значения выражения с любым из константных элементов.

2. После выполнения оператора, следующего за найденным константным элементом, остальные константные элементы не проверяются и выполнение оператора **case** заканчивается.

Примеры использования оператора case.

С целью сравнения возможностей операторов **case** и **if-else** при реализации комбинационных схем вначале рассмотрим описание булевой функции

$$y = f(a, b, c) = \sum m(a, b, c) = \sum (1, 2, 3, 4, 7) = \prod (0, 5, 6)$$

с помощью оператора **if-else**.

Листинг 10.9. Описание булевой функции *y* с помощью оператора **if-else**

```
module sm_IF (input a,b,c,
  output reg y);
  wire [2:0] w = {a,b,c}; // вспомогательный вектор,
                           // объединяющий входные сигналы
  always @(*) begin
    if (w==3'd0 || w==3'd5 || w==3'd6) y = 1'b0;
```

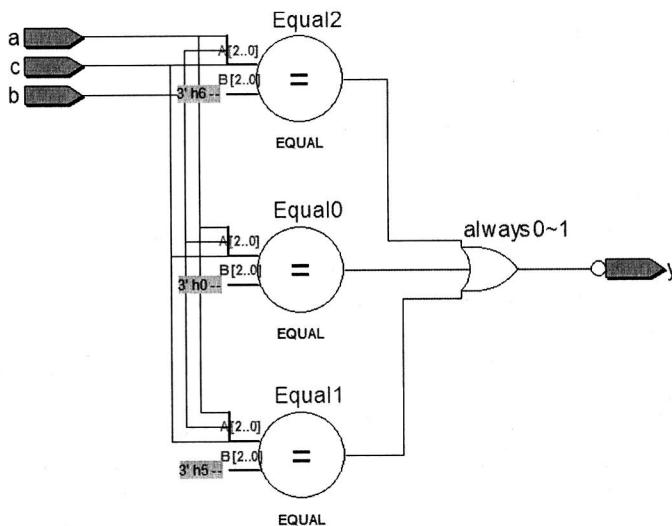


Рис. 10.7. Реализация модуля sm_IF из листинга 10.9: пример описания булевой функции с помощью оператора if

```

else y = 1'b1;
end
endmodule

```

Результаты синтеза схемы из листинга 10.9 показаны на рис. 10.7. А теперь рассмотрим различные способы описания той же функции y с помощью оператора case.

Листинг 10.10. Описание функции y с помощью оператора case в соответствии с таблицей истинности

```

module sm_Case1 (input a,b,c,
    output reg y);
    always @(*)
        case ({a, b, c}) // использование операции конкатенации
            3'b000: y = 1'b0; // все возможные входные комбинации
            3'b001: y = 1'b1; // и соответствующие им значения выходов
            3'b010: y = 1'b1;
            3'b011: y = 1'b1;
            3'b100: y = 1'b1;
            3'b101: y = 1'b0;
            3'b110: y = 1'b0;
            3'b111: y = 1'b1;
        endcase
    endmodule

```

Листинг 10.11. Описание функции y с использованием конструкции default

```

module sm_Case2 (input a,b,c,
    output reg y);
    always @(*)
        case ({a, b, c})
            3'b000: y = 1'b0; // анализируются только нулевые значения
            3'b101: y = 1'b0;
            3'b110: y = 1'b0;
            default: y = 1'b1; // единичное значение выхода принимается
                                по умолчанию
        endcase
    endmodule

```

Листинг 10.12. Вариант реализации предыдущего примера, когда одному оператору предшествует несколько константных элементов оператора case

```

module sm_Case3 (input a,b,c,
    output reg y);
    always @(*)
        case ({a, b, c})
            3'd0, 3'd5, 3'd6: y = 1'b0; // использование нескольких константных элементов
            default: y = 1'b1;
        endcase
    endmodule

```

В результате синтеза примеров из листингов 10.10–10.12 получается одна и та же схема, показанная на рис. 10.8.

Задание. Проанализируйте примеры из листингов 10.10–10.12. Отметьте сильные и слабые стороны каждого из способов описания комбинационной схемы. Сколько еще способов описания данной комбинационной схемы с использованием оператора case вы можете предложить? Какой из способов является лучшим?

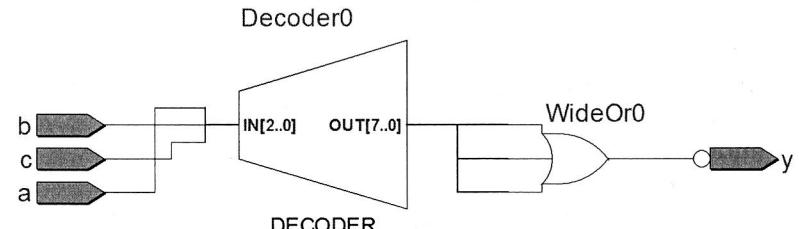


Рис. 10.8. Реализация модулей sm_Case1-3 из листингов 10.10–10.12: примеры описания булевой функции с помощью операторов case

Существенным недостатком оператора **case**, ограничивающим его использование, является то, что отдельные биты в значении вычисляемого выражения, а также отдельные биты константных элементов могут принимать только значения 0 и 1. С целью расширения возможностей оператора **case** в язык Verilog были введены операторы **casez** и **casex**.

10.4. Операторы casez и casex

Форматы операторов **casez** и **casex** полностью совпадают с форматом оператора **case**, за исключением того, что вместо ключевого слова **case** используются ключевые слова **casez** и **casex** соответственно.

Замечание. Операторы **casez** и **casex** заканчиваются одним и тем же ключевым словом **endcase**.

Оператор **casez** позволяет в вычисляемом выражении, а также в константных элементах использовать логическое значение **Z** для обозначения неопределенных (*don't care*) значений битов. Оператор **casex** подобен оператору **casez**, но позволяет использовать два логических значения **Z** и **X** для обозначения неопределенных (*don't care*) значений битов. Кроме того, в константных элементах операторов **casez** и **casex** для обозначения неопределенных и неизвестных значений битов можно использовать знак вопроса («?»).

Приведем несколько примеров использования операторов **casez** и **casex**.

Листинг 10.13. Пример использования неопределенных значений выходов в обычном операторе **case**

```
module sm_Case4 (input a,b,c,
    output reg f);
    always @(*)
    case ({a, b, c})
        3'b001: f = 1'b1;
        3'b010: f = 1'b1;
        3'b011: f = 1'b1;
        3'b100: f = 1'b1;
        3'b110: f = 1'b0;
        3'b111: f = 1'b1;
    default: f = 1'bx; // выход может иметь неопределенное
               // значение
    endcase
endmodule
```

Листинг 10.14. Пример использования неопределенных значений входов, использование оператора **casex**

```
module decoder (input [7:0] in,
    output reg [1:0] y);
    always begin
        casex (in) // входной вектор, значение которого проверяется
            8'b001100xx : y=2'b00;
            8'b1100xx00 : y=2'b01;
            8'b00xx0011 : y=2'b10;
            8'bxx001100 : y=2'b11;
        default: y=2'bxx;
    endcase
    end
endmodule
```

Листинг 10.15. Использование знака вопроса («?») в операторе **casex** при описании комбинационной схемы

```
module sm_Case5 (output reg f,
    input a, b);
    always @(*)
    casex ({a,b})
        2'b0? : f = 1'b1;
        2'b10 : f = 1'b0;
        2'b11 : f = 1'b1;
    endcase
endmodule
```

Результаты синтеза примеров из листингов 10.13–10.15 приведены на рис. 10.9–10.11 соответственно.

Задания.

- Попробуйте объяснить, почему при реализации примера из листинга 10.13 используется мультиплексор, а при реализации примеров из листингов 10.14–10.15 используются дешифраторы?
- Опишите с помощью оператора **casex** приоритетный шифратор на 8 входов.

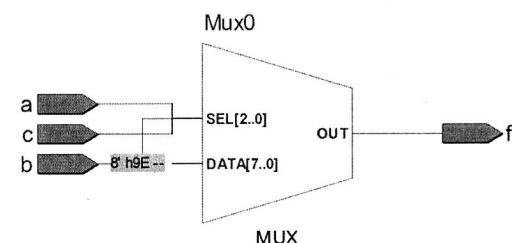


Рис. 10.9. Реализация модуля **sm_Case4** из листинга 10.13: пример использования неопределенных значений выходов в операторе **case**

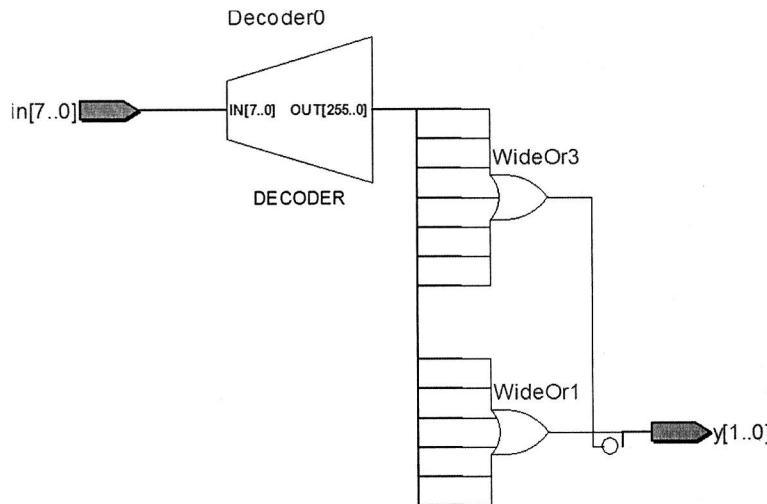


Рис. 10.10. Реализация модуля *decoder* из листинга 10.14: пример использования неопределенных значений входов в операторе *casex*

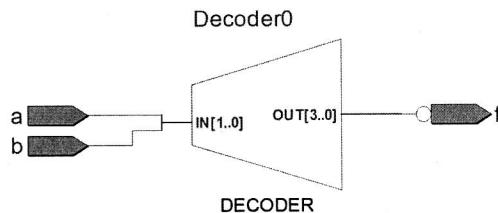


Рис. 10.11. Реализация модуля *sm_Case5* из листинга 10.15: пример использования знака «?» при описании значений входов в операторе *casex*

10.5. Оператор *for*

Оператор *for* предназначен для организации циклов. Его синтаксис, практически, совпадает с синтаксисом аналогичного оператора языка программирования С. Формат оператора *for*:

for (*initial_assignment*; *expression*; *step_assignment*) *statement*

где *initial_assignment* — оператор назначения, который выполняется один раз при старте оператора *for*; *expression* — выражение, значение которого проверяется перед выполнением каждого цикла; *step_assignment* — оператор назначения, который выполняется в конце каждого цикла.

Действие оператора *for*. При старте оператора *for* первым выполняется оператор *initial_assignment*, который присваивает начальное значение переменной цикла (обычно в качестве переменной цикла используется некоторая переменная типа *integer*). Затем вычис-

ляется значение выражения *expression*. Если вычисленное значение истина, то выполняется оператор *statement*; в противном случае выполняется оператор кода, следующий за оператором *for*. В каждом цикле после выполнения оператора *statement* выполняется оператор *step_assignment*, который обычно присваивает новое значение переменной цикла. После чего вновь проверяется выражение *expression* и процесс выполнения оператора *for* повторяется.

Замечание. Для увеличения (уменьшения) на единицу переменной цикла в операторе *for* нельзя использовать операции *++i* и *-i*, как в языке С, поскольку в языке Verilog нет операций *++* и *-*.

В отношении переменной цикла и используемых в операторе *for* операторов назначения, как правило, имеются и другие ограничения, накладываемые конкретными компиляторами.

Задание. Узнайте, какие ограничения на использование оператора *for* накладывает используемый вами компилятор.

Примеры использования оператора *for*.

Листинг 10.16. Определение числа нулей в 4-битовом слове с использованием цикла *for*

```

module count_zero1 (
    input [3:0] A,           // входное слово
    output reg [2:0] cnt);   // результат
    integer i;               // переменная цикла
    always @(*) begin
        cnt = 3'd0;          // начальное значение счетчика
        for (i=0; i < 4; i=i+1)
            if (!A[i]) cnt = cnt + 1; // подсчет числа нулей
        end
    endmodule

```

Результат синтеза примера из листинга 10.16 дан на рис. 10.12.

Замечание. Анализ рис. 10.12 показывает, что применение оператора *for* хотя и позволяет сократить описание, однако является достаточно затратным с точки зрения использования аппаратуры, а также приводит к построению многоуровневых схем, снижающих быстродействие проекта.

10.6. Оператор *while*

Оператор *while* предназначен для организации циклов, которые заканчиваются в случае невыполнения указанных условий. Формат оператора *while*:

while (*expression*) *statement*

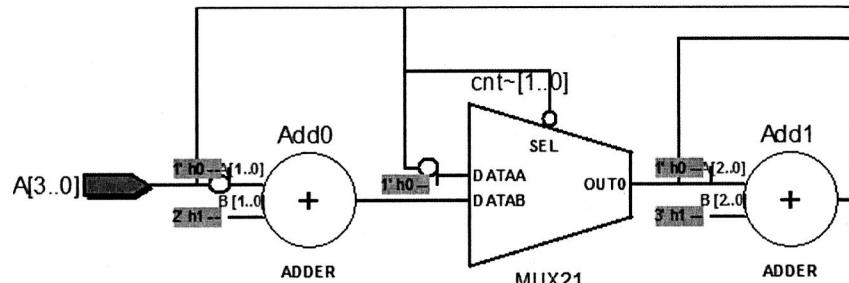


Рис. 10.12. Реализация модуля count_zero1 из листинга 10.16: пример

где **while** — проверяемое выражение; **statement** — выполняемый оператор (группа операторов).

Действие оператора **while**. Перед началом каждого цикла вычисляется значение выражения **expression**. Если оно истинно, то выполняется оператор **statement**; в противном случае выполняется оператор кода, следующий за оператором **while**. Другими словами, оператор **statement** выполняется до тех пор, пока значение выражения **expression** истинно.

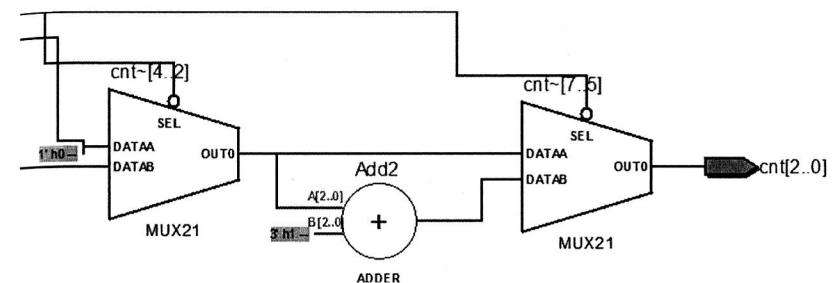
Замечание. Для окончания цикла **while** в группе операторов **statement** следует предусмотреть действия, которые изменяют значение выражения **expression**, иначе цикл оператора **while** будет выполняться бесконечно.

Пример использования оператора **while**:

Листинг 10.17. Определение числа нулей в 4-битовом слове с использованием цикла **while**

```
module count_zero2 (
    input [3:0] A,           // входное слово
    output reg [2:0] cnt);   // результат
    integer i;              // переменная цикла
    always @(*) begin
        i = 0;
        cnt = 3'd0;
        while (i < 4) begin
            if (!A[i]) cnt = cnt + 1;
            i = i + 1;
        end
    end
endmodule
```

В результате будет синтезирована точно такая же схема, как на рис. 10.12.



использования оператора **for** для определения числа нулей в 4-битовом слове

10.7. Оператор repeat

Оператор **repeat** предназначен для выполнения цикла заданное число раз. В отличие от оператора **while**, выполнение оператора **repeat** заканчивается всегда. Кроме того, в тело цикла нет необходимости вставлять операторы для изменения значения выражения **expression**. Единственный недостаток оператора **repeat**: перед его выполнением должно быть известно число повторений цикла. Формат оператора **repeat**:

repeat (*number*) **statement**

где *number* — число повторений цикла. Конструкция *number* в операторе **repeat** может быть выражением, значение которого вычисляется один раз при старте оператора **repeat** и не изменяется в процессе выполнения циклов.

Действие оператора **repeat**. Оператор **statement** выполняется столько раз, сколько повторений циклов указано с помощью числа или выражения *number*.

В качестве примера использования оператора **repeat** рассмотрим описание устройства умножения 8-битовых слов.

Листинг 10.18. Описание устройства умножения 8-битовых слов

```
module mult (
    output [15:0] y,           // выход
    input [7:0] a, b);         // входы
// использование функции для реализации алгоритма умножения
function [15:0] multiply (input [7:0] a, b);
begin: serialMult
    reg [7:0] mcnd, mpy;      // множитель
    mpy = b;                  // множимое
    mcnd = a;                 // результат умножения
    multiply = 0;
repeat (8) begin
    mpy = mpy + mcnd;
    mcnd = mcnd >> 1;
end
endfunction
```

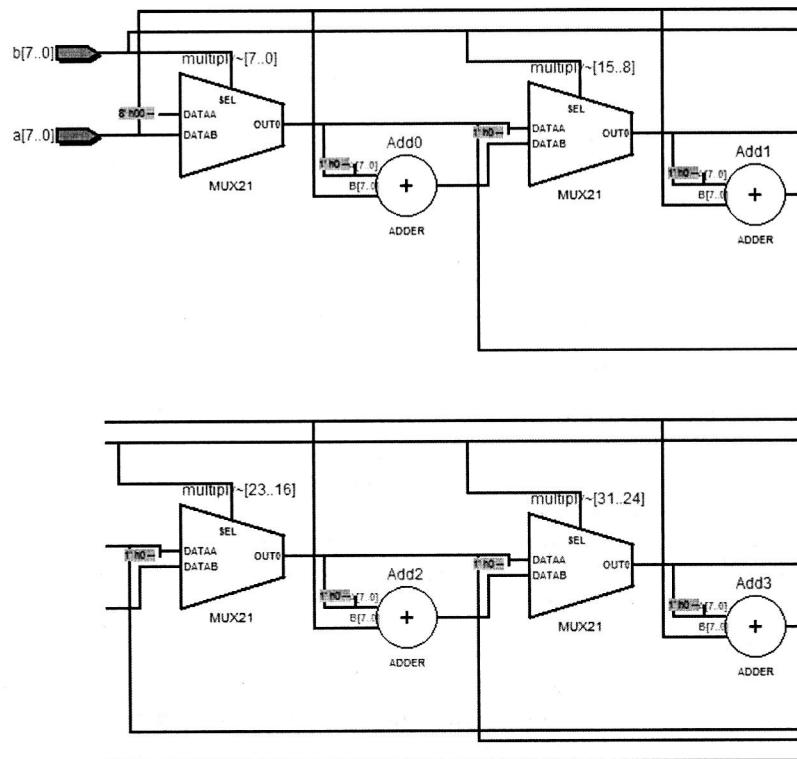


Рис. 10.13. Реализация модуля *mult* из листинга 1: пример использования

```

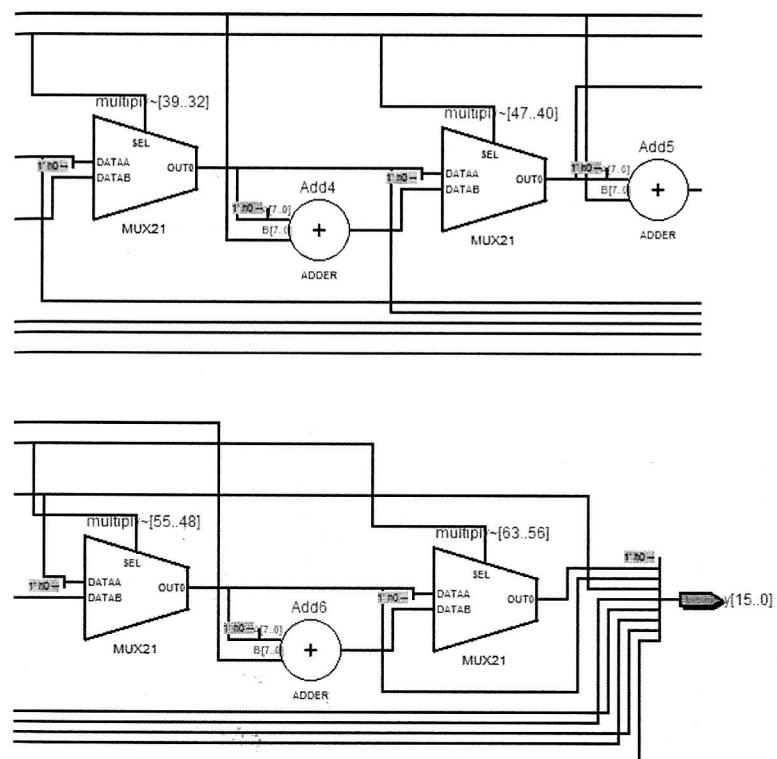
if (mpy[0])           // если младший бит равен 1, то
    multiply = multiply + {mcnd, 8'b00000000};
    multiply = multiply >> 1;
    mpy = mpy >> 1;
end
endfunction
assign y = multiply(a,b);
endmodule

```

Результат синтеза примера из листинга 10.18 дан на рис. 10.13.

Замечание. Как можно видеть из приведенных примеров, использование операторов цикла (**for**, **while** и **repeat**) приводит к построению многоуровневых каскадных схем с сумматором на каждом уровне. Поэтому некоторые компиляторы ограничивают максимальное число циклов при синтезе проектов.

Задание. Узнайте максимальное число циклов при синтезе для



оператора **repeat** для описания устройства умножения 8-битовых слов

каждого оператора **for**, **while** и **repeat**, которое поддерживает используемый вами компилятор.

10.8. Оператор forever

Оператор **forever** предназначен для организации бесконечных непрерывно повторяющихся циклов. Формат оператора **forever**:

forever statement

где *statement* — оператор, выполняемый в бесконечном цикле.

