Министерство образования и науки Российской Федерации
Федеральное агентство по образованию
Московский государственный институт электронной техники
(технический университет)

В.Б. Стешенко, Т.В. Попова, Д.Б. Малашевич

Основы HDL Verilog как средства проектирования цифровых устройств

Учебное пособие

Под редакцией кандидата технических наук, доцента
А.И. Сухопарова

Допущено Учебно-методическим объединением вузов
по университетскому политехническому образованию
в качестве учебного пособия для студентов высших
учебных заведений, обучающихся по направлению 230100 «Информатика и вычислительная техника», специальности 230104 «Системы автоматизированного проектирования»

Москва 2006

УДК 658.512.011.56:004.432:681.32
О75
Рецензенты: канд. техн. наук *Г.К. Лукошко*;
канд. техн. наук *Г.А. Манохин*;
доц. *В.А. Мартынюк*

Стешенко В.Б., Попова Т.В., Малашевич Д.Б.
О75 Основы HDL Verilog как средства проектирования цифровых устройств: Уч. пос. / Под ред. А.И. Сухопарова. - М.: МИЭТ, 2006. -
136 с.: ил.
ISBN 5-7256-0433-0

Изложены теоретические положения, составляющие основу языка высокого уровня Verilog. Освещен круг вопросов, знание которых необходимо начинающему пользователю языка. Рассмотрены основные понятия Verilog. Приводятся структуры как всего Verilog-проекта, так и различных по назначению модулей. Подробно излагаются конструкции операторов, блоков, циклов, системных директив и прочих составных частей языка, имеющих наиболее частое применение. Рассматриваются формы автоматного описания БИС, структура Verilog-проекта для конечных автоматов. Анализируются современные средства синтеза, а также методология создания моделе- и синтезопригодных Verilog-описаний цифровых устройств.

Изложение материала сопровождается примерами описания алгоритмов функционирования цифровых устройств как на уровне логических вентилей, так и на поведенческом уровне. Приводятся примеры проектирования автоматов Мили и Мура средствами языка Verilog.

Пособие предназначено для студентов, изучающих Verilog. Оно может также представлять интерес для специалистов - разработчиков цифровых устройств.

ISBN 5-7256-0433-0

Учебное пособие
Стешенко ***Владимир Борисович***
Попова ***Татьяна Викторовна***
Малашевич ***Денис Борисович***
Основы HDL Verilog как средства проектирования цифровых устройств
Редактор *Л.М. Рогачева*. Технический редактор *Л.Г. Лосякова*. Корректор *Л.Г. Лосякова*. Компьютерная верстка *А.А. Григорашвили.*
Подписано в печать с оригинал-макета 03.10.06. Формат 60×84 1/16. Печать офсетная. Бумага офсетная. Гарнитура Times New Roman. Усл. печ. л. 7,89. Уч.-изд. л. 6,8. Тираж 150 экз. Заказ 161.
Отпечатано в типографии ИПК МИЭТ.
124498, Москва, Зеленоград, проезд 4806, д. 5, МИЭТ.

Введение

В настоящее время проектирование больших систем осуществляется с применением так называемых языков описания высокого уровня (*Hardware Description Languages - HDL*) VHDL и Verilog. Язык описания аппаратуры Verilog был разработан фирмой *Gateway Design Automation* в 1984 г. После поглощения последней фирмой *Caddence* язык получил широкое распространение среди разработчиков и стал не менее популярен, чем VHDL.

В отличие от VHDL, структура и синтаксис которого напоминают такие "сложные" языки, как АДА или АЛГОЛ, Verilog обеспечивает более лаконичный и удобочитаемый синтаксис, характерный для очень популярного в среде программистов и разработчиков встроенных систем языка Си. Verilog позволяет достаточно эффективно выполнить описание и провести моделирование и синтез цифровых схем благодаря наличию развитых средств описания устройств, применению встроенных примитивов и примитивов пользователя, средств временного контроля, моделированию задержки распространения от входа до выхода, возможности задания внешних тестовых сигналов.

HDL Verilog изначально предназначался для моделирования цифровых систем и как средство описания синтезируемых проектов стал использоваться с 1987 г. Впоследствии этот язык начал применяться и для проектирования аналоговых схем. В настоящее время ведущие пакеты синтеза систем на программируемых логических интегральных схемах (ПЛИС) от таких фирм, как *Synopsis, Caddence, Mentor Graphics,* поддерживают синтез с описаниями на языке Verilog.

Языки описания аппаратуры долгое время были прерогативой довольно узкого класса разработчиков специализированных интегральных схем. С появлением ПЛИС резко расширился круг пользователей, заинтересованных в применении современных способов описания проекта, при этом испытывающих некоторый недостаток литературы, особенно русскоязычной, в которой бы достаточно подробно излагались основы синтаксиса языка описания аппаратуры Verilog и рассматривались примеры их применения при реализации устройств на ПЛИС. В данном пособии предпринята попытка восполнить этот пробел.

Цель данного пособия не только помочь читателю изучить основы языка Verilog, но и показать на конкретных примерах создание синтезо- и моделепригодных описаний небольших проектов.

Глава 1. Общие понятия

1.1. Особенности проектирования с применением Verilog

В Verilog используется независимый от технологии стиль описания устройств. Это позволяет проектировать системы, обладающие высокой мобильностью, т.е. возможностью переносить их на другую элементную базу. Тогда для создания законченного проекта необходимо лишь произвести компиляцию в соответствующей системе в элементную базу производителя. Но в этом состоит и основной недостаток языков высокого уровня - неполный учет специфических особенностей используемой элементной базы.

Языки описания аппаратуры являются формальной записью, которая может быть использована на всех этапах разработки цифровых электронных систем. Это обусловлено тем, что язык легко воспринимается и машиной, и человеком. Он может использоваться на этапах описания, верификации, синтеза и тестирования аппаратуры, а также передачи данных о проекте, его модификации и сопровождения.

Прежде чем приступить к рассмотрению синтаксиса языков описания аппаратуры, следует определить основные понятия, касающиеся применения языков при проектировании цифровых систем.

С помощью HDL Verilog цифровая система может быть описана на структурном, поведенческом (главная особенность и достоинство HDL) и потоковом уровнях. Такое описание называется Verilog-описанием или Verilog-моделью устройства.

Структурное описание является достаточно традиционным и представляет собой описание системы в виде совокупности компонентов и связей между ними. Как следует из названия, оно отражает структуру системы и может быть реализовано в "железе" с использованием соответствующего программного и аппаратного обеспечения.

Поведенческое описание представляет собой описание системы при помощи задания зависимости вход-выход. Такое описание используется для моделирования системы, функциональной и временной верификации и не всегда может быть синтезировано. Вопрос о том, насколько поведенческое описание может быть реализовано при синтезе проекта, решается в рамках некоторого подмножества языка, называемого "синтезируемым подмножеством". Преимущества языков описания высокого уровня в полной мере реализуются только тогда, когда разработчик грамотно использует конструкции языка для написания синтезируемых поведенческих описаний.

Потоковое описание не всегда выделяется в отдельный класс. По сути оно представляет собой поведенческое описание межрегистровых передач и обмена данными. Такой тип описания удобен для проектирования систем, имеющих шинную архитектуру.

Разработчики предпочитают выражать свой проект в поведенческой форме, откладывая подробности реализации на более поздний этап проектирования. Абстрактное представление позволяет рассматривать различные варианты устройства и обнаруживать узкие места до начала детальной проработки. Таким образом, можно уменьшить количество итераций при проектировании. Следует помнить, что описание должно быть полным и однозначным.

При написании модели на языке Verilog преследуются три цели: она должна правильно работать, быстро моделироваться и хорошо синтезироваться. Естественно, что эти требования часто конфликтуют между собой.

Итак, язык высокого уровня Verilog позволяет описать и ввести проект в САПР, а также решить задачи, связанные с синтезом и верификацией проекта. Поэтому важным является понимание основ логического проектирования цифровых устройств.

1.2. Синтаксис языка Verilog. Ключевые слова

Текст, написанный на языке высокого уровня Verilog, представляет собой проект заданного будущего устройства (Verilog-проект). Он состоит из конструкций, расположенных в определенном порядке. Любой фрагмент Verilog-проекта может быть назван Verilog-текстом или кодом.

В языке Verilog существует несколько видов конструкций:

- ключевые слова с атрибутами;
- структурные элементы (см. главу 4);
- операторы;
- комментарии;
- строки.

Конструкции записываются с помощью алфавита языка Verilog.

В алфавит входят:

- прописные и строчные буквы латинского алфавита (комментарии и строки могут быть написаны на русском языке); Verilog представляет собой регистро-зависимый язык, т.е. строчные и прописные буквы могут в нем различаться;
- цифры десятичной системы счисления;
- специальные знаки: $? + – ‘ “ < = и т.д.;
- неизображаемые символы: пробел, табуляция, переход на новую строку, которые применяются для обозначения в тексте пустого пространства;
- различного рода разделители: точка с запятой, пробел, запятая, двоеточие, скобки и подчеркивание. Точка с запятой ставится в конце каждой конструкции (за несколькими исключениями). Пробел, запятая, двоеточие и скобки используются внутри конструкции. Разделитель "подчеркивание" применяется исключительно для удобства чтения в именах и числах. Например, *rs_trigger* или *c = 16’b0010_1101_0101_1001*.

Ключевые слова зарезервированы в языке, т.е. их нельзя использовать в качестве имен. Ключевые слова представляют собой англоязычные термины, имеющие конкретный проектный смысл. Для их изображения используются только строчные буквы. Два соседних ключевых слова разделяются любым неизображаемым символом, например, *initial begin* и т.д. Перечень рассматриваемых в пособии ключевых слов приводится в Приложении 1. Всего в языке Verilog насчитывается около ста ключевых слов различного назначения.

Существуют две версии стандарта Verilog: 1995 г. и 2000 г., причем последняя полностью включила в себя свойства предыдущей версии [1]. В настоящее время есть сообщения о появлении третьей версии стандарта. Список конструкций и ключевых слов может меняться для разных систем синтеза. В данном пособии рассматривается свыше пятидесяти ключевых слов (в разных конструкциях) и системных директив, которые наиболее часто применяются в проектировании.

Большинство ключевых слов применяется вместе со своими атрибутами. Их запись может занимать несколько строк. Атрибуты представляют собой соответствующую данному ключевому слову информацию, записанную в заданном формате. Формат определяет принятый порядок записи. В языке также имеются ключевые слова, которые применяются без атрибутов.

В Verilog могут применяться комментарии [2], которые оформляются двумя способами:

// комментарий;

```
/* это
    комментарий */
```

Первый способ используется для записи комментария в одну строку, если необходимы пояснения к какой-либо конструкции или фрагменту текста. Второй способ используется для записи комментария в несколько строк. Этот способ применяется в основном перед текстом какого-либо фрагмента проекта для записи его назначения, а также при отладке проекта. Например, один фрагмент Verilog-текста может быть оформлен как комментарий (закомментирован) при отладке другого фрагмента.

Комментарии не могут быть вложенными.

Применение комментариев безусловно полезно, оно помогает разработчику избежать лишних затрат времени на поиски необходимой информации.

В Verilog допустимо применение так называемых строк, которые содержат вспомогательную информацию. Строка заключается в кавычки и не может занимать более одной линии. Например:

```
«привет, читатель»; // правильное использование строки
«привет,
        читатель»; // неправильное использование строки
```

Таким образом, для размещения информации могут быть использованы как комментарии, так и строки. Комментарии применяются в основном для Verilog-текста, а строки - для всего остального, хотя это деление достаточно условно.

1.3. Правила именования

В Verilog-проекте все модули (см. главу 4), сигналы и другие переменные, а также константы имеют свои имена. Выбор выразительного имени является важной частью процесса проектирования. Удачно выбранное имя модуля или сигнала объясняет, какую функцию он выполняет и что означает.

При выборе имен необходимо руководствоваться существующими правилами, которые включают в себя требования и рекомендации.

Требования стандарта языка Verilog состоят в следующем.

1. В качестве имен нельзя использовать ключевые слова; ключевые слова могут быть только составной частью имени; например, *q_output*.
2. Имена могут состоять из латинских букв, цифр, а также знаков: *$* и подчеркивание, например, *n$657*.
3. Цифры и знак *$* не могут быть первыми в имени.

Соблюдая требования, полезно помнить и о рекомендациях.

1. Использовать информативные, но короткие имена. Информативное имя избавит разработчика от дополнительных комментариев.
2. Использовать постоянно одни и те же сокращения в случае выполнения однотипных функций.
3. Применять в именах только строчные буквы и подчеркивание (в качестве разделителя).
4. Пользоваться одним и тем же именем во всей иерархии проекта.

К перечисленным правилам можно добавить, что ограничений на длину имен нет, но следует соблюдать разумные пределы. Например, *data_in*, *out_mux* и т.д.

1.4. Числа в Verilog. Неопределенное и высокоимпедансное состояние

Язык Verilog может работать со следующими величинами: константами, которые хранят фиксированную информацию, и переменными, которые могут изменяться во время моделирования. И константы, и переменные могут быть как целыми, так и действительными числами.

Наиболее часто применяются целые числа. Формат записи целого числа имеет следующий вид:

<разрядность>′<основание><число>

Здесь угловые скобки указывают содержание записи и в Verilog-тексте не применяются. Разрядность указывается в десятичной системе и определяет количество бит под представление числа.

У констант и десятичных чисел разрядность и основание могут не указываться. В этом случае разрядность рассчитывается из величины числа.

Основание определяет выбранную систему счисления:

B или *b* - двоичную;

D или *d* - десятичную;

H или *h* - шестнадцатеричную;

O или *o* - восьмеричную.

Например: 5′*b*10011 // 5 бит отводится под представление числа 19

4′*d* 12 // 4 бита отводится под представление числа 12

Приведем некоторые способы записи числа 8:

4′b1000 // запись сделана в двоичной системе

4′d8 // запись сделана в десятичной системе

8 // десятичная система может применяться без указания

// разрядности и основания

Разработчик сам выбирает удобную для работы систему счисления.

В любом месте числа может быть записан знак подчеркивания в качестве разделителя разрядов, улучшающий читабельность.

Например:

12′*b*0001_1010_1000.

Если разрядность заказана больше, то число дополняется слева либо нулями, либо знаком *x* (неопределенность) в зависимости от значения старшего разряда.

Например:

запись *a* = 4′*b*010; означает *a* = 0010,

запись *c* = 4′*b*x10; означает *c* = *x*x10.

Действительные числа могут быть представлены в двух формах: десятичной и экспоненциальной.

Например:

1.8; 1_2387.3980_3047; 4.8*E*10; 2.1*e*-9 // правильная запись

3. // неправильная запись

Для обозначения поведения сигналов в Verilog, кроме 0 и 1, предусмотрен учет неопределенного и высокоимпедансного состояний. Эти состояния обозначаются соответственно символами *x* и *z*. Символ *x* используется также для обозначения безразличного состояния.

В начале работы устройства появление *x* является естественным, так как из-за наличия задержек элементов не во всех узлах сразу устанавливаются определенные значения сигналов. Но в процессе моделирования появление *x* может свидетельствовать о конфликтной ситуации.

Появление высокоимпедансного состояния *z* (так называемого третьего состояния) говорит о наличии обрыва в цепи. В зависимости от особенностей проекта состояние *z* может использоваться в качестве рабочего, но может быть и признаком конфликта.

Как *x*, так и *z* указываются в числе вместо цифры в том месте, которое соответствует сигналу, принимающему эти значения. Для обозначения состояния *z* может использоваться символ ?. Это рекомендуется делать в операторах выбора *case* (см. параграф 7.6) для улучшения читаемости кода.

Например: 4′*b*10*x*0
4′*b*101*z*
12′dz
12′d?

Кроме 1; 0; *x*; *z* могут использоваться понятия сильного и слабого сигналов (***strong, weak***), а также высокого и низкого сигналов (***highz, false***). Например, *weak*1, *strong*0 и т.д. Наибольшую силу имеет сигнал ***strong,*** наименьшую - *z*.

1.5. Типы данных

Verilog поддерживает несколько различных типов данных. Поименованные константы и переменные могут быть объявлены, т.е. указаны в тексте целыми, а также действительными величинами или параметрами.

Для этого применяются ключевые слова и форматы:

integer <имена величин через запятую>;

real <имена величин через запятую>;

parameter <имя параметра> = <численное значение параметра>;

Например: *integer i,j;*
real a,c;
parameter st0 = 3'd0;

Если переменная объявлена как *integer*, то подразумевается 32-битное число. Такие числа применяются, например, при подсчете количества циклов.

Ключевое слово *parameter* применяется для введения и назначения величины, которая не меняется в данной редакции текста, но может измениться в новой редакции.

При проверке алгоритма функционирования (симуляции) используется временной тип данных - ***time*** (см. главу 10), который позволяет рассматривать поведение объекта во временном пространстве. Эти данные применяются встроенными функциями и являются обычно
64-битовыми целыми величинами.

И наконец, данные типа ***event*** (событие) - в языке существует ряд операторов и конструкций для работы с событиями.

Глава 2. Порты

2.1. Классификация портов

В языке Verilog используется модульный принцип описания устройства или алгоритма его функционирования (см. подробнее главу 4). Это значит, что различные части устройства описываются по типу модуля, т.е. описание носит законченный, автономный характер. Причем одни модули могут содержать другие модули (принципы иерархии).

Порты - это выводы модулей, выполняющие несколько функций:

- порты указывают на входы и выходы модулей;
- с помощью портов описываются связи одного модуля с другими;
- порты указывают на связи внутри модуля.

Вместо термина "порт" иногда используется термин "узел" (или "контакт", или "вывод"), если имеется в виду элемент устройства; а также термин "данные" (или "переменные"), если подразумеваются конкретные значения или поведение сигнала в данном узле.

У каждого порта объявляются (записываются в Verilog-тексте):

- назначение;
- характер (или тип сигнала).

2.2. Назначение портов

В зависимости от назначения каждый порт должен быть объявлен как выход, вход или двунаправленный вывод.

При этом применяются следующие ключевые слова и форматы:

output < имя порта или перечень имен через запятую>;

input <имя порта или перечень имен через запятую>;

inout < имя порта или перечень имен через запятую>;

В настоящее время в пользовательской среде языка Verilog приняты определенные стиль и соглашения, обусловленные производственной необходимостью. В соответствии с этим во всем Verilog-проекте порты перечисляются только в последовательности: выходы, входы, двунаправленные элементы. Причем внутри каждой группы имена перечисляются по алфавиту. Такой порядок необходим для обеспечения возможности обработки Verilog-проекта различными программами. Это правило следует соблюдать с ключевыми словами *output* и *input*, с остальными ключевыми словами его выполнять не обязательно, но желательно.

Например, для *D*-триггера со статическим управлением (триггер - защелка), изображенного на рис.2.1, порты могут быть объявлены следующим образом:

input data, enable, reset;

output q_tr;

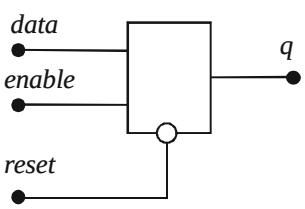


Рис.2.1. D-триггер со статическим управлением записью

А для фрагмента устройства, изображенного на рис.2.2, порты могут быть объявлены так:

output b;
input a;
inout c;

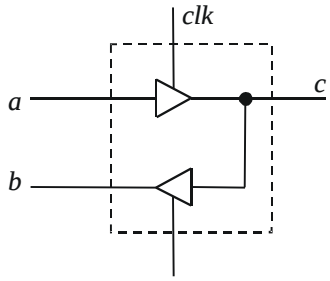


Рис.2.2. Фрагмент устройства
с использованием двунаправленного вывода

При объединении выходов логических элементов соответствующий контакт объявляется как элемент. Он может вести себя как монтажное И (***wand***) или монтажное ИЛИ (***wor***). Например, объединение во фрагменте устройства (рис.2.3) может быть описано следующим
образом:

wand (*out,le1_out,le2_out*);

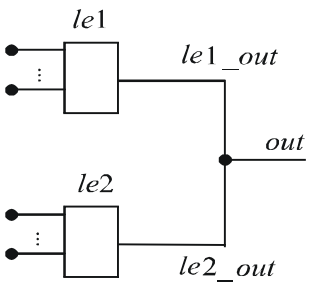


Рис.2.3. Фрагмент устройства
с объединением выходов элементов

Если, конечно, оно ведет себя как монтажное И.

В группе перечня портов в случае необходимости могут быть объявлены узлы подключения источника (источников) питания и заземления.

Ключевые слова и формат записи следующие:

supply1 <имя узла источника питания>;
supply0 <имя узла заземления>;

Например: *supply1 e*;
supply0 grnd;

Примечание: эти узлы не всегда обязательно объявлять с помощью ключевых слов; иногда достаточно присвоить им значения 1 и 0.

Например:

e = 1′*b*1;
grnd = 1′*b*0;

Однако следует помнить, что все назначения делаются в блоке *initial* (см. главу 3).

2.3. Типы сигналов в Verilog. Цепи. Регистры

При написании Verilog-проекта необходима следующая информация о сигналах: значения, с помощью которых описывается поведение сигналов, и характер поведения (или тип) сигналов.

Как указывалось в главе 1, существуют четыре значения, которые могут принимать сигналы: 0, 1, *z* и *x*. Первые два соответствуют логическим уровням, третье - состоянию с высоким импедансом, четвертое означает неопределенное состояние и используется во всех случаях, когда симулятор не может определить значение данного сигнала (для входного сигнала *x* означает безразличное состояние).

В зависимости от характера (типа) сигнала порт может быть объявлен в Verilog-тексте как цепь или как регистр.

Цепи характеризуют непрерывное изменение (модифицирование) сигнала на выходе цифрового устройства относительно изменения сигналов на его входах. Цепь принимает значение источника (драйвера) сигнала. Если драйверы цепи имеют различную силу (параграф 1.4), цепь принимает значение более сильного сигнала; если сигналы по силе равнозначны, то цепь принимает неопределенное состояние *x*. Наименьшую силу имеет сигнал *z*.

Ключевое слово и формат записи при объявлении порта (портов) цепью:

wire <имя порта или перечень имен через запятую>;

Например: *wire prm*;

wire a,b;

По умолчанию порту, объявленному как *wire*, назначается величина *z* (*default value*) и разрядность - 1 бит.

Данные типа *wire* не могут быть назначены явно (т.е. им не могут быть присвоены численные значения в тексте), они нуждаются в сигнале-драйвере. Такое назначение называется непрерывным.

Цепь может также называться проводом.

Цепи характеризуются тем, что не могут хранить информацию (в качестве аналога цепи можно представить комбинационные схемы).

Если тип сигнала не объявлен, то по умолчанию порту присваивается тип *wire*. Хотя опыт показывает, что у всех портов лучше указывать все характеристики.

Рассмотренные ранее ключевые слова *supply*0 и *supply*1 также используются для обозначения цепей.

Тип сигнала не является раз и навсегда установленной величиной, а может меняться от модуля к модулю.

Регистры. Основное различие между цепями и регистрами состоит в том, что значение регистра должно быть назначено явно. Эта величина сохраняется до тех пор, пока не сделано новое назначение. Такое назначение называется процедурным. Ключевое слово - *reg*, величина по умолчанию - *x* (величины *z* в регистре быть не может); разрядность по умолчанию - 1 бит.

Формат записи:

reg <имя порта или перечень имен через запятую>;

Например: *reg out_counter*;

reg out_nq,out_q;

В этом случае говорят: данные типа *reg* или данные типа *wire*.

Аналогом регистра является элемент памяти.

Для *D*-триггера, изображенного на рис.2.1, тип портов может быть объявлен следующим образом:

reg data, enable, reset;

wire q_tr;

Процедурное назначение характеризует не только данные типа *reg*, но также *integer и real*. Однако типы *integer* и *real* представляют собой абстрактные понятия, а типы *wire* и *reg* непосредственно отражаются в аппаратуре.

О правилах назначения цепей и регистров. Общий принцип состоит в том, что характеристики всех портов должны быть объявлены. При этом могут быть исключения.

Входы внутренних модулей всегда должны иметь тип *wire* (цепь), так как они управляются внешними сигналами. Входы промежуточных модулей могут быть как цепью, так и регистром. Входы внешнего по отношению ко всем другим модулям имеют тип *reg* (регистр).

Выходы внутренних модулей могут иметь тип как *reg*, так и *wire*. Это зависит от введенных понятий и от применяемых в модуле конструкций. Выходы внешнего модуля должны быть типа *wire*, поскольку управляются внутренними модулями (здесь также могут быть исключения).

Каждый модуль кроме входов и выходов может иметь также внутренние связи. Они не являются ни входами, ни выходами и должны быть объявлены с помощью ключевого слова *wire*.

Двунаправленный порт всегда имеет тип *wire*.

2.4. Векторы и массивы

Векторы. Как цепи, так и регистры могут иметь произвольную разрядность. В этом случае есть возможность объявить их как векторы.

Общий формат объявления векторов имеет следующий вид:

reg/*wire*[N_2:N_1]<имя порта1>,<имя порта2>…;

где N_2 - старший (или значимый) бит; N_1 - младший бит.

Например: *reg*[3:0]*q*; // выход 4-битного (4-разрядного) регистра. Здесь порты *q*[3], *q*[2], *q*[1] и *q*[0] являются отдельными элементами вектора; причем *q*[3] - старший бит (нумерация от большего к меньшему).

Если указано: *reg*[0:7]*a*; то старшим является бит *a*[0] (нумерация от меньшего к большему).

Смешивание двух способов нумерации вносит путаницу, и этого следует избегать. Так как способ от большего к меньшему более привычен, рекомендуется использовать его.

С каждым отдельным элементом вектора (портом) можно работать и выполнять различные операции независимо от других элементов.

Примеры записи векторов:

wire [31:0]*data*; // 32-разрядная цепь

reg [3:0]*a*,*b*; // здесь объявлены 4-разрядные переменные *a* и *b*.

Последняя запись эквивалентна двум следующим:

reg [3:0]*a*;

reg [3:0]*b*;

Если в характеристике *reg*/*wire* порты объявлены как векторы, то и в назначении *output* / *input* они должны быть объявлены как векторы (формат аналогичен).

Например:

output [3:0]*q*;

Перечисляемые в выходах или во входах отдельные элементы вектора указываются в порядке от старшего бита - к младшему.

Например:

output test_out[6], *test_out*[3], *test_out*[0];

Присваивание векторов. Лучший стиль - присваивать векторы только равной длины, поскольку разные моделирующие программы, а также разные версии программ неодинаково реагируют на разную длину векторов.

Если присваиваются векторы разной длины, то Verilog либо дополняет нулями, либо обрубает правую часть оператора присваивания (=) в зависимости от того, меньше или больше разрядов в правой части, чем в левой. Этот процесс происходит без диагностики.