Действие оператора forever. При встрече в коде оператора **forever** оператор *statement* начинает выполняться бесконечное число раз, т. е. до окончания времени симуляции.

Замечание. Оператор **forever** часто используется для организации бесконечной последовательности периодических сигналов, например, синхросигналов.

Примеры использования оператора **forever**.

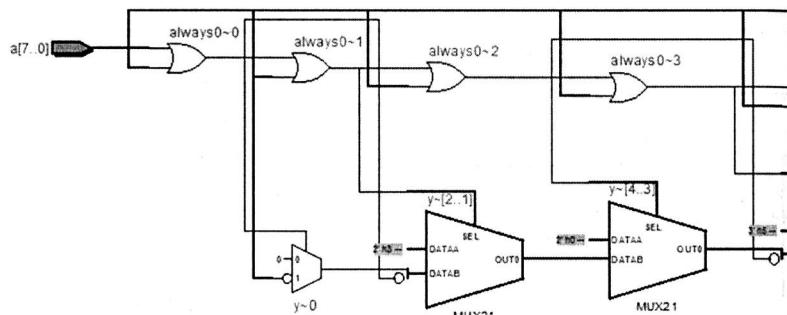


Рис. 10.14. Реализация модуля *index_of_left_one* из листинга 10.20: пример местоположения

Листинг 10.19. Генератор синхросигнала с периодом 50 временных единиц, Который старнет после 1000 временных единиц от начала симуляции.

```
module clock_oscillator (output reg clk);
initial begin
    clk = 0;
    #1000 forever #25 clk = ~ clk;
end
endmodule
```

Приведенный пример не может быть реализован синтезатором, поскольку синтезатор запрещает реализацию бесконечных циклов. Однако подобное описание может использоваться при моделировании проекта.

10.9. Оператор disable

Оператор **disable** предназначен для прекращения выполнения последовательности операторов в именованном блоке (*named block*). Формат оператора **disable**:

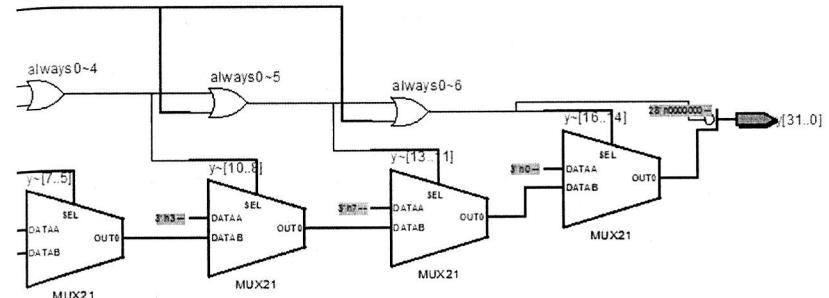
```
disable block_name
```

где *block_name* — имя именованного блока, выполнение операторов которого необходимо прекратить.

Действие оператора **disable**. Когда симулятор встречает в коде оператор **disable**, то осуществляется выход из группы операторов именованного блока с именем *block_name*, т. е. управление передается оператору, следующему за группой операторов с именем *block_name*.

Замечание. Оператор **disable** часто используется для принудительного выхода из некоторого цикла, завершения выполнения операторов цикла **for**, **while** и др.

Примеры использования оператора **disable**.



использования оператора **disable** в цикле **for** при поиске в битовом векторе первой единицы слева

Фрагмент кода при реализации функций операторов **continue** и **break** языка С с помощью оператора **disable**

```
begin: break
    for (i=0; i< n; i=i+1)
begin: cont
    if (a==0) disable cont; // переход к следующему циклу
                           // оператора for
                           // аналог команды continue языка С
    // другие операторы
    if (a==b) disable break; // окончание блока break
                           // аналог команды break языка С
    // другие операторы
end                                // end cont
end                                // end break
```

Листинг 10.20. Нахождение в битовом векторе местоположения первой единицы слева

```
module index_of_left_one # (parameter N=8)
(input [N-1:0] a,           // входной вектор
 output integer y);        // возвращаемое значение
always begin: block_1        // именованный блок block_1
    for (y=0; y< N; y=y+1)
        if (a[y]==1'b1)
            disable block_1;   // выход из блока block_1
        end                   // конец блока block_1
    endmodule
```

Результат синтеза примера из листинга 10.20 дан на рис. 10.14.

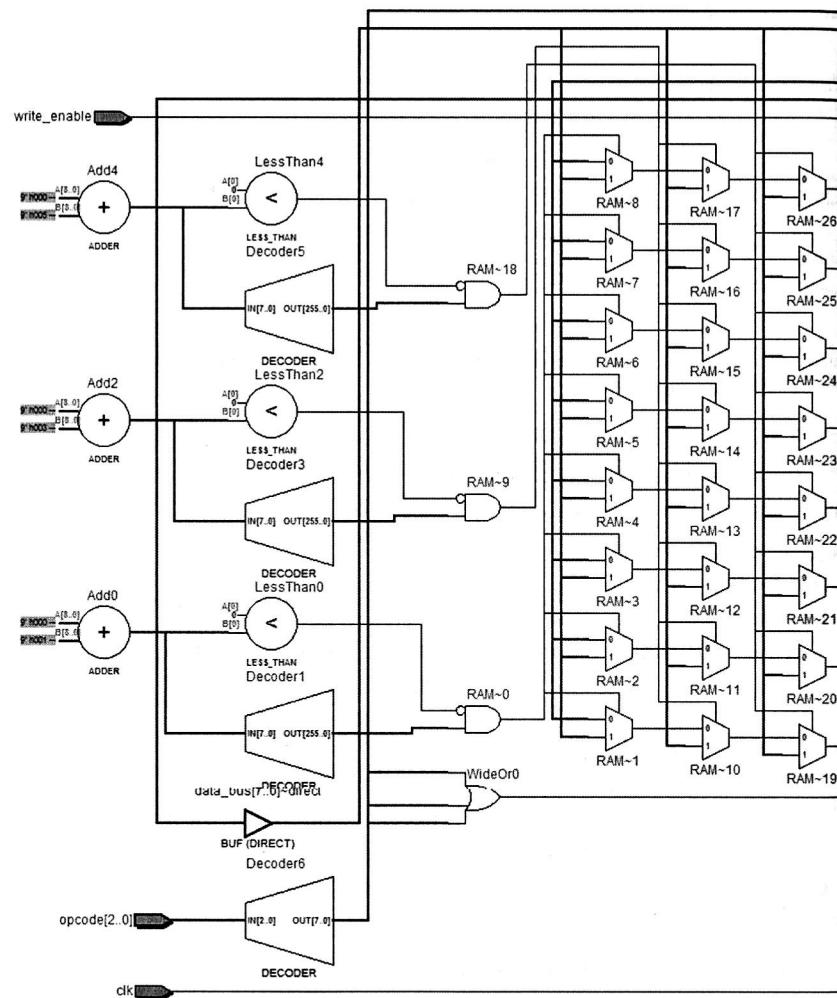


Рис. 10.15. Реализация модуля wr_re_mem из листинга 10.21:

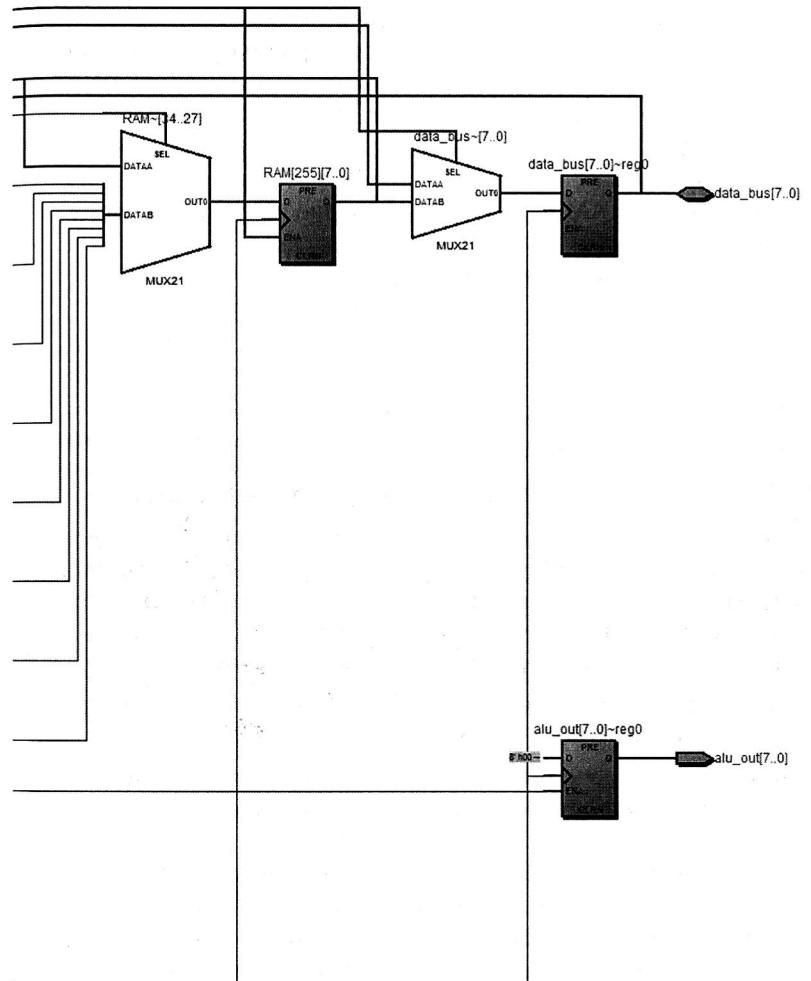
10.10. Пример использования операторов процедурного программирования

Рассмотрим пример совместного использования нескольких операторов процедурного программирования.

Листинг 10.21. Пример реализации операций записи и чтения из памяти

```
module wr_re_mem (
    input clk, write_enable,
    input [2:0] opcode,
```

// управляющие сигналы
// код операции



пример реализации операций записи и чтения из памяти

```
inout reg [7:0] data_bus,
output reg [7:0] alu_out); // шина данных
integer i; // выход АЛУ
reg [7:0] RAM [0:255]; // переменная цикла
reg [7:0] address = 0, // объявление памяти
        accum = 0; // переменная адреса
        // регистр аккумулятора
always @ (posedge clk) begin // управление возрастающим
    casez (opcode) // фронтом
        3'b1?: alu_out = accum; // проверка кода операции
        3'b0: data_bus = RAM[address]; // чтение значения из
```

```

// аккумулятора
3'b000: if (write_enable)          // проверка разрешения записи
    for (i=0; i<=5; i=i+1)        // запись пяти слов
        RAM[address+i] = data_bus; // с шины в память
3'b011: begin                     // чтение из памяти
    for (i=0; i<=255; i=i+1)      // организация цикла
        data_bus = RAM[i];         // чтение из памяти в шину
    end
default : $display("illegal opcode in module wr_re_mem");
           // сообщение об ошибке
endcase
end
endmodule

```

Результат синтеза примера из листинга 10.21 дан на рис. 10.15.

10.11. Различие между операторами **wait** и **while**

Различия между операторами **wait** и **while** лежат в природе этих операторов. Оператор **wait** управляет выполнением процесса, он может задержать выполнение процесса до момента удовлетворения некоторым условиям. В качестве таких условий могут выступать сигналы, являющиеся внешними по отношению к данному процессу.

Замечание. Процессами в языке Verilog являются процедурные операторы (блоки) **initial** и **always**.

Напомним, что процессы в языке Verilog выполняются параллельно и приостановление одного процесса не останавливает симуляцию всего проекта.

Оператор **while** управляет процессом выполнения блока процедурных операторов. При этом оператор **while** не приостанавливает выполнение данного процесса. Чтобы цикл **while** когда-либо закончился, в теле блока операторов должно быть предусмотрено изменение некоторой величины, которая проверяется в выражении оператора **while** перед выполнением каждого цикла. В то же время проверять в выражении оператора **while** значение сигналов, которые являются внешними по отношению к блоку операторов, а тем более к данному процессу, нельзя, поскольку это противоречит концепции оператора **while**.

Задание. Напишите фрагмент кода с оператором **while**, в котором проверяется значение сигнала: а) внешнего по отношению к данному оператору; б) внешнего по отношению к данному процессу. Какие сообщения при этом выдает компилятор?

Листинг 10.22. Пример использования оператора **while** для управления процессом.

```

module endlessloop (output reg [15:0] cnt);
reg a=0;
always #50 a=~ a; // первый процесс периодически изменяет
                   // значение a
always begin       // второй процесс
    cnt=0;
    while (a) begin // бесконечный цикл
        cnt=cnt+1;   // изменение значения переменной cnt
                       // не изменяет выражения
        $display("cnt=%0d", cnt);
    end
    $display("Это сообщение никогда не появится на экране.
Почему?");
end
endmodule

```

Задание. Проверьте, что произойдет с предыдущим примером, если оператор **while** заменить на оператор **wait**?

Г л а в а 11

Атрибуты

11.1. Атрибуты языка Verilog

Использование языка Verilog в практике инженерного проектирования показало, что часто отдельные элементы языка требуют уточнения при их реализации в компиляторах, синтезаторах, симуляторах и др. Для передачи такой уточняющей информации служат атрибуты. Другими словами, атрибуты языка Verilog определяют специфические свойства операторов и конструкций, реализованных в конкретном программном обеспечении, например, в компиляторе языка Verilog пакета Quartus II. Формат атрибутов языка Verilog:

(* attribute, attribute,...*)

где (*) — начало списка атрибутов; *) — конец списка атрибутов; attribute — определяемый атрибут.

Замечание. Атрибутные скобки (пары знаков «(*)» и «*)») являются обязательными даже при задании только одного атрибута.

Атрибуты языка Verilog могут приписываться в виде префикса к объявлениям, экземплярам модулей, операторам, портам и др. Атрибуты могут также приписываться в виде суффикса к операциям или вызовам функций. Атрибутам могут иметь значения. Если значение атрибута не определено, то по умолчанию принимается значение 1. Можно также одновременно определять несколько атрибутов. В этом случае в списке атрибутов они разделяются запятыми.

Примеры использования атрибутов:

```
(* INIT=>1 *) reg Q;           // использование атрибута при
                                // объявлении переменной Q
sum = a + (* ripple_adder *) b; // использование атрибута в выра-
                                // жении в виде суффикса
                                // к операции сложения
```

Атрибуты как возможная конструктивная единица впервые были введены в языке Verilog-2001. Однако стандарт языка Verilog-2001

Атрибуты

не определяет конкретные атрибуты. Конкретные атрибуты для языка Verilog вводят разработчики программного обеспечения (компиляторов, синтезаторов, симуляторов и др.), в котором поддерживается язык Verilog.

Задание. Узнайте атрибуты языка Verilog используемой вами системы проектирования.

Несмотря на то что конкретные атрибуты языка Verilog определяются разработчиками программных средств проектирования, все современные компиляторы языка Verilog поддерживают два атрибута: full_case и parallel_case.

11.2. Атрибут full_case

Использование оператора case при описании сложных комбинационных схем может привести к ситуации, когда значения выходов схемы определены не для всех наборов входных значений. Данная ситуация часто встречается в случае отсутствия в операторе case конструкции default. Однако, если не указать значения выходов схемы для всех возможных входных наборов, то синтезатор на выходе схемы автоматически поставит защелки (подобная ситуация в отношении оператора if рассмотрена в разделе 10.2). Чтобы избежать автоматического установления защелок на выходах комбинационных схем служит атрибут full_case.

Атрибут full_case указывает синтезатору трактовать значения выходов схемы как неопределенные (don't care) для входных наборов, которые не определены в операторах case. Рассмотрим пример использования атрибута full_case при описании комбинационной схемы.

Листинг 11.1. Пример использования атрибута full_case.

```
module use_full_case (input [2:0] data,
                      input [1:0] sel,
                      output reg y);
  always @ (*)
    (* full_case *) // применение атрибута full_case
    case (sel)      // к оператору case
      2'b00 : y = data[0];
      2'b01 : y = data[1];
      2'b10 : y = data[2];
      // отсутствует входная комбинация 2'b11
    endcase
endmodule
```

Результат синтеза примера из листинга 11.1 приведен на рис. 11.1.

Реализация этого же примера без атрибута full_case требует в два раза больше логических элементов и приведена на рис. 11.2.

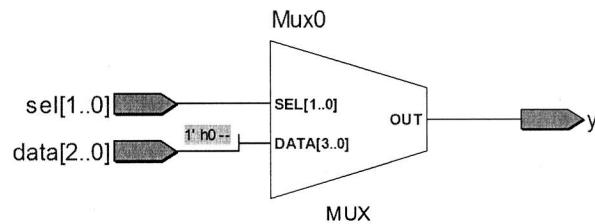


Рис. 11.1. Реализация модуля `use_full_case` из листинга 11.1: пример использования атрибута `full_case` в операторе `case`

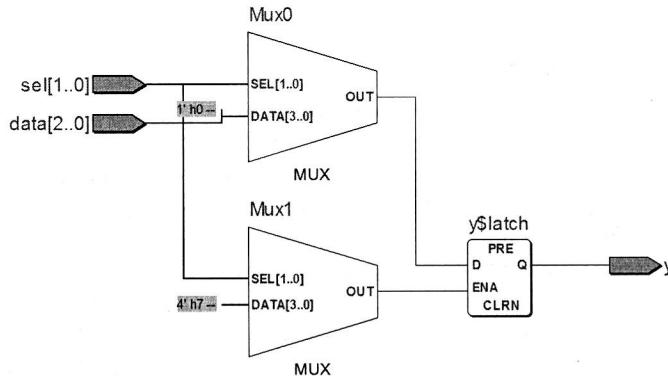


Рис. 11.2. Реализация примера из листинга 11.1 без атрибута `full_case`

Задание. В листинге 11.1 в оператор `case` добавьте конструкцию `default` для задания неопределенного значения выхода в случае входной комбинации $2'b11$. Результаты синтеза сравните с рис. 11.1 и 11.2.

11.3. Атрибут `parallel_case`

Напомним, как работает оператор `case`: вычисляется выражение; полученное значение последовательно сравнивается с константными элементами в том порядке, в котором они записаны в исходном коде; в случае совпадения выполняется оператор, следующий за данной константой; осуществляется выход из оператора `case`. При этом оставшиеся константные элементы не проверяются. Рассмотренный алгоритм при схемной реализации оператора `case` требует присутствия в схеме приоритетного шифратора, который будет заканчивать выполнение оператора `case` в случае совпадения значения выражения с одним из константных элементов.

Имеется две причины изменения рассмотренной стратегии реализации оператора `case`.

Во-первых, если известно, что в случае совпадения значения выражения с одним из константных элементов, совпадение значения

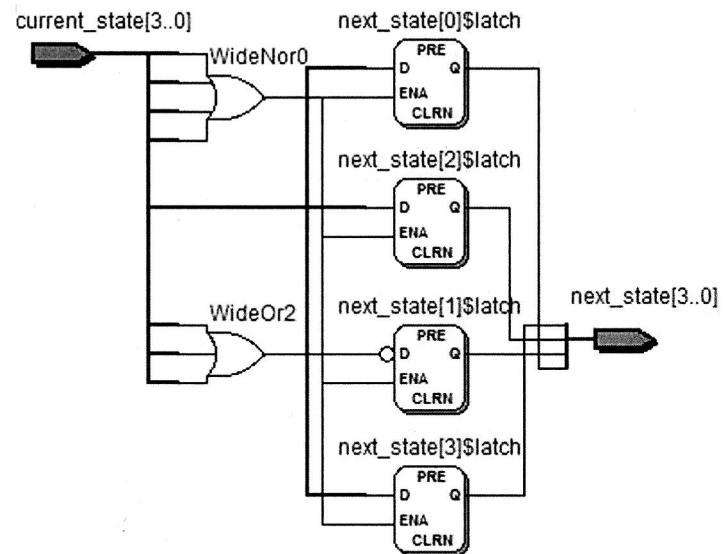


Рис. 11.3. Реализация модуля `one_hot_assign1` из листинга 11.2: пример использования атрибута `parallel_case` в операторе `case`

выражения с другими константами невозможно. Примером может служить описание с помощью оператора `case` полной таблицы истинности системы булевых функций. В данном случае для повышения быстродействия схемы, а иногда и для уменьшения стоимости реализации имеет смысл избавиться от приоритетного шифратора при схемной реализации оператора `case`.

Во-вторых, в константных элементах операторов `casex` и `casez` могут использоваться неопределенные значения битов (обозначаемые знаком «?»). В таких случаях одно и то же значение выражения может совпадать с несколькими константными элементами. Иногда требуется, чтобы операторы, управляемые такими константными элементами, функционировали параллельно. В подобной ситуации необходимо, чтобы проверка значения выражения со всеми константами и выполнение соответствующих операторов в случае совпадения выполнялись параллельно.

Для решения приведенных проблем служит атрибут `parallel_case`. Атрибут `parallel_case` указывает синтезатору вместо реализации приоритетного шифратора обеспечить параллельную реализацию операторов для всех константных элементов оператора `case`.

В качестве примера рассмотрим использование атрибута `parallel_case` при кодировании внутренних состояний конечного автомата унарным (one-hot) кодом.

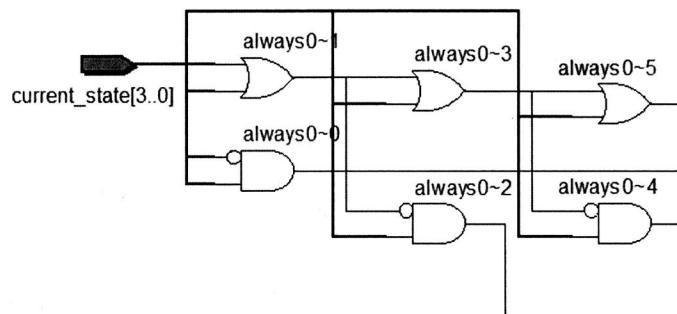


Рис. 11.4. Реализация примера из листинга 11.2

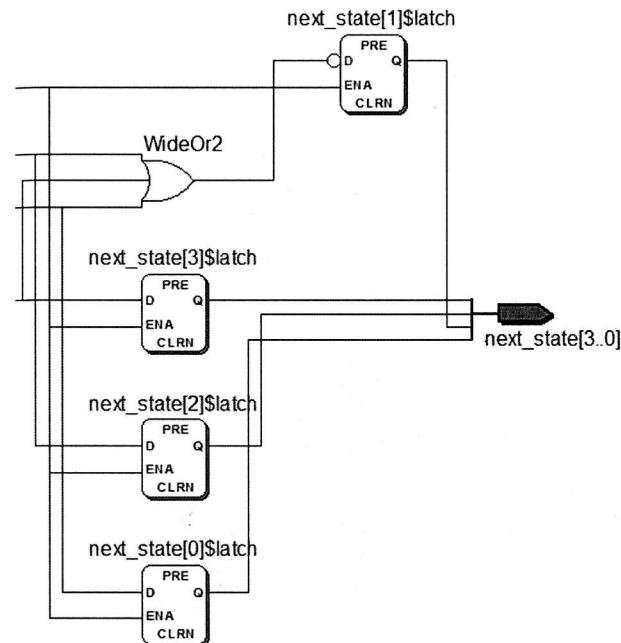
Листинг 11.2. Пример комбинационной схемы, реализующей функции переходов конечного автомата

```
module one_hot_assign1 (
    input [3:0] current_state,           // настоящее состояние
                                         // конечного автомата
    output reg [3:0] next_state);        // следующее состояние
                                         // конечного автомата

parameter s1 = 4'b0001, s2 = 4'b0010, s3 = 4'b0100, s4 = 4'b1000;
                                         // унарные коды

always @(*)
(* parallel_case *)
case (1'b1)                         // применение атрибута
                                         // parallel_case: ищется единица
    current_state[0] : next_state = s2; // в разрядах кода; поскольку
    current_state[1] : next_state = s3; // код унарный, будет выполнен
    current_state[2] : next_state = s4; // только один из операторов
    current_state[3] : next_state = s1;
endcase
endmodule
```

Реализация примера из листинга 11.2 требует 6 логических элементов и приведена на рис. 11.3.



без атрибута *parallel_case*

Реализация этого же примера без атрибута *parallel_case* требует 9 логических элементов и приведена на рис. 11.4.

Атрибуты *full_case* и *parallel_case* могут использоваться совместно.

Листинг 11.3. Пример совместного использования атрибутов *full_case* и *parallel_case*

```
module one_hot_assign2 (input[3:0] current_state,
                        output reg [3:0] next_state);
parameter s1 = 4'b0001, s2 = 4'b0010, s3 = 4'b0100, s4 = 4'b1000;
always @(*)
(* parallel_case, full_case *)
casex (current_state)
    4'b??1 : next_state = s2;
    4'b??1? : next_state = s3;
    4'b?1?? : next_state = s4;
    4'b1?? ? : next_state = s1;
endcase
endmodule
```

Реализация примера из листинга 11.3 вообще не требует логических элементов и приведена на рис. 11.5.



Рис. 11.5. Реализация модуля `one_hot_assign2` из листинга 11.3: пример одновременного использования атрибутов `full_case` и `parallel_case`

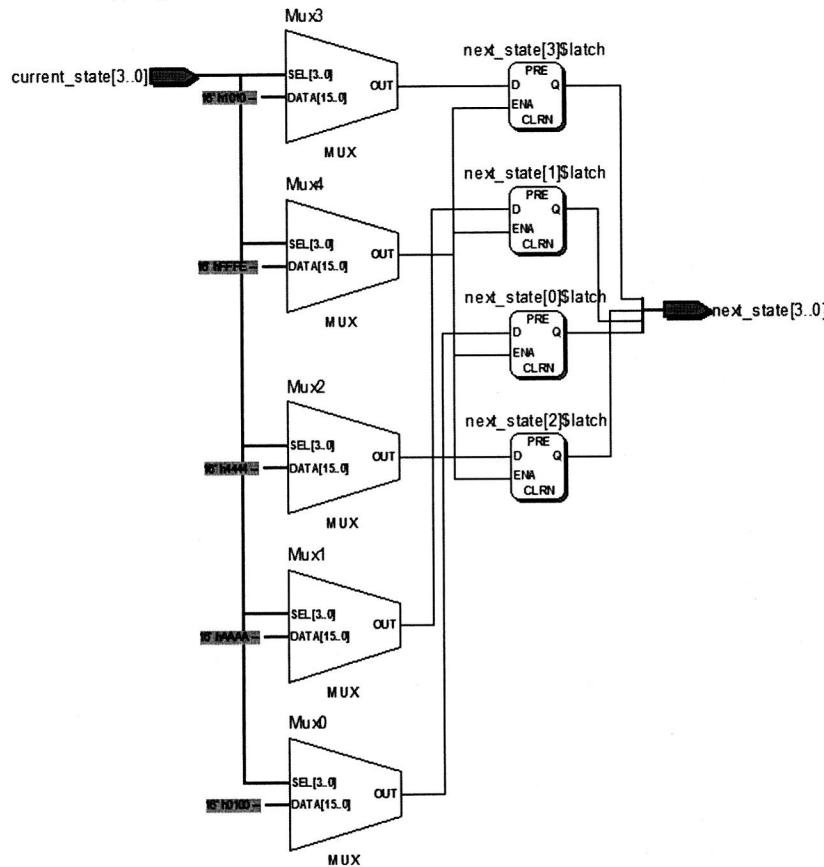


Рис. 11.6. Реализация примера из листинга 11.3 без атрибутов `full_case` и `parallel_case`

Реализация этого же примера без атрибутов `full_case` и `parallel_case` требует 8 логических элементов и приведена на рис. 11.6.

Задание. Объясните функционирование модуля из листинга 11.3. Проверьте ваш анализ с помощью симулятора.

Г л а в а 12

Блок генерации

12.1. Блоки генерации языка Verilog

Блоки генерации (generate blocks) предназначены для генерации фрагментов кода языка Verilog. Если сравнивать с языками программирования, то блоки генерации выполняют функции препроцессора по обработке исходного текста программы перед его компиляцией. Блоки генерации языка Verilog выполняют предварительную обработку исходного кода перед его выполнением с помощью симулятора или синтезатора.

Блоки генерации языка Verilog позволяют формировать последовательность повторяющихся фрагментов кода, незначительно отличающихся между собой; выбирать для выполнения симулятором фрагменты кода при выполнении определенных условий и др. Поскольку блоки генерации осуществляют преобразование исходного кода перед его выполнением, то все используемые в блоках генерации выражения должны быть константными.

В языках программирования часто операторы препроцессора и операторы языка значительно отличаются между собой (например, Ассемблер, С). В языке Verilog синтаксис операторов, используемых в блоках генерации, совпадает с синтаксисом операторов самого языка Verilog.

Для объяснения функционирования операторов генерации введем следующие понятия: *исходный код* и *компилируемый код*.

Исходный код — это исходный код проекта на языке Verilog, написанный разработчиком (файл проекта на языке Verilog).

Компилируемый код — это код проекта после его обработки с помощью блоков генерации. Если исходный код содержит блоки генерации, то исходный код будет отличаться от кода компиляции (он может быть как больше, так и меньше исходного кода, чаще — больше).

12.2. Формат блока генерации

Блоки генерации определяются внутри модуля и используются для генерации кода этого модуля. Формат блока генерации имеет вид:

```
genvar genvar_name, ...;
generate
    genvar genvar_name, ...;
    generate_items
endgenerate
```

где **genvar**, **generate** и **endgenerate** — ключевые слова; **genvar_name** — переменная типа **genvar**; **generate_items** — элементы блока генерации.

Переменная типа **genvar** должна быть целой положительной величиной. Она используется при создании фрагментов кода компиляции на основе некоторого фрагмента исходного кода (подобие переменной цикла в операторе **for**). Переменные типа **genvar** используются только внутри блока генерации, они могут иметь определенные значения только во время генерации кода и не существуют во время симуляции. Переменные типа **genvar** могут объявляться как внутри, так и вне блока генерации. Если переменная типа **genvar** объявлена вне блока генерации, то она может использоваться и в других блоках генерации.

Элементами блока генерации (**generate_items**) могут быть:

- оператор присваивания значения переменной типа **genvar** в формате
- ```
genvar_name = constant_expression;
```
- где **constant\_expression** — константное выражение;
- объявления сетей (**net**) и переменных (**reg**);
  - экземпляры модулей и примитивов;
  - процедурные блоки;
  - объявления задач и функций;
  - операторы генерации **if-else**, **case** и **for**.

**Замечание.** Следует различать операторы блока генерации **if-else**, **case** и **for** от аналогичных процедурных операторов. Операторы блока генерации **if-else**, **case** и **for** выполняются в самом начале компиляции с целью преобразования исходного кода, а процедурные операторы **if-else**, **case** и **for** выполняются во время симуляции.

## 12.3. Операторы генерации

### 12.3.1. Группа элементов генерации

Везде в приводимых ниже форматах операторов генерации вместо элемента генерации **generate\_item** может находиться группа элементов генерации, которая имеет следующий формат:

```
begin : generate_block_name
 generate_item
 generate_item
 ...
end
```

где **generate\_block\_name** — имя блока генерации.

### 12.3.2. Оператор **if-else**

Оператор **if-else** предназначен для включения по заданному условию в компилируемый код определенных фрагментов исходного кода. Формат оператора генерации **if-else**:

```
if (constant_expression)
 generate_item
else
 generate_item
```

где **if** и **else** — ключевые слова; **constant\_expression** — константное выражение; **generate\_item** — элемент генерации. В данном формате конструкция **else** может отсутствовать.

Действие оператора генерации **if-else**. В случае истинности константного выражения (**constant\_expression**) в компилируемом коде будет сформирован фрагмент исходного кода, соответствующий элементу генерации, следующему непосредственно после конструкции (**constant\_expression**). В противном случае в компилируемый код будет включен фрагмент исходного кода, соответствующий элементу генерации, следующему за ключевым словом **else**. В случае ложности выражения **constant\_expression** и отсутствия конструкции **else** в компилируемый код ничего не включается.

**Листинг 12.1.** Пример использования оператора генерации **if-else** для выбора из двух функций, реализующих различные алгоритмы умножения, в зависимости от размеров аргументов.

```
module multiplier (a, b, product);
 parameter a_size = 8, b_size = 8; // параметры модуля
 localparam product_size = a_size+b_size; // локальный параметр
 input [a_size-1:0] a; // используется старый
 input [b_size-1:0] b; // стиль
 output [product_size-1:0] product;
 generate
 if ((a_size < 8) || (b_size < 8)) // используется алгоритм
 // carry-look-ahead со схемой ускоренного переноса
 CLA_mult #(a_size, b_size) m (a, b, product);
 else
 // используется алгоритм
 // Wallace-tree
```

```

WALLACE_mult #(a_size, b_size) m (a, b, product);
endgenerate
endmodule

```

**Замечание.** В листинге 12.1 использован старый стиль объявления портов модуля.

### 12.3.3. Оператор case

Оператор **case** предназначен для включения в компилируемый код определенных фрагментов исходного кода в случае совпадения вычисленного константного выражения с определенным значением. Формат оператора генерации кода **case**:

```

case (constant_expression)
 genvar_value: generate_item
 ...
 default: generate_item
endcase

```

где **case**, **default** и **endcase** — ключевые слова; **constant\_expression** — вычисляемое константное выражение; **genvar\_value** — сравниваемое значение (константа); **generate\_item** — элемент генерации. В данном формате конструкция **default** может отсутствовать.

**Действие оператора case.** Вычисляется константное выражение **constant\_expression**. Полученное значение последовательно сравнивается с константами **genvar\_value**. В случае совпадения значения выражения с одной из констант **genvar\_value**, из исходного кода в компилируемый код включается элемент генерации **generate\_item**, следующий непосредственно за данной константой. Если значение выражения **constant\_expression** не совпало ни с одной из констант, то из исходного кода в компилируемый код включается элемент генерации, следующий за ключевым словом **default**; в случае отсутствия конструкции **default** в компилируемый код ничего не включается.

Пример использования оператора генерации **case** приведен в листинге 12.2.

**Листинг 12.2.** Пример использования оператора генерации **case** для создания экземпляра сумматора **x1** с помощью различных способов реализации в зависимости от ширины слова

```

generate
 case (WIDTH)
 1: adder_1bit x1(co, sum, a, b, ci); // 1-битовая реализация
 2: adder_2bit x1(co, sum, a, b, ci); // 2-битовая реализация
 default: adder_cla #(WIDTH) x1(co, sum, a, b, ci); // реализация
 endcase // с ускоренным переносом
endgenerate

```

### 12.3.4. Оператор for

Оператор генерации **for** предназначен для формирования в компилируемом коде последовательности фрагментов, отличающихся между собой значением переменной генерации или значением выражения, вычисленного на основе переменной генерации. Формат оператора генерации **for**:

```

for (genvar_name = constant_expression; constant_expression;
 genvar_name = constant_expression)
 generate_item

```

где **for** — ключевое слово; **genvar\_name** — переменная генерации; **constant\_expression** — константное выражение; **generate\_item** — элемент генерации.

**Действие оператора генерации for.** Функционирование оператора генерации **for** во многом совпадает с функционированием аналогичного процедурного оператора. Первое в круглых скобках константное выражение определяет начальное значение переменной генерации; второе выражение определяет условие окончания выполнения оператора **for**; третье выражение предназначено для изменения значения переменной генерации. В результате выполнения оператора **for** в компилируемом коде на основании элемента генерации **generate\_item** исходного кода будет сформирована последовательность фрагментов кода, которые отличаются между собой только значением переменной генерации **genvar\_name** или значением выражения, вычисленного на основе этой переменной.

Пример использования оператора генерации **for**.

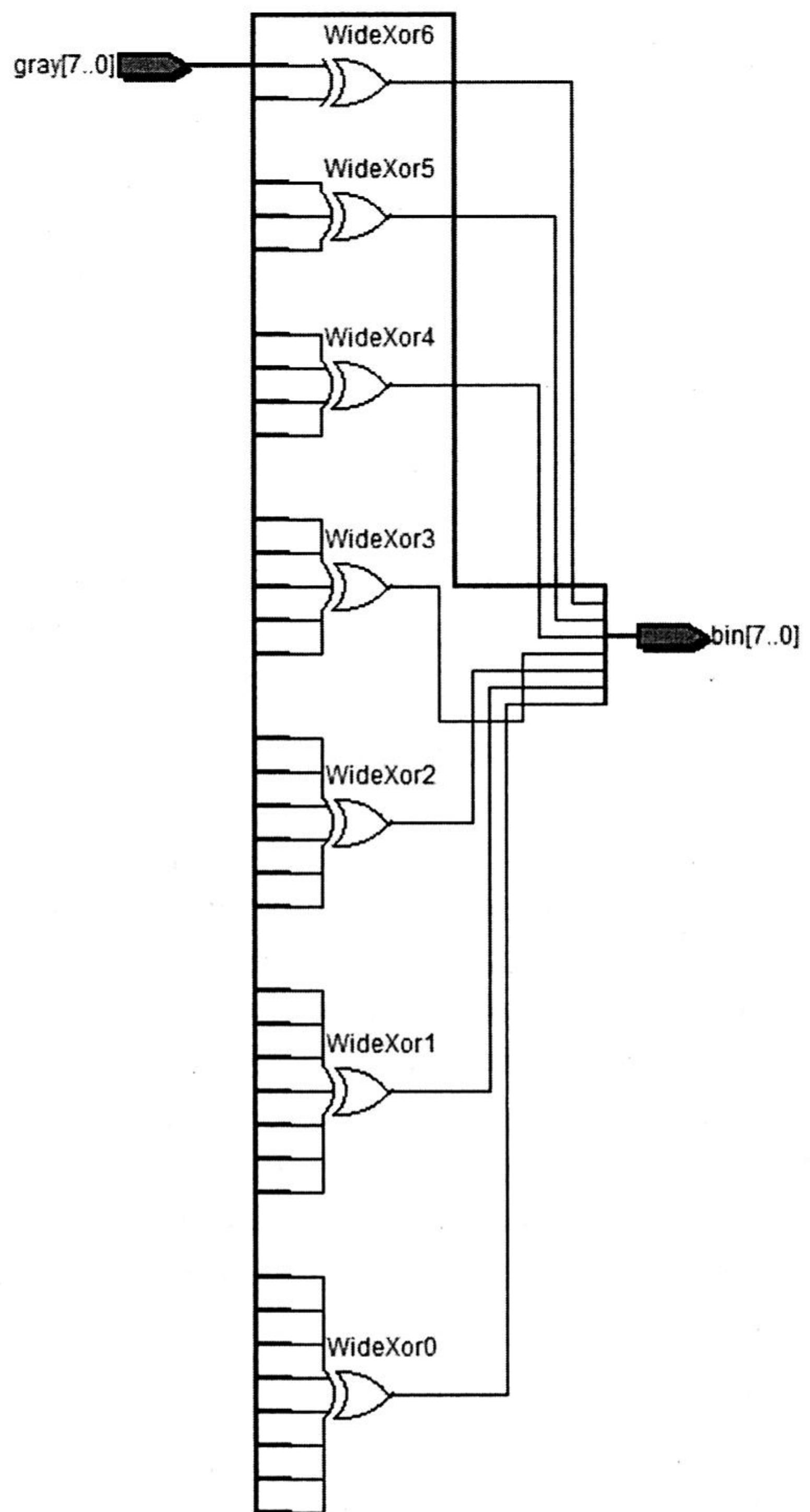
**Листинг 12.3.** Пример преобразователя кода Грэя в бинарный код с использованием оператора генерации **for**.

```

module gray_bin # (parameter N = 8)
 (input [N-1:0] gray, // входное значение — код Грэя
 output [N-1:0] bin); // выходное значение —
 // бинарный код
 genvar i; // i — переменная генерации
 generate // начало блока генерации
 for (i=0; i<N; i=i+1) begin: bl // оператор генерации for
 assign bin[i] = ^ gray[N-1:i]; // вычисление очередного
 // значения
 end
 endgenerate
endmodule

```

Результат синтеза примера из листинга 12.3 приведен на рис. 12.1.



**Рис. 12.1.** Реализация модуля *gray-bin* из листинга 12.3: пример использования блока генерации и оператора **for** при описании преобразователя кода Грэя в бинарный код

### Замечания.

1. Каждая задача или функция в операторах генерации может определяться только один раз.
2. Внутри цикла оператора **for** нельзя размещать определение задач или функций.

**Задание.** Опишите с помощью блока генерации последовательный сумматор на 8 битов из сумматоров на 1 бит.

# Г л а в а 13

## Задачи и функции

### 13.1. Задачи и функции языка Verilog

Задачи и функции относятся к структурным единицам языка Verilog, которые позволяют структурировать описание кода, делать его более читабельным и компактным. Поскольку задачи и функции допускают многократные вызовы, то таким образом можно сократить исходный код проекта. Кроме того, с помощью задач и функций можно скрывать локальные переменные. По аналогии с языками программирования задачи и функции относятся к элементам структурного программирования.

Задачи языка Verilog приблизительно напоминают подпрограммы в языках программирования, в то время как функции языка Verilog подобны функциям языков программирования. Кроме того, в языке Verilog введено понятие *константной функции* (*constant function*).

Не следует, однако, забывать, что главной конструкторской единицей языка Verilog является модуль, а задачи и функции используются как вспомогательные элементы при описании модулей.

В приводимых ниже форматах везде вместо процедурного оператора может записываться процедурный блок, т. е. группа операторов, взятая в операторные скобки.

### 13.2. Автоматические и статические задачи и функции

Понятие автоматических и статических задач и функций пришло из языков программирования и связано с работой симулятора. Все задачи и функции по умолчанию являются статическими. Это означает, что для всех переменных (входных, выходных, локальных) при симуляции задач и функций в памяти компьютера один раз выделяется статическая область памяти. При каждом обращении к задачам или функциям в эту область записываются значения переменных.

Проблемы статической памяти возникают при симуляции параллельных процессов, когда возможно одновременное выполнение нескольких экземпляров одной и той же задачи или функции. В этом случае значение переменных двух или более параллельно работающих экземпляров задачи и функции будут записываться в одну и ту же область памяти.

В случае автоматических задач и функций память для переменных выделяется всякий раз при каждом вызове задачи или функции и освобождается после завершения их работы. Использование автоматической памяти позволяет одновременно выполнять несколько экземпляров задач или функций. В отношении работы симулятора это означает, что становится возможным моделирование параллельных процессов.

Таким образом, если в алгоритме функционирования проекта допускается параллельная работа различных вызовов задач и функций, то такие задачи и функции должны быть объявлены как автоматические. Во всех остальных случаях задачи и функции могут быть статическими.

### 13.3. Задачи

Как было отмечено ранее, задачи языка Verilog напоминают подпрограммы или процедуры в языках программирования. Отличие заключается в том, что если в языках программирования используется список параметров для передачи значений аргументов подпрограммам и возврата результатов, то в задачах языка Verilog с этой целью используется список портов, через которые поступают и возвращаются определенные значения.

Задачи объявляются внутри модуля и являются локальными элементами по отношению к данному модулю. Задачи могут вызываться из процедурных блоков **initial** и **always**, а также из других задач.

Задачи могут иметь любое количество входных, выходных и двунаправленных портов, а также вообще не иметь портов (подобно модулям). Задачи могут включать операторы управления временем: #, @ или wait.