Например:

```
reg [7:0]a;
reg [3:0]c;
…….
a = c;   // переменная c дополняется четырьмя нулями в старшие
         // разряды; неявное дополнение; это некорректная запись
a = {4'h0,c};   // применение конкатенации (см. главу 7);
                // явное дополнение нулями; корректная запись
```

Приведенные выше комментарии относятся и к случаю сравнения векторов.

Массивы. Часть цифрового устройства или устройство в целом может иметь матричную структуру, как, например, в схемах памяти или в программируемых логических матрицах (ПЛМ). Матричная структура характеризуется строками и столбцами. Verilog-моделью такого устройства являются *n* однотипных векторов (или массивы). Массивами могут быть объявлены данные типа *reg*. Данные типа *wire* массивами быть не могут.

Общий формат объявления массива имеет следующий вид:

reg [N_2:N_1] <имя порта> [M_2:M_1];

где N_2 и N_1 означают то же, что и у векторов, т.е. разрядность данных (или количество столбцов); M_2 и M_1 означают разрядность адреса (или количество строк).

В отличие от векторов может быть указано только одно имя порта.

Так же, как и у векторов, левая цифра в скобках означает старший бит.

Например:

```
reg [3:0] a [31:0];          // 32 4-разрядных элемента,
reg c [7:0];                 // 8 1-разрядных элементов,
integer [3:0] out [31:0]; // 32 4-битных элемента.
```

Обращение к элементу массива имеет такой же синтаксис, как и обращение к разряду вектора. Таким образом, *data* [*i*] может быть *i*-м разрядом в векторе *data* или *i*-м элементом в массиве *data*, в зависимости от того, как объявлен порт *data*.

Для обращения к разряду элемента массива этот элемент сначала надо записать в рабочую переменную.

Например:

```
reg [15:0] array [7:0];
reg [15:0] temp; // temp - рабочая переменная
…….
temp = array[3]; // переменной temp присваивается значение
                 // элемента 3 массива array
…….
…….temp[7]…… // конструкция текста работает с разрядом 7
              // элемента 3 массива
```

Глава 3. Базовые блоки Verilog

В языке Verilog применяются два базовых блока: блок *always* и блок *initial* (инициализации). Они играют важную роль в Verilog-проекте: подавляющее большинство модулей имеет в своем составе хотя бы один из этих блоков.

3.1. Always-блок

Блок *always* означает, что действие, указанное в нем, выполняется до тех пор, пока процесс моделирования (или симуляции) не будет остановлен директивами *$finish* или *$stop*.

Блок *always* применяется только с данными типа *reg*. Отсюда следует и обратное положение. Если требуется использовать блок *always*, например, для определения значений выходов сумматора *s* и *p*, то их надо объявить как данные типа *reg*.

В Verilog используется несколько конструкций с ключевым словом *always*.

Наиболее частое употребление имеет конструкция с так называемым списком чувствительности.

Конструкция имеет следующий формат:

always @ (<имя переменной 1> *or* <имя переменной 2> *or*…)

Эта запись означает, что указанные за ней внутри блока действия (они, как правило, определяются с помощью различных операторов, в том числе и оператора присваивания) выполняются всегда при изменении хотя бы одной переменной (здесь - операнда). Список операндов для блока *always* называется списком чувствительности (или списком возбуждения). Вместо *or* в конструкции можно использовать запятую. Необходимо обратить внимание на то, что в конце данной конструкции точка с запятой не указывается. В качестве операндов могут использоваться векторы.

Примеры записи рассмотренной конструкции:

always @ (*in*1 *or in*2 *or in*3 *or in*4)

always @ (*data*)

always @ (*posedge clk*)

Последняя запись применяется для описания устройств с динамическим управлением записью. При управлении переходом 0→1 применяется ключевое слово ***posedge***, при управлении переходом 1→0 - ключевое слово ***negedge***. Здесь речь идет об управлении устройством фронтом (или спадом) тактового сигнала. Если у одной переменной ставится *posedge* (или *negedge*), то и у всех переменных из списка чувствительности должно стоять *posedge* (или *negedge*).

Например:

always @ (*posedge clk, negedge reset*)

При вычислениях, которые должны производиться при изменении любой из используемых в блоке переменных, Verilog позволяет не перечислять имена списка, достаточно указать символ обобщенного списка чувствительности:

*always @**

Если в вычислениях участвует одна переменная, то блок *always* применяется без списка чувствительности.

Например:

always # 10 *clk =~ clk;*

always # 20 *i* = *i* + 1;

Эти записи означают, что каждые 10 заданных единиц времени (см. главу 9) переменная *clk* меняет свое значение на противоположное, а каждые 20 единиц времени *i* увеличивается на единицу во все время моделирования. Символ # применяется для обозначения задержки.

Внутри блока *always* можно делать следующие записи:

```
@(negedge clk) a = b;   // a становится равным b при каждом
                        // переходе clk из 1 в 0
a = @(negedge clk)b;    // смысл записи аналогичен предыдущему
```

3.2. Блок инициализации (initial)

С помощью блока *initial* осуществляется собственно моделирование устройства. Блок инициализации включает в себя оператор или группу операторов, которые будут выполняться с момента старта моделирования. В первом случае конструкция блока имеет вид:

initial<оператор>;

а во втором:

```
initial begin
    <группа операторов>;
end
```

Ключевые слова ***begin*** и ***end*** называются операторными скобками. Группы операторов всегда заключаются в операторные скобки. Так как большие проекты, как правило, редактируются и дополняются, то предпочтительнее использовать второй вариант. Операторные скобки применяются также в блоке *always* и в других конструкциях, в которых используется больше одного оператора.

Блок *initial* применяется в основном для явного назначения переменных (процедурное назначение), поэтому он используется главным образом для задания входных воздействий (в том числе начальных значений), т.е. при формировании тестов. Следует помнить, что переменные в этом случае (имеется в виду процедурное назначение) должны быть объявлены как *reg*.

Все блоки *initial*, присутствующие в Verilog-проекте, выполняются одновременно, один раз в начале моделирования.

Примеры применения блока.

1.
```
initial in1 = 1'b0;   // задание начального значения in1
 initial begin
  # 10 in1 = 1'b1;  // выполняется в момент времени t = 10
                    // заданных временных единиц
  # 20 in1 = 1'b0;  // выполняется в момент t = 30
 end
```

2.
```
initial begin
   a   = 1'b0;      // задание начальных значений
   b   = 1'b0;      // входных сигналов для сумматора
   p_in = 1'b0;
end
initial begin
  # 10 b = 1'b1;  // выполняется в момент t = 10; b принимает
                  // значение 1,остальные сигналы не меняются
  # 10 a = 1'b1;
  # 10 b = 1'b0;    // выполняется в момент t = 30
p_in = 1'b1;    // выполняется в момент t = 30 и т.д.
```

end

3. Если некоторая величина объявлена вектором, то она может быть явно назначена с помощью блока *initial* следующим образом:

```
reg [3:0] data;
initial data = 4'b0110;  // запись 0110 может называться словом
```

В этом примере *data*[3] = 0, *data*[2] = 1, *data*[1] = 1 и *data*[0] = 0. Можно произвести частичное назначение вектора, например:

```
initial data[2:1] = 2'b11;
```

В блок *initial* могут входить системные директивы. Блок можно включать в модули всех уровней, но с учетом задач проектирования. На этом тезисе остановимся подробнее. Так как блок *initial* является несинтезируемой конструкцией (см. главу 13), то программа синтезатора проигнорирует его, как и все остальные несинтезируемые конструкции. Например, если в модуле, описывающем работу триггера, записать начальное значение выхода:

```
initial q = 1'b0;
```

то алгоритм функционирования может показать правильные результаты, а синтезируемое устройство - нет. Поэтому для того, чтобы при включении устройства в нужных узлах были фиксированные значения, необходимо иметь входы начальной установки или предварительной установки, но не использовать для этих целей блок *initial.*

Завершая главу, еще раз отметим, что блок *initial* служит для описания действий, которые выполняются один раз (при запуске моделирования), а блок *always* обозначает действия, которые выполняются постоянно.

Например, запись

```
always # 10 y = a + b;
```

означает, что вычисления будут производиться каждые 10 заданных временных единиц, а запись

```
initial # 10 y = a + b;
```

означает, что действие будет выполнено один раз в момент времени $t = 10$.

Блоков *initial* и *always* может быть сколько угодно. Порядок выполнения определяется только по заданному времени.

Глава 4. Модули Verilog

В Verilog используется модульный принцип описания проекта. Это значит, что различные части устройства или алгоритмы их функционирования описываются по типу модуля, т.е. описания носят автономный, законченный характер. Модуль является основной единицей в Verilog. Он представляет собой некий логический объект, который обычно реализуется в аппаратуре. Например, модуль может быть простым вентилем или 32-разрядным счетчиком, или подсистемой памяти, а также вычислительной системой или вычислительной сетью.

4.1. Иерархия Verilog-проекта

Всякий Verilog-проект носит иерархический (или многоуровневый) характер. То есть он состоит из одного или более модулей, любой из которых может содержать другие модули (рис.4.1).

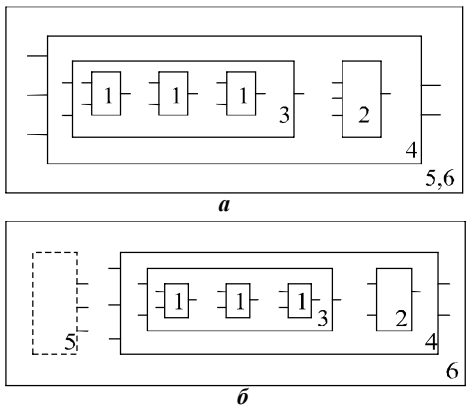


Рис.4.1. Многоуровневый характер Verilog-проекта: 1, 2 - базовые модули; 3 - субмодуль; 4 - головной модуль; 5 - модуль *TestBench*; 6 - модуль-оболочка: *а* - модуль *TestBench* в качестве оболочки головного модуля; *б* - модуль *TestBench* - составная часть модуля оболочки

В зависимости от иерархии различают следующие модули.

1. **Головной или корневой модуль** (корень дерева). Этот тип модуля не входит ни в один другой модуль описания устройства (модуль 4 на рис.4.1,*а*,*б*); он является внешним (или верхним) по отношению к другим модулям (субмодулям или базовым модулям).

2. **Субмодули** являются внутренними (или модулями нижнего уровня) по отношению к внешним (или модулям верхних уровней). В то же время они сами являются внешними по отношению к модулям нижних уровней, т.е. к тем, которые они включают в себя. Эти модули могут называться также узлами дерева. На рис.4.1,*а* и 4.1,*б* модуль 3 является субмодулем. При этом может быть несколько уровней вложенности. Количество уровней не ограничено.

3. **Базовые модули** (или листья дерева) не включают в себя никаких других модулей. На рис.4.1 базовыми являются модули 1 и 2.

4. **Тестовый модуль** (или модуль **TestBench**) содержит описание тестовых входных воздействий для проверки функционирования проектируемого устройства или его алгоритма. Модуль TestBench (на рис.4.1,*б* это модуль 5, он показан пунктиром) не является

частью устройства, но входит в структуру Verilog-проекта. При этом с точки зрения иерархии он может быть оболочкой для головного модуля (рис.4.1,*а*) или находиться внутри модуля-оболочки (рис.4.1,*б*).

5. **Модуль-оболочка** (модуль 6 на рис.4.1,*б*) объединяет тестовый и головной модули. Как и тестовый модуль, оболочка не является устройством или его частью, а служит лишь для удобства оформления и отладки проекта.

Если модули представляют собой одно и то же устройство и имеют одинаковое описание, то такие модули по отношению к внешним называются **экземплярами** (на рис.4.1 это модули 1). Например, в счетчике или регистре может быть использовано несколько однотипных триггерных подсистем. Достаточно один раз описать такую подсистему в модуле - и затем можно неоднократно ссылаться на это описание. В частном случае во внешнем модуле может быть один экземпляр внутреннего. Например, модуль 2 в модуле 4.

Описание Verilog-проекта начинается либо с базовых модулей, которые затем объе-

```
module <имя модуля> (<список имен выходов и входов>);
        <описание выходов и входов>;
        <описание локальных переменных>;
        <элемент модуля>;      }
        ……                    } ядро тела модуля
        <элемент модуля>;     }
endmodule
```

Рис.4.2. Типовая структура модуля

диняются в целый проект, либо с модуля верхнего уровня, к которому добавляются детали проекта. Первый подход называется ***bottom-up*** (снизу вверх), второй подход - ***topdown*** (сверху вниз).

Хотя Verilog допускает проектирование "леса", т.е. множества деревьев, иерархия будет понятной, если используется только одно дерево. Это означает, что структура Verilog-проекта имеет один корень, который может содержать только порты и входящие в него модули (см. рис.4.1).

При описании устройства должно быть соответствие иерархии проекта иерархии физического разбиения устройства. Это упрощает трансляцию модели в физическую структуру.

4.2. Описание модуля и его структура

Каждый модуль начинается с ключевого слова ***module***, имеет заголовок (имя) и список портов. В структуре модуля должно быть предусмотрено описание средств, с помощью которых он взаимодействует со своим окружением. Все выходы и входы модуля должны быть описаны.

На рис.4.2 представлена типовая структура модуля, которая характерна для головного модуля, а также для базовых и субмодулей. Структуры тестового модуля и модуля-оболочки имеют свои особенности.

Имя у модуля должно быть обязательно, так как по нему производится ссылка на данный модуль. В списке портов указываются только внешние выводы (выходы и входы).

В описании выходов и входов необходимо указать назначение портов (ключевые слова *output* и *input*) и характер поведения (или тип) сигналов (ключевые слова *reg* и *wire*). Требуется соблюдать порядок перечисления портов, указанный в главе 2.

Описание выходов и входов допустимо производить в списке с ключевым словом *module*.

Например:

module psm (*output reg [1:0] sum, input a, input b*);

Описание локальных переменных включает в себя описание внутренних связей модуля, параметров, а также вспомогательных переменных, например, для подсчета числа циклов и др.

Все используемые в модуле порты, параметры и переменные должны быть объявлены. Хотя, как и во всяком правиле, здесь есть исключения. Например, у цепочки трех последовательно включенных инверторов достаточно объявить имена портов начала и окончания цепочки, не объявляя промежуточные узлы. Пусть это будут *in_inv* и *out_inv*. Тогда

out_inv =(~(~(~in_inv))) ;

Ядро тела модуля представлено элементами модуля. Элементы могут быть различного типа. Наиболее распространенными из них являются **непрерывное присваивание** (или назначение), **функциональные** и **структурные элементы**.

В зависимости от иерархии модули могут иметь разный состав. Базовые модули имеют в составе ядра непрерывное присваивание и функциональные элементы. Субмодули имеют в составе ядра структурные элементы и один (любой) или оба оставшихся элемента, хотя в частном случае могут состоять из одних структурных элементов. В головном модуле избегают непрерывного присваивания (в качестве элементов ядра модуля).

Непрерывное присваивание - это лаконичный способ описания комбинационной части модуля. Оно выполняется с помощью конструкции *assign*.

4.2.1. Функциональные элементы модуля

Функциональные элементы определяют способ описания функциональной сущности базового модуля. Например, один из способов описания состоит в использовании примитивов языка Verilog, таких, как *and*, *or*, *nor* и т.д. (глава 5). Хотя проектирование на вентильном уровне может быть очень эффективным, предпочтительным является другой способ, который поддерживает Verilog. Он заключается в применении функциональных конструкций более высокого уровня. Это блоки Verilog, которым предшествуют ключевые слова *initial* или *always* (базовые блоки, которые рассматривались ранее). Именно они часто имеются в виду, когда говорят о функциональных элементах.

Все функциональные элементы в модуле выполняются одновременно. Каждый функциональный элемент может быть простым или составным. Простой функциональный элемент представляет собой одну конструкцию. Составной функциональный элемент образован группой конструкций, заключенных в операторные скобки *begin* и *end*. Относительно большим блокам может быть присвоено имя:

```
begin: <имя блока>
……
end
```

Поименованные блоки могут иметь внутри себя описание локальных переменных.

Описание модуля заканчивается ключевым словом ***endmodule***.

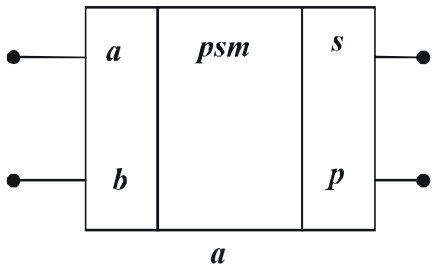
Рассмотрим пример построения базового модуля для описания функционирования полусумматора [3, 4] (рис.4.3).

```
/* модуль предназначен для вычисления функций суммы и переноса у полусумматора
*/
    module psm (p, s, a, b);   // см. рис.4.3,a
        output p, s;
        input a, b;
        wire a, b;
        reg p,s;
        always @ (a, b)
            begin
                s=a&~b|b&~a;
                p=a&b;
```

```
    end
endmodule
```

В этом примере перед первым ключевым словом приводится комментарий, в котором указывается назначение модуля. Затем записывается ключевое слово *module* вместе с именем модуля *psm*, по которому может осуществляться ссылка на модуль. В скобках перечислены выходы (*p*, *s*) и входы (*a*, *b*); строка заканчивается точкой с запятой.

Далее указывается назначение портов (ключевые слова *output* и *input*) и типы сигналов: входы объявлены как цепи, а выходы - как регистры (поскольку применяется блок *always*).



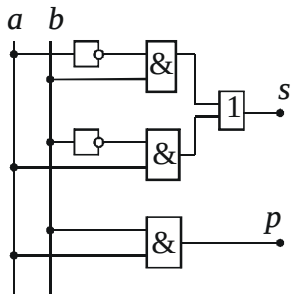
а

a	*b*	*s*	*p*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$s = \bar{a}b \vee a\bar{b}$

$p = ab$

б



в

Рис.4.3. Полусумматор: *а* - условное обозначение;
б - таблица истинности и алгебраические выражения;
в - логическая схема

С помощью блока *always* происходит вычисление функций полусумматора по формулам (рис.4.3,*б*). Так как в блоке записаны две конструкции для вычисления суммы и переноса, то используются операторные скобки. Имя блоку ввиду его небольшого размера не присвоено. Блок *always* постоянно следит за своими операндами, и когда хотя бы один из них изменяется, происходит вычисление значения функций выходов.

Модуль завершается ключевым словом *endmodule*.

Следует обратить внимание на структуру записей: каждая следующая группа записей делается через 2 - 3 пробела от предыдущей.

Рассмотренный вариант описания функционирования полусумматора является далеко не единственным. Verilog представляет возможность составить описание несколькими способами даже для такого простого устройства. В последующих главах это будет показано на различных примерах.

Функциональные элементы составляют основу, как правило, базовых модулей, но могут применяться и в субмодулях.

4.2.2. Структурные элементы модуля

Структурный элемент представляет собой ссылку на модуль более низкого уровня в текущем модуле. Другими словами, структурный элемент является описанием экземпляра внутреннего модуля во внешнем. В одном модуле может быть несколько структурных

элементов, которые используют описание как одного и того же внутреннего модуля, так и разных внутренних модулей. Все они должны иметь разные имена.

Конструкция структурного элемента имеет следующий вид:

<имя внутреннего модуля> <имя экземпляра внутреннего модуля в текущем> (список имен портов);

Здесь первым указывается имя внутреннего модуля, на который производится ссылка. Имена модуля и экземпляра могут совпадать, но это нежелательно. Например, при ссылке на модуль с описанием функционирования полусумматора из рассмотренного примера используется его имя *psm*, а имя экземпляра модуля выбирается, исходя из приведенных ранее правил, *psm*1.

С помощью списков портов осуществляется идентификация (или присоединение) портов внутреннего модуля в текущем модуле.

Присоединение реализуется двумя способами:

1) используя заданный во внутреннем модуле порядок перечисления портов (позиционное соответствие);

2) используя прямое назначение портов.

В первом случае должно быть строгое соответствие последовательностей перечисления портов во внутреннем модуле и в текущем. Неиспользованные порты не указываются и обозначаются запятыми. Во втором случае используется следующая конструкция для списка портов:

(.<имя порта во внутреннем модуле> (<имя порта в текущем модуле>),...));

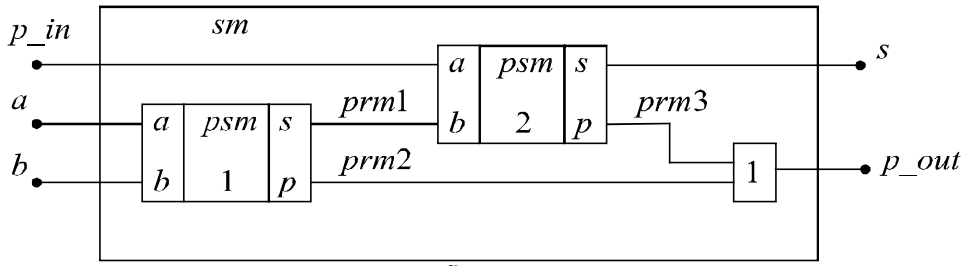
Здесь конкретно указывается соответствие портов внутреннего модуля и текущего независимо от последовательности их перечисления, т.е. порядок перечисления портов уже не имеет значения.

Рассмотрим использование структурных элементов на примере построения модуля для описания функционирования одноразрядного сумматора (рис.4.4,*а*). При этом воспользуемся уже готовым описанием функционирования полусумматора.

```
/* модуль предназначен для вычисления функций суммы и переноса у одноразрядного
сумматора */
module sm (p_out, s, a, b, p_in);
    output p_out, s;
    input a, b, p_in;
    wire prm1, prm2, prm3;
    reg p_out;
            psm psm1 (prm2, prm1, a, b);
            psm psm2 (.p(prm3), .s(s), .a(p_in), .b(prm1));
            always @ (prm2, prm3)
                    p_out = prm2 | prm3;
endmodule    // sm
```

Первые три строки нам уже знакомы, и добавить здесь нечего. Затем объявляются внутренние переменные; как уже указывалось, им присваивается тип *wire*. Выход *p_out* объявляется как регистр, так как его значение вычисляется в блоке *always*.

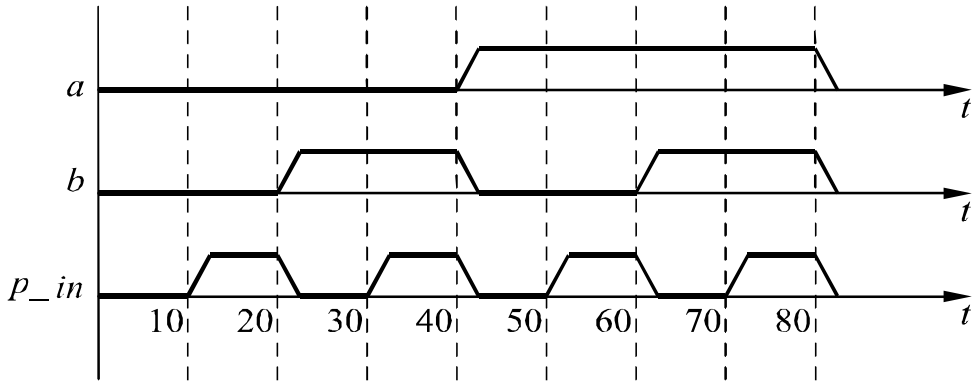
В модуле используются два структурных элемента. Это два экземпляра одного и того же модуля, описывающего работу полусумматора, который является по отношению к текущему модулю (сумматору) внутренним (или модулем низшего уровня). Первый структурный элемент - экземпляр полусумматора с именем *psm*1. В его списке портов используется позиционное соответствие, т.е. заданный в модуле *psm* порядок перечисления. Выходы *p* и *s* модуля *psm* соответствуют внутренним переменным *prm*2 и *prm*1 модуля *sm*, а входы *a* и *b* модуля *psm* - входам *a* и *b* модуля *sm*.



а

a	b	p_in	s	p_out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

б



в

Рис.4.4. Одноразрядный сумматор: *а* - функциональная схема (*prm*1, *prm*2, *prm*3 - внутренние цепи); *б* - таблица истинности; *в* - временные диаграммы

Второй структурный элемент - экземпляр модуля *psm* с именем *psm*2. В списке портов используется прямое назначение, т.е. явно указано, какому порту внутреннего модуля какой порт текущего модуля соответствует.

Если ссылки производятся на один и тот же модуль, как в рассматриваемом примере, то допустимо применять следующую конструкцию:

*psm psm*1 (*prm2*, *prm1*, *a*, *b*),
*psm*2 (.*p*(*prm*3), .*s*(*s*), .*a*(*p_in*), .*b*(*prm*1));

В блоке *always* при каждом изменении *prm*2 или *prm*3 (или обеих переменных сразу) происходит вычисление *p_out*.

При передаче портов Verilog предоставляет пользователю разные возможности. Можно передавать:

1) порты с использованием знаков операций. Например:

*rs_trig rs*1(*q*, ~*clk*, …)

То есть в модуль *rs_trig* из внешнего модуля передается инверсное значение сигнала *clk*;

2) векторы. Например, если у полусумматора выход объявлен как двухбитная переменная:

module psm (*sum*, *a*, *b*);
output [1:0] *sum*; // здесь *s* = *sum*[1], а *p* = *sum*[0]

то в модуле *sm* при описании экземпляра модуля порты можно указать следующим образом:

*psm psm*1 ({*prm*1, *prm*2}, *a*, *b*);

Здесь *prm*1 будет соответствовать значение *sum*[1] (т.е. *s*), а *prm*2 - *sum*[0] (т.е. *p*).

Векторы можно передавать не полностью, а частично. Например, из вектора *out*[7:0] можно передать четыре переменные: *out* [5:2];

3) численные значения. Например, структурный элемент *psm*1 в модуле *sm* может выглядеть следующим образом:

*psm psm*1 (*prm*2, *prm*1, 0, *b*);

т.е. здесь входная величина *a* принимает значение 0. Данный прием применяется довольно часто, например, при задании значений узлов питания и земли, начальных значений выходов триггеров в процессе отладки проекта и т.д.

Таким образом, применение структурных элементов позволяет осуществить идею иерархии проекта, а возможности Verilog предоставляют различные способы ее реализации, что дает разработчику некоторую свободу действий.

4.3. Тестовый модуль TestBench и его особенности

Как уже указывалось, в иерархии проекта тестовый модуль может занимать разные положения. Во-первых, наряду с выполнением основной функции - описанием тестов *TestBench* может являться оболочкой проекта (см. рис.4.1,*а*). Во-вторых, вместе с головным модулем он может входить в состав модуля-оболочки (см. рис.4.1,*б*).

Рассмотрим описание тестового модуля для первого варианта. Его типовая структура изображена на рис.4.5,*а*.

Первая строка содержит директиву масштаба модельного времени, которая начинается с обратной кавычки: **`*timescale***. Она относится к директивам компиляции, которые в данном пособии не рассматриваются.

С этой директивой указываются временные параметры:

*N*1 - масштабный коэффициент для чисел, следующих за символом # (задержка);

*N*2 - точность расчета.

Оба параметра указываются в нано- или пикосекундах и могут принимать значения только 1, 10 или 100. По умолчанию назначаются *ps*.

Например:

`*timescale* 1*ns*/1*ps* // *ns*, *ps* - англоязычные сокращения

Строка может отсутствовать, если симуляция или моделирование не производится. В некоторых версиях моделирующих программ такая строка должна содержаться во всех файлах перед модулем.

В следующей строке тестового модуля указывается ключевое слово *module* и имя модуля с обязательной приставкой *tb* (*TestBench*), указывающей на то, что это тестовый модуль проекта. В этом случае список портов отсутствует.

Далее указывается характер поведения выходов и входов (*reg* или *wire*).

В описании локальных переменных чаще всего объявляются используемые в данном модуле параметры.

```
`timescale <N1> [ns или ps] / <N2> [ns или ps]
module <имя модуля>_tb;
        reg <список имен выходов и входов>;
        wire <список имен выходов и входов>;
        <описание локальных переменных>;
        <имя тестируемого модуля> <имя экземпляра модуля>
        (<список имен портов>);
                initial begin
                        <начальные установки входных сигналов>
                end
                initial begin
                        <описание поведения входных сигналов во времени>
                end
                <одна из конструкции $finish>
endmodule
```

а

```
module <имя модуля>_tb (список имен портов);
        output <список имен портов>;
        reg <список имен портов>;
        <описание локальных переменных>
                initial begin
                <начальные установки входных сигналов>
                end
                initial begin
                <описание поведения входных сигналов во времени>
                end
                <одна из конструкций $finish>
endmodule
```

б

Рис.4.5. Типовая структура тестового модуля *TestBench*: *а* - тестовый модуль является оболочкой для головного; *б* - тестовый модуль находится внутри модуля-оболочки

Следующая строка представляет собой структурный элемент, она вызывает головной (тестируемый) модуль. При этом в списке портов устанавливается соответствие между портами тестового и тестируемого модулей. Модуль *TestBench* в общем случае может иметь отличные от головного имена портов. Дело в том, что при разработке проекта какого-либо устройства обычно рассматривается несколько вариантов реализации. Например, предполагаемое устройство имеет входы с именем *data*, тогда в анализируемых вариантах они могут быть названы *data*0, *data*1 и т.д. При отладке этих вариантов их поочередно подключают к тестовому модулю, в котором может использоваться одно имя *data*.

Собственно моделирование осуществляется с помощью конструкций, определяемых ключевыми словами *initial begin…end*. Конструкции содержат описания начальных значений и дальнейшего поведения входных сигналов во времени. Эти описания можно не разделять и размещать в одном блоке *initial*, однако рекомендуется использовать два блока для удобства восприятия и редактирования модуля. Следует отметить, что тестовый модуль без блока *initial* не работает.

В отдельных случаях модуль *TestBench* может содержать операторы обработки функции выхода, которые включаются в блок *initial*.

Далее указывается одна из конструкций системной директивы *$finish* (см. главу 10), которая завершает процесс моделирования (симуляции). И наконец, ключевое слово *endmodule* указывает на окончание описания тестового модуля.

Рассмотрим пример построения тестового модуля для одноразрядного сумматора:

```
`timescale 1ns/1ps
    module test_sm_tb;
```

```
        wire p_out, s;
        reg a, b, p_in;
        sm sm1 (p_out, s, a, b, p_in);
        initial begin
            a = 1'b0;
            b = 1'b0;
            p_in = 1'b0;
    end
    initial begin
        # 10 p_in = 1'b1;
        # 10 b = 1'b1;
            p_in = 1'b0;
        # 10 p_in = 1'b1;
        # 10 a = 1'b1;
            b = 1'b0;
            p_in = 1'b0;
        # 10 p_in = 1'b1;
        # 10 b = 1'b1;
            p_in = 1'b0;
        # 10 p_in = 1'b1;
        # 10 $finish
    end
endmodule
```

В этом примере масштабный коэффициент выбран равным 1 нс, а точность - равной 1 пс. Модулю присвоено имя *test_sm_tb*. Выходные порты *p_out* и *s* объявлены цепями, так как они управляются внутренними модулями. Входы *a*, *b* и *p_in* объявлены регистрами, так как для них используется явное назначение в блоках *initial*.

Затем следует структурный элемент, в котором происходит обращение к тестируемому модулю, в данном случае - модулю *sm*. В списке портов используется принятый в модуле *sm* порядок перечисления.

Далее следуют два блока *initial*. В первом задаются начальные значения входных сигналов, во втором - их поведение во времени в соответствии с таблицей истинности сумматора (см. рис.4.4,*б*). Так как физические задержки в данном случае не учитываются, то время для одного такта (10 нс) выбрано условно, только из соображений удобства визуального контроля результатов симуляции.

Описание тестового модуля заканчивается системной директивой *$finish* и ключевым словом *endmodule*.

В рассмотренном примере вместо второго блока *initial* можно применить одну из конструкций блока *always*, учитывая периодический характер изменения входных воздействий (рис.4.4,*в*). То есть поведение входных сигналов можно описать следующим образом:

```
    always # 40 a = ~ a;
    always # 20 b = ~ b;
    always # 10 p_in = ~ p_in;
```

Здесь указано, что величина *a* должна изменять свое значение на противоположное каждые 40 временных единиц, *b* - каждые 20 единиц, а *p_in* - каждые 10 единиц.

Рассмотренный тестовый модуль завершает построение Verilog-проекта для устройства одноразрядного сумматора, который с точки зрения иерархии включает в себя: базовый модуль *psm*, головной модуль *sm* и тестовый модуль *test_sm_tb*, являющийся в данном случае и модулем-оболочкой.

Отвлечемся от рассмотренного примера и приведем несколько комментариев. Пусть имеется 3-битный вектор *in*, и необходимо промоделировать устройство при всех значениях этого вектора.

В тестовом модуле изменения *in* можно задать следующим образом:

initial in=0;

always # 10 *in* = *in*+1;

Или использовать запись, тождественную последней:

always # 10 *in* ++;

При задании модельного времени не должно быть переполнения.

Если разработчику необходимо провести несколько экспериментов (расчетов) с одним описанием, но с разной длительностью такта, то можно воспользоваться параметром. То есть в группу описания портов добавляется конструкция с ключевым словом *parameter* и с подходящим именем, например, *t*:

parameter t = 10;

Тогда строка из блока *initial* будет выглядеть, например, следующим образом:

 # t p_in = 1′*b*1;

При изменении длительности такта можно, не меняя весь блок *initial*, изменить лишь численное значение параметра.

Типовая структура тестового модуля для второго варианта, изображенного на рис.4.1,*б*, приведена на рис.4.5,*б*.

Сравнение структур тестового модуля в первом и втором вариантах показывает, что их описания различаются в основном до блока *initial*, а далее совершенно одинаковы.

В первой строке указывается ключевое слово *module* и имя модуля с обязательной приставкой *tb*. В качестве списка имен портов, как и в последующих двух строках, используются имена входов тестируемого модуля.

Далее указывается ключевое слово *output* с тем же списком входов. Дело в том, что результатом работы тестового модуля является присвоение различных значений входным выводам. Поэтому входы головного модуля являются выходами для тестового.

Третья строка указывает на то, что всем переменным должно быть сделано процедурное назначение, так как они используются в блоке *initial*.

Рассмотрим пример описания тестового модуля для одноразрядного сумматора по второму варианту:

```
module test_sm_tb (a, b, p_in);
    output a, b, p_in;
    reg a, b, p_in;
        initial begin
        <см. предыдущий пример>
        end
endmodule    // test_sm_tb
```

В этом примере в качестве списка портов в первых трех строках фигурируют входы сумматора *a*, *b* и *p_in*.

Как видно, использование тестового модуля в качестве внутреннего делает его описание проще.

4.4. Модуль-оболочка для Verilog-проекта

Применение модуля-оболочки предоставляет разработчику дополнительные возможности в оформлении проекта. Типовая структура модуля-оболочки приведена на рис.4.6.

К первой строке комментарии уже были даны. Вторая строка содержит ключевое слово *module* и имя модуля-оболочки без указания списка портов. Третья строка содержит список имен выходов (здесь имеются в виду выходы и входы головного модуля). Наличие этой записи дает возможность наблюдать поведение выходов в консольной части экрана.

Если такой необходимости нет, то данная строка может отсутствовать. В этом случае при вызове головного модуля в списке портов выходы можно не указывать.

```
`timescale <N1> [ns или ps] / <N2> [ns или ps]
 module <имя модуля>;
        wire <список имен выходов>;
        wire <список имен входов>;
        <описание локальных переменных>;
        <имя тестового модуля> <имя экземпляра модуля> (<список имен входов>);
        <имя головного модуля> <имя экземпляра модуля> (<список имен выходов и входов>);
endmodule
```

Рис.4.6. Типовая структура модуля-оболочки

Всем портам в модуле-оболочке присваивается тип *wire*, так как они управляются внутренними (по отношению к данному) модулями.

Описание локальных переменных включает в себя описание внутренних связей модуля-оболочки (рис.4.7), которые затем должны быть указаны в списках портов структурных элементов. Наличие локальных переменных в этом модуле необязательно. В частности, в рассматриваемом далее примере они отсутствуют.

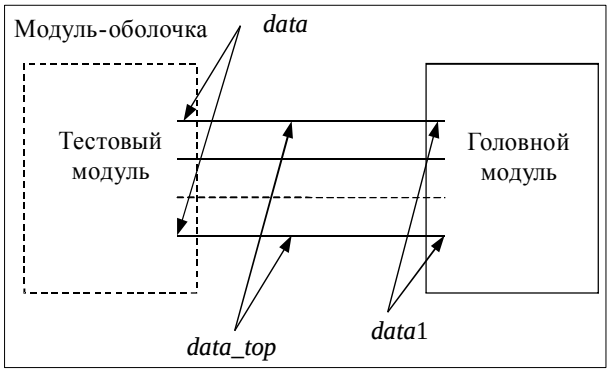


Рис.4.7. Внутренние связи модуля-оболочки

Затем вызывается тестовый модуль со списком входов и головной модуль со списком выходов и входов.

Заканчивается описание модуля-оболочки ключевым словом *endmodule*.

Для описания устройства одноразрядного сумматора модуль-оболочка может выглядеть следующим образом:

```
`timescale 1ns/1ps
module top;
        wire p_out, s;
        wire a, b, p_in;
        test_sm_tb tb (a, b, p_in);
        sm uut (.p_out(p_out), .s(s), .a(a), .b(b), .p_in(p_in));
endmodule
```

Жестких правил для выбора варианта оформления проекта нет. Поэтому каждый разработчик решает сам, какой вариант ему выбрать: с одним тестовым модулем или с тестовым модулем и модулем-оболочкой.

Глава 5. Встроенные примитивы языка Verilog

Язык Verilog позволяет описать алгоритм функционирования устройства и его конструкцию. Конструкцию устройства (или его частей) можно описать с помощью встроенных примитивов языка. Встроенным примитивом является Verilog-модель логического элемента (ЛЭ). Это, например, логические элементы (или вентили): *or*, *xor*, *not* и др. Встроенные примитивы входят в основном в состав базовых модулей, но могут входить и в субмодули. Проектирование на вентильном уровне может быть очень эффективным с точки зрения оценки конечного результата.

5.1. Описание встроенных примитивов

При описании встроенного примитива задаются: ключевое слово, имя, выходы и входы. При этом используется следующая конструкция:

<ключевое слово> <имя> (<список имен выходов и входов через запятую>);

В конструкции применяются ключевые слова:

and - логический элемент И;

nand - логический элемент И-НЕ;

or - логический элемент ИЛИ;

nor - логический элемент ИЛИ-НЕ;

xor - логический элемент исключающее ИЛИ;

xnor - логический элемент исключающее ИЛИ-НЕ;

buf - буферный элемент;

not - логический элемент НЕ.

Все элементы, за исключением *buf* и *not*, имеют один выход и два и более входов, причем ограничений на количество входов в разумных пределах нет. Элемент *buf* имеет один вход и несколько выходов.

Имена задаются в соответствии с указанными в главе 1 правилами. В некоторых версиях моделирующих программ имя указывать не обязательно.

В описании портов, как всегда, сначала указываются выходы, затем - входы. Как выходы, так и входы указываются в алфавитном порядке. Векторы указываются только в виде своих элементов в порядке от старшего бита к младшему.

Например:

*and la*1(*out*, *in*1, *in*2, *in*3);

*xor la*2(*out_xor*, *in*[1], *in*[0]);

*not la*3(*out_la*3, *data*);

Для примитивов с одинаковым ключевым словом допустима следующая конструкция записи:

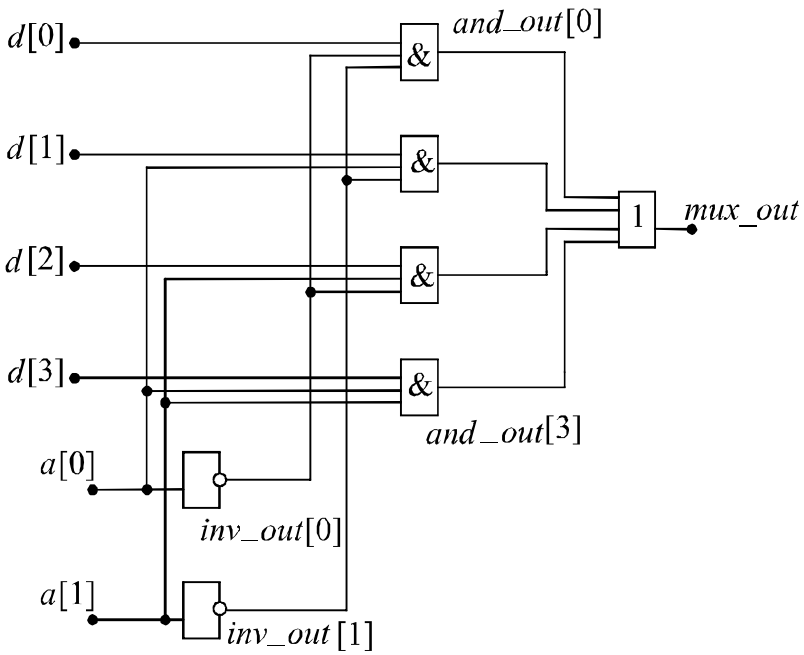
<ключевое слово> <имя>(<список имен выходов>, <список имен входов>),
<имя> (<список имен выходов>,
<список имен входов>),…;

Встроенные примитивы применяются с любым типом данных.

Если при моделировании необходимо учесть время задержки ЛЭ, то в конструкцию включается запись #<время задержки в заданных единицах>.

Например:

or #5 *myor*(*out*, *data*[1], *data*[0]);



а

a_1	a_0	f
0	0	d_0
0	1	d_1
1	0	d_2
1	1	d_3

б

$$f = \overline{a}_1\overline{a}_0 d_0 \vee \overline{a}_1 a_0 d_1 \vee a_1\overline{a}_0 d_2 \vee a_1 a_0 d_3$$

в

Рис.5.1. Мультиплексор "4 - 1": *а* - логическая схема; *б* - таблица истинности; *в* - алгебраическое выражение для функции выхода

Описание логических вентилей зарезервировано в Verilog и входит в любое средство моделирования.

5.2. Проектирование комбинационных схем на примере мультиплексора "4 - 1"

При проектировании логических устройств Verilog позволяет использовать различные способы их описания.

Рассмотрим описание, основанное на применении встроенных примитивов для схемы мультиплексора "4 - 1" (рис.5.1,*а*).

Начнем с описания базового модуля:

```
/* модуль предназначен для вычисления функции выхода мультиплексора "4 - 1" */
module mux 4_1(mux_out, a, d);
    output mux_out;
    input [3:0] d;  // 4-битная величина
    input [1:0] a;
    wire [3:0] and_out;
    wire [1:0] inv_out;
    not m_not_0(inv_out[0], a[0]);
    not m_not_1(inv_out[1], a[1]);
    and m_and_0(and_out[0], d[0], inv_out[1], inv_out[0]);
    and m_and_1(and_out[1], a[0], d[1], inv_out[1]);
    and m_and_2(and_out[2], a[1], d[2], inv_out[0]);
    and m_and_3(and_out[3], a[1], a[0], d[3]);
    or m_or(mux_out, and_out);
endmodule     // mux 4_1
```

В этом примере информационные и адресные входы, а также внутренние связи объявлены векторами. Применение векторной формы имеет и положительные, и отрицательные стороны: одни записи становятся короче и проще, другие - длиннее.

Тип сигналов для внешних входов и выходов вообще не объявлен. По умолчанию им присваивается тип *wire*.

Перейдем к построению тестового модуля. Субмодули в данном случае отсутствуют, а базовый модуль является также и головным.

```
/* тестовый модуль для мультиплексора "4 - 1" */
    timescale 1ns/10ps
    module mux_tb;
        wire mux_out;
        reg [1:0]a;
        reg [3:0] d;
        mux 4_1 nom1(mux_out, a, d,);
        initial begin
            a = 2'b00;
            d = 4'b0000;
        end
        initial begin
            #10 a = 2'b01;
                d = 4'b0010;
            #10 a = 2'b10;
                d = 4'b0000;
            #10 a = 2'b11;
                d = 4'b1000;
            #10 $finish
        end
    endmodule  // mux_tb
```

Данный тестовый модуль выполняет также и функцию модуля-оболочки.

В описании модуля использованы преимущества векторной формы представления переменных при задании входных воздействий. Входные воздействия (сигналы) соответствуют таблице истинности функционирования мультиплексора (рис.5.1,*б*).

Таким образом, получен Verilog-проект для мультиплексора "4 - 1", состоящий из базового и тестового модулей. Если нет ошибок, то функция выхода при заданных входных сигналах в течение четырех тактов (по 10 нс каждый) должна последовательно принимать значения: 0101.

Глава 6. Конструкция assign

6.1. Особенности конструкции assign

Непрерывное назначение, или присваивание, является характеристикой цепи. Обеспечение непрерывного назначения цепи реализует конструкция *assign*, которая имеет следующий формат:

assign <имя переменной> = <вычисления, операторы Verilog>;

Например:

assign result = *op*1 + *op*2;

assign y= ~|*data*[1:0];

Разумеется, переменная, которая определяется с помощью конструкции *assign*, должна быть объявлена цепью. Значение цепи вычисляется каждый раз заново, когда меняется хотя бы один из операндов, входящих в правую часть конструкции. В качестве операндов могут применяться векторные величины. С точки зрения классификации функциональных элементов ядра тела модуля конструкция *assign* относится к непрерывному назначению.

Непрерывное назначение для полусумматора, рассматриваемого в параграфе 4.2, выглядит следующим образом:

assign s = *a* &~ *b* | ~ *a* & *b*;

assign p = *a* & *b*;

а с применением векторной формы конструкция имеет вид

output [1:0]*sum*;

input a, *b*;

assign sum = *a* + *b*;

В случае необходимости в конструкции можно учитывать временные задержки.

Например:

assign # 2 *out_y* = *x*1 & *x*2 | *x*3 & *x*4;

т.е. новое значение *out_y* будет вычисляться через 2 единицы заданного времени по сравнению со временем последнего изменения операнда (или операндов). Хотя с точки зрения возможностей Verilog такая конструкция вполне допустима, применять ее при проектировании реальных устройств нежелательно (см. главу 13).

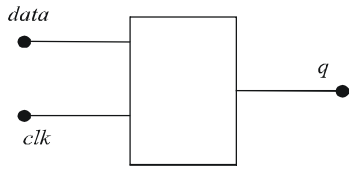


Рис.6.1. Условное графическое
изображение синхронного *D*-триггера

Все приведенные примеры относятся к так называемой явной форме, т.е. объявление типа переменной и собственно вычисления производятся в разных конструкциях:

wire sel;

assign sel= (*data*[0])? *data*[1]: *gnd;*

Наряду с явной может применяться и неявная форма, когда объявление типа переменной и вычисления производятся в одной конструкции:

wire sel= (*data*[0])? *data*[1]: *gnd;*

С конструкцией *assign* может применяться только одно выражение (один оператор присваивания). То есть группы операторов, а значит, и операторные скобки с этой конструкцией не используются.

Assign применяется для проектирования комбинационных частей устройства, а *always* - и для комбинационных, и для последовательных. Например, функционирование *D*-триггера (рис.6.1) с точки зрения возможностей Verilog может быть описано следующим образом:

assign q = ~clk & q|clk & data;

Однако на практике алгебраическое описание триггерных устройств не применяется, так как алгоритмический способ дает возможность синтезатору выбрать конфигурацию последовательностного устройства, а формула - нет. Кроме этого, по формуле всегда будет синтезировано устройство со статическим управлением, хотя в настоящее время применяется в основном динамическое управление.

Различные примеры записей на языке Verilog как для конструкции *assign*, так и для других конструкций, приведены в Приложении 2.

6.2. Пример описания логической схемы с применением конструкции assign

Рассмотрим устройство мультиплексора "4 - 1". Конструкция *assign* дает возможность использовать алгебраическое представление функции выхода (см. рис.5.1,*в*).

```
/* модуль предназначен для описания функционирования мультиплексора "4 - 1" с применением конструкции assign */
module mux4_1_as(out_mux, a, d);
    output out_mux;
    input [1:0]a;
    input [4:0]d;
    assign out_mux = (~a[1]& ~a[0]&d[0]|
            ~a[1]&a[0]&d[1]|
            a[1]&~a[0]&d[2]|
            a[1]&a[0]&d[3]);
endmodule
```

В этом примере к имени модуля добавлено *_as* для идентификации применяемой в нем конструкции.

Входы (как адресные, так и информационные) объявлены векторными переменными. Обе группы входов являются цепями, так как явного назначения в модуле им не делается, и они управляются внешними сигналами. Цепью также является и выход, поскольку он вычисляется с помощью конструкции *assign*. Тип сигналов (*wire*) в модуле не объявлен, так как он назначается по умолчанию. Применение скобок в правой части конструкции *assign* в данном случае необязательно.

Глава 7. Операторы Verilog

Основой описания различных алгоритмов является применение операторов, которых в языке Verilog насчитывается более тридцати (Приложение 3). Все операторы размещаются либо внутри блока *always*, либо в конструкции *assign* за некоторыми исключениями для блока *initial*.

7.1. Классификация операторов

Операторы классифицируются по трем признакам:

1) по количеству операндов;
2) по выполняемой функции;
3) по количеству бит (разрядов) у результата.

В зависимости от количества операндов существуют три типа операторов:

1) с одним операндом (унарные); они располагаются слева от операнда, например: *clock* =~ *clock*; // унарный оператор отрицания, *clock* - операнд

2) с двумя операндами (бинарные); они располагаются между операндами, например:

y = *a* + *b*; // бинарный арифметический оператор сложения,
// *a* и *b* - операнды

Все бинарные операторы выполняются над операндами с одинаковой разрядностью. Если разрядность одного операнда меньше другого, он дополняется нулями: неявно - системой, явно - разработчиком (см. параграф 2.4); предпочтителен второй вариант;

3) с тремя операндами (тернарный оператор); он разделяет три операнда двумя знаками. Такой оператор всего один, и его конструкция имеет вид

out = *a* ? *d*1 : *d*0; // тернарный оператор ?:

Оператор читается следующим образом: если *a* - истинно, то *out* = *d*1, иначе *out* = *d*0.

Ни к одному из перечисленных типов не относятся операторы: конкатенации, повторения и сдвига.

В зависимости от выполняемой функции существует 9 типов операторов: арифметические, поразрядные, конкатенация, операторы повторения, сдвига, отношения, эквивалентности, приведения и логические операторы. Первые пять типов операторов дают многобитный результат, остальные - однобитный.

Особую группу составляют условные операторы, которые будут рассмотрены в параграфе 7.6.

7.2. Арифметические и поразрядные операторы, конкатенация, операторы повторения и сдвига

Арифметические операторы являются бинарными и включают в себя: сложение, вычитание, умножение, деление и определение остатка от деления. Соответственно перечислению они обозначаются знаками: +, –, *, /, %.

Например:

y = *a* * *b*;

Если хотя бы один бит в одном из операндов неопределен (*x*), то и результат операции будет неопределен.

При использовании арифметических операторов надо в случае необходимости под результат зарезервировать на 1 или более бит больше, чем под операнды (см. пример описания полусумматора с применением векторной формы в параграфе 6.1).

Поразрядные операторы являются бинарными (за исключением поразрядного отрицания) и предназначены для выполнения основных логических функций: И, ИЛИ, НЕ, исключающее ИЛИ и исключающее ИЛИ-НЕ. Соответственно они называются: поразрядное И, поразрядное ИЛИ, поразрядное отрицание, поразрядное исключающее ИЛИ и поразрядное исключающее ИЛИ-НЕ. Эти операторы обозначаются знаками: &, |, ~, ^, ~^ (или ^~).

Если *a* и *b* являются *n*-разрядными векторами, то результатом выполнения поразрядного оператора является также *n*-разрядная величина (например, *g*), каждый разряд которой - результат выполнения оператора над одноименными разрядами исходных величин *a* и *b*. В частном случае *a* и *b* могут быть одноразрядными величинами.

Например, пусть $a = 4'b1100;$

$c = 4'b0101;$

тогда выполнение поразрядных операторов над этими векторами даст следующие результаты:

$y = \sim a;$ // $y = 4'b0011$

$out = a \& c;$ // $out = 4'b0100$

Поразрядные операторы применялись в модуле параграфа 6.2 при изучении конструкции *assign*.

Конкатенация (от английского слова "объединение") позволяет увеличить разрядность как цепей, так и регистров. При этом объединяемые величины заключаются в фигурные скобки и указываются через запятую.

Например:

$a = 1'b1;$

$b = 2'b00;$

$g = \{a, b\};$ // здесь $g = 3'b100;$

$h = \{2'b1x, 4'd7\};$ // здесь $h = 6'b1x0111$

Количество объединяемых величин не ограничивается.

Конкатенацию удобно применять при назначении входных воздействий.

Например, в тестовом модуле сумматора (параграф 4.3) входные воздействия можно задать следующим образом:

```
initial begin
……..……..……..……..
   # 10 {a ,b , p_in} = 3'b100;
……..……..
end
```

Повторение. Этот оператор расширяет возможности конкатенации и используется для многократного повторения объединения. Коэффициент кратности также указывается в фигурных скобках.