Объявление задачи имеет следующий формат:

```
task automatic task_name (
 port_declaration port_name, port_name, ...,
 port_declaration port_name, port_name, ...);
 local_variable_declarations
 procedural_statement
endtask
```

где **task**, **automatic** и **endtask** — ключевые слова; **task\_name** — имя задачи; **port\_declaration** — объявление порта задачи; **port\_name** — имя

порта; **local\_variable\_declaration** — объявление локальных переменных; **procedural\_statement** — процедурный оператор (блок), составляющий тело задачи.

Старый стиль объявления задачи имеет следующий формат:

```
task automatic task_name;
 port_declaration port_name, port_name, ...;
 port_declaration port_name, port_name, ...;
 local_variable_declarations
 procedural_statement or statement_group
endtask
```

Конструкция **automatic** является необязательной. Она объявляет задачу с автоматическим распределением памяти, то есть разрешается одновременная работа нескольких экземпляров одной и той же задачи.

Конструкция **port\_declaration** может иметь следующие форматы:

```
port_direction reg signed range и
port_direction port_type
```

Здесь конструкции **reg**, **signed** и **range** являются необязательными. Конструкция **port\_direction** определяет направление передачи данных, она может иметь значения: **input**, **output** или **inout**; использование ключевых слов **reg** и **signed** имеет прежнее значение (объявляет переменную типа **reg** и знаковую величину); конструкция **range**, как и прежде, имеет форму **[msb:lsb]** и определяет битовый диапазон переменной; конструкция **port\_type** может иметь значение **integer**, **time**, **real** или **realtime**.

Вызов задачи имеет следующий формат:

```
task_name (signal, signal, ...);
```

где **task\_name** — имя задачи; **signal**, **signal** — список подсоединяемых сигналов.

Пример объявления задачи приведен в листинге 13.1.

**Листинг 13.1.** Пример объявления и вызова задачи, которая читает данные из памяти

```
module read_mem_PC(
 input [15:0] PC,
 output [31:0] IR,
 input read_grant,
 input clock,
 input [31:0] data_bus,
 output reg read_request,
 output reg [15:0] addr_bus);
// определение задачи read_mem, которая читает данные из памяти
```

```

task read_mem (
 input [15:0] address, // значение адреса
 output [31:0] data); // возвращаемое значение
begin
 read_request = 1; // установка флага «запрос на чтение»
 wait (read_grant) // ожидание разрешения на чтение
 addr_bus = address; // установка адреса на адресную шину
 data = data_bus; // чтение данных с шины данных
 #5 addr.bus = 16'bzz; // перевод адресной шины в высокий
 // импедансное состояние
 read_request = 0; // сброс флага «запрос на чтение»
end
endtask
always @ (posedge clock)
 read_mem (PC, IR); // вызов задачи
endmodule

```

### 13.4. Функции

Функции языка Verilog очень похожи на функции языка С, только вместо списка переменных используется список портов. Функции, как и задачи, объявляются внутри модуля и являются локальными по отношению к данному модулю. В отличие от задач функции всегда возвращают значение, которое приписывается имени функции. Функция может вызываться в любом месте, где может использоваться значение выражения. Каждая функция должна иметь по крайней мере один входной порт. Функции не могут иметь выходных или двунаправленных портов. Функции не должны содержать операторы управления процедурным временем (#, @ и wait), а также операторы неблокирующего назначения (<<= >>). Считается, что функции всегда выполняются за нулевое (0.0) время симуляции.

Объявление функций имеет следующий формат:

```

function automatic range_or_type function_name (
 input range_or_type port_name, port_name, ...
 input range_or_type port_name, port_name, ...);
 local_variable_declarations
 procedural_statement
endfunction

```

где **function**, **automatic** и **endfunction** — ключевые слова. Конструкции **automatic** и **range\_or\_type** являются необязательными. Здесь ключевое слово **automatic** объявляет функцию с автоматическим распределением памяти. Это позволяет симулятору одновременно параллельно выполнять несколько экземпляров (вызовов) одной и той

же функции, а также использовать рекурсивный вызов функции (когда функция вызывает саму себя).

Старый стиль объявления функций имеет следующий формат:

```

function automatic [range_or_type]function_name;
 input range_or_type port_name, port_name, ...;
 input range_or_type port_name, port_name, ...;
 local_variable_declarations
 procedural_statement or statement_group
endfunction

```

Конструкция **range\_or\_type** определяет тип возвращаемого функцией значения, а также тип входных портов. В случае отсутствия конструкции **range\_or\_type** объявляется 1-битовая переменная типа **reg**. Конструкция **range\_or\_type** имеет следующие форматы:

```

signed [msb : lsb]
reg signed [msb : lsb]
integer, time, real or realtime

```

где **signed** объявляет знаковую переменную в дополнительном коде; **reg** объявляет переменную типа **reg**; **integer**, **time**, **real** и **realtime** объявляют переменные соответствующего типа; **msb** и **lsb** могут быть числами, константами, выражениями или константными функциями.

Вызов функции имеет следующий формат:

```
function_name (signal, signal, ...)
```

где **function\_name** — имя функции; **signal**, **signal** — список подсоединяемых сигналов.

Пример объявления функции приведен в листинге 13.2.

**Листинг 13.2.** Объявление и вызов функции для вычисления факториала

```

module fact_1 #(parameter LIMIT=32)
 (input [31:0] data,
 output reg [63:0] result);
 // описание функции для вычисления факториала
function automatic [63:0] factorial (input reg [31:0] n);
 if (n <= 1) factorial = 1;
 else factorial = n * factorial (n-1); // рекурсивный вызов функции
endfunction
always
 if (data <= LIMIT)
 result = factorial(data); // вызов функции factorial
 else result = 0;
endmodule

```

**Замечание.** Не все компиляторы поддерживают рекурсивный вызов функций.

**Задание.** Проверьте, поддерживает ли используемый вами компилятор рекурсивный вызов функций.

### 13.5. Константные функции

Константные функции языка Verilog аналогичны функциям предпроцессора в языке программирования. Другими словами, константная функция должна допускать свое выполнение (и возвращать некоторое значение) до начала работы симулятора. Отсюда следует ряд ограничений на объявление и использование константных функций, по сравнению с обычными функциями языка Verilog:

- в константных функциях могут использоваться только локальные переменные;
- запрещено использование сетевых (*net*) типов данных;
- значения параметров, используемых функцией, должны определяться до вызова функции;
- значения параметров нельзя переопределять с помощью оператора **defparam**;
- нельзя использовать обращение к константной функции при объявлении размеров портов;
- в константных функциях запрещено обращение к системным функциям и задачам;
- в константных функциях также запрещены иерархические ссылки и имена.

Однако константная функция может вызывать другую константную функцию.

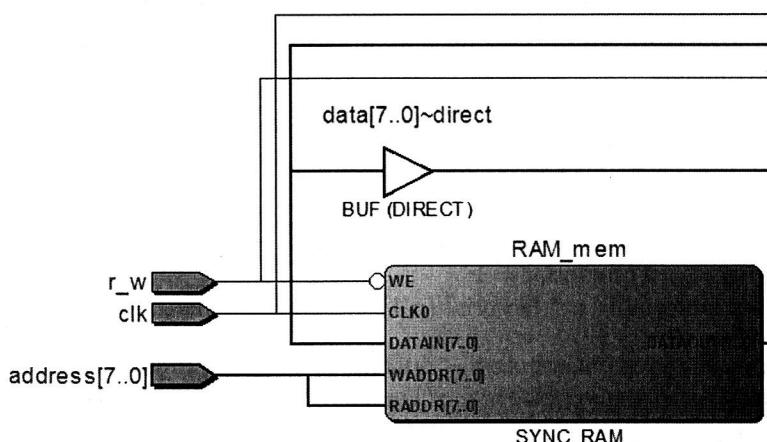


Рис. 13.1. Реализация модуля *ram\_model* из листинга 13.3: пример функция

Пример использования константной функции приведен в листинге 13.3.

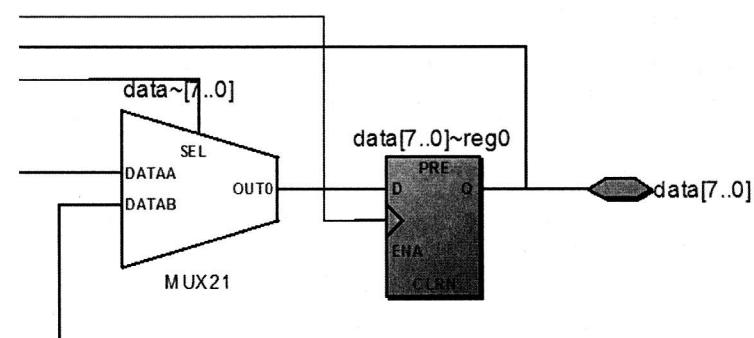
**Листинг 13.3.** Использование константной функции для определения размера адреса памяти

```
module ram_model
 // модуль синхронной
 // однопортовой памяти
 // ширина слова и
 // адресное пространство
 // шина адреса
 // сигналы управления
 // шина данных
 localparam adder_width = clogb2(ram_depth)-1;
 // ширина шины адреса

/* константная функция clogb2, которая по размеру адресного пространства ram_depth вычисляет достаточную ширину шины адреса */
function integer clogb2(input [31:0] value);
 for (clogb2=0; value > 0; clogb2=clogb2+1)
 value = value >> 1;
endfunction

reg [data_width-1:0] RAM_mem[0:ram_depth-1];
// объявление памяти

always @(posedge clk) begin
 // память синхронная
 if (r_w) data = RAM_mem[address];
 // чтение из памяти
 else RAM_mem[address] = data;
 // запись в память
end
endmodule
```



синхронной однопортовой памяти, при описании которой использовалась константная *clogb2*

Результат синтеза примера из листинга 13.3 приведен на рис. 13.1.

### 13.6. Сравнение функций и задач

Главное отличие между функциями и задачами в языке Verilog (как и в языках программирования) заключается в том, что результат выполнения функции может быть только один и он приписывается имени функции. Благодаря этому вызовы функций могут использоваться в выражениях в качестве обычных операндов. Задачи могут иметь несколько результатов, которые передаются через выходные и двунаправленные порты. Однако вызовы задач нельзя использовать в выражениях. В то же время язык Verilog имеет свои особенности при использовании задач и функций, которые отражены в табл. 13.1.

Подводя итоги сравнения задач и функций можно сделать следующий вывод: функции более удобны при описании комбинационных схем, а задачи — последовательностных схем.

Таблица 13.1

Сравнение функций и задач

| Категория                                                 | Функции                                                                                                                                                                                                              | Задачи                                                                                                                           |
|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Вызов (активация)                                         | Допускается вызов функции внутри выражения, а возвращаемое значение может использоваться в этом выражении. Возможен вызов функции в операторах непрерывного назначения <code>assign</code> , в том числе процедурных | Выполняется в виде отдельного процедурного оператора. Обращение к задаче нельзя выполнять в операторах непрерывного назначения   |
| Обращение к другим задачам и функциям                     | В теле функции можно обращаться к другим функциям, в том числе к самой себе (рекурсивно). Однако нельзя обращаться к задачам                                                                                         | Задачи могут обращаться как к функциям, так и к другим задачам                                                                   |
| Входы и выходы                                            | Функция должна иметь по крайней мере один порт, не может иметь выходных и двунаправленных портов. Возвращает одно значение, которое связано с именем функции                                                         | Задача может иметь любое количество портов произвольного типа, а также вообще не иметь портов                                    |
| Управление процедурным временем                           | Функции не могут управлять процедурным временем, т. е. не могут содержать операторы <code>#</code> , <code>@</code> и <code>wait</code>                                                                              | Задачи могут управлять процедурным временем, т. е. могут содержать операторы <code>#</code> , <code>@</code> и <code>wait</code> |
| Оператор неблокирующего назначения <code>&lt;=&gt;</code> | Функции не могут содержать оператор неблокирующего назначения <code>&lt;=&gt;</code>                                                                                                                                 | Задачи могут содержать оператор неблокирующего назначения <code>&lt;=&gt;</code>                                                 |

## Г л а в а 14

### Системные задачи и функции

#### 14.1. Системные задачи и функции языка Verilog

В языке Verilog, как и в языке программирования C, имеется множество предопределенных задач и функций, которые называются системными. Системные задачи и функции языка Verilog начинаются со знака доллара (`$(`). Стандарт языка Verilog IEEE 1364 определяет минимальное количество системных задач и функций. Разработчики программных средств проектирования часто определяют дополнительные свойства системных задач и функций, например, для отображения на экране временных диаграмм. Кроме того, пользователи могут сами определять системные задачи и функции с помощью языка программирования интерфейса Verilog (*Programming Language Interface — PLI*).

#### 14.2. Системные задачи для отображения текста

К системным задачам языка Verilog для отображения текста относятся `$display`, `$write`, `$strobe` и `$monitor`. Первые три задачи очень подобны между собой. Они выполняются, как только симулятор встретит их в коде проекта. Задачу `$monitor` можно записать в коде проекта только один раз и она будет выполняться при изменении значения любого из аргументов, кроме значения времени. По умолчанию числовые значения всех перечисленных выше задач выводятся в десятичной системе счисления. Добавление к названию задачи суффиксов `b`, `o` и `h` позволяют по умолчанию выводить числовые значения в двоичной, восьмиричной и шестнадцатиричной системах счисления.

Системная задача `$display` выводит на экран форматированное сообщение. Задача `$display` во многом напоминает функцию `printf()` языка программирования C. Формат задачи `$display`:

```
$display («text-format», list_of_arguments);
```

Список форматирующих символов задачи \$display

Таблица 14.1

| Символ | Описание                                                 |
|--------|----------------------------------------------------------|
| %b     | Бинарное значение                                        |
| %o     | Восьмиричное значение                                    |
| %d     | Десятичное значение                                      |
| %h     | Шестнадцатиричное значение                               |
| %e     | Действительное значение в экспоненциальном представлении |
| %f     | Действительное значение в десятичном представлении       |
| %t     | Отформатированное значение времени                       |
| %s     | Строка символов                                          |
| %m     | Иерархическое имя области действия                       |
| %l     | Привязка библиотеки конфигурации                         |
| \t     | Символ табуляции                                         |
| \n     | Символ новой строки                                      |
| \»     | Символ «двойные кавычки»                                 |
| \`     | Символ «обратный слеш»                                   |
| %%     | Знак процента                                            |

где `«text-format»` — текстовая строка с форматирующими символами; `list_of_arguments` — список аргументов.

Форматирующие символы представлены в таблице 14.1.

Кроме того, символы `%0b`, `%0o`, `%0d` и `%0h` подавляют любые лидирующие нули выводимого значения. Форматирующие символы `%e` и `%f` могут определять ширину поля (число печатаемых знаков) выводимого значения (например, `%5.2f` определяет для чисел с фиксированной запятой ширину поля размером 5 знаков перед запятой и 2 знака после запятой). Символы `%m` и `%l` не имеют аргументов, для них аргументы определяются неявно. Все форматирующие символы не чувствительны к регистру, т. е. символы `%b` и `%B` эквивалентны.

После вывода строки символов на экран задача `$display` добавляет к выводимой строке управляемый символ начала новой строки.

Системная задача `$write` совпадает с задачей `$display`, только к выводимой строке не добавляется управляемый символ начала новой строки.

Системная задача `$strobe` совпадает с задачей `$display` за исключением того, что вывод текста задерживается до тех пор, пока не будут выполнены все моделируемые события в текущем времени симуляции.

Системная задача `$monitor` имеет точно такой же формат как и задача `$display`. Однако задача `$monitor` функционирует совершенно иначе. Если задача `$display` выполняется тогда, когда симулятор в коде проекта встретит строку с данной задачей, то задача `$monitor` вызывает фоновый процесс, который непрерывно проверяет аргументы в списке, и как только хотя бы один из них, за исключением переменных времени, изменит свое значение, выводит на экран отформатированное сообщение.

Пример использования системной задачи `$display`:

```
$display ("Время симуляции %t", $time);
```

Пример использования системной задачи `$monitor`:

```
$monitor ($time, ": A=%b B=%b Cin=%b S=%b Cout=%b", A,
B, Cin, S, Cout);
```

### 14.3. Системные задачи и функции для работы с файлами

В случае большого объема информации, выводимой симулятором на экран, появляется явление «скроллинга», которое затрудняет анализ информации. В этом случае информацию можно выводить в текстовый файл, воспользовавшись соответствующими системными задачами и функциями, а затем проанализировать любую часть файла с помощью любого текстового редактора. Аналогично, в случае необходимости подготовки большого объема информации для работы симулятора информацию предварительно можно подготовить в виде текстового файла, а затем воспользоваться системными задачами и функциями чтения из файла.

#### 14.3.1 Открытие и закрытие файлов

Открыть файл в языке Verilog можно с помощью функции `$fopen`. Функция `$fopen` открывает на диске файл для записи и возвращает значение типа `integer` (32 бита), которая имеет следующие форматы:

```
mcd = $fopen («file_name»);
```

```
fd = $fopen («file_name», type);
```

где `mcd` — многоканальный дескриптор с единственным установленным битом; `fd` — одноканальный дескриптор с несколькими установленными битами; `«file_name»` — строка символов с именем файла; `type` — один или несколько символов, определяющих тип открываемого файла.

Для одновременной записи информации в несколько файлов дескрипторы типа `mcd` могут объединяться вместе с помощью операции `or`. В дескрипторе типа `mcd` нулевой бит зарезервирован для нулевого окна симулятора, а тридцать первый бит — для дескрипторов типа `fd`. В дескрипторе типа `fd` устанавливается бит 31 и, по крайней мере, еще один бит.

Файлы с дескрипторами типа `mcd` всегда открываются только для записи. Файлы с дескрипторами типа `fd` могут открываться как для записи, так и для чтения, а также могут открываться в режиме добавления. Используя дескриптор типа `fd` в каждый момент времени, может читаться или писаться только один файл.

Значение параметра *type* при открытии файлов

| Параметры             | Описание                                                     |
|-----------------------|--------------------------------------------------------------|
| «г» или «rb»          | Открыть для чтения                                           |
| «w» или «wb»          | Сократить до нулевой длины или создать для записи            |
| «а» или «ab»          | Добавление; открыть для записи в конец файла                 |
| «г+», «г+b» или «gb+» | Открыть для обновления (чтения и записи)                     |
| «w+», «w+b» или «wb+» | Сократить до нулевой длины или создать для обновления        |
| «а+», «a+b» или «ab+» | Добавление; открыть или создать для обновления в конец файла |

Параметр *type* представляется строкой символов, которая определяет тип открываемого файла. Значения параметра *type* представлены в таблице 14.2.

Закрыть ранее открытый файл можно с помощью функции **\$fclose**, которая имеет следующий формат:

**\$fclose (mcd\_or\_fd)**

где *mcd\_or\_fd* — дескриптор типа *mcd* или *fd*, используемый при открытии файла с помощью функции **\$fopen**.

Примеры открытия и закрытия файлов:

```
fp1 = $fopen("result.txt",w); // открывает файл нулевой длины
 // (если нет, то создает) для записи
fp2 = $fopen("result.txt",a); // открывает файл для записи в конец
 // файла
fp3 = $fopen("result.txt",r+); // открывает файл для чтения и записи
fp4 = $fopen("result.txt",r); // открывает файл только для чтения
fclose(fp1); // закрывает файл fp1
```

### 14.3.2. Вывод информации в файл

После того как файл открыт, в него можно записывать информацию с помощью системных задач **\$fmonitor**, **\$fdisplay**, **\$fwrite** и **\$fstrobe**, которые имеют следующий формат:

```
$fmonitor (mcd_or_fd, "text with formal specifiers", list_of_arguments);
$fdisplay (mcd_or_fd, "text with formal specifiers", list_of_arguments);
$fwrite (mcd_or_fd, "text with formal specifiers", list_of_arguments);
$fstrobe (mcd_or_fd, "text with formal specifiers", list_of_arguments);
где mcd_or_fd — дескриптор файла типа mcd или fd; "text_with_formal_specifiers" — текстовая строка с форматирующими символами; list_of_arguments — список аргументов.
```

Использование функций **\$fmonitor**, **\$fdisplay**, **\$fwrite** и **\$fstrobe** аналогично использованию функций **\$monitor**, **\$display**, **\$write** и **\$strobe**, только добавляется в качестве первого аргумента дескриптор файла, в который осуществляется запись.

Таблица 14.2

Примеры использования функций записи в файл:

```
fp = $fopen («result.adding.txt»); // открывает файл типа mcd
$timeformat (-9, 1, «ns», 8); // определяет формат времени
 // при выводе
$fmonitor (fp,«%t: A=%b B=%b Cin=%b S=%b Cout=%b», $time,
A, B, Cin, S, Cout);
```

### 14.3.3. Другие функции работы с файлами

Кроме рассмотренных системных функций, в стандарт языка Verilog-2001 был добавлен ряд функций для работы с файлами из языка программирования С, формат использования которых имеет вид:

```
c = $fgetc (fd);
code = $ungetc (c,fd);
code = $fgets (str,fd);
code = $fscanf (fd, format, arguments);
code = $fread (reg_variable, fd);
code = $fread (memory_array, fd, start, count);
position = $ftell (fd);
code = $fseek (fd, offset, operation);
code = $rewind (fd);
errno = $ferror (fd,str);
code = $fflush (mod_or_fd);
```

Таблица 14.3

Описание дополнительных функций работы с файлами

| Формат функции                                                                                          | Действие                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| c = \$fgetc (fd) code = \$ungetc (c,fd) code = \$fgets (str,fd) code = \$fscanf (fd, format, arguments) | Читает очередной символ <i>c</i> из файла <i>fd</i> Возвращает символ <i>c</i> в файл <i>fd</i> Читает строку символов <i>str</i> из файла <i>fd</i> Читает информацию из файла <i>fd</i> согласно формату <i>format</i> и присваивает значение аргументам <i>arguments</i> |
| code = \$fread (reg_variable, fd) code = \$fread (memory_array, fd, start, count)                       | Читает значение переменной типа <i>reg</i> из файла <i>fd</i> Читает содержимое элементов памяти <i>memory_array</i> из файла <i>fd</i> от позиции <i>start</i> ; количество читаемых значений указывается в <i>count</i>                                                   |
| position = \$ftell (fd)                                                                                 | Возвращает текущее значение указателя позиции файла <i>fd</i>                                                                                                                                                                                                               |
| code = \$fseek (fd, offset, operation) code = \$rewind (fd)                                             | Устанавливает указатель текущей позиции файла <i>fd</i> от позиции <i>operation</i> на смещение <i>offset</i> Перемещает указатель текущей позиции файла <i>fd</i> в начало                                                                                                 |
| errno = \$ferror (fd,str)                                                                               | Проверяет наличие ошибки при работе с файлом <i>fd</i> ; возвращает код ошибки <i>errno</i> и строку сообщений <i>str</i>                                                                                                                                                   |
| code = \$fflush (mcd_or_fd)                                                                             | В файл <i>mcd_or_fd</i> дописывает содержимое выходного буфера (если еще не было дописано)                                                                                                                                                                                  |

где *c* — символ, прочитанный из файла; *code* — код завершения работы функции (равен коду завершения аналогичных функций языка С); *position* — указатель позиции файла; *errno* — код ошибки при работе с файлом.