Например, для величин *a* и *b*, определенных для конкатенации, различные способы объединения выглядят следующим образом:

$c = \{4\{a\}\};$ // $c = 4'b1111$

$d = \{2\{a, b\}\};$ // $d = 6'b100\_100$

Если *c* объявлена как 4-разрядная величина, а в тексте ей присваивается значение: $c = \{2\{a\}\};$

то в старшие разряды будут добавлены нули, т.е. $c = 4'b0011$. Естественно, что для указанных переменных *a* и *c* нельзя написать: $c = \{5\{a\}\};$

Операторы сдвига позволяют осуществить сдвиг операнда вправо или влево на *n*-разрядов:

>> <*n*> - сдвиг вправо на *n* разрядов,

<< <n> - сдвиг влево.

Незаполненные разряды дополняются нулями. В этих конструкциях *n* может быть числом, переменной или функцией.

Например: *y*0 = 4'*b*0001;

*y*1 = *y*0 << 1; // *y*1 = 4'*b*0010

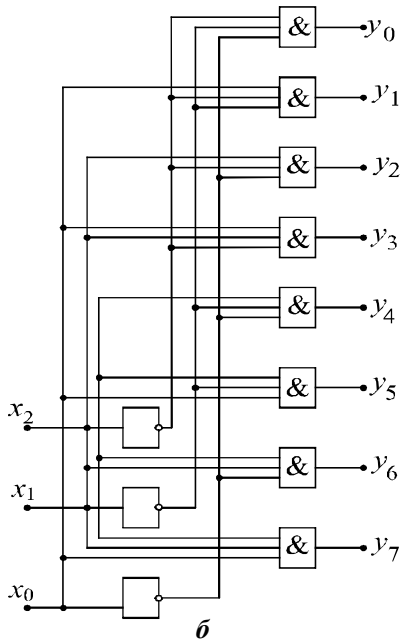
Операторы сдвига часто применяются для реализации на Verilog алгоритмов функционирования умножителей, регистров сдвига и т.д.

7.3. Пример описания функционирования дешифратора "3 - 8"

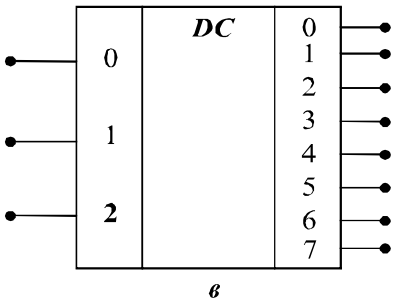
Алгоритм функционирования дешифратора может быть описан несколькими способами. Некоторые способы описания комбинационных устройств были рассмотрены на примерах сумматора и мультиплексора в предыдущих параграфах. Они применимы и к дешифратору (рис.7.1).

x_2	x_1	x_0	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0					
0	1	0		0	1	0				
0	1	1			0	1	0			
1	0	0				0	1	0		
1	0	1					0	1	0	
1	1	0						0	1	0
1	1	1							0	1

а



б



в

Рис.7.1. Дешифратор "3 - 8": *а* - таблица истинности (в незаполненных позициях - нули); *б* - логическая схема; *в* - условное обозначение

Рассмотрим описание поведения дешифратора с использованием оператора сдвига. Описание составлено на основе таблицы истинности дешифратора (рис.7.1,*а*).

Базовый модуль выглядит следующим образом:

```
/* модуль предназначен для описания дешифратора "3 - 8" */
      module dc3_8(dc_out, dc_in);
          output [7:0] dc_out;   // объявление выходов (рис.7.1,б,в)
          input [2:0] dc_in;       // объявление входов
          reg [7:0] dc_out;        // так как применяется блок always
          always @ (dc_in);
                 begin
                     dc_out = 8'b00000001;
                     dc_out = dc_out << dc_in;
                 end
endmodule   // dc3_8
```

В приведенном модуле сначала в младший разряд записывается единица, затем происходит сдвиг этой единицы на количество разрядов, соответствующее значению входного набора. Например, если *dc_in* = 3′*b*100, то *dc_out* = 8′*b*00010000, т.е. *dc_out*[4] = 1, а значения остальных выходов равны нулю, что и требовалось. Если *dc_in* = 0, то сдвига не происходит.

Описание поведения дешифратора с применением оператора сдвига можно выполнить еще проще - используя только одну конструкцию. В этом случае лучше воспользоваться конструкцией *assign*.

7.4. Операторы отношения, эквивалентности, приведения, логические операторы

Результатом выполнения данных операторов является однобитная величина.

Операторы отношения являются бинарными и предназначены для сравнения двух операндов. Операторами отношения являются: "больше", "больше или равно", "меньше" и "меньше или равно", которые обозначаются соответственно знаками: >, >=, <, <=.

Результат выполнения операторов равен единице или нулю (истина или ложь). Если хотя бы один операнд неопределен (*x*), то и результат операции будет неопределен.

Операторы отношения применяются с условными операторами, в тернарном операторе и др.

Например:

```
y = (a > b) ? 1 : c;       // если a > b, то y = 1, иначе y = c
if (a <= b) y = f [4];     // если a <= b, то y = f [4]
```

В качестве операндов применяются как однобитные величины, так и многобитные (векторы). При сравнении оценивается значение вектора в целом (а не побитно).

Операторы эквивалентности также являются бинарными и также предназначены для сравнения двух операндов. Но сравнение носит иной характер. Во-первых, операторы эквивалентности, в отличие от операторов отношения, сравнивают операнды побитно; а во-вторых, проверяется равенство операндов. Результат операции также равен единице или нулю (истина или ложь).

К операторам эквивалентности относятся оператор логического равенства (==) и оператор логического неравенства (!=). Если операнды содержат значения *x* или *z*, то результат всегда будет равен *x*. В этом случае применяются операторы выборочного равенства (===) и выборочного неравенства (!==).

Например:

```
a = 4'b010;     // при объявленном размере вектора (4) a будет
```

// дополнено нулем: a = 0010

$c = 4'bx10$; // c будет дополнено x: $c = xx10$

Применение операторов эквивалентности к этим величинам даст следующие результаты:

a != c; // результат x

a ! == c; // результат 1

a == c; // результат x

a === c; // результат 0

Операторы эквивалентности применяются со всеми конструкциями, в которых может проводиться проверка.

Например:

if (R == 1) q = 0;

Операторы приведения, так же как и поразрядные операторы, позволяют выполнять основные логические функции: И, ИЛИ, И-НЕ, ИЛИ-НЕ, исключающее ИЛИ, исключающее ИЛИ-НЕ. Однако это унарные операторы; они выполняются над многоразрядным операндом (вектором) пошагово, бит за битом, начиная с двух крайних левых разрядов, выдавая однобитный результат.

Операторы приведения имеют следующие обозначения (в соответствии с порядком перечисления функций): &, |, ~&, ~|, ^, ~^ (или ^~).

Например, если $a = 4'b1100$, то результатом выполнения оператора ИЛИ (|a) будет 1, а результатом выполнения оператора исключающее ИЛИ (^a) будет 0.

Операторы приведения позволяют, например, реализовать проверку на четность (и соответственно нечетность) того или иного многобитного операнда.

Логические операторы, так же как поразрядные операторы и операторы приведения, позволяют выполнять логические функции: И, ИЛИ, НЕ, которые соответственно обозначаются символами: &&, ||, !. Это бинарные операторы; в качестве операндов здесь могут выступать как переменные, так и логические выражения, т.е. такие выражения, результатом выполнения которых является истина (1) или ложь (0). Другими словами, все ранее рассмотренные операторы этого параграфа.

При выполнении логических операторов сначала происходит оценка операндов (векторов): если в векторе присутствует хотя бы одна единица, то весь операнд считается равным единице, иначе - нулю. И лишь после этой оценки выполняется сам оператор (уже над однобитными значениями). Результат - также однобитная величина (нуль или единица). Результат может принимать и значение x (неопределенное состояние).

Если операнды принимают значения x или z, то им присваивается низкий логический уровень - *false*.

Например, если $a = 2'b10$ и $b = 1'b0$, то результаты выполнения логических операторов имеют следующий вид:

$c = a\&\&b$; // c = 0

$d = a \,||\, b$; // d = 1

e =! a; // e = 0

7.5. Особенности выполнения операторов. Приоритет операторов

Принципы выполнения операторов были рассмотрены ранее. Однако существуют аспекты, которые носят общий характер и относятся ко всем операторам.

1. В одном и том же выражении могут быть использованы разные операторы. Под разными понимаются как операторы одного типа, так и разнотипные операторы.

2. Все операторы Verilog могут быть вложенными друг в друга (как разные, так и одинаковые). Количество уровней вложенности не ограничено в разумных пределах.

Например:

y = *a*& (*b*&(~*c*| *d*));

3. Все операторы выполняются в соответствии с определенным приоритетом (см. Приложение 4). При необходимости изменения порядка применяются скобки. Без нарушения приоритета заключать выражение в скобки не обязательно, но желательно для удобства восприятия.

Например:

assign out = (*x*1 &~ *a*1)|(*x*2 & *a*0);

4. В языке Verilog предусмотрены два типа операторов присваивания, которые обозначаются: = (оператор блокирующего присваивания) и <= (оператор неблокирующего присваивания). Первый тип (=) означает последовательное выполнение действий.

Например, если в тексте записано:

a = *c*;

b = *a*;

то переменной *b* присваивается значение *c* (*b* = *c*). Такой же результат будет при указании одинаковых задержек:

#5 *a* = *c*;

#5 *b* = *a*; // здесь также *b* = *c*

Часто этот оператор используется при составлении тестов.

Второй тип (<=) означает параллельное выполнение действий.

Например, если в тексте записано:

a <= *c*;

b <= *a*;

то переменной *b* присваивается значение *a* (*b* = *a*). Такой же результат будет при указании одинаковых задержек:

#5 *a* <= *c*;

#5 *b* <= *a*; // здесь *b* = *a*.

Если в операторах присваивания (обоих типов) указаны разные задержки, то действия выполняются в соответствии с указанным временем.

Конструкция *assign* применяется только с оператором присваивания первого типа, конструкция *always* - с операторами обоих типов. Описание комбинационных частей устройства происходит с оператором =, а последовательностных - в основном с <=.

Второй тип оператора присваивания имеет одинаковое написание с оператором отношения "больше или равно". Конкретная интерпретация этих операторов вытекает из контекста.

7.6. Условные операторы (процедурные)

К условным относятся три оператора: оператор условного перехода *if*, оператор выбора *case* и оператор ветвления (тернарный оператор). Первые два оператора применяются, как правило, в блоке *always*; оператор ветвления может применяться и в блоке *always*, и в конструкции *assign*.

7.6.1. Оператор условного перехода if

В этом операторе вычисляется условие, и в зависимости от результата выполняется либо первый указанный оператор (или группа операторов), либо второй. Группа операторов, как и в предыдущих случаях, заключается в операторные скобки *begin…end* и может быть поименована.

Конструкция оператора имеет следующий вид:

if (условие)

```
    begin [:<имя>]        // выполняется, если результат равен 1 (истина)
        <группа операторов>
    end
else
    begin [:<имя>]    // выполняется, если результат равен 0 (ложь)
        <группа операторов>
    end
```

В качестве условия могут использоваться простые и сложные функции, которые записываются с помощью операторов Verilog. Оператор логического равенства с единичной правой частью (== 1) может не указываться; тогда в качестве условия фигурирует одна независимая переменная (в том числе вектор). Например, записи *if* (*clk*) и *if* (*clk* == 1) тождественны. Результатом проверки условия должна быть однобитная величина (истина или ложь).

Допускается вложенность операторов *if*. Например, описание алгоритма работы полусумматора (см. рис.4.3,*б*) с применением оператора условного перехода может выглядеть следующим образом:

```
always @ (a or b)
    if (a^b) begin
        s = 1'b1;
        p = 1'b0;
    end
else if (a&&b) begin
        s = 1'b0;
        p = 1'b1;
     end
      else begin
        s = 1'b0;
        p = 1'b0;
     end
```

В приведенном примере проверка осуществляется на основе данных таблицы истинности полусумматора. Если *a* и *b* принимают разные значения и результат выполнения оператора *a*^*b* равен 1, то *s* = 1, а *p* = 0. В противном случае проверяется результат работы оператора *a*&&*b*: при *a* = 1 и *b* = 1 он равен 1; тогда *s* = 0, а *p* = 1. Иначе, т.е. при *a* = 0, и *b* = 0, *s* = 0 и *p* = 0.

Таким образом, оператор условного перехода требует выполнения каких-либо действий при условии "истина" или других действий при условии "ложь". Если результат "ложь" маловероятен или невозможен, то Verilog допускает при очевидности ситуации не указывать *else*. Однако нарушение пары *if* - *else* может вызвать синтез нежелательной логики. Чтобы указать, что ничего не должно происходить при условии "ложь", используется пустой оператор *else* и точка с запятой:

```
        else    ;
```

Если в операторе условного перехода *if* сравниваются векторы, то необходимо учитывать комментарии, изложенные в параграфе 2.4.

В случае, когда в операторе *if* переменной может не присвоиться какое-либо значение, ей нужно присвоить это значение перед условным оператором.

Рассмотрим пример базового модуля, содержащего описание функционирования дешифратора "3 - 8" (см. рис.7.1) с применением оператора условного перехода *if*.

```
module dc 3_8_if (dc_out, dc_in);
    output [7:0] dc_out;
    input [2:0] dc_in;
    reg [7:0] dc_out;
    always @ (dc_in)
```

```
        begin
          dc_out = 0;
          if (|dc_in ==1)
                dc_out [dc_in] = 1;
          else dc_out [0] = 1;
    end
endmodule     // dc 3_8_if
```

В модуле *dc* 3_8_*if* в начале алгоритма всем выходам присваивается значение нуля. Затем в зависимости от значения входов (т.е. если они отличны от 0) соответствующему выходу присваивается значение единицы. Например, если *dc_in* [2] = 0, *dc_in* [1] = 1 и *dc_in* [0] = 0, то *dc_out* [2] будет равен 1 (значения сигналов на остальных выходах остаются при этом равными 0), иначе (т.е. на всех входах присутствует нуль) *dc_out* [0] = 1.

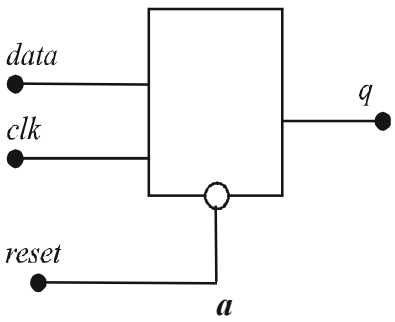
Для описания работы мультиплексора "4 - 1" может быть использована следующая конструкция оператора *if* (см. рис.5.1,*б*):

```
if ((a1 == 0) && (a0 == 0)) out_mux = d0;        // здесь внутренние
else if ((a1== 0) && (a0== 1)) out_mux = d1;    // скобки в соответствии
   else… и т.д.                                          // с приоритетом
                                                  // не обязательны
```

Подобная запись оператора *if* является корректной.

Некорректна следующая запись:

```
if (a1 == a0 == 0) out_mux = d0;
```



а

reset	*clk*	*data*	*q*
0	*x*	*x*	0
1	1	0	0
1	1	1	1
1	0	*x*	*

* - режим хранения

x - безразличное состояние

б

Рис.7.2. Синхронный *D*-триггер со статическим управлением записью и инверсным сбросом: *а* - условное (графическое) обозначение; *б* - таблица истинности

До сих пор рассматривались описания алгоритмов работы комбинационных устройств. Пример описания функционирования *D*-триггера, приведенного на рис.7.2,*а*, имеет следующий вид:

```
/* модуль содержит описание алгоритма работы синхронного D-триггера со статическим управлением записью и инверсным сбросом */
module d_tr_if (q, clk, data, reset);
    output q;
    input clk, data, reset;
```

```
    reg q;
    always @ (clk or reset)
        if (~ reset) q =0;
        else if (clk) q =data;
            else   ;
endmodule    // d_tr_if
```

В этом примере в соответствии с таблицей истинности *D*-триггера (рис.7.2,*б*) при значении сигнала *reset* = 0 выход принимает значение 0 (*q* = 0). В противном случае при наличии разрешающего сигнала (*clk* = 1) сигнал на выходе принимает значение входного сигнала (*q* = *data*). Иначе (*clk* = 0) на выходе остается старое значение сигнала (режим хранения), так как в Verilog переменные хранят свое значение до присвоения нового.

Следует отметить, что способы описания алгоритмов работы рассмотренных в этом параграфе устройств с использованием оператора *if* не являются единственными.

Приведем еще один пример записи конструкции *if* с использованием конкатенации:

```
reg [7:0] a;
reg [3:0] c;
…….
    if (a == {4'b0, c}) begin
    ……
    end
    else        ;
```

7.6.2. Оператор выбора case. Примеры описания функционирования триггерных устройств

В операторе выбора *case* вычисления производятся так же, как и в операторе *if*, в зависимости от условия. Однако условие может быть только численной величиной или, в частном случае, параметром.

Конструкция оператора имеет вид:

```
case ({переменная 1, переменная 2, ...})
    <условие 1>:<оператор 1>;
    <условие 2>:<оператор 2>;
    …….....
    default: <оператор>;
endcase
```

Независимые переменные в операторе *case* указываются с помощью конкатенации:

```
case ({in1, in2})          // in1, in2 - однобитные переменные; например,
                           // входы устройства
2'b01: y = 0;              // y - выход устройства
default: y = 1;            // во всех остальных случаях
endcase
```

В приведенном примере оценивается величина оператора {*in*1, *in*2}. Если он равен 2′*b*01 (это и есть условие, т.е. если *in*1 = 0, а *in*2 = 1), то выходу присваивается значение нуля. В случае несовпадения значения входных переменных с условием (ключевое слово *default*) выходной сигнал должен быть равен единице. Во избежание проблем при моделировании устройства оператор *case* всегда должен включать случай несовпадения, даже если он пустой:

```
default:    ;
```

Если переменная объявлена как вектор, то в операторе *case* она указывается без конкатенации.

Например:

case (*a*)

Если в условии присутствуют безразличные состояния, то используются модификации оператора *case*: *casex* и *casez*. Их применение означает, что условие может содержать безразличное состояние, которое в первом случае (*casex*) обозначается как *x* или *z*, а во втором - только как *z*.

В операторе выбора могут использоваться операторные скобки.

Например:

```
case(in)
3'b000: begin  y1 = 0;
               y2 = 1;
               …….
        end
 3'b001: …..
```

Конструкция *case* успешно применяется для описания как комбинационных, так и последовательностных устройств.

Описание алгоритма функционирования *D*-триггера (рис.7.2,*б*) с применением оператора выбора *case* может выглядеть следующим образом:

```
always @ (clk, reset)
   casex ({clk, reset})
      2'bx0: q= 1'b0;      // x - безразличное состояние
      2'b11: q = data;
      default:    ;
endcase
```

Здесь значения выходного сигнала формируются в зависимости от значения входных сигналов.

При описании алгоритмов работы различных устройств можно использовать комбинацию условных операторов *if* и *case*. В этом случае конструкции записей выглядят следующим образом:

```
   if ...                 if ...              case...
   case ...               .....               if ...
   .....                  else                .....
   endcase                case                else
или                   или                    .....
   .....                  .....               endcase
   else                   endcase
```

Рассмотрим пример построения Verilog-проекта для проверки алгоритма функционирования двухступенчатого устройства - *MS*-триггера (рис.7.3,*а*) - с использованием условных операторов. В этом проекте можно выделить модули трех уровней: базовый модуль, который описывает поведение синхронного одноступенчатого *RS*-триггера (на рисунке одноступенчатые *RS*-триггеры обведены пунктирной линией), головной модуль, который включает в себя два экземпляра базового модуля, и тестовый модуль, который выполняет также функцию модуля-оболочки.

Приведем текст проекта:

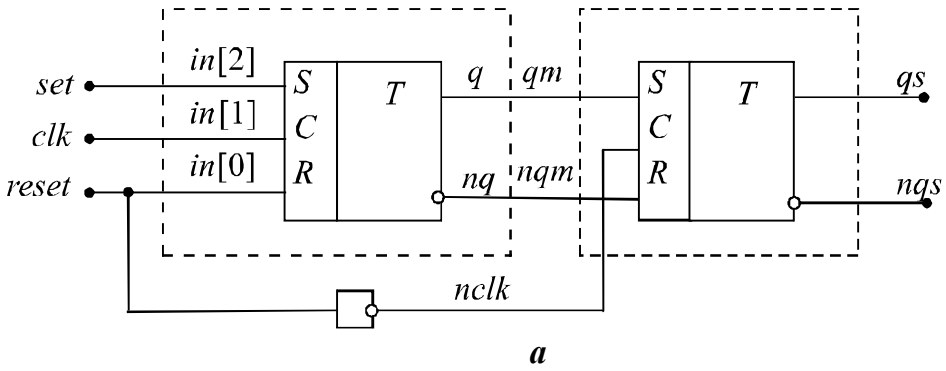
```
/* тестовый модуль проекта */
timescale 1ns/1ps
module test_ms_tb;
   wire nq, q;
   reg clk, r, s;
   ms_tr ms (nqs, qs, clk, r, s);   // вызов головного модуля
```

```
    initial { clk, r, s} = 0;          // начальные установки входных
                                       // воздействий
    always #10 clk = ~clk;
    initial begin
       # 10 { r, s} = 2'b01;   // проверка режима установки (рис.7.3,б);
                               // здесь r = 1, s = 0
       # 20 {r, s} = 2'b00;   // проверка режима хранения
       # 20 {r, s} = 2'b11;   // проверка запрещенного режима
       # 20 {r, s} = 2'b10;   // проверка режима сброса
       # 20 {r, s} = 2'b00;
       # 10 $finish
    end
endmodule   // test_ms_tb
/* головной модуль проекта */
module ms_tr (nqs, qs, clk, r, s);  // выходы и входы указаны в
                                    // алфавитном порядке
    output nqs, qs;
    input clk, r, s;
    wire nclk, nqm, qm;        // объявление внутренних цепей
    rs_tr base1 (nqm, qm, s, clk, r);
    rs_tr base2 (nqs, qs, qm, nclk, nqm);
    not base3 (nclk, clk);
endmodule   // ms_tr
module rs_tr (nq, q, in);      // базовый модуль проекта
    output nq, q;
    input [2:0] in;
    reg nq, q;
    always @ (in)
       if (in[1]) begin
        case ({in[2], in[0]})
             2'b01: q = 1'b0;
             2'b10: q = 1'b1;
             2'b11: q = 1'bx;
             2'bxx: q = 1'bx;
             default: $display ("неверный код") ;
       endcase
       nq = ~ q;
      end
     else   ;
endmodule // rs_tr
```

В приведенном примере в основе описания базового модуля лежит знание таблицы истинности. При передаче портов необходимо учитывать структуру устройства.

Условие 2′*bxx*: *q* =1′*bx*; введено для учета работы второй ступени устройства. Если с первой ступени поступают сигналы *xx* (т.е. входы *in*[2] и *in*[0] равны 1), то на выходах *qs* и *nqs* должна быть неопределенность, что и записано в конструкции.

Остановимся подробнее на описании режима хранения. В приведенном примере он может быть учтен в конструкции *default*: *q* = *q*; хотя в предыдущих примерах при описании *D*-триггера этого не указывалось. Дело в том, что при синтезе устройства оператор *q* = *q* может привести к появлению лишней связи. Поэтому его следует избегать и использовать с осторожностью только на этапе проверки работы алгоритма. Таким образом, в данном случае можно обойтись пустым оператором *default*: величина *q* и так будет хранить свое значение до присвоения нового, обеспечивая тем самым режим хранения.



а

c	*r*	*s*	*q*
0	*x*	*x*	*
1	0	0	*
1	0	1	1
1	1	0	0
1	1	1	*x*

все равно что (безразличное состояние)

* - режим хранения

неопределенность

б

Рис.7.3. Двухступенчатый *RS*-триггер (*MS*): *а* - логическая схема; *б* - таблица истинности

На данном примере в качестве тренировки можно отработать механизмы передачи портов, взаимодействие двух ступеней, учет неопределенного состояния и описание тестового модуля. Однако в практических разработках нужно учитывать следующее. Первое - в настоящее время применяется в основном динамическое управление записью. И второе - при описании триггерных устройств используется в большинстве случаев оператор неблокирующего присваивания (но с учетом особенностей моделирования и очереди событий, см. главы 11 и 13).

7.6.3. Оператор ветвления

Оператор ветвления является тернарным оператором. Конструкция оператора имеет вид:

<переменная> = <условное выражение>?
<выражение 1> : <выражение 2>;

Если условное выражение истинно, то переменная принимает значение выражения 1, иначе - значение выражения 2.

Условное выражение (иначе - проверяемая величина) может быть переменной или параметром, а также может включать в себя проверку результата выполнения различных операторов.

Например:

flag = (*in* == 4'*b*1100)?1:0;

В данном случае, если векторная величина *in* принимает значение 1100, *flag* = 1, иначе *flag* = 0.

По умолчанию условное выражение сравнивается с единицей.

В качестве выражения 1 и 2 могут быть указаны: числа, переменные (в том числе векторы), а также функции.

Например:

out = *clk* ? *q*0: (*q*1&*q*2);

Оператор ветвления применяется в основном с конструкцией *assign* и может быть вложенным.

Например, алгоритм работы мультиплексора "4 - 1" с использованием тернарного оператора имеет вид

assign out_mux = *a*1?(*a*0?*d*3:*d*2):(*a*0?*d*1:*d*0);

а работу полусумматора можно описать следующим образом:

assign s = (~*a*&*b*|*a*&~*b*)?1:0;

assign p = (*a*&*b*)?1:0;

7.6.4. Элементы с третьим состоянием. Пример описания с применением оператора ветвления

Высокоимпедансное (или третье) состояние в описаниях модулей обычно вызывает у студентов ряд вопросов. Поэтому в данном пособии приведен пример ключевой схемы (рис.7.4), реализующей функцию ЛЭ с тремя состояниями (0, 1, отключение - *z*), а также описание модуля с использованием *z*.

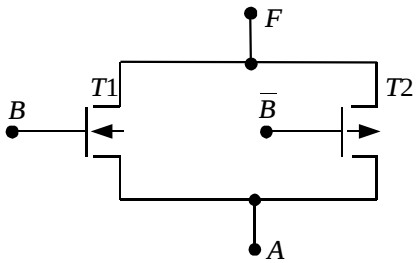


Рис.7.4. Ключевая схема, реализующая функцию элемента с тремя состояниями

Рассмотрим МОП-схему, изображенную на рис.7.4. При параллельном соединении *p*- и *n*-канальных транзисторов получаем ключевую схему, на выходе которой реализуется функция

F = *A* при *B* = 1.

Действительно, если *A* = 0, то потенциал, поступающий на вход *A*, передается на выход через открытый транзистор *T*1 (*T*2 закрыт). Если *A* = 1, то потенциал передается через *T*2 (*T*1 закрыт). При *B* = 0 оба транзистора закрыты, т.е. выход оказывается в отключенном состоянии (состояние *z*). Таким образом, данная схема реализует функцию элемента с тремя состояниями (0, 1, *z*).

Логические элементы с тремя состояниями широко применяются на практике. С их помощью можно, например, записывать и считывать информацию с *n* регистров, подключенных к одной шине. В каждый момент времени работа производится только с одним регистром, а остальные находятся в отключенном состоянии *z*.

Пример Verilog-модуля с третьим состоянием:

```
module tri_z (bus, enable, value);
    inout [3:0] bus;
    input enable;
    input [3:0] value;
        assign #2 bus = (enable == 1)? value:4'bz;
endmodule  // tri_z
```

В данном примере, когда управляющий сигнал принимает высокий уровень (1), шина принимает значение входной величины *value*; в противном случае она переходит в третье состояние *z*.

При описании устройств следует помнить, что сигнал *z* имеет наименьшую силу и может не влиять на итоговое значение сигнала цепи.

Глава 8. Циклы Verilog

При описании цифровых устройств дополнительные (к операторам) возможности разработчику предоставляют циклы. В Verilog существуют четыре типа циклов: *while*, *for*, *repeat* и *forever*. Все циклы можно использовать только внутри блоков *always* и *initial*.

Цикл ***while*** предназначен для выполнения одного или группы операторов, следующих за ним. Операторы выполняются до тех пор, пока записанное в цикле условие верно. В качестве условия может быть использован любой оператор параграфа 7.4, т.е. такой, результатом выполнения которого является однобитная величина.

Конструкция цикла *while* имеет следующий вид:

while (<условие>) *begin*
    <группа операторов>
end

Например:

```
always@ (a,b)
while (a < b) begin
    y1 = 1'b1;
    y2 = 1'b0;
end
```

Цикл ***for*** дает возможность инициализировать, проверять и увеличивать переменную явным способом.

Конструкция цикла *for* имеет следующий вид:

for (<инициализация, начальное назначение переменной>;
    <условие продолжения(проверка)>; <оператор, указывающий способ увеличения переменной>) *begin*
    <группа операторов>;
end

Операторы выполняются, пока условие продолжения истинно. Увеличение переменной выполняется в конце каждого прохода цикла.

Например:

```
integer [31:0] list;
integer index;
initial begin
    for (index = 0, index < 31; index = index + 1)
      list [index] = index + index;
end
```

Цикл ***repeat*** - это цикл с заданным числом повторений. Он выполняется конечное число раз.

Конструкция цикла имеет следующий вид:

repeat (<условие>) *begin*
<группа операторов>;
end

Здесь условие может быть константой или выражением (значение выражения вычисляется при первом прогоне цикла), но обязательно целым числом.

Например:

```
integer count;
reg [127:0] a;
initial begin
     count = 128;
     repeat (count)
        begin
```

```
            count = count - 1;
            a [count] = count/2;
        end
    end
```

Цикл ***forever*** называется вечным циклом. Следующие за ним операторы (или оператор) выполняются непрерывно, пока не поступит команда *$finish*, т.е. до конца моделирования.

Конструкция цикла *forever* имеет следующий вид:

```
forever begin
    <установка> <группа операторов>
end
```

Например:

```
initial begin
     a = 2; b = 4;
    forever begin
     # 5 a = a + b;
          b = a - 1;
    end   // forever
end     // initial
```

В этом примере указано, что каждые 5 заданных временных единиц надо вычислять *a* и *b* и делать это до тех пор, пока моделирование не будет остановлено командой *$finish*.

Приведем еще один пример:

```
reg clk;
initial begin
    clk = 1'b0;
    # 100 forever # 5 clk = ~ clk;
end
initial # 200 $finish;
```

Эта запись означает, что в течение 100 единиц заданного модельного времени сигнал *clk* не меняется и равен 0. Затем в период от 100 до 200 единиц он должен менять свое значение на противоположное каждые 5 временных единиц.

Глава 9. Верификация в Verilog

В языке Verilog предусмотрены различные возможности для организации верификации (проверки, контроля) проекта.

9.1. Формирование начальных значений входных воздействий

Работоспособность проекта (в том числе алгоритмов, устройств) должна быть проконтролирована на соответствие техническому заданию (определенному, например, в виде таблицы истинности). Поэтому в ходе проверки входным портам присваиваются все возможные значения (далее - входные воздействия или сигналы), отображающие реальные варианты работы.

Все входные воздействия формируются в тестовом модуле. Они задаются с помощью базовых блоков:

```
initial                          always
begin                            begin
.....            и               .....
end                              end
```

Блок *initial* выполняется с начала моделирования, поэтому в нем производится инициализация сигналов, т.е. задание начальных значений. Приведем несколько примеров:

1.
```
initial begin
    a0 = 0;
    a1 = 0;
    d0 = 0;
    .....
    d3 = 0;
end
```

2. Те же сигналы в векторной форме:
```
initial begin
    a = 2'b00;    // допускается a = 0; или a = 2' d0;
    d = 4'b0000;
end
```
3. Те же сигналы с использованием конкатенации:
```
initial
    {a0, a1, d0,..., d3} = 6'b000_000;
```

Итак, установлено значение входных сигналов в момент старта моделирования ($t = 0$). Далее проект проверяется во времени (временной контроль).

9.2. Временной контроль

Для анализа поведения объекта в процессе работы в языке Verilog предусмотрены три варианта временного контроля: с использованием задержек (*delay*), событий (*event*) и списка чувствительности.

Задержка обозначается символом # и означает время либо между сменой входных сигналов, либо между поступлением сигнала на вход элемента или модуля и появлением результата на его выходе.

В первом случае задержка применяется в тестовом модуле для задания поведения входных воздействий во времени и указывается в следующем формате:

<τ> <одна из трех конструкций операторов, указанных в предыдущем разделе>;

Здесь τ может быть числом (целым или действительным), параметром, постоянной или переменной величиной или, наконец, функцией.

Например:

```
initial begin
    # 10 a = 1;    // выполняется в момент t = 10
    # 2.5 b = 2;   // выполняется в момент t = 12.5
    # b/2 c = a;   // выполняется в момент t = 13.5
    # prm c = 4;   // выполняется в момент t = 13.5 + prm
end
```

Возможны и другие конструкции тех же записей:

```
initial begin
    a = # 10 1;
    b = # 2.5 2;
    ..........................
end
```

Время измеряется в тех единицах, которые определяются в директиве *`timescale*.

В качестве примера задания поведения входных воздействий во времени можно привести описание сигналов в тестовом модуле для сумматора (параграф 4.3):

```
initial begin
    # 10 a = 1;   // выполняется в момент t = 10
    # 2 b = 0;    // выполняется в момент t = 12
end
```

Входные воздействия задаются в тестовом модуле, в основном с помощью оператора блокирующего присваивания (=). Однако может применяться и оператор неблокирующего присваивания (<=). Приведенная запись с этим оператором будет иметь другой смысл:

```
initial begin
    # 10 a <= 1;   // выполняется в момент t = 10
    # 2 b <= 0;    // выполняется в момент t = 2
end
```

Задержка может применяться при описании встроенных примитивов, а также в конструкции *assign*.

Например:

```
nor # 2 mynor (out, in1, in2);
assign # 5 out = (cntrl = 4'b0111) ? 1 : 0;
```

Эта форма временного контроля и две следующие применяются во всех модулях, описывающих устройство (базовых, субмодулях, редко - в головном модуле).

При **событийном контроле** происходит вычисление результата при каждом изменении входных сигналов.

Например, запись

```
@ (clk) a = b;
```

означает, что при каждом изменении тактового сигнала *clk* (это и есть событие) выполняется *a = b*. Данная запись относится к группе конструкций блока *always*.

Приведем еще примеры событийного контроля:

```
@ (negedge clk) a = b;
a = @ (negedge clk) b;
```

Эти две записи тождественны и означают, что при переходе тактового сигнала *clk* из 1 в 0 выполняется *a* = *b*.

В настоящее время эта форма контроля применяется достаточно редко.

И наконец, разновидностью временного контроля является конструкция блока *always* со списком чувствительности (возбуждения).

Например:

```
always @ (clk or in1 or in2)
   begin
   ……..
end
```

Здесь изменение хотя бы одной переменной приводит к выполнению группы операторов, заключенных в операторные скобки *begin…end*.

9.3. Периодически изменяющиеся величины

Для задания периодически изменяющихся величин используются конструкции Verilog, у которых есть возможность изменять (чаще всего инвертировать) сигнал через заданный промежуток времени (или заданное число раз) во все время моделирования (или в указанный интервал).

Приведем несколько примеров, в которых используется это свойство.

1.
```
initial begin
   clk = 1'b0;
   forever # 5 clk = ~ clk;
end
```

Здесь каждые 5 единиц времени сигнал *clk* переходит из низкого уровня в высокий и наоборот.

2.
```
always # 10 clk = ~ clk;
```

Здесь сигнал *clk* должен быть проинвертирован каждые 10 единиц времени; это по сути означает, что период тактового сигнала составляет 20 заданных временных единиц.

Наиболее часто подобные конструкции применяются для задания тактового сигнала. Однако они вполне применимы ко всем периодическим сигналам.

3.
```
initial begin
   count = 10;
   repeat (count) begin
      # 40 in = 0;
      # 20 in = 1;
   end    // repeat
end      // initial
```

В этом примере период изменения величины *in* составляет 60 единиц времени и повторяется 10 раз.

4.
```
initial clk = 1'b0;   // запуск такта
   always # 10 clk = ~ clk;
```

Здесь приведен пример смешанного использования блоков *always* и *initial* при задании периодически изменяющейся величины.

5. Как уже указывалось, векторные величины можно периодически изменять с помощью следующих конструкций:

```
always # 10 in = in+1;
always # 10 in ++;      //запись, тождественная предыдущей
```

Глава 10. Системные директивы Verilog

Язык Verilog является не только средством описания проекта, но и довольно мощным инструментом для поведенческого моделирования. Встроенные директивы компилятора (системные директивы) позволяют выполнить моделирование и провести контроль и анализ его результатов.

К системным директивам относятся директивы вывода результатов моделирования и директивы окончания моделирования.

10.1. Директивы вывода результатов моделирования

Для наблюдения результатов моделирования применяются директивы: ***$display, $write*** и ***$monitor***. Они указываются внутри блоков *initial* или *always* (в основном в *initial*). Наиболее часто используется директива ***$display***. Она позволяет вывести на экран монитора строки, выражения или переменные. Информация, указанная в *$display*, выводится в нижней (консольной) части экрана после компиляции.

Конструкция директивы имеет следующий вид:

$display(<*expr* 1>, <*expr* 2>,…, <*expr n*>);

где <*expr* i> может быть строкой, выражением или переменной.

Приведем несколько характерных примеров записи директивы:

1. *$display* (*"Classic hits"*);

В консольной части экрана появится выражение *Classic hits*.

2. *$display* (*$time*);

Такая конструкция записи позволяет наблюдать на экране текущее время.

3. *counter* = 4*'b*10;
   $display (*"the counter is* % *b"*, *counter*);

Здесь *"the counter is"* является строкой; % *b* - формат вывода переменной (табл.10.1); *counter* указывает, какую переменную надо вывести (формат всегда предшествует переменной). В результате на экране будет наблюдаться следующее: *the counter is* 0010. Из приведенного примера видно, что строки и форматы заключаются в кавычки.

Таблица 10.1

Форматы вывода переменных для директив вывода результатов моделирования

Форма представления чисел	Формат		
Десятичная	% *d*	или	% *D*
Двоичная	% *b*	или	% *B*
Восьмеричная	% *o*	или	% *O*
Шестнадцатеричная	% *h*	или	% *H*

Примечание. Существуют и другие форматы, которые в данном пособии не рассматриваются.

Формат вывода может быть указан в самой директиве (например, *$displayb*); по умолчанию результаты выводятся в десятичной форме.

Директива *$display* предоставляет пользователю много различных возможностей. Она может применяться с задержкой и операторами цикла; внутри директивы могут использоваться векторы и различные операторы.

Применение конструкции # <*N*> перед директивой *$display* означает, что вывод на экран осуществляется через указанное время после назначения сигналов на входы.

Для мультиплексора "4 - 1" директива может выглядеть следующим образом:

1 *$display* (*"a*1 = %*b*, *a*0 = %*b*,…, *d*0 = %*b*, *out_mux is* % *b"*,
*a*1, *a*2,…, *out_mux*);

или в векторной форме:

1 *$display* (*"a* = %*b*, *d* = %*b*, *out_mux* = % *b"*, *a*, *d*, *out_mux*);

Применение вечного цикла с директивой *$display* дает возможность наблюдать результаты через заданный промежуток времени до конца моделирования.

Например:

forever # 10 *$display* (*$time " %d %d \ n"*, *in_out*);

Из этого примера видно, что форматы можно перечислять без запятой. Символы \ *n* означают переход на новую строку.

Приведем еще несколько примеров конструкций директивы *$display*:

$display (*a* / *b*);

$display (*a* && *b*);

always # 10 *$display* (*$time*, *"%b %b %b %d"*, *clk*, *reset*, *data*, *q*);

Если переменные *a* и *c* определены следующим образом:

a = 1′*b*1;

c = 6′*b*101001;

то применение директивы вывода

$displayb ({*c*[5:3], *a*});

покажет результат: 4′*b*1011;

Директива ***$write*** идентична директиве *$display*, однако она не осуществляет автоматический переход на новую строку в конце вывода информации. Если формат вывода не определен, то по умолчанию назначается десятичная форма представления.

Формат директивы ***$monitor*** аналогичен *$display*. Разница состоит в том, что вывод на экран формируется при любом изменении переменных из списка. Наблюдение за результатами моделирования может быть включено директивой ***$monitoron*** или отключено директивой ***$monitoroff***. По умолчанию в начале моделирования наблюдение за его ходом включено.

10.2. Окончание моделирования

Для организации окончания моделирования применяются директивы ***$finish*** и ***$stop***.

Директива ***$finish*** завершает моделирование и передает управление операционной системе. Эта директива всегда должна содержаться в одной из конструкций *initial*.

Применяются следующие способы оформления окончания моделирования.

1. *initial begin*
   # 10 *a*1 = 1′*b*1;
   ……
   # 20 *$finish*;
   end

В этом примере используется последовательный счет времени; окончание моделирования произойдет через 20 заданных временных единиц после последнего изменения переменной (или переменных).

2. *initial* # 100 *$finish*;

На этом примере остановимся подробнее. Здесь задано время моделирования. Если эта конструкция указана при моделировании, например, счетчика, то его работа будет проверяться в течение 100 единиц времени. Однако если проверяется, например, комби-

национное устройство, то будут промоделированы все заданные в тестовом модуле варианты, даже если сумма временных отрезков (# *T*) превышает заданное время - 100.

3. *initial $finish*;

Здесь окончание моделирования произойдет сразу после вычислений, соответствующих последнему изменению переменных (переменной). Такая запись аналогична следующей.

4. *initial begin*
   # 5 {*data*} = 4′*b*0010;
   ……...
   $finish;
   end

Директива ***$stop*** применяется достаточно редко. Она приостанавливает моделирование и переводит систему в интерактивный режим.

Глава 11. Проектирование БИС на основе автоматного представления

В рассмотренных ранее примерах применялись элементы теории алгебры логики (АЛ). В полной мере математический аппарат АЛ применим только к комбинационным схемам, в которых отсутствуют элементы памяти (ЭП). Поэтому для синтеза и анализа последовательностных устройств, содержащих ЭП (триггеры), необходимы сведения из теории цифровых автоматов (ЦА). Эта теория предоставляет разработчикам возможность использовать такие описания (модели) цифровых устройств, которые позволяют получить их быструю и эффективную реализацию.

В данной главе показывается, каким образом на основе теории цифровых автоматов можно получить модель последовательностного устройства и затем составить его Verilog-описание.

11.1. Элементы теории цифровых автоматов

В теории ЦА изучаются наиболее общие законы поведения устройств (или автоматов - согласно терминологии данной теории) без учета их конечной структуры и физической природы обрабатываемой информации.

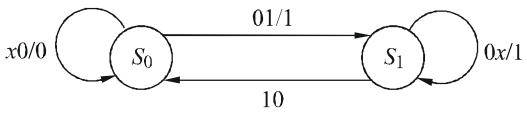


Рис.11.1. Автоматный граф для *RS*-триггера

Под цифровым автоматом понимается устройство, служащее для преобразования дискретной информации (т.е. перехода из одного состояния в другое) под воздействием входных сигналов и сохранения состояния в отсутствие последних [5, 6].

Основными формами (или моделями) описания работы автоматов являются табличные, графические и аналитические. Все формы описывают один и тот же алгоритм работы, поэтому существует взаимно-однозначное соответствие между всеми формами представления автомата.

Среди табличных форм в теории ЦА применяются таблицы истинности, таблицы переходов, таблицы состояний, таблицы выходов, таблицы работоспособности (или расширенная таблица) и совмещенные таблицы. Таблицы истинности неоднократно применялись в данном пособии. Таблица переходов составляется для одного триггера; в ней указывается, какие входные воздействия надо подать, чтобы получить нужный переход триггера из заданного состояния. Состояния ЭП называются внутренними состояниями автомата. Во всех остальных таблицах в разных вариантах (в зависимости от пожелания разработчика) указываются используемые входные наборы (входные состояния), соответствующие им состояния элементов памяти (как правило, значения сигналов на прямых выходах триггеров) и значения сигналов на выходах автомата (состояния выходов).

Граф автомата (или автоматный граф) - это ориентированный граф, в котором множеству вершин соответствует множество внутренних состояний, а множеству дуг - множество возможных переходов из одного состояния в другое. Например, для *RS*-триггера (его таблица истинности в синхронном варианте приведена на рис.7.3,*б*) автоматный граф приведен на рис.11.1. Здесь вершины соответствуют внутренним состояниям триггера (S_0 и S_1),

а дуги указывают на возможные переходы ЭП из одного состояния в другое под

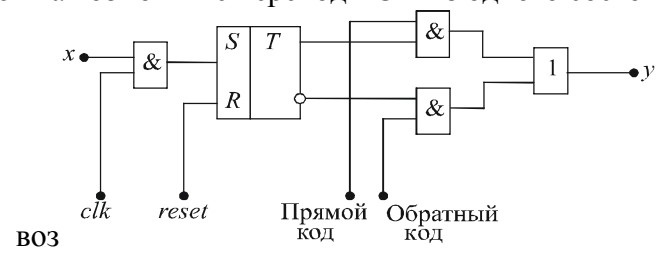


Рис.11.2. Разрядная схема параллельного регистра

действием входных сигналов. Запись на дугах означает: в числителе - состояния входов, в знаменателе - состояния выходов (в данном случае выходы автомата и ЭП совпадают).

Аналитической моделью цифрового автомата служит так называемый конечный автомат - КА (поэтому иногда теорию цифровых автоматов называют теорией конечных автоматов):

$$\text{КА} = \{M_x, M_y, M_s, \varphi, \psi),$$

где M_x, M_y и M_s - множество входных, выходных и внутренних состояний; φ - функция переходов; ψ - функция выходов.

Для представления информации принят алфавитный способ задания:

$X = \{x_0, x_1, \ldots, x_{m-1})$ - входной алфавит;

$Y = \{y_0, y_1, \ldots, y_{k-1})$ - выходной алфавит;

$S = \{s_0, s_1, \ldots, s_{n-1})$ - алфавит внутренних состояний.

Обычно рассматриваются два типа конечных автоматов: автомат Мили и автомат Мура.

Автомат Мили - это конечный автомат, работа которого описывается уравнениями:

$$\begin{aligned} S(t+1) &= \varphi\{s(t), x(t)\}; \\ Y(t) &= \psi\{s(t), x(t)\}, \end{aligned} \qquad (11.1)$$

где t - время.

Автомат Мура - это конечный автомат, работа которого описывается уравнениями:

$$\begin{aligned} S(t+1) &= \varphi\{s(t), x(t)\}; \\ Y(t) &= \psi\{s(t)\}. \end{aligned} \qquad (11.2)$$

Функция переходов φ реализует отображение $S \times X \to S$, а функция выходов ψ - отображение $S \times X \to Y$ (для автомата Мили) и $S \to Y$ (для автомата Мура).

Автомат Мили является более общим по сравнению с автоматом Мура. В теории цифровых автоматов доказывается, что с точки зрения выполняемой функции устройство может быть представлено как в виде автомата Мили, так и в виде автомата Мура. На практике наиболее часто встречаются автоматы Мура.

С точки зрения теории цифровых автоматов комбинационный автомат - это автомат вида

$$\text{КА} = \{M_x, M_y, M_s, \psi),$$

а последовательностный автомат - это конечный автомат, алгоритм функционирования которого описывается уравнениями вида (11.1) или (11.2).

В качестве примера автомата Мили можно привести разрядную схему параллельного регистра (рис.11.2), выполняющую функции сброса, приема и хранения информации, а также выдачи информации в прямом и обратном кодах. В этой схеме состояние выхода

зависит от состояния не только элемента памяти, но и входов "Прямой код", "Обратный код".

В качестве примера автомата Мура можно привести устройство счетчика, рассматриваемого в главе 14.

11.2. Структура Verilog-описания для конечных автоматов

Структурная модель автомата Мили представлена на рис.11.3. В этой модели можно выделить три части: логику переходов, регистр текущего состояния и логику формирования выходов.

Конечный автомат в каждый конкретный момент времени может находиться только в одном состоянии. Тактовый сигнал вызывает переход автомата из одного состояния в другое. Правила перехода и определяются комбинационной схемой, называемой логикой переходов. Следующее состояние определяется как функция текущего состояния и входных воздействий (11.1).

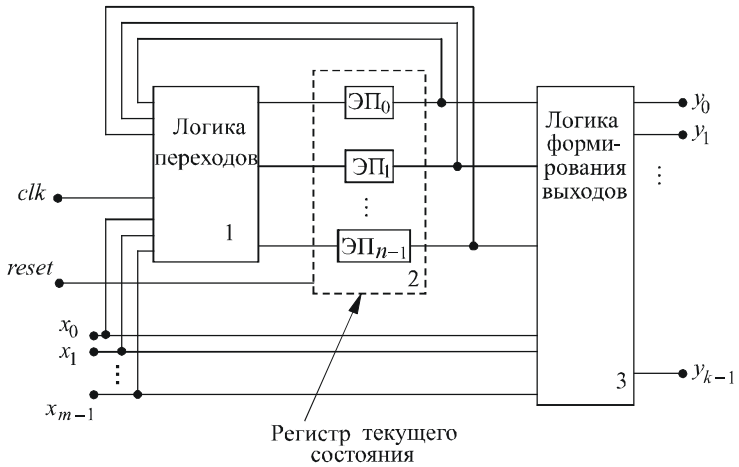


Рис.11.3. Структурная схема автомата Мили

Регистр текущего состояния (или триггерная подсистема) представляет собой набор тактируемых триггеров, которые используются для хранения текущего внутреннего состояния.

Определение объема памяти или числа триггеров, необходимых для реализации заданного алгоритма функционирования цифрового автомата, является одним из основных вопросов, возникающих при его проектировании. Очевидно, что этот вопрос однозначно связан с определением необходимого числа состояний автомата. Для n триггеров регистра текущего состояния автомата возможно 2^n состояний (каждый триггер может быть в одном из двух состояний: 0 и 1). Отсюда следует и обратное положение: если число состояний, необходимых для работы автомата, равно A, то триггерная подсистема должна содержать $n \geq \log_2 A$ триггеров, где n - ближайшее целое число. Неравенство возникает из-за того, что часть возможных при данном числе триггеров состояний может быть нерабочей. Например, при $n = 4$ возможно 16 состояний, из которых рабочими могут быть только 10.

Выход цифрового автомата определяется как функция текущего внутреннего состояния и входных воздействий. Эту функцию выполняет комбинационная схема, которая называется логикой формирования выхода.

Структурную модель автомата Мура можно получить на основании структурной модели автомата Мили, исключив из нее связи $x_0, x_1, \ldots, x_{m-1}$ с блоком 3.

Структурные модели автоматов Мили и Мура фактически отображают структуру Verilog-описания, которое можно составить, опираясь на приведенные к рис.11.3 комментарии.

Для функционирования цифрового автомата необходимы еще два сигнала: сигнал синхронизации *clk* и сигнал сброса *reset*.

Сигнал синхронизации инициирует переключение триггерной подсистемы. Формирование этого сигнала непосредственно связано с алгоритмом работы всего устройства и поэтому в каждом конкретном случае оговаривается особо.

Для обеспечения стабильной и безотказной работы используется сброс автомата в начальное состояние, как правило - нулевое (вход *reset*). Таким образом всегда обеспечивается инициализация автомата из заранее предопределенного состояния при первом тактовом сигнале. Если сброс не предусмотрен, невозможно предсказать, с какого начального состояния начнется функционирование, и это может привести к сбоям в работе всей системы. Эта ситуация особенно актуальна при включении питания системы. Если требуется начинать работу с произвольного состояния, то необходимы дополнительные входы предварительной установки (см. пример счетчика в главе 14). Как правило, применяется асинхронный вариант сброса и предварительной установки автомата.

Обычно при рассмотрении различных моделей цифровых автоматов сигналы сброса и синхронизации не указываются, но их наличие подразумевается, и в Verilog-описаниях они присутствуют уже в явном виде.

11.3. Примеры описаний на Verilog автоматов Мили и Мура

Пусть необходимо получить описание работы цифрового устройства, о котором известно следующее. В устройстве используется инверсный сброс и динамическое управление записью, оно имеет два входа, один выход и может находиться в одном из пяти внутренних состояний (S_0, S_1, …, S_4). Автоматный граф устройства представлен на рис.11.4. Так как число состояний автомата равно пяти, то для их реализации триггерная подсистема должна содержать, как указывалось, три триггера. Поскольку конкретные значения состояний в задании не оговорены, то примем их равными 000,001, 010, 011 и 100; за начальное примем состояние 000.

Представленный граф соответствует автомату Мили, о чем свидетельствует переход $S_4 \rightarrow S_0$. При этом переходе значение выхода является функцией не только внутреннего состояния, но и входов, что соответствует выражению (11.1).

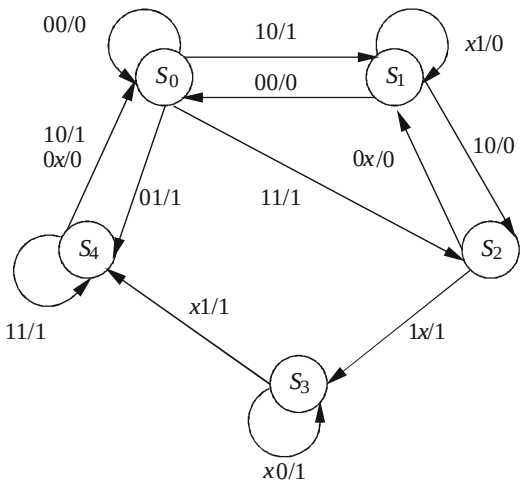


Рис.11.4. Автоматный граф заданного устройства
(в виде автомата Мили)

Поскольку триггерная подсистема содержит три триггера, то на них можно реализовать восемь состояний. Так как рабочими являются только пять состояний, то во всех остальных случаях определим внутреннее состояние устройства как начальное, а состояние выхода как нулевое.