Действия перечисленных функций описаны в табл. 14.3.

**Замечание.** Не все компиляторы языка Verilog поддерживают дополнительные функции работы с файлами.

**Задание.** Проверьте, поддерживает ли используемый вами компилятор языка Verilog дополнительные функции работы с файлами.

## 14.4. Другие системные задачи и функции

### 14.4.1. Управление процессом симуляции

Для управления процессом симуляции служат две функции: **\$finish** и **\$stop**, которые имеют следующий формат:

```
$finish (n);
$stop (n);
```

Задача **\$finish** завершает выполнение процесса симуляции. Аргумент *n* может принимать значение 0, 1 или 2, которое управляет объемом выводимой на экран информации о завершенном процессе. Задача **\$stop(n)** задерживает процесс симуляции и вводит интерактивный отладочный режим.

### 14.4.2. Управление временем симуляции

Текущее время симуляции можно получить с помощью функций **\$time**, **\$stime** и **\$realtime**. Эти функции аргументов не имеют. Функция **\$time** возвращает текущее время симуляции в виде 64-битового вектора; **\$stime** — в виде 32-битового целого значения; **\$realtime** — в виде действительного числа.

Вид, в котором выводится время симуляции, определяется с помощью задания **\$timeformat**, которая имеет следующий формат:

```
$timeformat (unit, precision, <suffix>, min_field_width);
```

Параметр *unit* представляет собой значение степени числа 10 для определения единиц времени, например 0 соответствует  $10^0 = 1$  секунда;  $-3$  соответствует  $10^{-3} = 1$  миллисекунда;  $-6$  соответствует  $10^{-6} = 1$  микросекунда;  $-9$  соответствует  $10^{-9} = 1$  наносекунда;  $-12$  соответствует  $10^{-12} = 1$  пикосекунда.

Параметр *precision* (точность) определяет количество чисел, отображаемых после десятичной точки.

Параметр *suffix* представляет собой строку символов, которая выводится после значения числа, например, «ns».

Параметр *min\_field\_width* определяет минимальное число отображаемых символов.

Пример использования задания **\$timeformat**:

```
$timeformat (-9, 2, <ns>, 10);
```

которое определяет единицы измерения времени наносекунды, будет отображаться две цифры после запятой, в конце выводимого значения добавляется строка символов «ns», а всего для представления значения времени используется поле из 10 знаков.

Функция **\$printtimescale** выводит на экран временнюю шкалу (*time scale*) определенного модуля, а если модуль не определен, то выводит область действия проекта, из которой данный модуль вызывается.

### 14.4.3. Преобразование знаковых и беззнаковых величин

Две функции **\$signed** и **\$unsigned** служат для преобразования знаковых чисел в беззнаковые и наоборот. Формат функций:

```
signed_value = $signed (unsigned_value)
unsigned_value = $unsigned (signed_value)
```

где *unsigned\_value* — число без знака; *signed\_value* — число со знаком.

Примеры использования функций **\$signed** и **\$unsigned**:

```
reg [7:0] a; // объявление беззнаковой переменной a
reg signed [7:0] b; // объявление знаковой переменной b
a = $unsigned (-4); // переменной a будет присвоено значение
 // 4'b1100
b = $signed (4'b1100); // переменной b будет присвоено значение -4
```

### 14.4.4. Запись и чтение в переменные и из строки символов

Функция **\$swrite** позволяет осуществлять запись значений в переменную типа **reg**. Действует функция **\$fwrite** подобно функции **\$fwrite**, только запись осуществляется не в файл, а в переменную типа **reg**. Формат функции **\$swrite**:

```
$swrite (reg_variable, format, arguments, format, arguments, ...);
$format (reg_variable, format, arguments);
```

где *reg\_variable* — переменная типа **reg**; *format* — текстовая строка с форматирующими символами; *arguments* — аргументы.

Функция **\$sformat** подобна функции **\$swrite**, но содержит только одну строку с форматирующими символами в качестве второго аргумента.

К имени функции **\$swrite** в качестве суффиксов могут добавляться символы **b**, **o** и **d**, которые определяют систему счисления по умолчанию.

Функция **\$sscanf** позволяет читать значения из строки символов. Формат функции **\$sscanf**:

```
code = $sscanf(str, format, arguments);
```

где *code* — возвращаемый код; *str* — строка символов; *format* — текстовая строка с форматирующими символами; *arguments* — аргументы.

#### 14.4.5. Загрузка содержимого памяти

Для загрузки содержимого имеющейся в проекте памяти служат две функции: **\$readmemb** и **\$readmemh**, которые имеют следующий формат:

```
$readmemb ("file_name", variable_array, start_address, end_address);
$readmemh ("file_name", variable_array, start_address, end_address),
```

где *file\_name* — имя файла, из которого читаются данные и загружаются в память; *variable\_array* — двумерный массив, представляющий блок памяти; *start\_address* и *end\_address* — начальный и конечный адреса элементов памяти, куда осуществляется загрузка данных.

Адреса *start\_address* и *end\_address* являются необязательными параметрами. Файл, из которого читаются данные, представляет собой текстовый файл в коде ASCII, данные в котором должны быть представлены в двоичном (для функции **\$readmemb**) или шестнадцатиричном (для функции **\$readmemh**) виде.

Примеры использования функции **\$readmemh**:

```
reg [7:0] mem [1:256]; // определение памяти
 // из 256 8-битовых слов
initial $readmemh ("mem.data",mem); // загрузка содержимого
 // всей памяти
initial $readmemh ("mem.data",mem,16); // загрузка от адреса
 // 16 до 256
initial $readmemh ("mem.data",mem,128,1); // загрузка от адреса
 // 128 до 1
```

#### 14.4.6. Преобразование переменных типа real в 64-битовый вектор

Как было указано ранее, значение переменных типа **real** не может быть передано непосредственно через порт. Для этого переменную типа **real** предварительно необходимо преобразовать в 64-битовый вектор. Чтобы выполнить такое преобразование (и обратно) служат системные функции **\$realtobits** и **\$bitstoreal**, которые имеют следующий формат:

```
64-bit_reg_variable = $realtobits(real_variable);
real_variable = $bitstoreal(64-bit_reg_variable);
```

где *64-bit\_reg\_variable* — представление значения переменной типа **reg** в виде 64-битового вектора; *real\_variable* — переменная типа **real**.

Пример использования функций **\$realtobits** и **\$bitstoreal** приведен в листинге 14.1.

**Листинг 14.1.** Передача переменных типа **real** через порт

```
module driver (net_r);
 output net_r;
 real r;
 wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
 input net_r;
 wire [64:1] net_r;
 real r;
 initial assign r = $bitstoreal(net_r);
endmodule
```

**Замечание.** Не все компиляторы поддерживают тип данных **real**.

**Задание.** Проверьте, поддерживает ли используемый вами компилятор тип данных **real**.

#### 14.4.7. Функции для работы с командной строкой

Функция **\$test\$plusargs** позволяет тестировать командную строку на наличие в ней определенной опции. Опция в командной строке должна начинаться со знака плюс («+»), но знак + не включается в строку символов. Если указанная опция командной строки найдена, то функция возвращает нулевое значение. Формат функции **\$test\$plusargs**:

```
integer = $test$plusargs («invocation_option»)
```

где *integer* — возвращаемое значение типа **integer**; *invocation\_option* — строка символов, содержащая искомую опцию.

Пример использования функции **\$test\$plusargs**. Пусть вызов симулятора из командной строки содержит опцию +HELLO. Тогда в результате выполнения следующего кода:

```
initial begin
 if ($test$plusargs("HELLO")) $display("Hello argument found.");
 if ($test$plusargs("HE")) $display("The HE subset string is detected.");
 if ($test$plusargs("H")) $display("Argument starting with H found.");
 if ($test$plusargs("HELLO_HERE")) $display("Long argument.");
 if ($test$plusargs("HI")) $display("Simple greeting.");
 if ($test$plusargs("LO")) $display("Does not match.");
end
```

будет выведено сообщение:

Hello argument found.  
The HE subset string is detected.  
Argument starting with H found.

Функция `$value$plusargs` преобразует текстовую строку в определенный формат и помещает значение во второй аргумент. В качестве строки символов может использоваться опция вызова командной строки с форматирующими символом. В качестве форматирующих символов могут использоваться следующие символы: `%b`, `%o`, `%d`, `%h`, `%e`, `%f`, `%g` и `%s`.

Обычно функция `$value$plusargs` служит для представления определенной опции вызова из командной строки в виде значения некоторой переменной. Формат функции `$value$plusargs`:

```
integer = $value$plusargs ("invocation_option=format", variable)
```

**Замечание.** Не все рассмотренные в главе 14 функции и задачи языка Verilog поддерживаются компиляторами.

#### Задания.

1. Проверьте, какие из рассмотренных в главе 14 функций и задач поддерживаются используемым вами компилятором.
2. Проверьте, какие дополнительные функции и задачи поддерживаются используемым вами компилятором.

## Г л а в а 15

### Директивы компилятора

#### 15.1. Директивы компилятора языка Verilog

Директивы компилятора являются командами компилятора и дают указание компилятору, как интерпретировать исходный код проекта, написанный на языке Verilog.

Все директивы компилятора начинаются со знака апостроф («'»). Поскольку директивы языка Verilog не являются операторами, то в конце директивы знак точки с запятой («;») не ставится.

Область действия каждой директивы компилятора не ограничивается модулем или файлом. Если компилятор встретил в исходном коде директиву компилятора, то ее действие сохраняется на весь проект до тех пор, пока другая директива не модифицирует или не отменит действие предыдущей директивы.

#### 15.2. Возврат к умалчиваемым значениям директив компилятора

Директива `'resetall` возвращает значение по умолчанию для тех директив компилятора, которые имеют значение по умолчанию.

Формат директивы `'resetall`:

`'resetall`

#### 15.3. Определение значения единицы времени

Единицы времени широко используются в различных конструкциях языка Verilog при описании моделей проектов. Значение (длительность) одной единицы времени определяется с помощью директивы `'timescale`, которая имеет следующий формат:

`'timescale time_unit base / precision base`

где `time_unit` — задержка для одной единицы времени, может принимать значение 1, 10 или 100; `base` — база временных единиц времени,

может принимать значения *s* (секунды), *ms* (миллисекунды), *us* (микросекунды), *ns* (наносекунды), *ps* (пикосекунды) и *fs* (фемтосекунды); *precision* — точность представления времени, определяет временные единицы для представления времени после десятичной точки; *base* после слова *precision* — база временных единиц для точности представления времени.

Пример использования директивы ‘timescale’:

```
'timescale 1ns / 10ps
```

определяет временные единицы длительностью в 1 наносекунду с точностью представления 10 пикосекунд (0.01 наносекунд) и двумя позициями после десятичной точки.

**Замечание.** Директива ‘timescale’ не имеет значения по умолчанию, поэтому должна всегда определяться в проектах, где используются временные единицы.

#### 15.4. Макроопределения

Макроопределения используются для замены в исходном коде одной текстовой строки на другую и широко используются в языках программирования. В языке Verilog макроопределения или макросы задаются с помощью директивы ‘define’, которая имеет следующие два формата:

```
'define macro_name text_string
```

```
'define macro_name (arguments) text_string (arguments)
```

где *macro\_name* — имя макроопределения, при последующем использовании в исходном коде имени макроопределения должен предшествовать знак апостроф («'»); *text\_string* — текстовая строка, на которую будет заменяться имя макроопределения в исходном коде; *arguments* — список аргументов, передаваемых в макроопределении.

**Замечание.** Текстовая строка *text\_string* в макроопределении заканчивается символом возврата каретки (*carriage return*), т. е. может распространяться не более чем до конца строки.

С помощью макроопределений можно определять константы, простые функции и др. При описании макроопределений могут использоваться комментарии.

Примеры макроопределений:

```
'define cycle 20 // период синхросигнала cycle
always # ('cycle/2) clk = ~ clk; // генерация синхросигнала clk
 // с периодом cycle
'define NAND (dval) nand # (dval) // запись примитива nand
 // заглавными буквами
'NAND (3) i1 (y, a, b); // определение экземпляра i1
```

```
// примитива nand
'NAND (3:4:5) i2 (o, c, d); // определение экземпляра i2
 // примитива nand
```

Для отмены ранее определенного имени макроопределения служит директива ‘*undef*’, которая имеет следующий формат:

```
'undef macro_name.
```

#### 15.5. Директивы условной компиляции

К директивам условной компиляции относятся следующие директивы: ‘*ifdef*’, ‘*ifndef*’, ‘*else*’, ‘*elsif*’ и ‘*endif*’. Они позволяют включать в исходный код отдельные фрагменты кода в зависимости от того, было ли ранее определено с помощью директивы ‘*define*’ или вызывающей опции +*define*+ имя макроопределения *macro\_name*. Формат директив условной компиляции:

```
'ifdef macro_name
'ifndef macro_name
 verilog_source_code
'else
 verilog_source_code
'elsif macro_name
 verilog_source_code
'endif
```

где *macro\_name* — имя макроопределения; *verilog\_source\_code* — фрагмент исходного кода.

Пример использования директив условной компиляции:

```
'ifdef RTL
 wire y = a & b; // реализация y в виде сетевой переменной
'else
 and #1 (y, a, b); // реализация y с помощью примитива and
'endif
```

#### 15.6. Включение файлов

В исходный код на языке Verilog могут включаться любые текстовые файлы с помощью директивы ‘*include*’. Формат директивы ‘*include*’:

```
'include «file_name»
```

где *file\_name* — имя включаемого файла.

Пример использования директивы ‘*include*’:

```
'include «project_macros.v»
```

## 15.7. Определение умалчивающего типа цепей

В языке Verilog цепи по умолчанию имеют тип **wire**. Тип цепей по умолчанию можно изменить с помощью директивы '**default\_nettype**', которая имеет следующие форматы:

```
'default_nettype net_data_type
'default_nettype none
```

где **net\_data\_type** — новый тип цепей по умолчанию.

При использовании второго формата со значением типа цепей **none** запрещается неявное (по умолчанию) определение типа цепей, т. е. тип каждой цепи должен при определении цепи задаваться явно.

Пример использования директивы '**default\_nettype**:

```
'default_nettype wand // определение wand как тип цепей
 // по умолчанию
```

## 15.8. Определение логических значений для неподсоединенных входов

В некоторых случаях для более точного результата симуляции необходимо определять значения неподсоединеных входов модулей. Для этой цели служат директивы '**unconnected\_drive**' и '**'nounconnected\_drive**', которые имеют следующие форматы:

```
'unconnected_drive pull1
'unconnected_drive pull0
'nounconnected_drive
```

где ключевое слово **pull1** означает, что неподсоединеные входы модулей по умолчанию будут «подтягиваться» к единице, а **pull0** — к нулю. Использование директивы '**'nounconnected\_drive**' указывает, что неподсоединеные входы модулей находятся в высокомпеданском состоянии, например:

```
'unconnected_drive pull0 // подтягивание неподсоединеных
 // входов к «земле»
```

## 15.9. Определение пользовательских библиотек

С помощью директивы '**uselib**' можно указать библиотеку, в которой компилятор будет искать определения модулей и пользовательских примитивов (UDP) в первую очередь.

**Замечание.** Директива '**uselib**' не определена в стандарте языка IEEE 1364, однако имеется в большинстве компиляторов.

**Задание.** Проверьте, поддерживает ли используемый вами компилятор директиву '**uselib**'.

Формат директивы '**uselib**:

```
'uselib file = <file> dir = <directory> libext = <extension>
```

где **file** — имя файла (может включать путь к директории) пользовательской библиотеки; **directory** — путь к пользовательской библиотеке; **extension** — расширение файлов пользовательской библиотеки.

В данном формате каждая из конструкций **file**, **dir** и **libext** является необязательной. Использование директивы '**uselib**' без параметров отменяет использование пользовательских библиотек.

Примеры использования директивы '**uselib**:

```
'uselib file=/models/rtl.lib
 ALU i1 (y1, a, b, op); // использование модели RTL
'uselib dir=/models/gate.lib libext=.v
 ALU i2 (y2, a, b, op); // использование вентильной модели
'uselib // отключить пользовательскую
 // библиотеку
```

**Замечание.** Не все рассмотренные в главе 15 директивы языка Verilog поддерживаются компиляторами.

**Задание.** Проверьте, какие из рассмотренных в главе 15 директив поддерживают используемый вами компилятор.

**Задание.** Проверьте, какие дополнительные директивы поддерживает используемый вами компилятор.

# Г л а в а 16

## Блоки спецификаций

### 16.1. Блоки спецификаций языка Verilog

Блоки спецификаций предназначены для определения временных параметров, которые могут использоваться при работе симулятора и временного анализатора.

### 16.2. Формат блоков спецификаций

Формат блоков спецификаций имеет следующий вид:

**specify**

```
specparam_declarations
simple_path_delay
edge-sensitive_path_delay
state-dependent_path_delay
timing_constraint_checks
endspecify
```

где **specify** и **endspecify** — ключевые слова. Конструкция **specparam\_declarations** определяет параметры блока спецификаций, она имеет следующий формат:

```
specparam constant_name =value, ...;
```

где **specparam** — ключевое слово; **constant\_name** — имя константы (параметра); **value** — значение константы. В качестве значений параметров могут выступать целые и действительные числа, задержки, а также строки символов в коде ASCII.

Примеры определения параметров спецификаций:

```
specparam delay_1 = 10, // задание константе delay_1 значения 10
t_2 = 3:4:6 ; // задание t_2 значения в виде строки
// символов
```

Конструкция **simple\_path\_delay** (задержка простого пути) имеет следующий формат:

## Блоки спецификаций

173

Таблица 16.1  
Наборы значений задержек для различного числа переключений сигнала

| Задержка | Представляемые переключения выходов                                                                                                                                  |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1        | Все переключения                                                                                                                                                     |
| 2        | Возрастающие и падающие фронты                                                                                                                                       |
| 3        | Возрастающие и падающие фронты, а также переход в высокоимпедансное состояние                                                                                        |
| 6        | Возрастающие и падающие фронты, а также следующие переключения:<br>$0 > Z$ , $Z > 1$ , $1 > Z$ , $Z > 0$                                                             |
| 12       | Возрастающие и падающие фронты, а также следующие переключения:<br>$0 > Z$ , $Z > 1$ , $1 > Z$ , $Z > 0$ , $0 > X$ , $X > 1$ , $1 > X$ , $X > 0$ , $X > Z$ , $Z > X$ |

$(input\_port\ polarity : path\_token\ output\_port) = (delay);$

Здесь **input\_port** — входной порт; **output\_port** — выходной порт; **delay** — задержка; конструкция **polarity** является необязательной.

Конструкция **polarity** (полярность) может принимать значения «+» или «-». Знак минус («-») указывает, что вход будет инвертироваться. Конструкция **polarity** игнорируется большинством симуляторов, но может использоваться временными анализаторами.

Конструкция **path\_token** (признак пути) обозначается двумя символами и может иметь следующие значения: «\*>» — для полного соединения; «=>» — для параллельного соединения. Полное соединение «\*>» указывает, что сигнал (бит входного вектора) может распространяться (проходить) к любому выходному биту. Параллельное соединение «=>» указывает, что каждый бит входного вектора соединяется с соответствующим битом выходного вектора (нулевой — с нулевым, первый — с первым и так далее).

Конструкция **delay** (задержка) может представляться одним числом или в виде следующего формата: **min : typ : max**, где **min** — минимальное значение; **typ** — типовое значение; **max** — максимальное значение задержки.

Отдельные наборы задержек могут определяться для 1, 2, 3, 6 или 12 переключений (переходов) сигнала, как показано в табл. 16.1.

Конструкция **edge\_sensitive\_path\_delay** (задержка пути, чувствительного к фронту) имеет следующий формат:

$(edge\ input\_port\ path\_token\ (output\_port\ polarity : source)) = (delay);$

где конструкции **edge** **input\_port**, **path\_token**, **output\_port** и **polarity** имеют прежние значения; **edge** (необязательный элемент) может принимать значение **posedge** или **negedge**; если не определен, то используются все переключения входов; **source** (необязательный элемент) является входным портом или значением, которое примет выход.

Конструкция **state\_dependent\_path\_delay** (задержка пути, зависящего от состояния) может иметь следующие форматы:

```
if (first_condition) simple_or_edge-sensitive_path_delay
if (next_condition) simple_or_edge-sensitive_path_delay
ifnone simple_path_delay
```

где **if** и **ifnone** — ключевые слова; *first\_condition* и *next\_condition* — выражения, значения которых рассматриваются как условия; *simple\_or\_edge-sensitive\_path\_delay* — задержка простого пути или пути, чувствительного к фронту; *simple\_path\_delay* — задержка простого пути.