Введем следующие обозначения:

data_in, data_out - входы и выходы устройства,

*s*0, *s*1, *s*2, *s*3 и *s*4 - внутренние состояния,

pres_state, next_state - текущее и следующее состояния устройства.

Описание функционирования заданного устройства на Verilog в виде базового модуля имеет вид

```
/* описание автомата Мили на Verilog */
module mealy (data_out, clk, data_in, reset);
output data_out;
input [1:0] data_in;
reg data_out;
reg [2:0] pres_state, next_state;
parameter s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3, s4 = 3'd4;
  /* при изменении способа кодирования  состояний или их значений меняется  только эта конструкция */
always @ (posedge clk or negedge reset)
begin: statereg
 /* в блоке statereg происходит управление регистром текущего состояния: сброс в начальное состояние или переход в новое  состояние */
    if (! reset)  pres_state <= s0;
    else  pres_state <= next_state;
end                              // конец блока statereg
always @ (pres_state or data_in)
    begin: cnsl                  // аббревиатура от Combinational Next State
                                 // Logic - логика переходов
    case(pres_state)
    s0: case(data_in)
        2'b00: next_state = s0;
        2'b01: next_state = s4;
        2'b10: next_state = s1;
        2'b11: next_state = s2;
        endcase
    s1: case(data_in)
        2'b00: next_state = s0;
        2'b10: next_state = s2;
        default: next_state = s1;
        endcase
    s2: case(data_in [1])
        1'b0: next_state = s1;
        1'b1: next_state = s3;
        endcase
    s3: casex(data_in)
        2'bx1: next_state = s4;
        default: next_state = s3;
        endcase
    s4: case(data_in)
        2'b11: next_state = s4;
        default: next_state = s0;
```

```
            endcase
        default: next_state = s0;      // для неиспользуемых состояний
        endcase
    end     // cnsl
    always @ (pres_state or negedge reset)
        begin: col_mealy            // аббревиатура от Combinational Output Logic -
                                    // логика формирования выхода
        case(pres_state)
        s0: case(data_in)
              2'b00: data_out = 1'b0;
              default: data_out = 1'b1;
            endcase
        s1: data_out = 1'b0;
        s2: casex(data_in)
            2'b0x: data_out = 1'b0;
            default: data_out = 1'b1;
            endcase
        s3: data_out = 1'b1;
        s4: casex(data_in)
            2'b1x: data_out = 1'b1;
            default: data_out = 1'b0;
        endcase
     default: data_out = 1'b0;      // для нерабочих случаев
    endcase
  end     // col_mealy
endmodule     // mealy
```

Рассмотренный автомат Мили можно преобразовать в эквивалентный по выполняемой функции автомат Мура. Однако это преобразование увеличивает количество внутренних состояний. Для рассматриваемого устройства один из возможных вариантов преобразования представлен на рис.11.5.

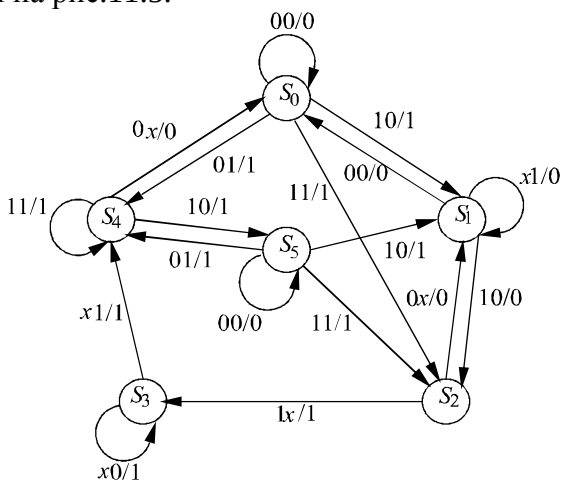


Рис.11.5. Автоматный граф заданного устройства
(в виде автомата Мура)

В данном случае для преобразования автомата Мили в автомат Мура необходимо преобразовать переход $S_4 \rightarrow S_0$, так как ему соответствуют два варианта входных и выходных значений. Оставим для этого перехода только один вариант (0*x*/0), а для реализации второго варианта (10/1) введем дополнительное внутреннее состояние S_5, которое в соответствии с теорией ЦА должно быть эквивалентно [6] одному из существующих состоя-

ний (в данном случае S_0). Это означает, что все переходы из состояния S_5 должны быть такие же, как и из S_0. Остальные состояния и переходы остаются без изменений. Присвоим новому состоянию значение 101.

Описание автомата Мура на Verilog (одинаковые с предыдущим описанием фрагменты опущены) имеет вид

```
/* описание автомата Мура на Verilog */
module moor (data_out, clk, data_in, reset);
……..
parameter s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3, s4 = 3'd4, s5 = 3'd5;
/* в описании блока statereg изменений нет */
   always @ (pres_state or data_in)
     begin: cnsl
       case(pres_state)
         s0: case(data_in)
         .....
         s4: casex(data_in )
              2'b0 x: next_state = s0;
              2'b10: next_state = s5;
              2'b11: next_state = s4;
              endcase
         s5: case(data_in)
              2'b00: next_state = s5;
              2'b01: next_state = s4;
              2'b10: next_state = s1;
              2'b11: next_state = s2;
              endcase
         default: next_state = s0;
         endcase
     end     // cnsl
   always @ (press_state)
     begin:  col_moor
        case(pres_state)
         s0: if (next_state == s0) data_out = 1'b0;
              else data_out = 1'b1;
         s1: data_out = 1'b0;
         s2: if (next_state == s1) data_out = 1'b0;
              else data_out = 1'b1;
         s3: data_out = 1'b1;
         s4: if (next_state == s0) data_out = 1'b0;
              else data_out = 1'b1;
         s5: if (next_state == s5) data_out = 1'b0;
              else data_out = 1'b1;
         default: data_out = 1'b0;
         endcase
     end    // col_moor
endmodule   // mealy
```

Глава 12. Автоматизированное проектирование цифровых схем с применением высокоуровневого описания на языке Verilog

Современные цифровые интегральные схемы (ИС) состоят из миллионов транзисторов. Разработать столь сложное устройство невозможно без применения систем автоматизации и особых подходов к проектированию.

12.1. Декомпозиция проекта

Одним из основных приемов проектирования, позволяющим разрабатывать схемы любой сложности, является декомпозиция - разбиение объекта проектирования на составные части. Декомпозиция применяется не только при разработке ИС, но и в программировании (выделение участков программного кода в отдельные файлы, использование процедур и функций, создание библиотек универсальных и часто применяемых функций и т.д.) и во многих других задачах, не связанных с электроникой.
При разработке ИС применяется термин "**функциональная декомпозиция**", отражающий суть применяемой декомпозиции - выделение функционально завершенных блоков (участков схемы). Под функциональной завершенностью понимается очевидная, легко формализуемая функция, выполняемая выделенным блоком устройства.
Функциональная декомпозиция выполняется на различных уровнях иерархии проекта, таких как:

- сложные функциональные блоки;
- иерархические модули высокоуровневого описания;
- библиотечные стандартные ячейки.

Применение декомпозиции проекта позволяет эффективно разделить задачу проектирования между несколькими разработчиками или коллективами.

Верхним уровнем декомпозиции являются **сложные функциональные блоки** (СФБ, или *IP*-блоки в англоязычном варианте, *IP - intellectual proprietary*). СФБ представляют собой самостоятельные устройства: ядра микроконтроллеров и процессоров, интерфейсы, устройства цифровой обработки сигналов, памяти и т.д., которые могут быть объединены в одном кристалле. СФБ не только составные части конечного проекта, сами по себе они являются законченным продуктом и товаром, востребованным на рынке.

Примером декомпозиции на средних уровнях может служить применение **Verilog-модулей**. Разработчик описывает и отлаживает относительно небольшие части проекта, а затем использует их на более высоких уровнях иерархии. Функциональная декомпозиция позволяет выделять многократно используемые участки схемы устройства и с помощью механизма ссылок многократно применять экземпляр единожды созданного и отлаженного описания.

Декомпозицией на низшем уровне можно считать применение **библиотек стандартных ячеек** (далее - библиотек). В качестве стандартных ячеек (элементов) библиотек выбираются простейшие логические вентили, выполняющие функции (обычно на 2 - 4 входа): И-НЕ, ИЛИ-НЕ, исключающее ИЛИ, а также некоторые их комбинации и более сложные элементы, такие как одноразрядные сумматоры, мультиплексоры, элементы памяти (триггеры) и т.п.

12.2. Представления стандартных ячеек

Каждая библиотечная ячейка содержит набор представлений (view), описывающих ее на разных модельных уровнях.

Логическое представление определяет логическую функцию, выполняемую данной ячейкой. Для описания функции библиотечной ячейки часто используется язык Verilog.

Представление физических характеристик ячейки содержит временные характеристики ячейки (задержки от входов к выходам, длины фронтов, специальные параметры для ячеек памяти: время установки и удержания данных и т.д.), энергетические характеристики (динамическая и статическая рассеиваемая мощность), площадь, занимаемая ячейкой на кристалле (для приблизительной оценки площади конечного устройства как функции суммы площадей элементов), и некоторые другие характеристики. Физические характеристики ячейки формируются на основании измерений или экстракции (получения характеристик полупроводниковых приборов и межсоединений из топологического описания, см. далее). Они задаются в виде аналитической или табличной зависимости характеристики от значений сигналов на входах, а также внешних условий (например, емкостных нагрузок входов и выходов).

Логическое описание элемента и его физические характеристики применяются при синтезе схемы из описания на языке высокого уровня, при оптимизации схемы на всех этапах проектирования, а также при функциональном и временном моделировании.

Схемотехническое представление определяет электрическую схему ячейки. Она представляется в виде *SPICE*-моделей, которые описывают содержащиеся в схеме элементы (транзисторы, емкости, сопротивления). Сама ячейка представляется в виде списка цепей (*net list*). Схемотехническое описание используется при решении многих задач. Это автоматизированный синтез топологии ячейки; проверка конечной топологии устройства с помощью сравнения экстрагированного списка цепей со списком цепей, полученным путем объединения схемотехнических описаний ячеек; точное физическое моделирование критических участков схемы; смешанное аналого-цифровое моделирование. В то время как все остальные представления библиотечных ячеек описывают их как "черный ящик", схемотехническое представление помогает разработчику разобраться в особенностях работы элемента.

Топологическое представление содержит геометрическое описание слоев полупроводниковой структуры ячейки. На его основе формируются шаблоны для изготовления микросхемы, определяются физические характеристики ячейки и параметры моделей элементов схемотехнического описания.

Способы описания представлений библиотечных ячеек, а также состав самих библиотек зависят от типов интегральных схем, для разработки которых они предназначаются.

12.3. Типы интегральных схем

В соответствии с особенностями проектирования и производства различают три вида интегральных схем: заказные, полузаказные и программируемые.

Заказные ИС предназначены для массового производства. Их изготовление осуществляется в ходе одного полного технологического цикла по специальному комплекту конструкторской и технологической документации. Проектирование подобной схемы выполняется на основе таких библиотек стандартных ячеек, которые обеспечивают максимальную плотность заполнения площади кристалла. В результате размеры и себестоимость кристалла заказной ИС по сравнению с функциональными аналогами ИС других типов минимальны. Но цикл "разработка - производство" для заказных ИС является

самым длинным и дорогостоящим, поэтому их производство целесообразно только в больших количествах.

Полузаказные ИС предназначены для средне- и малосерийного производства. Технологический цикл их разработки и изготовления разбит на несколько этапов. Сначала осуществляется создание базовых матричных кристаллов (БМК) в неразделенной пластине.

БМК представляет собой совокупность полупроводниковых структур (в основном транзисторов) одного или нескольких типов. Структуры существуют в виде ячеек с элементарными соединениями внутри или без них. Совокупность ячеек БМК образует периодическую структуру с "каналами трассировки" между ними. Для полузаказных ИС применяются библиотеки элементов, представляющих собой описание соединений полупроводниковых приборов, размещенных в ячейках. БМК в виде неразделенных пластин изготавливаются в технологическом маршруте массового производства и складируются.

На этапе изготовления полузаказной ИС со склада берутся пластины с БМК, и в условиях штучного или малосерийного (на уровне пластин) производства на них наносятся слои металлизации, реализующие топологию межсоединений согласно библиотечным элементам и проектируемому устройству.

Размер кристалла полузаказной ИС всегда больше, иногда значительно, функционально эквивалентной заказной ИС по следующим причинам:

- при самой оптимальной разработке никогда не удается использовать все транзисторы в БМК, так как их заранее заданное физическое размещение на поверхности кристалла невозможно оптимизировать в соответствии с электрической схемой;
- БМК выпускается в виде рядов с существенно различным числом ячеек. При выборе БМК, естественно, берется тот, у которого число ячеек больше требуемого.

Длительность процесса "разработка - производство" для полузаказных ИС измеряется неделями, причем она значительно короче аналогичного процесса для заказных ИС. В настоящее время многие производители выбирают именно этот тип ИС.

Программируемые логические интегральные схемы (ПЛИС) - это завершенные электронные изделия, не требующие для реализации конкретной схемы дополнительных промышленных технологических операций. Основу ПЛИС составляют массивы конфигурируемых ячеек (не путать с термином "библиотечная ячейка") с выполненными межсоединениями. Ячейки ПЛИС состоят из нескольких управляющих элементов (мультиплексоров, коммутирующих соединений, триггеров и т.д.) и одного или нескольких табличных элементов с таблицами в виде долговременной памяти, значения которых могут быть изменены. Конфигурирование такой схемы под конкретную задачу выполняется с помощью программирования - записи во внутреннюю память микросхемы конфигурации ячеек. Программирование ПЛИС выполняется с помощью специального устройства - программатора, подключаемого к ПК, и легко может быть выполнено самим проектировщиком.

ПЛИС применяются:

- при макетировании устройств с последующим переводом их в зависимости от планируемых объемов производства в заказные или полузаказные ИС;
- при штучном или малосерийном производстве.

Язык Verilog может быть использован для проектирования цифровых устройств всех типов интегральных схем. Но в отличие от заказных и полузаказных ИС для программируемых схем не существует понятия библиотек стандартных ячеек. Каждый производитель ПЛИС выпускает собственный пакет программного обеспечения САПР, реализующего алгоритм генерации конфигурации ячеек ПЛИС из описания на языке высокого уровня.

12.4. Проектирование заказных схем с помощью языка Verilog

Проектирование цифровой ИС с применением библиотек стандартных ячеек и высокоуровневого описания на языке Verilog состоит из этапов, изображенных на рис 12.1.

12.4.1. Разработка высокоуровневого описания схемы

Данный этап проектирования, наиболее важный, является самым трудоемким, поскольку в большинстве случаев он не может быть полностью автоматизирован (исключение составляют регулярные структуры, например блоки памяти). От описания схемы на языке высокого уровня во многом зависят результаты дальнейших этапов проектирования. Разработчику приходится учитывать не только эффективность описания алгоритмов разрабатываемого устройства, но и то, каким образом системы автоматизации проектирования смогут создать из этого описания конечную логическую и электрическую схему.

Помощь разработчику оказывают специализированные редакторы **систем автоматизированного проектирования (САПР)** (например, редактор, встроенный в систему *Active-HDL*), способные проверять правильность описания языковых конструкций Verilog и указывающие на ошибки.

12.4.2. Функциональная верификация

На этом этапе решается задача проверки правильности созданного описания алгоритмов функционирования схемы. Проверка выполняется с помощью САПР, поддерживающих язык Verilog. На входы тестируемого модуля подаются последовательности сигналов. Программа моделирования обрабатывает высокоуровневое описание схемы и формирует сигналы на выходах схемы. Разработчик вручную или с помощью специальных программ сравнивает выходные последовательности сигналов с эталоном (например, техническим заданием) и на основании полученных результатов принимает решение о правильности функционирования алгоритма. В случае несоответствия полученных сигналов эталонным необходим анализ результатов моделирования внутренних узлов описания для локализации ошибки.

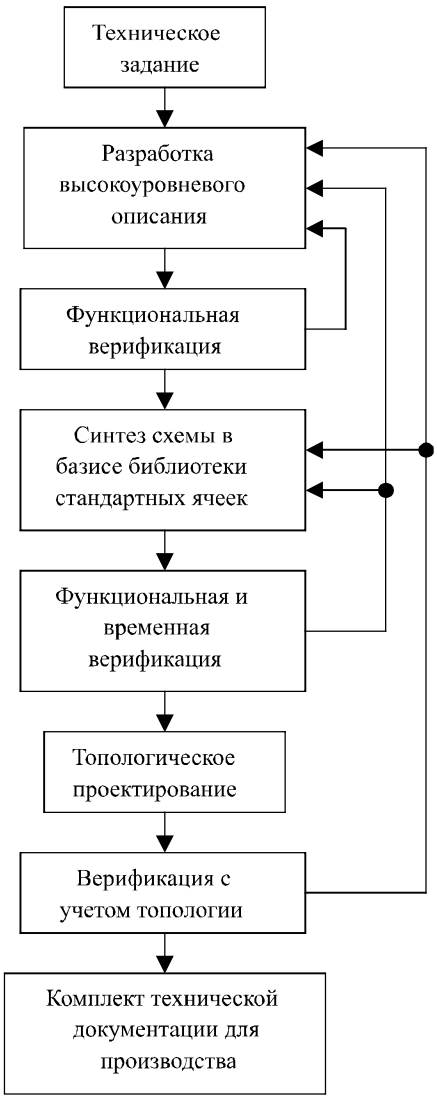


Рис.12.1. Маршрут проектирования цифровых ИС на основе библиотеки стандартных ячеек

Язык Verilog имеет широкий набор средств для описания входных тестовых последовательностей, а также для анализа результатов. Поэтому разработка тестовых входных последовательностей сигналов ведется в той же среде, что и всего проекта в целом.

12.4.3. Синтез схемы в базисе библиотеки стандартных ячеек

Синтезом схемы является в данном случае автоматизированная генерация из высокоуровневого описания проектируемой ИС (или ее фрагмента) логической схемы в базисе библиотеки стандартных ячеек. Эта задача решается с помощью системы логического синтеза (**синтезатора**).

Управление процессом синтеза со стороны разработчика происходит с помощью задания ограничений, таких как максимальная площадь схемы, временные характеристики схемы в целом и отдельных ее участков, максимальная потребляемая статическая и динамическая мощность, нагрузки входов и выходов головного модуля проекта. При подготовке проекта к синтезу следует учитывать особенности обработки синтезатором различных конструкций языка Verilog. Некоторые конструкции, правильные с точки зрения разработчика, могут быть преобразованы синтезатором в схемы, не соответствующие функциональным требованиям.

Первым этапом синтеза является перевод описания схемы во **внутренний формат синтезатора**. Каждый синтезатор имеет свой внутренний формат представления логических схем. Это обусловлено следующими причинами. Во-первых, большинство синтезаторов поддерживают несколько входных языков описания схемы (например, Verilog, EDIF, VHDL). В этом случае выгодно осуществить программную декомпозицию синтезатора - ввести внутренний формат представления логической схемы и использовать его в качестве связующего звена между системой синтеза и системой интерпретации входных языков описания схемы. Во-вторых, форма внутреннего представления схемы сильно влияет на эффективность процесса синтеза и является интеллектуальной собственностью разработчиков САПР. На этом этапе наибольшее значение имеет стиль высокоуровневого описания схемы.

Далее синтезатор выполняет отображение описания схемы в базис библиотеки. В результате должна получиться функционально правильная, но неоптимизированная схема. Следующим этапом синтеза является **оптимизация** (минимизация) схемы. Синтезатор анализирует различные варианты схемы (путем замены элементов), чтобы привести ее в соответствие с требованиями, ранее заданными разработчиком. Очень важно задавать ограничения, максимально приближенные к реальным условиям. Если задать слишком жесткие ограничения, то система логического синтеза, не получив варианта схемы, удовлетворяющего им, остановится на каком-то, как правило, неоптимальном варианте.

Для всех внутренних участков схемы синтезатор определяет ограничения автоматически, опираясь на характеристики ячеек, описанные в библиотеке. С помощью этого механизма можно формировать некоторые ограничения и для верхнего модуля проекта. Например, если указать системе логического синтеза, какие элементы будут подключены к внешним портам проекта, то информацию о нагрузках и некоторые временные параметры портов синтезатор сможет определить сам.

Одним из ограничений при решении задачи оптимизации является условное количество ресурсов (в первую очередь времени), которые система может использовать (с учетом быстродействия аппаратной системы). Поскольку оптимизация основывается на эвристических методах, то этот процесс может занять слишком большое время или вообще оказаться бесконечным. Поэтому синтезатор, затратив некоторое время, принимает решение остановиться и выбирает наиболее оптимальный из полученных вариант схемы. В большинстве систем синтеза разработчик может указать относительное количество времени, которое система должна затратить на решение задачи оптимизации. Чаще всего выбор сводится к трем вариантам: мало, средне или много.

Если к схеме предъявляются высокие требования и имеется достаточно времени, то оптимизация проводится дважды (**двухпроходная оптимизация**). Для первой оптимизации задаются заведомо завышенные ограничения, и по результатам синтеза определяются оптимальные ограничения. Вторая оптимизация выполняется с более точными ограничениями и большими затратами времени. Такой подход плохо автоматизируется и требует от разработчика некоторого опыта в определении оптимальных ограничений. В случаях двухпроходного синтеза имеет смысл тратить меньше времени на первый грубый проход, поскольку его результат в любом случае будет неточен из-за завышенных требований, и больше времени на второй - более точный.

С ограничением вычислительных ресурсов связана еще одна особенность синтеза, на которую разработчик должен обращать внимание. Для сложных устройств имеет смысл выполнять синтез не всего проекта сразу, а отдельных его частей - модульный синтез**.** **Модульный синтез** осуществляется следующим образом: выполняется синтез и грубая оптимизация отдельных модулей проекта, затем производится определение ограничений для межмодульных связей. Большинство систем синтеза определяют ограничения автоматически, поскольку известны характеристики соседних модулей. С полученными ограничениями выполняется точная оптимизация каждого модуля. В завершение проводится **ин-**

крементальная оптимизация (вторичная оптимизация, при которой используются результаты первичной) всего проекта.
Результатом работы на этапе синтеза является оптимизированный (минимизированный) вариант схемы устройства в базисе библиотеки стандартных ячеек.

12.4.4. Функциональная и временная верификация синтезированной логической схемы

Современные системы синтеза - очень мощные, надежные программные продукты, обеспечивающие высокую достоверность получаемой схемы и ее функциональное соответствие исходному описанию. Тем не менее к результатам синтеза необходимо применять функциональную и временную верификацию. Ошибки в алгоритм работы схемы могут внести два фактора: некорректность исходного высокоуровневого описания, которое в результате синтеза приводит к схеме, не соответствующей заданию, и влияние задержек элементов и межсоединений на функционирование схемы. Для обнаружения подобных ошибок после каждого этапа проектирования необходимо выполнять верификацию схемы.
Существует несколько основных способов **функциональной верификации** схемы.

1. Для схем небольшой сложности или отдельных модулей разработчик описывает последовательность входных тестовых сигналов, а затем с помощью системы моделирования выполняет расчет выходных сигналов схемы и анализирует результаты, представленные в виде диаграмм. Язык Verilog предоставляет гибкие средства для описания входных тестовых последовательностей, а большинство систем проектирования имеют интегрированные средства, позволяющие автоматизировать процессы подготовки описания тестов и моделирования. Например, система *Active-HDL* в автоматизированном режиме создает тестовый модуль на языке Verilog, в котором описываются входные тестовые сигналы, выполняет расчет выходных сигналов и выводит их в виде диаграмм.
2. Для более сложных схем визуальный анализ результатов моделирования затруднителен, поэтому они сравниваются с эталоном. В этом случае разработчик подготавливает последовательность входных тестовых сигналов и последовательность сигналов, которые он ожидает увидеть на выходах схемы, - эталон. Причем процесс сравнения можно автоматизировать стандартными средствами языка Verilog. Тогда система моделирования будет оповещать разработчика о наличии и моменте сбоя, а в случае соответствия результатов эталону сообщать лишь об удачном завершении. Поскольку такой способ верификации требует от разработчика больших усилий на этапе подготовки, позволяя, однако, сократить время анализа результатов, важно правильно оценивать, когда следует применять этот способ, а когда - предыдущий.

Следует отметить, что кроме задачи верификации схемы существует гораздо более сложная задача - контроль готовой микросхемы. Сложность заключается в недоступности внутренних узлов и ограниченном количестве внешних контактов. Тем не менее тестирование готовой микросхемы необходимо, и есть средства, позволяющие автоматизировать решение этой задачи (в данном пособии не рассматриваются). Эти же средства могут применяться и для верификации схемы на модельном уровне.

Для оценки требуемого быстродействия проводится **временная верификация** устройства. Поскольку последующие этапы проектирования вносят дополнительные задержки (задержки межсоединений и др.), необходимо иметь некоторый запас быстродействия.

Временную верификацию можно выполнить способами, описанными выше для функциональной верификации. Для этого по запросу разработчика система синтеза создает список цепей схемы и таблицу задержек. На основе полученной информации выполняется моделирование схемы уже с учетом задержек. Список цепей можно получить в виде описания схемы на языке Verilog, а таблицу задержек - в формате *SDF - Standard Delay For-*

mat. Все современные средства моделирования поддерживают оба этих формата. Однако такой способ верификации является весьма трудоемким (необходимо точно определять временные характеристики входных тестовых последовательностей) и не позволяет проводить поиск критических путей распространения сигнала.

Современные САПР имеют набор средств для эффективного временного анализа схемы. По команде разработчика система анализирует схему и выдает отчет обо всех или только о самых длинных путях распространения сигналов. В комбинационных схемах рассматриваются все возможные пути от входов к выходам, а в синхронных - пути на комбинационных участках схемы между синхронными триггерами. Можно также получить отчет только о путях распространения сигналов, не соответствующих требованиям.

Если результаты верификации (функциональной или временной) не соответствуют требованиям, то проводится частичный повторный синтез схемы с дополнительными ограничениями для критических цепей или изменение Verilog-описания схемы.

12.4.5. Размещение и трассировка топологических абстрактов библиотечных элементов

Топологический абстракт - это упрощенное топологическое представление библиотечного элемента, в котором содержится информация о размере ячейки, областях слоев металлизации, занятых
межсоединениями ячейки, и координатах ее контактов. Этой информации достаточно, чтобы выполнить размещение и трассировку элементов.

Поскольку вопросы топологического проектирования выходят далеко за рамки данного пособия, рассмотрим лишь основные моменты.

Для заказных схем применяется построчное размещение элементов. Один из геометрических размеров топологии элементов, например высота, выбирается фиксированным и равным для всех элементов библиотеки; при размещении элементы выстраиваются рядами.

Размещение и трассировка выполняются с теми же ограничениями характеристик схемы, которые заданы проектировщиком для синтеза. Система проектирования следит, чтобы выполнялись все топологические ограничения. Иногда (например, при слишком жестких ограничениях) САПР либо не сможет выполнить трассировку, либо выполнит с нарушением правил проектирования. В этом случае может потребоваться возврат к этапу синтеза или даже описанию схемы.

12.4.6. Экстракция и верификация характеристик схемы с учетом топологических особенностей

После решения топологических задач вновь должны быть выполнены функциональная и временная верификации (методами, описанными ранее) с учетом возможных изменений характеристик. Для этого из полученной топологии уточняется список цепей и таблица задержек элементов и межсоединений (экстракция параметров). Можно также получить точные значения емкостных нагрузок элементов.

Если характеристики полученной схемы удовлетворяют всем требованиям, то результаты топологического проектирования схемы сохраняются в специальных форматах и передаются на завод для изготовления опытного образца.

Затем образец тестируется, и при положительных результатах маршрут проектирования схемы можно считать завершенным.

12.5. Особенности проектирования полузаказных схем

Базовый матричный кристалл, в отличие от заказных схем, имеет заранее размещенные на кристалле полупроводниковые приборы. Однако маршрут проектирования полузаказной ИС на базе БМК, с точки зрения разработчика, является почти таким же, как и для проектирования заказной ИС.

Разница состоит в применяемой САПР и учете особенностей конкретного БМК:

1) заранее известное количество полупроводниковых приборов на кристалле и средний процент использования позволяют заблаговременно прогнозировать максимальный размер схемы;

2) точно известные характеристики приборов и фиксированные каналы трассировки позволяют создавать более точные модели и, следовательно, получать более достоверные результаты верификации;

3) задача размещения сводится к выбору набора транзисторов для реализации элемента.

12.6. Особенности проектирования программируемых логических схем

Маршрут проектирования ИС на основе ПЛИС сильно отличается от маршрутов для заказных и полузаказных схем. Поскольку в ПЛИС все элементы и межсоединения выполнены заранее, то задачи размещения и трассировки отсутствуют. Синтез схемы тоже отличен от других типов ИС.

Следует отметить, что все производители ПЛИС поставляют вместе с микросхемами собственные специализированные системы проектирования, включающие в себя полный набор средств разработки и верификации.

Проектирование начинается с разработки описания схемы на языке высокого уровня (например, Verilog) и ее функциональной верификации. Далее, как и в случае заказных и полузаказных схем, задаются ограничения схемы для системы синтеза.

После синтеза выполняется функциональная и временная верификация. Для этого применяются встроенные средства среды разработки (системы проектирования), немного отличающиеся друг от друга в зависимости от выбранного типа ПЛИС. Поскольку ПЛИС имеет заранее размещенные элементы и цепи межсоединений и их точные характеристики известны, то временная верификация дает очень точные результаты. Следовательно, гарантия работоспособности такой схемы после успешной верификации выше, чем у других типов схем.

Далее выполняется аппаратное программирование ПЛИС и тестирование схемы на макете.

Благодаря тому, что схемы на основе ПЛИС можно аппаратно реализовать за считанные минуты, они часто применяются для макетной отладки других типов схем, т.е. сначала описание схемы отлаживается на ПЛИС, а затем выполняется маршрут проектирования для другого типа. Многие производители БМК предоставляют средства, дающие возможность конвертировать описание схемы на основе ПЛИС в БМК, что позволяет обойтись без повторного цикла разработки и дает хорошо предсказуемые результаты.

Следует отметить, что самым трудоемким и ответственным этапом любого маршрута проектирования остается создание описания схемы.

Глава 13. Создание моделе- и синтезопригодного описания схемы на языке Verilog

Язык Verilog позволяет описывать устройство различными способами, что может привести к разным результатам при моделировании и синтезе. А один и тот же вариант описания может дать разные результаты до и после синтеза.

13.1. Моделе- и синтезопригодные описания

Под моделе- и синтезопригодностью описания будем понимать следующее. Во-первых, разработанное Verilog-описание схемы должно моделироваться и синтезироваться без ошибок. Во-вторых, результаты моделирования до и после синтеза должны соответствовать техническому заданию (функционально, по физическим характеристикам и др.). Для достижения этого результата нужен некоторый опыт работы. Иногда даже не совсем корректное описание может дать вполне приемлемый результат.
Например, описание комбинационного устройства имеет вид

```
// Пример1
always @ (d or clk)
if (clk)      q = 1'b0;
else if (e)  q = d;
```

Некорректность описания заключается в том, что внутри блока происходит считывание значения сигнала *e*, который не объявлен в списке чувствительности. В этом случае при моделировании не будет возникать событие, вызванное изменением значения сигнала *e* (событием считается любое изменение значения входа или внутреннего узла схемы). Это приведет к неправильным результатам моделирования. Однако большинство систем синтеза создадут на основе такого описания корректную схему, и результаты ее моделирования будут соответствовать техническому заданию (ТЗ).
Чтобы избежать недоразумений при моделировании до и после синтеза, достаточно использовать следующий корректный вариант описания приведенного примера:

```
// Пример2
always @ (d or clk or e)
    if (clk)      q = 1'b0;
    else if (e)  q = d;
```

В этом случае результаты синтеза будут идентичны Примеру 1, а результаты моделирования на всех этапах будут соответствовать ТЗ.
Следует помнить, что не все конструкции языка Verilog обрабатываются синтезатором и влияют на синтез логической схемы (т.е. являются **синтезопригодными**). Такие конструкции носят вспомогательный характер и игнорируются синтезатором. В результате неправильного применения **несинтезопригодных** конструкций может возникнуть несоответствие результатов моделирования до и после синтеза.
К синтезопригодным относятся конструкции со следующими ключевыми словами: ***and, always, assign, case, casez, casex, default, else, endcase, for, forever, if, inout, input, integer, nand, negedge, nor, not, or, output, parameter, posedge, reg, supply0, supply1, wand, wire, while, wor, xnor, xor.***
К несинтезопригодным относятся конструкции со следующими ключевыми словами: ***initial, highz0, highz1, real, repeat, strong0, strong1, time, weak0, weak1.***

Также в языке Verilog существуют операторы, непосредственно не участвующие в синтезе схемы. К ним относятся операторы отношения: >, >=, <, <= в том случае, когда они применяются во вспомогательных управляющих конструкциях (например, в цикле).
Часть ключевых слов не нарушают синтезопригодность описания и не влияют на результаты моделирования и синтеза. В основном они выполняют вспомогательные функции. Например, ключевые слова ***begin*/*end*** и ***module*/*endmodule*** структурируют и обеспечивают завершенность Verilog-описания.

13.2. Очередь событий при моделировании Verilog-описания

Все конструкции языка Verilog, управляющие значениями сигналов, при моделировании выполняются в определенной последовательности. Эта последовательность связана с такими понятиями, как событие и очередь событий. Механизм моделирования основан на событийном принципе.

В процессе моделирования выполняется расчет значений внутренних и выходных сигналов в зависимости от подаваемых входных тестовых последовательностей, т.е. происходят события.

Все события заносятся в очередь. Очередь событий формируется на основе пяти областей, определяемых стандартом IEEE Std 1364-2001:

1) **область активных событий**. Под активным понимается такое событие, которое происходит в текущий момент времени моделирования. Активные события выполняются первыми в произвольном порядке;

2) **область неактивных событий**. Под неактивным понимается такое событие, которое происходит в текущий момент времени моделирования и выполняется после всех активных событий. Отличия активных и неактивных событий описаны далее;

3) **область обновления результата неблокирующего назначения**. Такое обновление (назначение) называется **неблокирующим событием**. Оно содержит присвоение выходу или внутреннему узлу схемы значения, рассчитанного ранее, и должно быть выполнено в текущий момент времени моделирования после всех активных и неактивных событий;

4) **область мониторных событий**. Это область системных директив Verilog. Они выполняются последними в текущий момент времени;

5) **область будущих событий**. К будущим относятся все неактивные и неблокирующие события, запланированные на будущее время моделирования.

В соответствии с такой группировкой выполнение событий (моделирование) происходит в следующей последовательности:

1) выполняются все активные события;

2) если нет активных событий, неактивные события перемещаются в активную область и затем выполняются;

3) если нет активных и неактивных событий, неблокирующие события перемещаются в активную область и затем выполняются;

4) если нет активных, неактивных и неблокирующих событий, мониторные события перемещаются в активную область и выполняются;

5) если все события, запланированные на текущий момент времени, выполнены, то моделирование переходит на следующий момент времени (через заданный дельта-интервал).

Особенностью механизма моделирования является то, что события могут добавляться в активную область и выполняться затем в произвольном порядке. При этом возможна неоднозначность результатов
моделирования.

Например, пусть Verilog-описание содержит следующие четыре конструкции:

always @ (*q*) *p* = *q*; // 1
always @ (*q*) *p*2 = ~ *q*; // 2
always @ (*p or p*2) *clk* = *p* & *p*2; // 3
always @ (*posedge clk*)…; // 4

Зададим начальные значения: *q* = 0, *p* = 0, *p*2 = 1, *clk* = 0.

В данном примере возможны два варианта последовательности вычислений (моделирования) при изменении *q* из 0 в 1, представленные в табл.13.1, 13.2. Каждая таблица содержит два столбца. В столбце "обрабатываемое событие" показано обрабатываемое в текущий момент времени моделирования событие, в столбце "область активных событий" - текущее содержимое области активных событий. Номерами обозначены события, соответствующие конструкциям примера. События при назначении им нового результата обозначены по принципу <имя>(старое значение • новое значение).

Таблица 13.1

Первый вариант последовательности вычислений при моделировании

№ п/п	Обрабатываемое событие	Область активных событий
1		*q*(0• 1)
2	*q*(0• 1)	1, 2
3	1	*p*(0• 1), 2
4	*p*(0• 1)	3, 2
5	3	*clk*(0• 1), 2
6	*clk*(0• 1)	4, 2
7	4	2
8	2	*p*2(1• 0)
9	*p*2(1• 0)	3
10	3	*clk*(1• 0)
11	*clk*(1• 0)	<пусто>

Рассмотрим последовательность вычислений, описанную в табл.13.1. В начале моделирования в области активных событий находится переключение сигнала *q* из 0 в 1 (строка 1), вызванное внешним источником. В результате обработки этого события в область активных событий попадают конструкции 1 и 2 (строка 2). Далее выполняется конструкция 1 (строка 3) и т.д.

Таблица 13.2

Второй вариант последовательности моделирования

Обрабатываемое событие	Очередь активных событий
-	*q*(0• 1)
q(0• 1)	1, 2
1	*p*(0• 1), 2
2	*p*2(1• 0), *p*(0• 1)
p(0• 1)	3, *p*2(1• 0)
*p*2(1• 0)	3
3	<пусто> (*clk* не изменился)

Оба варианта корректны и могут иметь место. Однако в первом варианте сигнал *clk* переключился, а во втором - нет. Следовательно, во втором варианте не произойдет событие, запускающее обработку конструкции 4. В более сложных вариантах описания, содержащих не только комбинационные участки схемы, такая произвольность может привести к неправильному функционированию модели устройства.

Разработчик может управлять очередностью выполнения событий, помещая их в разные области. Выполняется это с помощью блокирующего и неблокирующего назначений.

Блокирующее назначение помещает событие в область активных, неактивных или будущих событий, что зависит от заданной задержки:

- блокирующее назначение без задержки попадает в область активных событий;
- блокирующее назначение с заданной задержкой 0 попадает в область неактивных событий;
- блокирующее назначение с ненулевой задержкой попадает в область будущих событий.

Неблокирующее назначение попадает либо в область обновления результата неблокирующего назначения, либо в область будущих событий. Неблокирующие назначения без задержки и с задержкой 0 трактуются одинаково.

Для выходов триггерных устройств должно использоваться только неблокирующее назначение. Это обеспечивает неизменность выходов триггеров до тех пор, пока не будут вычислены значения на их входах.

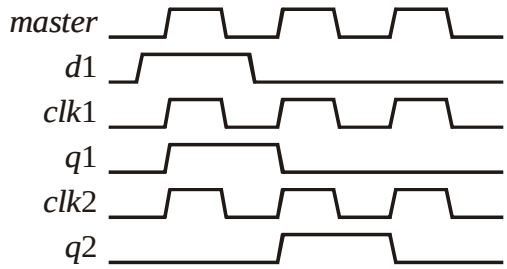


Рис.13.1. Временные диаграммы работы

сдвигового регистра для оператора неблокирующего присваивания

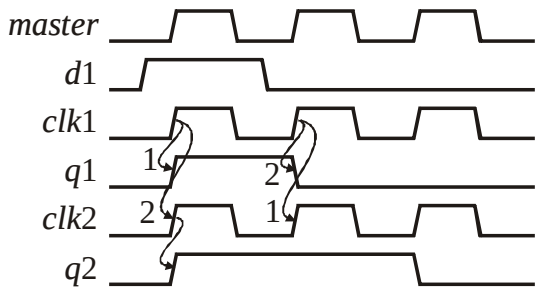


Рис.13.2. Временные диаграммы работы сдвигового регистра для оператора блокирующего присваивания

Рассмотрим пример использования блокирующего и неблокирующего назначений с помощью следующего фрагмента Verilog-описания:

```
always @ (master)        clk1 = master;
always @ (clk1)          clk2 = clk1;
always @ (posedge clk1)  q1 <= d1;
always @ (posedge clk2)  q2 <= q1;
```

Данное описание соответствует модели двухразрядного сдвигового регистра. На рис.13.1 приведена диаграмма результата его моделирования. Если заменить присваивания *q*1<= *d*1 и *q*2<= *q*1 на блокирующие, то может возникнуть ситуация, показанная на рис.13.2. В этом случае при переключении сигнала *clk*1 из 0 в 1 сначала может сработать конструкция *q*1 = *d*1, а затем *clk*2 = *clk*1 (поскольку оба события попадают в активную область и выполняются в произвольном порядке). Это приведет к записи данных сразу в оба

разряда регистра. При этом во время второго фронта 0-1 сигнала *clk*1 регистр может сработать правильно, что отражено на диаграмме. Возможна ситуация, при которой сигнал *q*2 вообще не переключится и сохранит нулевое значение.

13.3. Учет задержек при моделировании

В некоторых случаях возникает необходимость включения задержек в описание (модель) устройства [7]. Язык Verilog позволяет описывать задержки несколькими способами (см. главу 9), но не все из них моделируют задержку реального устройства.

Задержки могут указываться в операторах блокирующего и неблокирующего присваивания как справа, так и слева от оператора.

Варианты указания задержек для процедурного назначения:

```
always @ (a)
y <= # 5 ~a; // неблокирующее назначение, задержка указана справа;
always @ (a)
#5 y <= ~a; // неблокирующее назначение, задержка указана слева;
always @ (a)
#5 y = ~a; // блокирующее назначение, задержка указана слева;
always @ (a)
y = # 5 ~a; // блокирующее назначение, задержка указана справа.
```

Рассмотрим пример описания четырехразрядного сумматора, в котором задержка указана в правой части оператора неблокирующего присваивания.

```
// Пример 1
module sm_4bit (p_out, s, a, b, p_in);
    output p_out;
    output [3:0] s;
    input [3:0] a, b;
    input p_in;
    reg p_out;
    reg [3:0] s;
always @ (a or b or p_in)
    {p_out, s} <= # 12 a + b + p_in;
endmodule
```

На рис.13.3 приведена временная диаграмма результата моделирования для описания Примера 1. Оператор неблокирующего назначения с задержкой справа моделируется так, что при каждом изменении одного из операндов в тот же момент времени моделирования будет вычисляться результат операции. А назначение результата выходному сигналу будет помещено в очередь событий на 12 нс позже. Следовательно, каждое изменение одного из входных сигналов приведет к появлению на выходе схемы соответствующего переключения. Такой способ указания задержки соответствует работе реального устройства, а значит, является корректным и правильным.

```
// Пример 2
always @ (a or b or p_in)
    # 12 {p_out, s} <= a + b + p_in;
```

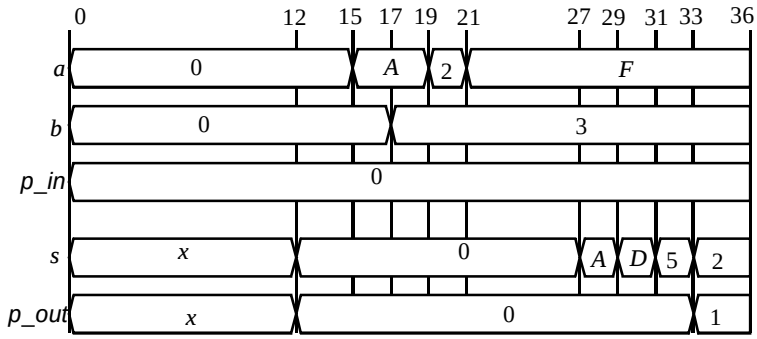


Рис.13.3. Временные диаграммы работы сумматора (Пример 1)

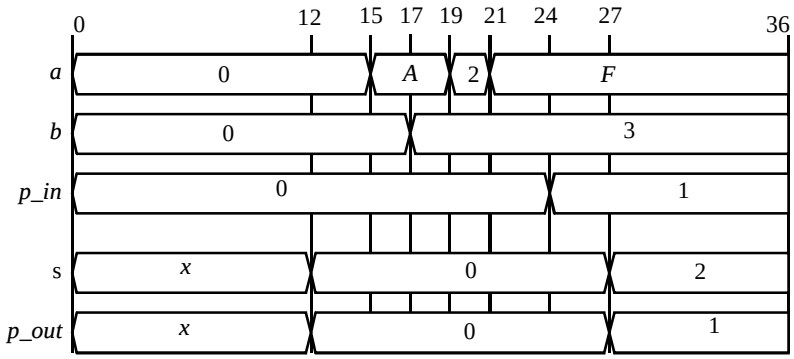


Рис.13.4. Временные диаграммы при указании задержки в левой части неблокирующего присваивания (Пример 2)

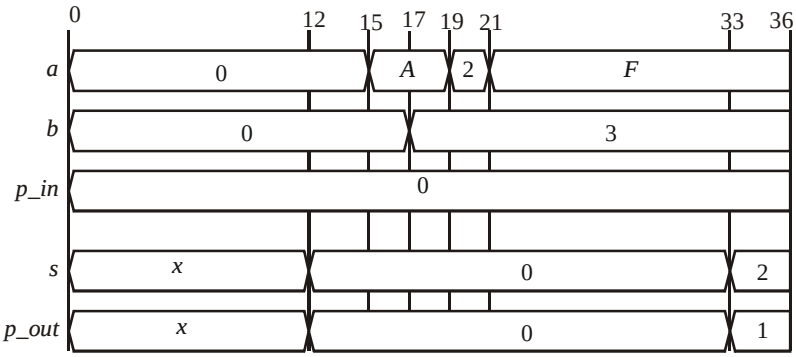


Рис.13.5. Временные диаграммы при указании задержки в непрерывном назначении (Пример 3)

При указании задержки в левой части неблокирующего назначения вычисление и назначение результата операции будут запланированы на будущий момент моделирования, соответствующий задержке (рис.13.4). Таким образом, через 12 нс после первого изменения (t = 15 нс) входного сигнала (например, a) произойдет вычисление значений выходных сигналов (t = 27 нс). Все изменения остальных входных сигналов, произошедшие в течение этих 12 нс, повлияют на выходы устройства без учета задержки. Следовательно, способ указания задержки в левой части неблокирующего присваивания не пригоден для описания устройств.

Указание задержки с блокирующим назначением также не дает результатов моделирования, соответствующих работе реальных устройств.

Задержку в левой части блокирующего назначения рекомендуется применять лишь при создании тестовых модулей для описания входных последовательностей.

При непрерывном назначении единственным корректным способом является указание задержки в левой части оператора.

```
// Пример 3
   module sm_4bit (p_out, s, a, b, p_in);
      output p_out;
      output [3:0] s;
      input [3:0] a, b;
      input p_in;
      assign # 12 {p_out, s} = a + b + p_in;
   endmodule
```

Такой способ указания задержки приведет к тому, что выходы схемы будут переключаться через 12 нс после последнего изменения входных сигналов. Диаграмма с результатами моделирования этого описания представлена на рис.13.5. Несмотря на то, что выходы схемы принимают правильные значения, такой способ указания задержки не позволяет проводить анализ переходных процессов в схеме и не рекомендуется для использования.

13.4. Использование конструкции *casex*

Конструкция *casex* трактует *x* как безразличное состояние, если оно встречается в операторе или в одном из условий. Проблемы с *casex* появляются, когда один из входных сигналов инициализируется в неопределенном состоянии. Моделирование, предшествующее синтезу схемы, обработает неопределенный входной сигнал как безразличный, а моделирование после синтеза обработает распространение неопределенного состояния входного сигнала через вентильную модель устройства.

Рассмотрим описание неполного дешифратора "2 - 4" с разрешающим сигналом.

```
module badcase (memce0, memce1, cs, en, addr);
   output memce0, memce1, cs;
   input en;
   input [31:30] addr;
   reg memce0, memce1, cs;
   always @ (addr or en) begin
      {memce0, memce1, cs} = 3'b0;
   casex ({addr, en})
         3'b101: memce0 = 1'b1;
         3'b111: memce1 = 1'b1;
         3'b0?1: cs = 1'b1;
   endcase
   end
endmodul
```

Если в начальный момент моделирования в приведенном примере разрешающий сигнал *en* будет находиться в неопределенном состоянии, то конструкция *case* ошибочно выполнит одно из условий, опираясь на значение адреса *addr*. Из-за этого при моделировании могут быть не видны некоторые проблемы, связанные с начальной установкой схемы, которые обнаружатся после синтеза. Аналогичная проблема возникнет и при неопределенном старшем разряде адреса, что приведет к установке *memce*0 или *memce*1 вместо *cs*.

Таким образом, при описании устройств рекомендуется избегать использования конструкции *casex*, так как ее применение может привести к ошибочной обработке случайного неопределенного сигнала.

Для моделирования устройств, в которых необходима обработка безразличного состояния, рекомендуется применять конструкцию *casez*.

13.5. Неполный перебор условий в конструкциях if и case

В конструкциях *if* и *case* неполный перебор условий может привести к тому, что в результате синтеза описание комбинационной схемы будет заменено защелкой (*latch* - триггер, управляемый уровнем) [8].

Рассмотрим пример:

```
always @ (data or gate)
    if (gate) q = data;
```

В результате синтеза приведенное описание превратится в защелку, поскольку из описания непонятно, что должно происходить с выходом схемы при *gate* = 0. Чтобы избежать подобной ситуации, необходимо либо явно описывать альтернативное состояние условного оператора, либо инициализировать значение выхода перед конструкцией *if*:

```
always @ (data, gate)      //  первая рекомендация
begin if (gate) q = data;
    else q = 0;
end
always @ (data, gate)      // вторая рекомендация
begin q = 0;
    if (gate) q = data;
end
```

Аналогичные рекомендации даются относительно оператора *case*: необходимо либо явно указывать полный набор условий, либо использовать ключевое слово *default*.

Итак, рассмотрены лишь некоторые, наиболее важные аспекты создания моделе- и синтезопригодного описания на языке Verilog. Для создания корректного описания также зачастую приходится учитывать особенности применяемой системы автоматизированного проектирования, с помощью которой выполняется разработка.

Глава 14. Примеры Verilog-проектов для различных устройств

В данной главе рассматриваются примеры описаний на Verilog как комбинационных, так и последовательностных устройств. Это описания устройств, реализующих различные функции на основе дешифратора и мультиплексора; два варианта описаний функционирования синхронного суммирующего счетчика и описание параллельного четырехразрядного регистра.

В главах 5 - 7 приводились различные способы описания функционирования дешифратора и мультиплексора. Эти схемы в общем случае являются универсальными, т.е. на их основе может быть реализована любая логическая функция (ЛФ). В настоящее время они находят широкое применение в программируемых устройствах.

Рассмотрим построение произвольной ЛФ на основе дешифратора (см. рис.7.1,*б*,*в*). На выходах *DC* вырабатываются все минтермы [3, 4], которые можно составить из данного числа аргументов. С другой стороны, любую логическую функцию можно представить в совершенной дизъюнктивной нормальной форме (СДНФ). Собирая с выходов дешифратора нужные (в соответствии с заданной ЛФ) минтермы и подавая их на ЛЭ ИЛИ, можно получить реализацию этой функции. Если ЛФ задана в виде таблицы истинности, то, минуя представление ее в виде СДНФ, можно лишь указать номера выходов *DC* (в соответствии со входными наборами), на которых функция принимает значение единицы.

Пример 1. Пусть требуется реализовать на основе дешифратора логическую функцию, зависящую от трех аргументов. Функция принимает значение единицы на наборах: 3, 5, 6 и 7. Составим описание ее работы на Verilog. Описание включает в себя базовый, головной и тестовый модули. При этом воспользуемся базовым модулем dc3_8, приведенным в параграфе 7.3.

```
/*  Описание  функционирования  произвольной  ЛФ  на  основе
DC "3 - 8" */
module log_f_dc (out_f, in_f);   // головной модуль
    output out_f;
    input [2:0] in_f;
    wire [7:0] dc_out;   // внутренние цепи
    dc3_8 ex (dc_out, in_f);
    or_base1 (out_f, dc_out[7], dc_out[6], dc_out[5], dc_out[3]);
endmodule   // log_f_dc
timescale 1ns/1ps   // тестовый модуль
module log_f_dc_tb;
    wire out_f;
    reg [2:0] in_f;
    log_f_dc UUT (out_f, in_f);
    initial in_f = 3'b000;
    always # 10 in_f[0] = ~ in_f[0];
    always # 20 in_f[1] = ~ in_f[1];
    always # 40 in_f[2] = ~ in_f[2];
    initial # 100 $finish;
endmodule   // log_f_dc_tb
```

Чтобы головной модуль не содержал никакой логики, можно функцию ИЛИ описать в отдельном базовом модуле.

Рассмотрим реализацию произвольной логической функции на основе мультиплексора (*MUX*). Для этого применяются два способа. Остановимся на одном из них. Суть его заключается в следующем. На адресные входы мультиплексора подается часть аргументов

воспроизводимой функции (в общем случае произвольным образом). Информационные входы *MUX* выполняют роль так называемых настроечных, на которые подаются либо логические константы, либо переменные, либо вспомогательные функции ($F_{настр}$).