Конструкций с ключевыми словами **if** может быть несколько; конструкция с ключевым словом **ifnone** является необязательной.

Для одного и того же пути могут определяться различные задержки. Условия *first\_condition* и *next\_condition* могут определяться значениями только входных портов. В условиях могут использоваться любые операции, но вычисленное значение должно быть истиной или ложью (**X** или **Z** рассматриваются как истина; если результатом вычисления условия является вектор, то используется младший значащий бит). Каждая задержка для одного и того же пути должна иметь различные условия или различные фронты чувствительности. После ключевого слова **ifnone** может записываться задержка только для простого пути. Эта задержка принимается в том случае, если ни одно из условий **if** не имеет значение истина.

Примеры определения задержки пути:

```
(a => b) = 1.8; /* путь для параллельных соединений, одна
 задержка для всех переключений выходов */
(a -*> b) = 2:3:4; /* путь для полного соединения, для всех
 переключений выходов диапазон задержек
 задан в форме min : typ : max, b получает
 инверсное значение a */
specparam t1 = 3:4:6; // задержки различных путей для возраста-
 t2 = 2:3:4; // ющих и падающих фронтов переключений
(a => y) = (t1, t2);
(posedge clk => (qb -: d)) = (2.6, 1.8); /* задержка пути, чувствитель-
 ного к возрастающему фронту синхросигнала
 clk; db получает инверсное значение d */
if (rst && pst) // задержка пути, зависимого от состояния и
(posedge clk => (q +: d)) = 2; // чувствительного к возраста-
 ющему фронту сигнала clk
if (opcode = 3'b000) // задержки путей, зависимых от состояний;
(a,b *> o) = 15; // для АЛУ с различными задержками для
 // определенных операций
if (opcode = 3'b001)
(a,b *> o) = 25;
ifnone (a,b *> o) = 10; // умалчивающее значение задержки
```

### 16.3. Обнаружение путей импульсов (сбоев)

Импульс является сбоем на входных путях модуля, который меньше, чем задержка пути. Специальная константа **specparam** может использоваться для управления: будет ли импульс проходить (распространяться) к выходу (транспортная задержка), не будет проходить к выходу (инерционная задержка) или приводить к неопределенному (**X**) результату на выходе.

Формат специальной константы **specparam**:

```
specparam PATHPULSE$input$output = (reject_limit, error_limit);
specparam PATHPULSE$ = (reject_limit, error_limit);
```

где *reject\_limit* — (предел отклонения) задержка или набор задержек в форме *min* : *typ* : *max*, которая меньше или равна задержке пути модуля; любой импульс на входе, который меньше или равен пределу отклонения, будет погашен, т. е. не распространяется на выход;

*error\_limit* — (предел ошибки) задержка или набор задержек в форме *min* : *typ* : *max*, которая больше или равна величине *reject\_limit* и меньше или равна задержке пути модуля; любой импульс на входе, больше чем *error\_limit*, будет распространяться на выход; любой импульс, меньший или равный *error\_limit* и больший чем *reject\_limit*, будет распространяться на выход как неизвестное значение **X**.

Второй формат с ключевым словом **PATHPULSE\$** применяется ко всем модулям, которые не имеют определенную специальную константу **specparam**. В случае, когда предел ошибки *error\_limit* и предел отклонения *reject\_limit* совпадают, может указываться только одно значение задержки, а круглые скобки могут опускаться.

Для большей точности распространения импульсов в языке Verilog-2001 были введены дополнительные ключевые слова **pulsestyle\_onevent**, **pulsestyle\_ondetect**, **showcancelled** и **noshowcancelled**, которые могут использоваться в следующих форматах:

**pulsestyle\_onevent** *list\_of\_path\_outputs*;

— указывает, что импульс распространяется по пути на выход как неизвестное значение **X**, с той же самой задержкой, как обычное изменение входа распространяется на выход; такое функционирование принято по умолчанию;

**pulsestyle\_ondetect** *list\_of\_path\_outputs*;

— указывает, что как только импульс будет обнаружен, неизвестное значение **X** распространяется по пути на выход без задержки пути;

**showcancelled** *list\_of\_path\_outputs*;

— указывает, что негативный импульс (когда задний фронт импульса встречается до ведущего фронта) не распространяется на выход; такое функционирование принимается по умолчанию;

**noshowcancelled** *list\_of\_path\_outputs*;

— указывает, что негативные импульсы распространяются на выход как неизвестное значение **X**.

#### 16.4. Проверки временных ограничений

Проверки временных ограничений выполняются с помощью системных задач, которые моделируют ограничения изменений входных сигналов, такие, как время установки и время задержки. Данные системные задачи имеют следующий формат:

```
$setup (data_event, reference_event, limit, notifier);
$hold (reference_event, data_event, limit, notifier);
$setuphold (reference_event, data_event, setup_limit, hold_limit, notifier,
stamptime_condition, checktime_condition, delayed_ref, delayed_data);
$recovery (reference_event, data_event, limit, notifier);
$removal (reference_event, data_event, limit, notifier);
$recrem (reference_event, data_event, recovery_limit, removal_limit,
notifier, stamptime_cond, checktime_cond, delayed_ref, delayed_data);
$skew (reference_event, data_event, limit, notifier);
$timeskew (reference_event, data_event, limit, notifier, event_based_flag,
remain_active_flag);
$fullskew (reference_event, data_event, data_skew_limit,
ref_skew_limit, notifier, event_based_flag, remain_active_flag);
$period (reference_event, limit, notifier);
$width (reference_event, limit, width_threshold, notifier);
$nochange (reference_event, data_event, start_edge_offset,
end_edge_offset, notifier);
```

Временные проверки измеряют разницу между эталонным событием *reference\_event* и данным событием *data\_event*. Здесь:

*reference\_event* и *data\_event* — входные порты модуля;

*limit* — константное выражение, которое определяет количество единиц времени, по прошествии которых наступает нарушение ограничения; выражение может быть набором задержек в форме *min : typ : max*;

*notifier* (необязательный параметр) — однобитовая переменная типа **reg**, которая автоматически переключается, как только временная проверка обнаружит нарушение;

*stamptime\_condition* и *checktime\_condition* (необязательные переменные) — условия для разрешения или запрещения отрицательных временных проверок;

*delayed\_ref* и *delayed\_data* (необязательные параметры) — задерживающие сигналы для отрицательных временных проверок;

*event\_based\_flag* (необязательный параметр) — флаг, основанный на событии; когда установлен, вызывает временную проверку, которая основана на событиях, вместо основы на времени (таймере);

*remain\_active\_flag* (необязательный параметр) — забыть активный флаг; когда установлен, вызывает временную проверку, которая становится активной после того, как получено сообщение о первом нарушении;

*start\_edge\_offset* и *end\_edge\_offset* — значения задержки (могут быть положительными или отрицательными), которые увеличивают или уменьшают время, в котором изменение не может случиться.

## Конфигурация проекта

### 17.1. Конфигурации

Конфигурации (*configurations*) являются набором указаний для определения точного местонахождения исходного кода с описанием каждого экземпляра модуля или примитива, которые используются в проекте. Конфигурации задаются с помощью конфигурационных блоков (*configuration blocks*).

Конфигурационный блок является частью исходного кода проекта на языке Verilog и описывается вместе с исходным кодом проекта. Однако конфигурационные блоки описываются вне модулей проекта. Они могут находиться как в одних и тех же файлах с исходными кодами проекта, так и в отдельных файлах.

Основной единицей при конфигурации проекта является ячейка (*cell*). Ячейка — это имя модуля, примитива или другой конфигурации.

В конфигурационных блоках используются имена символьических библиотек (*symbolic library*). Символьическая библиотека представляет собой логическую совокупность ячеек (т. е. имен). Имя каждой ячейки должно совпадать с соответствующим именем модуля, примитива или конфигурации.

Для отображения имен символьических библиотек на физическое местоположение файлов с исходными кодами используются файлы карты библиотеки (*library map files*). Информация, связывающая символьическую библиотеку и экземпляры модулей, может отображаться во время симуляции путем использования форматирующего символа `%l`, который будет печатать имя библиотеки и имя модуля в виде:

`library_name.cell_name`

где `library_name` — имя символьической библиотеки; `cell.name` — имя ячейки, соответствующей модулю, в котором выполняется оператор печати (вывода информации).

### 17.2. Конфигурационные блоки

Конфигурационный блок содержит набор указаний для поиска исходного описания на языке Verilog для того, чтобы связать определенный экземпляр проекта.

Конфигурационный блок имеет следующий формат:

```
config config_name;
 design lib_name.cell_name;
 default liblist list_of_library_names;
 cell lib_name.cell_name liblist list_of_library_names;
 cell lib_name.cell_name use lib_name.cell_name:config_name;
 instance hierarchy_name liblist list_of_library_names;
 instance hierarchy_name use lib_name.cell_name:config_name;
endconfig
```

Здесь `config` и `endconfig` — ключевые слова, определяющие конфигурационный блок.

Оператор `design` определяет библиотеку и ячейку модуля верхнего уровня или модулей в иерархии проекта. В конфигурационном блоке может быть только один оператор `design`, однако может быть перечислено несколько модулей верхнего уровня. Оператор `design` должен быть первым оператором в блоке конфигурации. Необязательное имя библиотеки `lib_name` определяет, какая символьическая библиотека содержит ячейку. Если имя библиотеки опущено, то для поиска ячейки используется библиотека, которая содержит блок конфигурации. Обязательное имя ячейки `cell_name` является именем модуля наивысшего уровня в иерархии проекта, который представляет блок конфигурации.

Оператор `default liblist` определяет, в каких библиотеках искать исходное описание экземпляров, для которых не совпал определенный пункт выбора. Библиотеки ищутся в порядке перечисления. Для нескольких проектов список библиотек может включать все библиотеки, которые необходимы для определения конфигурации. Список имен библиотек `list_of_library_names` представляет собой список имен символьических библиотек, разделенных запятыми.

Оператор `cell` определяет конкретное множество библиотек, в которых выполняется поиск исходного кода для имени модуля или примитива (вместо библиотек и порядка, определенных в операторе `default`). Необязательный элемент `lib_name` определяет символьическую библиотеку, которая содержит ячейку; обязательный элемент `cell_name` является именем модуля или примитива.

Оператор `instance` определяет конкретное множество библиотек, в которых выполняется поиск исходного кода для конкретного экземпляра модуля или примитива (вместо библиотек и/или порядка, оп-

ределенных в операторе **default**). Элемент *hierarchy\_name* является полным иерархическим путем имени экземпляра модуля или примитива.

Необязательная конструкция **use** определяет местоположение конкретной ячейки или экземпляра ячейки (вместо поиска ячейки в библиотеках оператора **default**). Элемент *lib\_name* определяет имя соответствующей библиотеки, содержащей ячейку; элемент «**:config-name**» указывает, что должен использоваться другой конфигурационный блок для определения экземпляра ячейки. Оператор **design** в таком конфигурационном блоке определяет актуальную связывающую информацию.

### 17.3. Файлы карты библиотеки

Для отображения символьических библиотек на расположение физических файлов используется отдельный файл, который называется *файлом карты библиотеки* (*library map file*). Если в процессе работы над проектом по какой-либо причине изменяется местоположение файлов с исходными кодами, то необходимо модифицировать только файл карты библиотеки. При этом исходные коды языка Verilog и конфигурационные блоки не изменяются. Файл карты библиотеки может включать операторы **library** и **include**, а также комментарии языка Verilog. Файл карты библиотеки не является исходным кодом языка Verilog.

Операторы **library** и **include** имеют следующий формат:

```
library lib_name list_of_file_paths, -includelist_of_file_paths;
include library_map_file_path;
```

Элемент *lib\_name* определяет имя символьической библиотеки, которая будет упоминаться в конфигурационных блоках. Элемент *list\_of\_file\_paths* является списком путей операционной системы к конкретным файлам. Путь, который оканчивается символом слеша («/») включает все файлы в определенном каталоге (идентичен пути, который оканчивается символом «/\*»). Путь, который не оканчивается символом слеша («/»), относится к каталогу, в котором расположен данный файл карты библиотеки. В обозначении пути могут использоваться следующие специальные символы:

- ? — знак подстановки, соответствует любому одному символу;
- \* — знак нескольких символов подстановки, соответствует любому числу любых символов;
- ... — иерархический знак подстановки, соответствует любому числу иерархических директориев;
- .. — определяет родительский (ближайший верхний) каталог;
- . — определяет каталог, содержащий файл *lib.map*.

Конструкция **-includelist** определяет, где искать включаемые файлы, на которые ссылаются директивы **'include** в исходном коде языка Verilog.

Конструкция **include** позволяет в один файл карты библиотеки включать другой такой же файл.

### 17.4. Примеры конфигурации проекта

#### 17.4.1. Исходное описание проекта

Пусть проект описывается с помощью четырех файлов *top.v*, *adder.v*, *adder.vg* и *lib.map*, которые имеют следующее содержимое:

```
file top.v:
 module top(...);
 ...
 adder a1(...);
 adder a2(...);
 endmodule
 module foo(...);
 ... // rtl
 endmodule
file adder.v:
 module adder(...);
 ...
 foo f1(...);
 foo f2(...);
 endmodule
 module foo(...);
 ... // rtl
 endmodule
file adder.vg:
 module adder(...);
 ...
 foo f1(...);
 foo f2(...);
 endmodule
 module foo(...);
 ...
 endmodule
file lib.map:
 library rtlLib top.v;
 library aLib adder.*;
 library gateLib
 adder.vg;
```

В нашем примере файлы `top.v`, `adder.v` и `adder.vg` компилируются вместе с заданным библиотечным файлом `lib.map`, который имеет следующую структуру:

```
rtlLib.top // from top.v
rtlLib.foo // from top.v
aLib.adder // from adder.v
aLib.foo // rtl from adder.v
gateLib.adder // from adder.vg
gateLib.foo // from adder.vg
```

#### 17.4.2. Использование конфигурации, заданной в файле карты библиотек

В случае отсутствия в проекте каких-либо конфигурационных блоков библиотеки ищутся согласно порядку объявления библиотек в файле карты библиотек. Это означает, что все экземпляры модуля `adder` должны использовать файл `aLib.adder`, поскольку `aLib` является первой библиотекой, спецификация которой содержит ячейку с именем `adder`. Кроме того, все экземпляры модуля `foo` должны использовать файл `rtlLib.foo`, поскольку `rtlLib` является первой библиотекой, которая содержит ячейку с именем `foo`.

#### 17.4.3. Использование оператора `default`

Для того чтобы всегда использовать модуль `foo`, описанный в файле `adder.v`, можно воспользоваться следующей конфигурацией:

```
config cfg1;
 design rtlLib.top
 default liblist aLib rtlLib;
endconfig
```

Здесь оператор `default liblist` изменяет порядок поиска библиотеки в файле `lib.map`. Поэтому библиотека `aLib` всегда просматривается перед библиотекой `rtlLib`. Поскольку библиотека `gateLib` не включена в `liblist`, описания модулей `adder` и `foo` на вентильном уровне использоваться не могут.

Для возможности использования описаний модулей `adder` и `foo` на вентильном уровне следует добавить следующую конфигурацию:

```
config cfg2;
 design rtlLib.top
 default liblist gateLib aLib rtlLib;
endconfig
```

Данная конфигурация дает указания в первую очередь просматривать библиотеку `gateLib` и выбирать описание модулей `adder` и `foo` на вентильном уровне из файла `adder.vg`.

#### 17.4.4. Использование оператора `cell`

Следующая конфигурация позволяет использовать описание `rtl` для модуля `adder` и вентильное представление модуля `foo` из библиотеки `gateLib`:

```
config cfg3;
 design rtlLib.top
 default liblist aLib rtlLib;
 cell foo use gateLib.foo;
endconfig
```

Здесь оператор `cell` выбирает все ячейки с именем `foo` и явно привязывает их к вентильному представлению в библиотеке `gateLib`.

#### 17.4.5. Использование оператора `instance`

Следующая конфигурация позволяет для экземпляра `top.a1` модуля `adder` (и его потомков) использовать вентильное описание, а для экземпляра `top.a2` модуля `adder` и его потомков) использовать `rtl` представление из библиотеки `aLib`:

```
config cfg4
 design rtlLib.top
 default liblist gateLib rtlLib;
 instance top.a2 liblist aLib;
endconfig
```

Поскольку `liblist` наследуется, все потомки экземпляра `top.a2` наследуют его `liblist` из оператора выбора экземпляра `instance`.

#### 17.4.6. Использование иерархической конфигурации

Пусть наш проект описан таким образом, что следующая конфигурация использует ячейку `rtlLib.foo` для экземпляра `f1` и ячейку `getLib.foo` для экземпляра `f2`:

```
config cfg5;
 design aLib.adder;
 default liblist gateLib aLib;
 instance adder.f1 liblist rtlLib;
endconfig
```

Для того чтобы использовать эту конфигурацию `cfg5` для экземпляра `top.a2` модуля `adder` и взять полностью умалчиваемый модуль `adder` из библиотеки `aLib` для экземпляра `top.a1`, используем следующую конфигурацию:

```
config cfg6;
 design rtlLib.top;
 default liblist aLib rtlLib;
 instance top.a2 use work.cfg5:config
```

**endconfig**

Здесь оператор **instance** привязывает конфигурацию *work.cfg5: config*, которая используется для привязывания к экземпляру *top.a2* и его потомкам. Оператор **design** в конфигурации *cfg5* определяет точное привязывание для экземпляра *top.a2*. Остальная часть конфигурации *cfg5* определяет правила для построения потомков экземпляра *top.a2*. Заметим, что оператор **instance** в конфигурации *cfg5* является родственником своего собственного модуля *adder* верхнего уровня.

**Г л а в а 18****Синтезируемые конструкции языка Verilog****18.1. Общие положения**

Поскольку язык Verilog предназначен как для синтеза, так и для моделирования цифровых устройств, то не все конструкции языка могут быть синтезируемыми, т. е. физически реализованы в аппаратуре. Стандарт IEEE 1364.1 Verilog Register Transfer Level Synthesis определяет те конструкции языка Verilog, которые должны быть реализованы программными средствами синтеза (синтезаторами).

**Замечание.** Не все производители компиляторов языка Verilog придерживаются рекомендаций стандарта IEEE 1364.1.

**Задание.** Уточните, какие конструкции языка Verilog поддерживаются, а какие не поддерживаются синтезатором используемого вами программного средства проектирования.

Знание конструкций языка Verilog, поддерживаемых конкретным синтезатором, необходимо для уверенности, что описываемый вами проект может быть реализован в аппаратуре. Синтезируемые конструкции языка Verilog, которые определены стандартом IEEE 1364.1, представлены в табл. 18.1.

Кроме того, синтезаторами должны поддерживаться следующие конструкции языка Verilog-2001:

- списки чувствительности;
- `@*` — для синтеза комбинационных схем;
- совмещение объявлений портов и данных;
- объявление портов в стиле ANSI C;
- неявные сети с операторами непрерывного назначения **assign**;
- многомерные массивы;
- выбор битов и битовых полей элемента массива;
- знаковые типы данных;
- знаковые числовые литералы;
- операции арифметического сдвига `<<< и >>>`;

Таблица 18.1

Поддерживаемые синтезатором конструкции языка Verilog

| Конструкция                                                  | Обозначение                                                                                                                                              | Описание                                                                                                                                                                                                       |
|--------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Объявление модулей                                           |                                                                                                                                                          | Полностью поддерживается                                                                                                                                                                                       |
| Объявление портов                                            | <b>input</b> , <b>output</b> , <b>inout</b>                                                                                                              | Полностью поддерживаются, поддерживаются любые размеры векторов                                                                                                                                                |
| Сетевые типы данных                                          | <b>wire</b> , <b>wand</b> , <b>wor</b> , <b>supply0</b> , <b>supply1</b>                                                                                 | Полностью поддерживаются скаляры и векторы                                                                                                                                                                     |
| Типы данных переменные                                       | <b>reg</b> , <b>integer</b>                                                                                                                              | Могут быть скаляры, векторы или массивы переменных; могут ограничиваться выполнением назначений переменной только в одном процедурном операторе; по умолчанию переменная типа <b>integer</b> занимает 32 бита. |
| Параметры                                                    | <b>parameter</b>                                                                                                                                         | Константные переменные могут ограничиваться целыми числами; может не поддерживаться переопределение параметров                                                                                                 |
| Целые числа                                                  |                                                                                                                                                          | Полностью поддерживаются; поддерживаются все размеры и системы счисления                                                                                                                                       |
| Экземпляры модулей                                           |                                                                                                                                                          | Полностью поддерживаются; поддерживается передача значений сигналов как по именам портов, так и по порядку                                                                                                     |
| Экземпляры примитивов                                        | <b>and</b> , <b>nand</b> , <b>or</b> , <b>nor</b> , <b>xor</b> , <b>buf</b> , <b>not</b> , <b>bufif1</b> , <b>bufif0</b> , <b>notif1</b> , <b>notif0</b> | Полностью поддерживаются                                                                                                                                                                                       |
| Оператор непрерывного назначения                             | <b>assign</b>                                                                                                                                            | Полностью поддерживается как явная, так и неявная формы                                                                                                                                                        |
| Процедурный оператор непрерывного назначения                 | <b>assign</b>                                                                                                                                            | Полностью поддерживается; может не поддерживаться конструкция <b>deassign</b>                                                                                                                                  |
| Объявление функций                                           | <b>function</b>                                                                                                                                          | Могут использоваться только поддерживаемые конструкции                                                                                                                                                         |
| Объявление задач                                             | <b>task</b>                                                                                                                                              | Могут использоваться только поддерживаемые конструкции                                                                                                                                                         |
| Процедурный блок <b>always</b>                               | <b>always</b>                                                                                                                                            | Должен поддерживаться список чувствительности                                                                                                                                                                  |
| Оператор блока                                               | <b>begin-and</b>                                                                                                                                         | Полностью поддерживаются как именованные, так и неименованные блоки                                                                                                                                            |
| Оператор блока Оператор блокирующего процедурного назначения | <b>fork-join</b>                                                                                                                                         | Не поддерживается                                                                                                                                                                                              |
| =                                                            | =                                                                                                                                                        | Полностью поддерживается; может ограничиваться использованием только одного типа назначения для всех назначений одной и той же переменной                                                                      |
| <=                                                           | <=                                                                                                                                                       | Полностью поддерживается; может ограничиваться использованием только одного типа назначения для всех назначений одной и той же переменной                                                                      |

Окончание табл. 18.1

| Конструкция                          | Обозначение                                                                                                    | Описание                                                                                                                                                                                 |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Операторы выбора                     | <b>if_else</b> , <b>case</b> , <b>casez</b> , <b>casex</b>                                                     | Логическое значение битов <b>X</b> и <b>Z</b> поддерживаются как неопределенные значения «don't care»                                                                                    |
| Оператор цикла                       | <b>for</b>                                                                                                     | Переменная цикла может увеличиваться или уменьшаться только на единицу                                                                                                                   |
| Операторы цикла                      | <b>while</b> , <b>forever</b>                                                                                  | Цикл должен иметь один синхросигнал для каждой итерации цикла, т. е. внутри цикла должны использоваться конструкции @ ( <b>posedge clk</b> ) или @ ( <b>negedge clk</b> )                |
| Оператор <b>disable</b>              | <b>disable</b>                                                                                                 | Должен использоваться внутри одного и того же именованного блока, который прерывается                                                                                                    |
| Операции                             | &, ~ &,  , ~  , ^,<br>~ ^, ^ ~, ==, !=,<br><, >, <=, >=, !,<br>&&,   , <<, >>,<br>{}, {{ }}, ?:, +, -,<br>*, / | Операндами операций могут быть: скаляр или вектор константа или переменная                                                                                                               |
| Операции                             | ==, !=<br>[i], [msb:lsb]                                                                                       | Не поддерживаются                                                                                                                                                                        |
| Выбор бита вектора или части вектора |                                                                                                                | Полностью поддерживается в правой стороне оператора назначения; значения <i>i</i> , <i>msb</i> и <i>lsb</i> должны быть константами при использовании в левой части оператора назначения |

- операция возведения в степень \*\*, однако, может иметь ограничения;
- рекурсивные функции, может быть ограничено число рекурсивных вызовов функций;
- размерные параметры;
- явная передача параметров в командной строке;
- директивы компилятора 'ifndef' и 'elsif'.