Пример 2. Пусть требуется реализовать на мультиплексоре "4 - 1" следующую логическую функцию:

$$F(X) = X_1\overline{X}_0 \vee X_2 \vee \overline{X}_1X_0 .$$

Выберем в качестве адресных входов переменные: X_2 (*a*[1], см. рис.5.1,*а*) и X_0 (*a*[0]). Тогда сигналы настройки определим на основании приведенной формулы и сведем их в следующую таблицу:

X_2	X_0	$F_{настр}$
0	0	X_1
0	1	$\overline{X}_1$
1	0	1
1	1	1

Таким образом, для реализации функции $F(X)$ на информационный вход мультиплексора *d*[0] подается переменная X_1 (см. рис.5.1,*б*), на *d*[1] - $\overline{X}_1$, а на *d*[2] и *d*[3] подается сигнал логической единицы. Переменным в тексте присвоено имя *in_F*.

Составим Verilog-описание работы заданной функции. Описание состоит из базового, головного и тестового модулей. В качестве базового воспользуемся описанием мультиплексора "*mux* 4_1_*as*" из параграфа 6.2.

```
/*  Описание  функционирования  произвольной  ЛФ  на  основе
MUX  "4 - 1" */
module log_f_mux (out_F, in_F);   // головной модуль
    output out_F;
    input [2:0] in_F;
    mux 4_1_as ex (out_F, in_F[2], in_F[0], in_F[1], ~ in_F[1], 1, 1);
endmodule   // log_f_mux
timescale 1ns/1ps   // тестовый модуль
module log_f_mux_tb;
    wire out_F;
    reg [2:0] in_F;
    log_f_mux UUT (out_F, in_F);
    initial in_F = 3'b000;
    always # 10 in_F[0] = ~ in_F[0];
    always # 20 in_F[1] = ~ in_F[1];
    always # 40 in_F[2] = ~ in_F[2];
    initial # 100 $finish;
endmodule   // log_f_mux_tb
```

Пример 3. Рассмотрим построение Verilog-проекта для синхронного суммирующего двоично-кодированного счетчика по модулю 13. Счетчик (рис.14.1) производит сложение поступающих на счетный вход (*increment*) импульсов. Его автоматный граф изображен на рис.14.2.

В счетчике предусмотрена возможность предварительной установки любого из возможных состояний (см. рис.14.2). Предварительная установка производится по входам *data* при разрешающем сигнале *load_data* = 1. Описание устройства осуществим на уровне вентилей и триггеров.

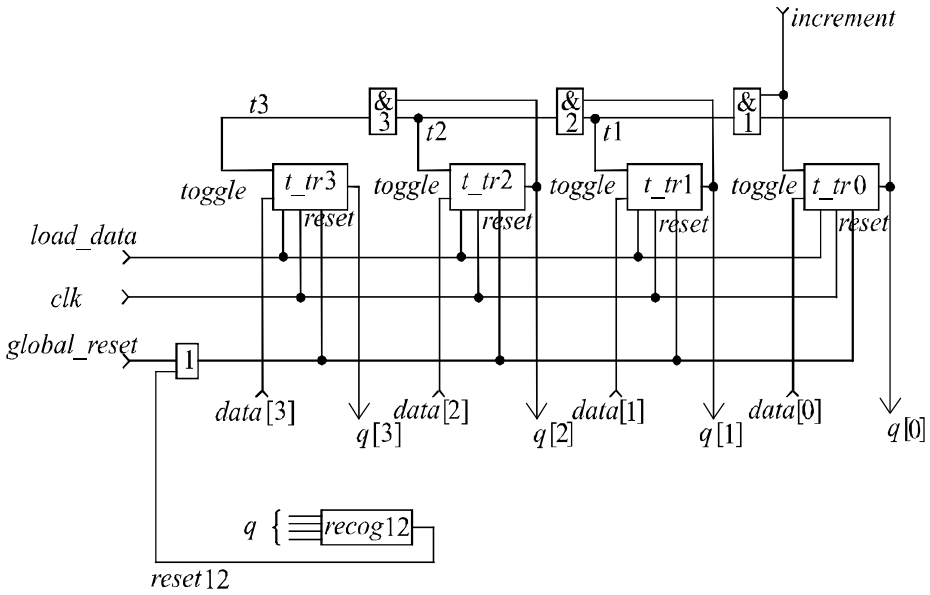


Рис.14.1. Функциональная схема счетчика по модулю 13 на сложение

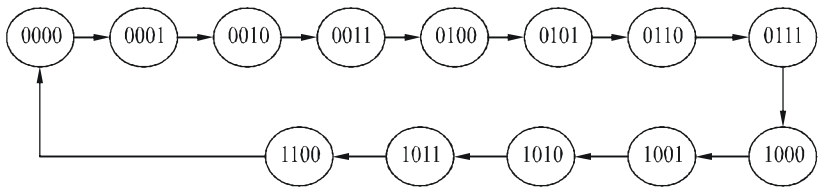


Рис.14.2. Автоматный граф счетчика

Счетчик построен по принципу двоичных устройств. Он состоит из четырех одинаковых разрядных схем. Заданный модуль счета обеспечивает комбинационная схема (*recog* 12), которая при появлении числа 12 на выходах переводит счетчик в 0, и счет начинается заново.

Каждый разряд счетчика представляет собой *T*-триггер (рис.14.3),

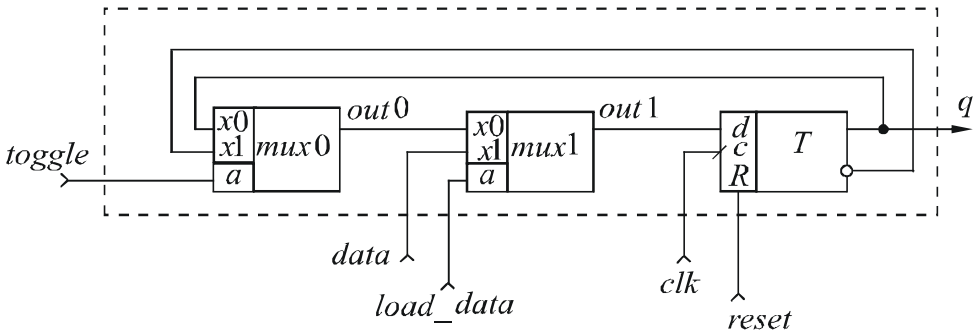


Рис.14.3. Функциональная схема *T*-триггера
с динамическим управлением записью

построенный на основе *D*-триггера с динамическим управлением записью и асинхронным сбросом. Если сигнал на входе *reset* = 1, то выход *q* = 0. Иначе при разрешающем сигнале на входе *clk* (вернее, при переходе его из 0 в 1) выход принимает значение сигнала, действующего на его информационном входе *d*: *q* = *d*.

Входной сигнал *d* формируется следующим образом. Если сигнал *load_data* = 1 (производится предварительная установка), то на выход мультиплексора *mux*1 проходит сиг-

нал со входа *x*1, т.е. *d* = *data*; иначе *d* принимает значение сигнала с выхода мультиплексора *mux*0. В этом случае, если сигнал на счетном входе *mux*0 *toggle* = 1, $d = \overline{q}$, иначе (*toggle* = 0) *d* = *q* (режим хранения).

toggle	q_{n+1}
0	q_n
1	$\overline{q}_n$

Описание счетчика проведем от базовых модулей к головному (снизу-вверх). Базовыми здесь являются модули, описывающие работу мультиплексора (*mux*), *D*-триггера (*d_tr*) и схемы, формирующей сигнал *reset*12 (*recog*12). В субмодуле описывается работа *T*-триггера.

И наконец, в головном модуле описывается весь счетчик.

```
/* Описание счетчика*/
module mux (out, a, x0, x1);
    output out;
    input a, x0, x1;
    assign out = a ? x1 : x0;
endmodule   // mux

module d_tr (q, clk, d, reset);
    output q;
    input clk, d, reset;
    reg q;
    always @ (posedge clk or posedge reset)
        if (reset) q <= 0;
        else q <= d;
endmodule   // d_tr

module t_tr (q, clk, data, load_data, reset, toggle);
    output q;
    input clk, data, load_data, reset, toggle;
    wire out0, out1;
    mux mux0 (out0, toggle, q, ~q);
    mux mux1 (out1, load_data, out0, data);
    d_tr d_tr0 (q, clk, out1, reset);
endmodule   // t_tr

module recog12 (out_recog12, in_recog12);
    input [3:0] in_recog12;
    output out_recog12;
    assign out_recog12 = (in_recog12 == 4'b1100);
endmodule   // recog12

module counter (q, clk, data, increment, global_reset, load_data);
    output [3:0] q;
    input [3:0] data;
    input clk, increment, global_reset, load_data;
    wire t1, t2, t3;
    wire reset, reset_12;
    t_tr t_tr0 (q[0], clk, data[0], load_data, reset, increment);
    t_tr t_tr1 (q[1], clk, data[1], load_data, reset, t1);
    t_tr t_tr2 (q[2], clk, data[2], load_data, reset, t2);
```

```
    t_tr t_tr3 (q[3], clk, data[3], load_data, reset, t3);
    and and1 (t1, q[0], increment);
    and and2 (t2, q[1], t1);
    and and3 (t3, q[2], t2);
    recog12 recog12_1 (reset12, q);
    or or1 (reset, global_reset, reset12);
endmodule   // counter

/* тестовый модуль для счетчика */
`timescale 1ns/10ps
module counter_tb;
    wire [3:0] q
    reg [3:0] data;
    reg clk, increment, global_reset, load_data;
    counter UUT (q, clk, data, increment, global_reset, load_data);
    initial begin
        {clk, increment, load_data, data, global_reset} = 0;
    initial forever # 25 clk = ~ clk;
    initial begin
        # 5 global_reset = 1;   // проверка установки счетчика в 0
        # 5 global_reset = 0;
        # 5 load_data = 1;   // проверка предварительной установки
            data = 4'b0011;
        # 5 load_data = 0;
    end
    initial begin
        # 20 increment = 1;   // проверка счета
        # 300 $finish;
    end
endmodule   // counter_tb
```

Пример 4. Приведем описание этого же счетчика на поведенческом уровне. Описание состоит из одного модуля; тестовый модуль можно взять из предыдущего примера.

```
module counter (q, clk, data, increment, global_reset, load_data);
    output [3:0] q;
    input [3:0] data;
    input clk, increment, global_reset, load_data;
    reg [3:0] q;
    always @ (posedge clk or posedge global_reset)
        if (global_reset) q <= 4'b0000;
        else if (load_data) q <= data;
                else if (increment)
                    if (q == 12) q <= 0;
                    else q< = q + 1;
                else   ;
endmodule      //  counter
```

При наличии достаточно мощных средств синтеза поведенческое описание позволяет получить более наглядное описание объекта.

Пример 5. Рассмотрим описание параллельного четырехразрядного регистра с динамическим управлением записью (рис.14.4). Регистр построен на *D*-триггерах, снабжен входом для асинхронного инверсного сброса и выполняет прием и хранение информации. Все описание состоит из базового и тестового модулей.

```
module rg (q, clk, data, reset);   // описание базового модуля
```

```
    output [3:0] q;
    input [3:0] data;
    input clk, reset;
    reg [3:0] q;
    always @ (posedge clk or negedge reset)
        if (~reset) q<= 4'b0000;
        else q<= data;
endmodule   // rg
`timescale 1ns/1ps   // описание тестового модуля
module rg_tb;
wire [3:0] q;
reg [3:0] data;
reg clk, reset;
rg UUT (q, clk, data, reset);
initial {clk, data, reset} = 0;   // проверка сброса регистра в 0
initial forever # 10 clk = ~ clk;
initial begin
        # 5 data = 4'b1001; // проверка режима записи
        # 15 data = 4'b0111; // запись не должна производиться, так как
                             // нет разрешения
        # 25 data = 4'b1100;   // запись должна быть
        # 50 $finish;
    end
endmodule   // rg_tb
```

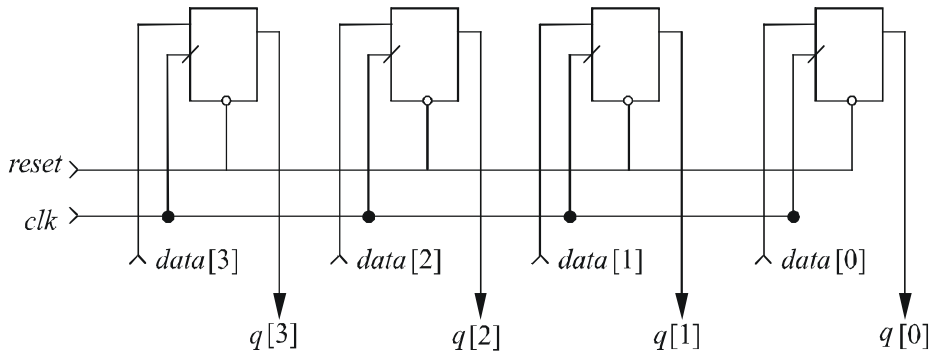


Рис.14.4. Функциональная схема четырехразрядного регистра

Литература

1. ***Поляков А.К.*** Языки VHDL и Verilog в проектировании цифровой аппаратуры. - М.: СОЛОН-Пресс, 2003. - 313 с.
2. ***Стешенко В.Б.*** EDA. Практика автоматизированного проектирования радиоэлектронных устройств. - М.: Нолидж, 2002. - 766 с.
3. ***Угрюмов Е.П.*** Цифровая схемотехника. - СПб.: БХВ-Петербург, 2002. - 528 с.
4. ***Попова Т.В.*** Основы логического проектирования интегральных схем. - М.: МИЭТ, 2002. - 91 с.
5. Автоматизация проектирования БИС / ***Под ред. Г.Г. Казеннова:*** В 6 кн. - Кн. 2: ***Савельев П.В., Коняхин В.В.*** Функционально-логическое проектирование БИС. - М.: Высшая школа, 1990. - 156 с.
6. ***Киносита К., Асада К., Карацу О.*** Логическое проектирование СБИС. - М.: Мир, 1988. - 308 с.
7. ***Clifford E.*** Cummings. Correct Methods For Adding Delays To Verilog Behavioral Models. Sunburst Design, Inc 2001. http://www.sunburst-design.com/papers/CummingsHDLCON1999_BehavioralDelays_Rev1_1.pdf
8. Mentor Graphics ModelSim Verilog/PLI Technology notes http://model.com/support/technotes.asp?id=103

Приложение 1. Ключевые слова и символы языка Verilog,их названия, назначения, форматы

<разрядность>'<основание> <число> - формат записи целого числа, в котором разрядность определяет количество бит под представление числа, а основание определяет выбранную систему счисления:

b или *B* - двоичную,

d или *D* - десятичную,

h или *H* - шестнадцатеричную,

o или *O* - восьмеричную.

Десятичные числа могут быть записаны без указания разрядности и основания.

x, z - символы для обозначения неопределенного и высокоимпедансного состояния, указываются вместо цифры в числах; например: $4'b1x01$.

integer <имя величины или перечень имен через запятую>; - объявление целых величин.

real <имя величины или перечень имен через запятую>; - объявление действительных величин.

parameter <имя параметра> = <численное значение параметра>; - объявление параметра.

input <имя порта или перечень имен через запятую>; - объявление портов как входов.

output <имя порта или перечень имен через запятую>; - объявление портов как выходов.

inout <имя порта или перечень имен через запятую>; - объявление порта как двунаправленного элемента.

wand <имена входов и выхода через запятую>; - объявление объединения выходов как монтажного И.

wor <имена входов и выхода через запятую>; - объявление объединения выходов как монтажного ИЛИ.

***supply* 0** <имя узла подключения заземления>; - объявление узла подключения источника заземления.

***supply* 1** <имя узла подключения источника питания>; - объявление узла подключения источника питания.

wire <имя порта или перечень имен через запятую>; - назначение данным типа цепь (назначается также по умолчанию).

reg <имя порта или перечень имен через запятую>; - назначение данным типа "регистр"; обязательно назначается данным, определяемым в блоках *always* и *initial.*

[N_2:N_1] <имя порта или перечень имен через запятую>; - объявление порта векторной величиной; N_2 - старший разряд.

always @ (<имя переменной 1> *or* <имя переменной 2> *or*…) - блок *always*; запись означает, что следующее за ней действие выполняется при изменении хотя бы одной переменной из списка (называется списком чувствительности) до тех пор, пока процесс моделирования не будет остановлен директивами *$finish* или *$stop*. Вместо *or* может использоваться запятая. Блок *always* работает только с данными типа *reg*.

***always* # <*T*>** <оператор> - одна из форм блока *always*, применяется при изменении одной переменной.

 # <*T*> - признак задержки, где *T* может быть целым, действительным числом или функцией.

posedge, negedge - ключевые слова для обозначения перехода сигнала из 0 в 1 и из 1 в 0 соответственно; применяются в списке чувствительности блока *always.*

initial begin <группа операторов> ***end*** - блок инициализации, с помощью которого осуществляется моделирование; включает в себя группу операторов, которые будут выполняться с момента старта моделирования.

initial <оператор> - частный случай блока *initial* для одного оператора.

begin [: <имя блока>] .... ***end*** - операторные скобки, в которые заключается группа операторов; применяется в конструкциях *always, initial* и др.

или

begin ... end

module <имя модуля> (<список портов>); - формат первой строки базовых, головного и субмодулей.

endmodule - ключевое слово, завершающее описание модуля.

<имя модуля> <имя экземпляра модуля> (<список портов>); - описание экземпляра модуля или обращение к модулю нижнего уровня из модуля верхнего уровня (ссылка на внутренний модуль).

(. <имя порта в модуле> (<имя порта в экземпляре модуля>),...)); - список портов при использовании прямого назначения (применяется при описании экземпляра модуля).

`*timescale* <*N*1> [*ns* или *ps*] / <*N*2> [*ps* или *ns*] - первая строка тестового модуля.

*N*1 - масштабный коэффициент, соответствующий числам с символом #;

*N*2 - точность моделирования.

module <имя модуля>*_tb* - формат записи для обозначения тестового модуля; ***tb*** (*TestBench*) - обязательная составная часть имени тестового модуля.

<ключевое слово> <имя> (<список выходов> <список входов>); - формат для описания встроенных примитивов.

Перечень ключевых слов:

and - ЛЭ И; один выход и несколько входов;

nand - ЛЭ И-НЕ; один выход и несколько входов;

or - ЛЭ ИЛИ; один выход и несколько входов;

nor - ЛЭ ИЛИ-НЕ; один выход и несколько входов;

xor - ЛЭ исключающее ИЛИ; один выход и несколько входов;

xnor - ЛЭ исключающее ИЛИ-НЕ; один выход и несколько входов;

buf - буферный элемент; один вход и несколько выходов;

not - ЛЭ НЕ; один вход и несколько выходов.

assign <имя переменной> = (операторы Verilog); значение переменной вычисляется заново каждый раз при изменении хотя бы одного оператора в правой части; конструкция работает только с переменными типа *wire*.

if (<условие>) <1-й оператор> ***else*** <2-й оператор>; - оператор условного перехода; вычисляется условие, и в зависимости от результата выполняется либо первый оператор (если результат - истина), либо второй (ложь); если используется группа операторов, то она заключается в операторные скобки *begin ... end.*

case ({объединяемые величины}) <условие1>: <оператор1>; ……. ***default*:** <оператор>; ***endcase*** - оператор выбора (разновидности *casex, casez*); операторы выполняются в зависимости от условия; условие является численной величиной; в *default*

указывается неучтенная в условиях ситуация; несколько операторов заключаются в операторные скобки. Допускается вложенность операторов *if* и *case*: *if ... case* или *case ... if*.

<условное логическое выражение> ? <истинно> : <ложно>; - оператор ветвления; запись *y* = *s* ? *x* : *r* означает что если *s* - истинно, то *y* = *x*, иначе *y* = *r*.

while (<условие>) - оператор цикла; следующие за *while* операторы выполняются, пока заданное условие истинно.

for (<инициализация переменной>;<условие продолжения>;<увеличение переменной>) - оператор цикла; дает возможность инициализировать, проверять и увеличивать индексную переменную явным способом.

repeat (<условие>) - оператор цикла с заданным числом повторений.

forever <установка> <оператор>; - оператор вечного цикла; вычисления повторяются до конца моделирования.

$display***[<ключевой символ формата>] (<строки>, <выражения>, <переменные>); - директива для вывода результатов моделирования; по умолчанию ***$displayd.

$monitor - формат аналогичен *$display*; в отличие от *$display* вывод формируется при любом изменении переменных.

$finish - директива для обозначения окончания моделирования.

$stop - директива приостанавливает моделирование и переводит систему в интерактивный режим.

Приложение 2. Примеры записей на языке Verilog

```
a = 4'b0111;                          c = 4'd5;
in_reg = 8'b10x1_0011;                data = 4'b0z01;
grnd = 0;                             integer n;
real p, r;                            parameter st1 = 3'd1;

input d, clk;                         input [3:0] in_dc;
input [3:0] d, [1:0] a;               input [3:0] a, c;
output out_psm;                       output [7:0] out_dc;
inout vent;                           supply0 grnd;
supply1 e;

wire prm1, prm2;                      wire [1:0] prm;
reg out_q;                            reg [7:0] out_dc;

always @ (r, s, clk)                  always @ (reset)
always @ (posedge clk or negedge reset)
always # 10 clk = ~ clk;              always # 10 {d} = ~ {d};
@ (clock) a = b;                      @ (negedge clock) a = b;
{p_out,s}<= #12 a+b+p_in;

initial out_q = 1'b0;                 initial begin
initial begin                           r = 1'b1;
  reset = 1'b0;                         # 10 r = 1'b0;
  set = 1'b1;                           s = 1'b1;
end                                   end
initial {a, b} = 2;
initial begin
   {a, b, p_in} = 3'b000;
   # 10 {a, b, p_in} = 3'b001;     // допустима запись {a, b, p_in} = 1;
……….
   $finish;
end
initial begin
   a = 2'b00;     // a - двухбитная величина; можно записать a = 0;
   d = 4'b0000;   // d - четырехбитная величина; допустима запись d = 0;
end
initial    # 400  $finish;
initial begin                 initial begin
   # 10 a = 1;                  # 10 a <= 1;     // выполняется в момент t = 10
   # 2.5 b = 2;                 # 2 b <= 0;      // выполняется в момент t = 2
   # b/2 c = a;                 # 10  $finish;
   # prm c = 4;               end
end

module psm (p, s, a, b);              module psm (sum, in);
module psm (sum, a, b);                 output [1:0] sum;
    output [1:0] sum;                   input [1:0] in;
```

```
psm psm1 (. sum [0] (prm3), . sum [1] (s), a (p_in), . b (prm1));
psm psm2 (prm2, prm1, in);      // при этом  in - двухбитная величина
mymod mymod1 (out, a&b, in2, in3);
rs rs1 (nq, q, 1, s);

`timescale 1ns/10ps                module test_tb;
`timescale 100/10                  ……..
                                   endmodule

and  my_ and  (out_or, in1, in2);
or  #5 or_but (f_out, out_dc [7], out_dc [3], out_dc [0]);

assign out = (in1 | in2) & in3;              assign # 1 out_psm = a + b;
assign y = (y1 || y2) ? a : 0;
assign out = a1 ? (a0 ? d3 : d2) : (a0 ? d1 : d0);

if (a ^ b)   begin              if (x == 1)
        s = 1'b1;               y [x] = 1;
        p = 1'b0;               else
end                             y [0] = 1;
    else     if  (a && b)       if (~ reset) q = 0;
begin                           else if (clk) q = data;
                s = 1'b0;             else     ;
               p = 1'b1;
  end
     else  ...

case ({in1, in2})
  2'b01: y = 0;
  default: y = 1;
endcase

while (a < b) begin             initial begin
  y1 = 1;                         for (index = 0; index < 31; index = index + 1)
  y2 = o;                         list [index] = index + index;
end                             end
integer count;                  initial begin
reg [127:0] a;                    clk = 0;
initial begin                     forever # 10 clk = ~ clk;
  count = 128;                  end
  repeat (count) begin          initial begin
    count = count - 1;            a = 2; b = 4;
    a                             forever begin
[count] = count / 2;                # 10 a = a + b;
  end                               b = a - 1;
end                               end
                                end
$display ($time, "the counter is %b", counter);
# 1 $display ("a = %b; d = %b, out = %d", a, d, out);
$display (a &~ b | ~ a & b);
always # 10 $display ($time, "%b %b %b", clk, data, q);
```

Приложение 3. Таблица операторов Verilog

Название оператора	**Обозначение**	**Примечание**
Арифметические операторы		
Умножение Деление Сложение Вычитание Определение остатка от деления	* / + – %	Бинарные (+ и – могут быть унарными). Результат - многобитный. Если хотя бы один бит *x*, то и результат *x*
Поразрядные операторы		
Отрицания И ИЛИ Исключающее ИЛИ Исключающее ИЛИ-НЕ	~ & | ^ ~^, ^~	Бинарные. Результат имеет столько же бит, сколько и операнды
Конкатенация (объединение)	{<имя> ,.., }	Позволяет увеличивать разрядность данных
Повторения	{<число> {<имя>,.., }}	Многократное повторение объединения (конкатенации)
Операторы сдвига		
Вправо Влево	>>*n* <<*n*	Бинарные. Сдвиг вправо на *n* разрядов. Сдвиг влево на *n* разрядов.
Операторы отношения		
Больше Больше или равно Меньше Меньше или равно	> >= < <=	Бинарные. Результат: 0, 1 или *x*. Если хотя бы один операнд *x*, то и результат *x*
Операторы эквивалентности		
Логического равенства Логического неравенства Выборочного равенства Выборочного неравенства	== != === !==	Бинарные. Операнды сравниваются побитно. Результат однобитный
Операторы приведения		
И	&	Унарные.

И-НЕ ИЛИ ИЛИ-НЕ Исключающее ИЛИ Исключающее ИЛИ-НЕ	~& | ~| ^ ~^, ^~	Выполняются над многоразрядным операндом пошагово, начиная с двух крайних левых разрядов. Результат однобитный
Логические операторы		
И ИЛИ НЕ	&& || !	Бинарные. Результат операции: 1, 0 или *x*. Операндом может быть переменная и логическое выражение

Приложение 4. Таблица приоритета операторов Verilog

1	+, – (унарные), !, ~, операторы приведения	Все унарные операторы
2	*, /, %	Арифметические
3	+, –	Арифметические
4	<<, >>	Сдвига
5	<, <=, >, >=	Отношения
6	==, !=, ===, !==	Эквивалентности
7	&, ~&	Поразрядные
8	^, ^~, ~^	Поразрядные
9	|, ~|	Поразрядные
10	&&	Логические
11	||	Логические

Примечание. Операторы расположены в порядке понижения приоритета.