## 18.2. Конструкции языка Verilog, поддерживаемые пакетом Quartus II фирмы Altera

В табл. 18.2 приведены конструкции языка Verilog стандарта IEEE 1364-2001, которые поддерживаются пакетом Quartus II фирмы Altera. В первом столбце таблицы приводится номер соответствующего стандарта IEEE 1364-2001, в котором описывается данная конструкция.

Таблица 18.2  
Конструкции языка Verilog, поддерживаемые пакетом Quartus II

| Раздел стандарта                                    | Конструкция                                                                  | Поддержка в пакете Quartus II                                      |
|-----------------------------------------------------|------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <b>Типы данных</b>                                  |                                                                              |                                                                    |
| 3.2                                                 | Сети ( <i>nets</i> ) и переменные ( <i>reg</i> )                             | Поддерживаются, за исключением типов <i>real</i> и <i>realtime</i> |
| 3.3                                                 | Векторы                                                                      | Поддерживаются                                                     |
| 3.4                                                 | Мощности ( <i>strengths</i> )                                                | Игнорируются при синтезе                                           |
| 3.5                                                 | Невявные объявления                                                          | Поддерживаются                                                     |
| 3.6                                                 | Инициализация сетей                                                          | Поддерживается                                                     |
| 3.7.1                                               | <i>wire</i> и <i>tri</i>                                                     | Поддерживаются                                                     |
| 3.7.2                                               | <i>wor</i> , <i>wand</i> , <i>trior</i> и <i>triant</i>                      | Поддерживаются                                                     |
| 3.7.3                                               | <i>trireg</i>                                                                | Не поддерживается                                                  |
| 3.7.4                                               | <i>tri0</i> и <i>tri1</i>                                                    | Поддерживаются                                                     |
| 3.7.5                                               | <i>supply0</i> и <i>supply1</i>                                              | Поддерживаются                                                     |
| 3.9                                                 | <i>integer</i> и <i>time</i>                                                 | Поддерживаются                                                     |
| 3.9                                                 | <i>real</i> и <i>realtime</i>                                                | Не поддерживаются                                                  |
| 3.11                                                | Параметры ( <i>parameters</i> )                                              | Поддерживаются                                                     |
| <b>Операции</b>                                     |                                                                              |                                                                    |
| 4.1.5                                               | Арифметические операции                                                      | Поддерживаются                                                     |
| 4.1.7                                               | Операции отношения                                                           | Поддерживаются                                                     |
| 4.1.8                                               | Операции равенства                                                           | Поддерживаются                                                     |
| 4.1.9                                               | Логические операции                                                          | Поддерживаются                                                     |
| 4.1.10                                              | Побитовые операции                                                           | Поддерживаются                                                     |
| 4.1.11                                              | Операции редукции                                                            | Поддерживаются                                                     |
| 4.1.12                                              | Операции сдвига                                                              | Поддерживаются                                                     |
| 4.1.13                                              | Условная операция                                                            | Поддерживается                                                     |
| 4.1.14                                              | Операция конкатенации                                                        | Поддерживается                                                     |
| 4.1.15                                              | Операция <i>or</i> в списке чувствительности                                 | Поддерживается                                                     |
| 4.2.1                                               | Выбор битов и битовых полей                                                  | Поддерживается                                                     |
| 4.2.2                                               | Адресация элементов памяти                                                   | Поддерживается                                                     |
| 4.2.3                                               | Операции над строками: назначения, конкатенации и сравнения                  | Поддерживается                                                     |
| 4.3                                                 | Выражения для минимальной, типовой и максимальной задержек                   | Поддерживаются. Игнорируются при синтезе                           |
| <b>Операторы назначения</b>                         |                                                                              |                                                                    |
| 6.1                                                 | Непрерывного назначения                                                      | Поддерживается                                                     |
| 6.2                                                 | Процедурные назначения                                                       | Поддерживаются                                                     |
| <b>Примитивы</b>                                    |                                                                              |                                                                    |
| 7.2                                                 | <i>and</i> , <i>nand</i> , <i>nor</i> , <i>or</i> , <i>xor</i> и <i>xnor</i> | Поддерживаются                                                     |
| 7.3                                                 | <i>buf</i> и <i>not</i>                                                      | Поддерживаются                                                     |
| 7.4                                                 | <i>bufif1</i> , <i>bufif0</i> , <i>notif1</i> и <i>notif0</i>                | Поддерживаются                                                     |
| 7.5                                                 | Примитивы MOS                                                                | Не поддерживается                                                  |
| 7.6                                                 | Двунаправленные переключательные элементы                                    | Не поддерживается                                                  |
| 7.7                                                 | Примитивы CMOS                                                               | Не поддерживается                                                  |
| 7.8                                                 | Примитивы <i>pullup</i> и <i>pulldown</i>                                    | Не поддерживается                                                  |
| <b>Примитивы, определяемые пользователем (UDPs)</b> |                                                                              |                                                                    |
| 8.2                                                 | Комбинационные                                                               | Поддерживается                                                     |
| 8.3, 8.4                                            | Последовательностные                                                         | Поддерживается                                                     |

Продолжение табл. 18.2

| Раздел стандарта                     | Конструкция                                                                | Поддержка в пакете Quartus II                                                          |
|--------------------------------------|----------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <b>Процедурные операторы и блоки</b> |                                                                            |                                                                                        |
| 9.2.1                                | Блокирующие процедурные назначения                                         | Поддерживаются                                                                         |
| 9.2.2                                | Неблокирующие процедурные назначения                                       | Поддерживаются                                                                         |
| 9.3                                  | Процедурные операторы непрерывного назначения                              | Не поддерживаются                                                                      |
| 9.3.1                                | Операторы <i>assign</i> и <i>deassign</i>                                  | Не поддерживаются                                                                      |
| 9.3.2                                | Операторы <i>force</i> и <i>release</i>                                    | Не поддерживаются                                                                      |
| 9.4                                  | Оператор <i>null</i>                                                       | Поддерживается для всех операторов, за исключением оператора <i>for</i>                |
| 9.4                                  | Условный оператор <i>if-else</i>                                           | Поддерживается                                                                         |
| 9.5                                  | Оператор <i>case</i>                                                       | Поддерживается                                                                         |
| 9.6                                  | Операторы цикла <i>forever</i> , <i>repeat</i> , <i>while</i> и <i>for</i> | Поддерживается                                                                         |
| 9.7.1                                | Оператор задержки <i>#</i>                                                 | Поддерживается. Игнорируется при синтезе                                               |
| 9.7.2                                | Оператор чувствительности @                                                | Поддерживается только в начале оператора <i>always</i>                                 |
| 9.7.3                                | Именованные события                                                        | Не поддерживаются                                                                      |
| 9.7.4                                | Операция <i>or</i> в списке чувствительности                               | Поддерживается                                                                         |
| 9.7.5                                | Оператор <i>wait</i>                                                       | Не поддерживается                                                                      |
| 9.7.6                                | Внутренние задержки                                                        | Поддерживаются. Игнорируется при синтезе                                               |
| 9.8.1                                | Последовательные блоки (операторные скобки <i>begin-end</i> )              | Поддерживаются                                                                         |
| 9.8.2                                | Параллельные блоки (операторные скобки <i>fork-join</i> )                  | Не поддерживается                                                                      |
| 9.9.1                                | Оператор <i>initial</i>                                                    | Поддерживается для определения условий по включению питания                            |
| 9.9.2                                | Оператор <i>always</i>                                                     | Поддерживается                                                                         |
| 11                                   | Оператор <i>disable</i>                                                    | Поддерживается                                                                         |
| <b>Задачи и функции</b>              |                                                                            |                                                                                        |
| 10.2                                 | Задачи                                                                     | Поддерживаются                                                                         |
| 10.3                                 | Функции                                                                    | Поддерживается, только когда переменные в функции являются локальными для этой функции |
| <b>Иерархические структуры</b>       |                                                                            |                                                                                        |
| 12.1                                 | Модули                                                                     | Поддерживаются                                                                         |
| 12.2                                 | Параметры                                                                  | Поддерживаются                                                                         |
| 12.2.1                               | Оператор <i>defparam</i>                                                   | Поддерживается                                                                         |
| 12.2.2                               | Оператор назначения значений параметров экземпляру модуля                  | Поддерживается                                                                         |
| 12.3                                 | Порты                                                                      | Поддерживаются                                                                         |
| 12.4                                 | Иерархические имена                                                        | Поддерживаются. Но допускаются ссылки только к объектам внутри текущего модуля         |

Окончание табл. 18.2

| Раздел стандарта                  | Конструкция                                                            | Поддержка в пакете Quatrus II                                                  |
|-----------------------------------|------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <b>Блоки спецификации</b>         |                                                                        |                                                                                |
| 13.1                              | Блоки спецификации                                                     | Игнорируются при синтезе                                                       |
| 13.2                              | Параметры блока спецификации ( <i>specparams</i> )                     | Игнорируются при синтезе                                                       |
| 13.3                              | Задержки путей модуля в блоке спецификации                             | Игнорируются при синтезе                                                       |
| <b>Системные задачи и функции</b> |                                                                        |                                                                                |
| 14.1                              | Системные задачи отображения                                           | Поддерживаются. Игнорируются при синтезе                                       |
| 14.2                              | Системные задачи ввода-вывода в файл                                   | Поддерживаются только задачи <code>\$readmemb</code> и <code>\$readmemh</code> |
| 14.3                              | Системная задача <code>\$timescale</code>                              | Не поддерживается                                                              |
| 14.4                              | Системные задачи управления симуляцией                                 | Не поддерживаются                                                              |
| 14.5                              | Системные задачи временных проверок                                    | Не поддерживаются                                                              |
| 14.6                              | Системные задачи моделирования программируемых логических матриц       | Не поддерживаются                                                              |
| 14.7                              | Системные задачи стохастического анализа                               | Не поддерживаются                                                              |
| 14.8                              | Системные функции времени симуляции                                    | Не поддерживаются                                                              |
| 14.9                              | Системные функции для преобразования переменных типа <code>real</code> | Не поддерживаются                                                              |
| 14.10                             | Функции вероятностных распределений                                    | Не поддерживаются                                                              |
| <b>Директивы компилятора</b>      |                                                                        |                                                                                |
| 16.1                              | <code>'celldefine</code> and <code>'endcelldefine</code>               | Не поддерживаются                                                              |
| 16.2                              | <code>'default_nettype</code>                                          | Поддерживается                                                                 |
| 16.3                              | <code>'define</code> и <code>'undef</code>                             | Поддерживаются                                                                 |
| 16.4                              | <code>'ifdef</code> , <code>'else</code> и <code>'endif</code>         | Поддерживаются                                                                 |
| 16.5                              | <code>'include</code>                                                  | Поддерживается                                                                 |
| 16.6                              | <code>'resetall</code>                                                 | Поддерживается                                                                 |
| 16.7                              | <code>'timescale</code>                                                | Игнорируются при синтезе                                                       |
| 16.8                              | <code>'unconnected_drive</code> и <code>'nounconnected_drive</code>    | Поддерживаются                                                                 |

## Заключение

Прежде всего, отметим, что из стандарта языка Verilog-2001 не вошло в данную книгу:

- семантики планирования (*scheduling semantics*);
- описание формата SDF (*Standard Delay Format*);
- VCD-файлы;
- стандарт процедурного интерфейса PLI языка C.

Семантику планирования (глава 5 стандарта) касаются организации работы симулятора и интересны разработчикам компиляторов. При необходимости читатель может самостоятельно ознакомится с главой 5 стандарта языка Verilog для того, чтобы более подробно узнать, как работает симулятор. Временные проверки (глава 15 стандарта) используются в блоках спецификаций для верификации временных ограничений сигналов. Формат SDF (глава 16 стандарта) содержит временные значения для определения задержек, значений параметров блока спецификаций, проверки временных ограничений и задержек межсоединений. VCD-файлы (глава 18 стандарта) содержат информацию об изменениях значений для выбранных переменных проекта, которые запоминаются с помощью системных задач. Стандарт процедурного интерфейса PLI языка C (главы 20-27 стандарта) является частью стандарта языка Verilog. С помощью интерфейса PLI можно создавать собственные системные задачи и функции.

Отметим некоторые особенности языка Verilog, которые делают его все более популярным среди разработчиков цифровой аппаратуры. Прежде всего, это простота синтаксиса, во многом совпадающая с языком программирования C. Поскольку большинство инженеров-проектировщиков и программистов хорошо знают язык C, то освоение языка Verilog для этой категории пользователей не составляет большого труда. С другой стороны, язык Verilog предоставляет большие возможности для описания цифровых устройств и систем. При этом описание может быть выполнено как с целью синтеза (автоматической аппаратной реализации), так и с целью моделирования (для проверки различных свойств проектируемой системы). Отметим, что данным свойством обладают далеко не все языки описания аппаратуры.

Кроме того, язык Verilog позволяет описывать проектируемое изделие на всех используемых в настоящее время в практике инженерного проектирования уровнях: транзисторов; вентилей; регистровых

передач; логических уравнений; поведенческом (функциональном); в виде иерархических структур.

Язык Verilog предоставляет также возможности для своего расширения. Для этого служит механизм определения пользовательских примитивов UDP и язык программирования интерфейса PLI. С помощью механизма UDP пользователь может задавать собственные примитивы, отсутствующие в языке Verilog. Интерфейс PLI служит для связи языка Verilog с языком программирования С. Возможно написание в языке С собственных системных задач и функций и подключение их к языку Verilog с помощью интерфейса PLI. Таким образом, возможно расширение языка Verilog в соответствии с пользовательскими требованиями.

Удачная, хорошо продуманная концепция языка Verilog подтверждается его стабильностью на протяжении последних 12 лет (стандарт 2005 года внес незначительные изменения). Поэтому можно ожидать, что в ближайшие несколько десятилетий язык Verilog будет оставаться основным языком проектирования цифровой аппаратуры.

Все вопросы и замечания к автору книги можно направлять по адресу: [valsol@mail.ru](mailto:valsol@mail.ru).

## Литература

1. Емец С. Verilog — инструмент разработки цифровых электронных схем // Компоненты и технологии. 2001. № 11. С. 86–88; № 12. С. 134–136; № 13. С. 76–79. № 14. С. 66–69.
2. Степченко В. Школа схемотехнического проектирования устройств обработки сигналов. Занятие 12, 13. Языки описания аппаратуры. Язык описания аппаратуры Verilog HDL // Компоненты и технологии. 2001. № 15. С. 110–115; № 16. С. 124–128.
3. Карштенбойм И. Краткий курс HDL. Часть 2. Описание языка Verilog // Компоненты и технологии. 2008. № 81. С. 164–171.
4. Поляков А.К. Языки VHDL и Verilog в проектировании цифровой аппаратуры. — Москва: СОЛООН-Пресс, 2003. — 313 с.
5. IEEE Std 1364-1995, IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language / The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA. — 653 р.
6. IEEE Std 1364-2001 (Revision of IEEE Std 1364-1995), IEEE Standard Verilog Hardware Description Language, IEEE Computer Society / The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA. — 828 р.
7. IEEE Std 1364.1, IEEE Standard for Verilog Register Transfer Level Synthesis / The Institute of Electrical and Electronics Engineers, Inc. New York, NY, USA. — 100 р.
8. Thomas D.E., Moorby P.R. The Verilog Hardware Description Language, Fifth Edition. — Kluwer Academic Publishers, New York, USA, 2002. — 381 р.
9. Palnitkar S. Verilog HDL: A guide to digital design and synthesis, Second Editions. — Mountain View, California, USA, Prentice Hall PTR, 2003. — 496 р.
10. Ciletti M.D. Advanced digital design with the Verilog HDL. — New Jersey, USA, Prentice Hall, 2003. — 985 р.
11. Ramachandran S. Digital VLSI system design. A design manual for implementation of projects on FPGAs and ASICs using Verilog. — Dordrecht, The Netherlands, Springer, 2007. — 709 р.
12. Сухарев С.А., Спицын В.Ю., Юсим И.Е. Современные средства транслирования моделей устройств из языка высокого уровня

Verilog-A во внутреннее представление системы SPECTRE // Всероссийская научно-техническая конференция «Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС)». Сборник трудов. 2006. № 1. С. 97–102.

13. Буллах Д.А. Использование языка Verilog-A в современных схемотехнических САПР // Известия высших учебных заведений. Электроника. 2007. № 1. С. 55–58.

14. Соловьев В.В., Заброцкий Л. Снижение стоимости реализации конечных автоматов путем выбора способа описания на языке Verilog // Тезисы докладов Седьмой Международной НТК «Информационные технологии в промышленности» (Information Technologies in Industry – ITI\*2012), 30-31 октября 2012 г., Минск, Объединенный институт проблем информатики НАН Беларуси, 2012. — С. 135–136.

15. Курганский С.И., Цырлов М.А., Матюшин Д.В. Моделирование программируемых логических интегральных схем с использованием Verilog HDL // Информационные технологии в проектировании и производстве. 2010. № 4. С. 48–51.

16. Palnitkar S., Saggurti P., Kuang S.-H. Finite state machine trace analysis program // Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1994. P. 52–57.

17. Arnold M., Wallace A., Cupal J., Cowles J., Engineer F. Towards a formal model of hardware synthesized from Verilog // Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1996. P. 60–66.

18. Cheng S.-T., Brayton R. K. Synthesizing multi-phase HDL programs // Proc. of the IEEE Int. Conf. Verilog HDL Conference, 1996. P. 67–76.

19. Wang T.-H., Edsall T. Practical FSM analysis for Verilog // Proc. of the IEEE Int. Conf. Verilog HDL Conference and VHDL International Users Forum, 1998. P. 52–58.

20. Balarin F., Passerone R. Specification, synthesis, and simulation of transactor processes // IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2007. Vol. 26, No. 10. P. 1749–1762.

21. Zhen Z., Hui Z. The hardware interface design in SoC with Verilog language // Proc. of the IITA Int. Conf. Service Science, Management and Engineering (SSME'09), 2009. P. 30–33.

22. Соловьев В.В. Объектно-ориентированное программирование на языке C++. Методическое пособие по курсу «Программирование» для студентов специальности «Вычислительные машины, комплексы, системы и сети». — Минск, БГУИР, 1994. — 162 с.

23. Соловьев В.В., Посредникова А.А. Реализация на ПЛИС компараторов большой размерности // Chip-News. Инженерная микроЭлектроника. 2005. № 9. С. 20–25.

24. Соловьев В.В., Посредникова А.А. Иерархическая реализация на программируемых интегральных схемах компараторов большого размера // Автоматизация проектирования дискретных систем (CAD-DD): материалы шестой международной конференции, Минск, 14–15 ноября 2007 г. С. 108–115.

25. Соловьев В.В., Посредникова А.А. Иерархический метод синтеза на программируемых логических интегральных схемах компараторов большого размера // Радиотехника и электроника. 2009. Т. 54, № 3. С. 354–362.

26. Solov'ev V.V., Posrednikova A.A. The Hierarchical Method of Synthesis of Large-Capacity Comparators with the Use of Programmable Logic Integrated Circuits // Journal of Communications Technology and Electronics. 2009. Vol. 54, No. 3. P. 338–346.

## Список сокращений

БМК – базовый матричный кристалл  
КМОП – Комплементарная металл-оксид-полупроводник структура  
ПЛИС – программируемая логическая интегральная схема  
САПР – система автоматизированного проектирования  
СБИС – сверх большая интегральная схема  
ЭСЛ – эмиттерно-связанная логика  
CMOS – Complementaty Metal-Oxide-Semiconductor (КМОП)  
ECL – Emitter-Coupled Logic (ЭСЛ)  
HDL – Hardware Description Language (язык описания аппаратуры)  
OVI – Open Verilog International  
PLI – Programming Language Interface (язык программирования интерфейса)  
RTL – Register Transfer Level (уровень регистровых передач)  
SDF – Standard Delay Format (формат стандартных задержек)  
UDP – User Defined Promotoves (определение пользовательских примитивов)  
VCD – Value Change Dump (дамп изменений значений)  
VHDL – Very High Speed Integrated Circuits Hardware Description Language (язык описания аппаратуры интегральных схем)

## Предметный указатель

Атрибут 136  
– `full_case` 137  
– `parallel_case` 138  
База числа 21  
Библиотека символическая 178  
Блок генерации 34, 143  
– именованный 38  
– конфигурационный 179  
– процедурный 81  
– – именованный 82  
– спецификаций 38, 172  
Время процедурное 86  
Выбор битов 59  
– битового поля 59  
– элемента массива 60  
Выполнение операций 71  
Выражения битовые 74  
Группа элементов генерации 144  
Диапазон битового поля 27  
Директива 167  
– включения файлов 169  
– возврата к умалчивааемым значениям 167  
– определения значения единиц времени 167  
– – макроопределений 168  
– – значений неподсоединенных входов 170  
– – пользовательских библиотек 170  
– – умалчивающего типа цепей 170  
– условной компиляции 169  
Драйвер 19  
Единица времени 167  
– логическая 19  
Задача 150  
– системная 157  
– – `@display` 157, 160  
– – `@fmonitor` 160  
– – `@fwrite` 160  
– – `@fstrobe` 160  
Задержка 41, 51, 173  
– внутренняя 93, 98, 104  
– инерционная 175  
– передачи данных 104  
– простого пути 174  
– пути, зависимого от состояния 173  
– пути, чувствительного к фронту 173  
– транспортная 175  
Знак белый 18  
Значение логическое 19  
– неизвестное 19  
– неопределенное 19  
– частично неизвестное 19  
Идентификаторы 17  
Иерархия модулей 36  
– имен 36  
Импульс 175  
Имя модуля иерархическое 37  
– относительное 37  
– полное 37  
Код исходный 143  
– компилируемый 143  
Комментарии 18  
Конструкции синтезируемые 185  
Конфигурации 178

Макроопределение 168  
 Массив 51  
   – экземпляров модулей 33  
 Моделирование проекта 14  
 Модуль 10, 24  
   – высшего уровня 36  
   – главный 36  
 Мощность источника сигнала 43  
   – логическая 19, 20  
**Направление передачи сигналов**  
 26  
 Ноль логический 19  
**Область действия** 37  
 Объявление портов 26  
 Оператор **cell** 183  
   – **default** 184  
   – **design** 179  
   – **instance** 183  
   – задержки # 86  
   – генерации 144  
   – – **case** 146  
   – – **for** 147  
   – – **if-else** 145  
   – назначения 93  
   – блокирующий 93  
   – – неблокирующий 93  
   – – непрерывного **assign** 11, 75, 94, 110  
   – – непрерывного **deassign** 110  
   – ожидания **wait** 87, 134  
   – процедурного назначения 93  
   – – блокирующий 94  
   – – – неблокирующий 101  
   – чувствительности @ 86  
   – процедурного программирования 114  
   – – **case** 118  
   – – **casez** 122, 139  
   – – **casex** 122, 139  
   – – **defparam** 33  
   – – **disable** 130  
   – – **for** 124  
   – – **forever** 129  
   – – **if-else** 114  
   – – **repeat** 127

  – – – **while** 125, 134  
   – процедурный 12  
   – **always** 81  
   – **initial** 81  
   – **force** 111  
   – **release** 111  
   – управления временем 97, 102  
 Операции арифметические 69  
   – идентичности 68  
   – конкатенации 32, 70  
   – логические 65  
   – наступления события 70  
   – отношения 67  
   – побитовые 62  
   – повторения 70  
   – редукции 64  
   – разносторонние 69  
   – условная 70  
 Описание вентиля 10  
   – проекта 10  
   – структурное 10  
**Память** 60  
 Параметры 31, 56  
   – блока спецификаций 57  
   – локальные 57  
 Передача значений параметров неявная 33  
 Переменные 49  
   – генерации 57  
 Подсоединение сигналов по именам портов 28  
   – по местоположению 28  
 Поле битовое 59  
 Полярность 175  
 Предел отклонения 175  
   – ошибки 175  
 Представление чисел с десятичной точкой 22  
   – экспоненциальное 22  
 Признак пути 173  
 Примитивы 40  
   – буферные 42  
   – вентильные 40  
   – переключательные 40  
   – пользовательские 44  
 Приоритет операций 71

Проверки временных ограничений 176  
 Пропуск сигнала 28  
 Путь иерархический 37  
   – импульса 175  
   – относительный 37  
   – полный 37  
**Размер** числа 21  
   – битовых выражений 74  
   – регистр 55  
**Сбой** 175  
**Сети** 18, 49  
**Сигнал** 18  
   – сети 52  
   – скалярный 56  
   – векторный 56  
**Симулятор** 19  
**Симуляция** 19  
**Синтезатор** 89, 185  
**Скобки** атрибутные 136  
   – операторные **begin-end** 82  
   – **fork-join** 82  
**Слово** ключевое 10  
**Состояние** высокой импеданции 19  
   – плавающее 19  
**Список** параметров 25, 31  
   – портов 25  
   – чувствительности 88  
**Стиль** описания модулей поведенческий 25  
   – структурный 25  
   – смешанный 26  
**Схема комбинационная** 89  
   – последовательностная 91  
**Таблица истинности** 45  
   – примитива 45  
**Терминал** 41  
**Тип** вентиля 10  
   – данных 49  
   – локальные параметры 57  
   – параметры 56  
   – параметры блока спецификации 57  
   – переменные 49, 54  
   – генерации 57  
**Управление временем** 106  
**Уравнение логическое** 9  
**Уровень** вентиляй 14  
   – логических уравнений 14  
   – поведенческий 14  
   – регистровых передач 14  
   – системный 14  
**Файл** карты библиотеки 178, 180  
**Фронт** возрастающий 86  
   – падающий 86  
**Функция** 149, 152  
   – загрузки памяти 164  
   – константная 149, 154  
   – преобразования величин 163  
   – – переменных 164  
   – работы с командной строкой 165  
   – – с файлами 161  
   – управления временем симуляции 162  
   – чтения и записи 163  
   – **@fopen** 159  
   – **@fclose** 160  
**Цепи** 10  
**Числа** 20  
   – восьмиричные 20  
   – двоичные 20  
   – действительные 22  
   – десятичные 20  
   – целые 20  
   – шестнадцатиричные 20  
**Экземпляр** модулей 28  
   – примитива 43  
**Элемент** генерации 144  
   – массива 60  
   – модуля 25  
**Эмуляция** модели 15  
**Язык** программирования интерфейса 157  
**Ячейка** 180

|                  |                       |
|------------------|-----------------------|
| HDL 3            | System Verilog 8      |
| IEEE 1364-1995 8 | System Verilog-2005 8 |
| IEEE 1364-2001 8 | Verilog-2001 16       |
| IEEE 1364-2005 8 | Verilog XL 7          |
| IEEE 1800-2005 8 | VHDL                  |
| Quartus II 187   |                       |

## Оглавление

|                                                               |    |
|---------------------------------------------------------------|----|
| Введение .....                                                | 3  |
| <b>Глава 1. Предварительное знакомство с языком Verilog .</b> | 7  |
| 1.1. История языка Verilog .....                              | 7  |
| 1.2. Первый проект на языке Verilog .....                     | 8  |
| 1.2.1. Описание проекта.....                                  | 8  |
| 1.2.2. Моделирование проекта .....                            | 14 |
| 1.3. Базовые элементы языка Verilog .....                     | 16 |
| 1.3.1. Ключевые слова .....                                   | 16 |
| 1.3.2. Идентификаторы .....                                   | 17 |
| 1.3.3. Белые знаки .....                                      | 18 |
| 1.3.4. Комментарии .....                                      | 18 |
| 1.4. Сигналы, сети, драйверы .....                            | 18 |
| 1.4.1. Логические значения .....                              | 19 |
| 1.4.2. Логическая мощность (сила) сигналов .....              | 20 |
| 1.5. Числа .....                                              | 20 |
| 1.5.1. Представление целых чисел .....                        | 20 |
| 1.5.2. Представление действительных чисел .....               | 22 |
| 1.6. Параллелизм языка Verilog .....                          | 23 |
| <b>Глава 2. Модули .....</b>                                  | 24 |
| 2.1. Определение модулей .....                                | 24 |
| 2.2. Элементы модулей .....                                   | 25 |
| 2.3. Объявления портов .....                                  | 26 |
| 2.4. Экземпляры модулей .....                                 | 28 |
| 2.5. Параметры .....                                          | 31 |
| 2.6. Неявная передача значений параметров .....               | 33 |
| 2.7. Массивы экземпляров модулей .....                        | 33 |
| 2.8. Иерархия модулей и иерархия имен .....                   | 36 |
| 2.9. Области иерархии и области действия имен .....           | 37 |
| <b>Глава 3. Примитивы и библиотечные модули .....</b>         | 39 |
| 3.1. Где можно найти готовое решение .....                    | 39 |
| 3.2. Примитивы языка Verilog .....                            | 40 |
| 3.3. Примитивы, определяемые пользователем .....              | 44 |
| <b>Глава 4. Типы данных .....</b>                             | 49 |
| 4.1. Два класса типов данных .....                            | 49 |
| 4.2. Сетевые типы данных .....                                | 50 |

|                                                                                    |           |
|------------------------------------------------------------------------------------|-----------|
| 4.3. Значение сигнала сети .....                                                   | 52        |
| 4.4. Типы данных переменные .....                                                  | 54        |
| 4.5. Другие типы данных .....                                                      | 56        |
| 4.5.1. Параметры .....                                                             | 56        |
| 4.5.2. Локальные параметры .....                                                   | 57        |
| 4.5.3. Параметры блока спецификации .....                                          | 57        |
| 4.5.4. Переменные генерации .....                                                  | 57        |
| 4.5.5. Тип данных событие .....                                                    | 58        |
| 4.5.6. Строки .....                                                                | 58        |
| 4.6. Выбор битов и битовых полей .....                                             | 59        |
| 4.7. Выбор элементов массива и битовых полей элементов массива .....               | 60        |
| 4.8. Объявление памяти .....                                                       | 60        |
| <b>Глава 5. Операции .....</b>                                                     | <b>62</b> |
| 5.1. Операции языка Verilog .....                                                  | 62        |
| 5.2. Побитовые операции .....                                                      | 62        |
| 5.3. Операции редукции .....                                                       | 64        |
| 5.4. Логические операции .....                                                     | 65        |
| 5.5. Операции отношения .....                                                      | 67        |
| 5.6. Операции идентичности .....                                                   | 68        |
| 5.7. Арифметические операции .....                                                 | 69        |
| 5.8. Разносторонние операции .....                                                 | 69        |
| 5.9. Выполнение операций .....                                                     | 71        |
| 5.10. Приоритет операций .....                                                     | 71        |
| 5.11. Размеры битовых выражений .....                                              | 74        |
| <b>Глава 6. Оператор непрерывного назначения assign .....</b>                      | <b>75</b> |
| 6.1. Присваивание значений в языке Verilog .....                                   | 75        |
| 6.2. Форматы оператора непрерывного назначения .....                               | 76        |
| 6.3. Использование оператора непрерывного назначения .....                         | 77        |
| <b>Глава 7. Процедурные операторы и блоки .....</b>                                | <b>81</b> |
| 7.1. Процедурные операторы <b>initial</b> и <b>always</b> , процедурные блок ..... | 81        |
| 7.2. Операторные скобки <b>begin-and</b> и <b>fork-join</b> .....                  | 82        |
| 7.3. Именованные процедурные блоки .....                                           | 82        |
| 7.4. Формат процедурных блоков .....                                               | 83        |
| <b>Глава 8. Управление процедурным временем .....</b>                              | <b>86</b> |
| 8.1. Оператор задержки <b>#</b> .....                                              | 86        |
| 8.2. Оператор чувствительности <b>@</b> .....                                      | 86        |
| 8.3. Оператор ожидания <b>wait</b> .....                                           | 87        |
| 8.4. Список чувствительности .....                                                 | 88        |
| 8.5. Список чувствительности в комбинационных схемах .....                         | 89        |

|                                                                                           |            |
|-------------------------------------------------------------------------------------------|------------|
| 8.6. Список чувствительности в последовательностных схемах .....                          | 91         |
| <b>Глава 9. Операторы процедурного назначения .....</b>                                   | <b>93</b>  |
| 9.1. Общие положения .....                                                                | 93         |
| 9.2. Оператор блокирующего назначения <b>&lt;=&gt;</b> .....                              | 94         |
| 9.2.1. Формат .....                                                                       | 94         |
| 9.2.2. Управление временем .....                                                          | 97         |
| 9.2.3. Внутренние задержки .....                                                          | 98         |
| 9.2.4. Особенности синтеза .....                                                          | 100        |
| 9.3. Оператор неблокирующего назначения <b>&lt;&lt;=</b> .....                            | 101        |
| 9.3.1. Формат .....                                                                       | 101        |
| 9.3.2. Управление временем .....                                                          | 102        |
| 9.3.3. Внутренние задержки .....                                                          | 104        |
| 9.3.4. Особенности синтеза .....                                                          | 105        |
| 9.4. Управление временем в процедурных операторах назначения во время моделирования ..... | 106        |
| 9.5. Процедурные операторы <b>assign</b> и <b>deassign</b> .....                          | 110        |
| 9.6. Процедурные операторы <b>force</b> и <b>release</b> .....                            | 111        |
| <b>Глава 10. Операторы процедурного программирования .....</b>                            | <b>114</b> |
| 10.1. Общие положения .....                                                               | 114        |
| 10.2. Оператор <b>if-else</b> .....                                                       | 114        |
| 10.3. Оператор <b>case</b> .....                                                          | 118        |
| 10.4. Операторы <b>casez</b> и <b>casex</b> .....                                         | 122        |
| 10.5. Оператор <b>for</b> .....                                                           | 124        |
| 10.6. Оператор <b>while</b> .....                                                         | 125        |
| 10.7. Оператор <b>repeat</b> .....                                                        | 127        |
| 10.8. Оператор <b>forever</b> .....                                                       | 129        |
| 10.9. Оператор <b>disable</b> .....                                                       | 130        |
| 10.10. Пример использования операторов процедурного программирования .....                | 132        |
| 10.11. Различие между операторами <b>wait</b> и <b>while</b> .....                        | 134        |
| <b>Глава 11. Атрибуты .....</b>                                                           | <b>136</b> |
| 11.1. Атрибуты языка Verilog .....                                                        | 136        |
| 11.2. Атрибут <b>full_case</b> .....                                                      | 137        |
| 11.3. Атрибут <b>parallel_case</b> .....                                                  | 138        |
| <b>Глава 12. Блок генерации .....</b>                                                     | <b>143</b> |
| 12.1. Блоки генерации языка Verilog .....                                                 | 143        |
| 12.2. Формат блока генерации .....                                                        | 144        |
| 12.3. Операторы генерации .....                                                           | 144        |
| 12.3.1. Группа элементов генерации .....                                                  | 144        |
| 12.3.2. Оператор <b>if-else</b> .....                                                     | 145        |
| 12.3.3. Оператор <b>case</b> .....                                                        | 146        |

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| 12.3.4. Оператор <b>for</b> .....                                            | 147 |
| <b>Глава 13. Задачи и функции</b> .....                                      | 149 |
| 13.1. Задачи и функции языка Verilog .....                                   | 149 |
| 13.2. Автоматические и статические задачи и функции .....                    | 149 |
| 13.3. Задачи .....                                                           | 150 |
| 13.4. Функции .....                                                          | 152 |
| 13.5. Константные функции .....                                              | 154 |
| 13.6. Сравнение функций и задач .....                                        | 156 |
| <b>Глава 14. Системные задачи и функции</b> .....                            | 157 |
| 14.1. Системные задачи и функции языка Verilog .....                         | 157 |
| 14.2. Системные задачи для отображения текста .....                          | 157 |
| 14.3. Системные задачи и функции для работы с файлами .....                  | 159 |
| 14.3.1. Открытие и закрытие файлов .....                                     | 159 |
| 14.3.2. Вывод информации в файл .....                                        | 160 |
| 14.3.3. Другие функции работы с файлами .....                                | 161 |
| 14.4. Другие системные задачи и функции .....                                | 162 |
| 14.4.1. Управление процессом симуляции .....                                 | 162 |
| 14.4.2. Управление временем симуляции .....                                  | 162 |
| 14.4.3. Преобразование знаковых и беззнаковых величин .....                  | 163 |
| 14.4.4. Запись и чтение в переменные и из строки символов .....              | 163 |
| 14.4.5. Загрузка содержимого памяти .....                                    | 164 |
| 14.4.6. Преобразование переменных типа <b>real</b> в 64-битовый вектор ..... | 164 |
| 14.4.7. Функции для работы с командной строкой .....                         | 165 |
| <b>Глава 15. Директивы компилятора</b> .....                                 | 167 |
| 15.1. Директивы компилятора языка Verilog .....                              | 167 |
| 15.2. Возврат к умалчиваемым значениям директив компилятора .....            | 167 |
| 15.3. Определение значения единицы времени .....                             | 167 |
| 15.4. Макроопределения .....                                                 | 168 |
| 15.5. Директивы условной компиляции .....                                    | 169 |
| 15.6. Включение файлов .....                                                 | 169 |
| 15.7. Определение умалчивающего типа цепей .....                             | 170 |
| 15.8. Определение логических значений для неподсоединенных входов .....      | 170 |
| 15.9. Определение пользовательских библиотек .....                           | 170 |
| <b>Глава 16. Блоки спецификаций</b> .....                                    | 172 |
| 16.1. Блоки спецификаций языка Verilog .....                                 | 172 |
| 16.2. Формат блоков спецификаций .....                                       | 172 |
| 16.3. Обнаружение путей импульсов (сбоев) .....                              | 175 |
| 16.4. Проверки временных ограничений .....                                   | 176 |

|                                                                                       |     |
|---------------------------------------------------------------------------------------|-----|
| <b>Глава 17. Конфигурация проекта</b> .....                                           | 178 |
| 17.1. Конфигурации .....                                                              | 178 |
| 17.2. Конфигурационные блоки .....                                                    | 179 |
| 17.3. Файлы карты библиотеки .....                                                    | 180 |
| 17.4. Примеры конфигурации проекта .....                                              | 181 |
| 17.4.1. Исходное описание проекта .....                                               | 181 |
| 17.4.2. Использование конфигурации, заданной в файле карты библиотек .....            | 182 |
| 17.4.3. Использование оператора <b>default</b> .....                                  | 182 |
| 17.4.4. Использование оператора <b>cell</b> .....                                     | 183 |
| 17.4.5. Использование оператора <b>instance</b> .....                                 | 183 |
| 17.4.6. Использование иерархической конфигурации .....                                | 183 |
| <b>Глава 18. Синтезируемые конструкции языка Verilog</b> ...                          | 185 |
| 18.1. Общие положения .....                                                           | 185 |
| 18.2. Конструкции языка Verilog, поддерживаемые пакетом Quartus II фирмы Altera ..... | 187 |
| Заключение .....                                                                      | 191 |
| Список литературы .....                                                               | 193 |
| Список сокращений .....                                                               | 196 |
| Предметный указатель .....                                                            | 197 |

*Адрес издательства в Интернет WWW.TECHBOOK.RU*

**Соловьев Валерий Васильевич**

**Основы языка проектирования цифровой аппаратуры Verilog**

Редактор Ю. Н. Чернышов  
Компьютерная верстка Ю. Н. Чернышова  
Обложка художника В. Г. Ситникова

Подписано в печать 15.12.2013. Формат 60×90/16. Усл. печ. л. 12,88.  
Тираж 500 экз. (1-й завод 100 экз.) Изд. №140353  
ООО «Научно-техническое издательство «Горячая линия–Телеком»

**В. В. СОЛОВЬЕВ**

# **Основы языка проектирования цифровой аппаратуры VERILOG**

Рассмотрен популярный язык проектирования цифровой аппаратуры Verilog. В книге достаточно полно описаны основные синтаксические элементы и конструкции языка с точки зрения их практического использования. Каждая конструкция языка сопровождается примером. Изложение материала не привязано к определенной элементной базе или конкретному программному средству проектирования, поэтому материал книги может использоваться при разработке проектов как на заказных СБИС и БМК, так и на ПЛИС. Популярность языку Verilog придает простота синтаксиса, во многом совпадающего с языком программирования C, а также большие возможности при описании цифровых устройств и систем, как для синтеза, так и для моделирования, от уровня транзисторов до сложных иерархических структур. Язык Verilog предоставляет также возможности для своего расширения. Для этого служит механизм определения пользовательских примитивов UDP и язык программирования интерфейса PLI.

Настоящая книга, в первую очередь, предназначена разработчикам цифровых устройств и систем для самостоятельного изучения языка Verilog. Поскольку изложение основных элементов и конструкций языка достаточно полное, книга может также использоваться как справочник по языку Verilog. Кроме того, материал книги может быть использован преподавателями, аспирантами и студентами соответствующих специальностей вузов.

**САЙТ ИЗДАТЕЛЬСТВА:**

**WWW.TECHBOOK.RU**

ISBN 978-5-9912-0353-1



9 785991 203531