Дональд Томас

# Логическое проектирование и верификация систем на SystemVerilog

# Логическое проектирование и верификация систем на SystemVerilog

*Дональд Томас*


Москва, 2019

**УДК 004.438**
**ББК 32.973.2**
**Т56**

**Дональд Томас**

Т56 Логическое проектирование и верификация систем на SystemVerilog / пер. с анг. А. А. Слинкина, А. С. Камкина, М. М. Чупилко; науч. ред. А. С. Камкин, М. М. Чупилко. – М.: ДМК Пресс, 2019. – 384 с.: ил.

**ISBN 978-5-97060-619-3**

Книга посвящена SystemVerilog – языку описания аппаратуры, используемому для моделирования электронных систем. Разработчики SystemVerilog сделали его синтаксис похожим на синтаксис языка C, что упрощает освоение. Предполагается, что у читателя есть базовая подготовка в области схемотехники и программирования. Материал по языку дается вместе с материалом по логическому проектированию, так что книга может использоваться в качестве учебного пособия для курсов цифровой схемотехники и архитектуры компьютеров. В современных подходах к проектированию аппаратуры проверка модели (верификация) не менее важна, чем ее разработка. SystemVerilog предлагает конструкции, позволяющие лучше отразить инженерный замысел в моделях, программные абстракции, упрощающие разработку тестовых окружений, утверждения, обеспечивающие проверку поведения сложных систем, а также средства измерения функционального покрытия в процессе верификации.

Издание будет полезно студентам, проходящим вводный курс цифровой схемотехники, а также разработчикам, которые знакомы с Verilog или VHDL, но желают освежить свои навыки или нуждаются в кратком справочнике по SystemVerilog.

УДК 004.438
ББК 32.973.2



ISBN 978-1-52336-402-2 (анг.)
ISBN 978-5-97060-619-3 (рус.)

*Посвящается моей семье, Сэнди, Джону и Холланд*

# Оглавление

# Предисловие: об этой книге

Язык Verilog появился зимой 1983–1984 годов и первоначально был коммерческим продуктом, предназначенным для моделирования и верификации цифровой аппаратуры. С тех пор он несколько раз подвергался стандартизации в Институте инженеров электротехники и электроники (Institute of Electrical and Electronics Engineers – IEEE); наконец, этот процесс завершился стандартом IEEE Std 1364™-2005, известным также как Verilog-2005[1]. SystemVerilog – это набор расширений Verilog-2005, определенных в стандарте IEEE Std 1800™-2005. И хотя это действительно надстройка над Verilog-2005, слово «расширение» не передает всей мощи нового языка. Получившаяся в результате комбинация того и другого описана в стандарте IEEE Std 1800™-2009[2].

SystemVerilog позволяет разработчикам работать с моделями более высокого уровня абстракции, что отвечает сложности современных цифровых систем. SystemVerilog – это не язык описания аппаратуры, с которым работали ваши родители, когда типичная микросхема содержала несколько тысяч транзисторов, ПЛИС (FPGA)[3] еще только маячили на горизонте, а логический синтез пребывал в младенческой стадии. В современных подходах к проектированию аппаратуры проверка модели (верификация) не менее важна, чем ее создание и имитационное моделирование (симуляция). SystemVerilog предлагает конструкции, позволяющие лучше отразить инженерный замысел в моделях, программные абстракции, упрощающие разработку тестовых окружений, утверждения, обеспечивающие проверку поведения сложных систем, а также средства измерения функционального покрытия в процессе верификации.

Хорошо это или плохо, но язык стал настолько большим, что охватить его целиком в одной книге трудно. В этой книге мы даже не пытаемся это сделать! У такого большого языка есть преимущество – это единая среда, объединяющая разработчиков и верификаторов и охватывающая многие уровни абстракции, используемые при проектировании интегральных схем. Книга начинается с описания новых конструкций SystemVerilog; при этом она содержит объяснения, позволяющие без труда читать унаследованные модели. Даже если вы никогда не сталкивались с языком Verilog, вам не придется изучать его перед освоением SystemVerilog; в книге есть все необходимое, чтобы начать работать с SystemVerilog.

---

[1] Предыдущими стандартами языка Verilog являются IEEE Std 1364™-1995 (так называемый Verilog-95) и IEEE Std 1364™-2001 (так называемый Verilog-2001). – *Здесь и далее прим. перев.*

[2] В феврале 2018 года стандарт языка SystemVerilog был обновлен – IEEE Std 1800™-2017.

[3] ПЛИС – программируемая логическая интегральная схема. Здесь и далее под ПЛИС понимается программируемая пользователем вентильная матрица (Field-Programmable Gate Array – FPGA).

Предполагается, что у читателя есть базовая подготовка в области схемотехники и программирования. Материал по языку дается вместе с материалом по логическому проектированию, так что книга может использоваться в качестве учебного пособия для курсов цифровой схемотехники и архитектуры компьютеров. Абстракции языка соответствуют абстракциям, используемым при проектировании, поэтому темы схемотехники и языка переплетаются. Книга ориентирована на следующие группы читателей:

- студентов, проходящих вводный курс цифровой схемотехники, на котором также преподается SystemVerilog;
- разработчиков, знакомых с Verilog или VHDL и желающих освежить свои навыки или нуждающихся в кратком справочнике по SystemVerilog;
- студентов, слушающих курсы по разработке СБИС/ПЛИС, затрагивающие дополнительные темы, в т. ч. вопросы верификации.

По сравнению с предыдущими книгами автора, написанными в соавторстве с Филипом Мурби (Philip Moorby)[4] («The Verilog Hardware Description Language, Fifth Edition», издательство Springer), в которых подробно описывался язык Verilog, эта книга в большей степени посвящена проектированию и верификации сложных цифровых систем.

Вы не найдете в книге полного описания SystemVerilog. Все верно – язык огромен! Задача книги – познакомить читателя с широким спектром возможностей языка; она дополняет вводные и продвинутые курсы по проектированию и верификации аппаратуры и закладывает фундамент для дальнейшего изучения. В книге имеются части, посвященные схемотехнике, в которых язык затрагивается лишь мимоходом (примером может служить глава 6). Такие части будут интересны студентам и преподавателям университетов.

В конце некоторых глав имеются задачи. За решениями, по крайней мере, некоторых из них можете обращаться ко мне по адресу dthomas@cmu.edu или dthomas1611@gmail.com. Возможно, я отвечу не сразу, но я постараюсь ответить. Если же вы студент, решайте самостоятельно. Только практикуясь, можно научиться!

Чтобы получить текст примеров для использования в учебных слайдах, обращайтесь ко мне по адресам, указанным выше.

Напоследок отметим, что в этом переработанном издании был обновлен материал о спецификации несущественных ситуаций в комбинационных схемах.

Получайте удовольствие от проектирования хороших систем!

*– Дон Томас –*

[4] Филип Мурби – один из создателей языка Verilog и разработчик первого Verilog-симулятора.

# Предисловие от издательства

## Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

## Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

## Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

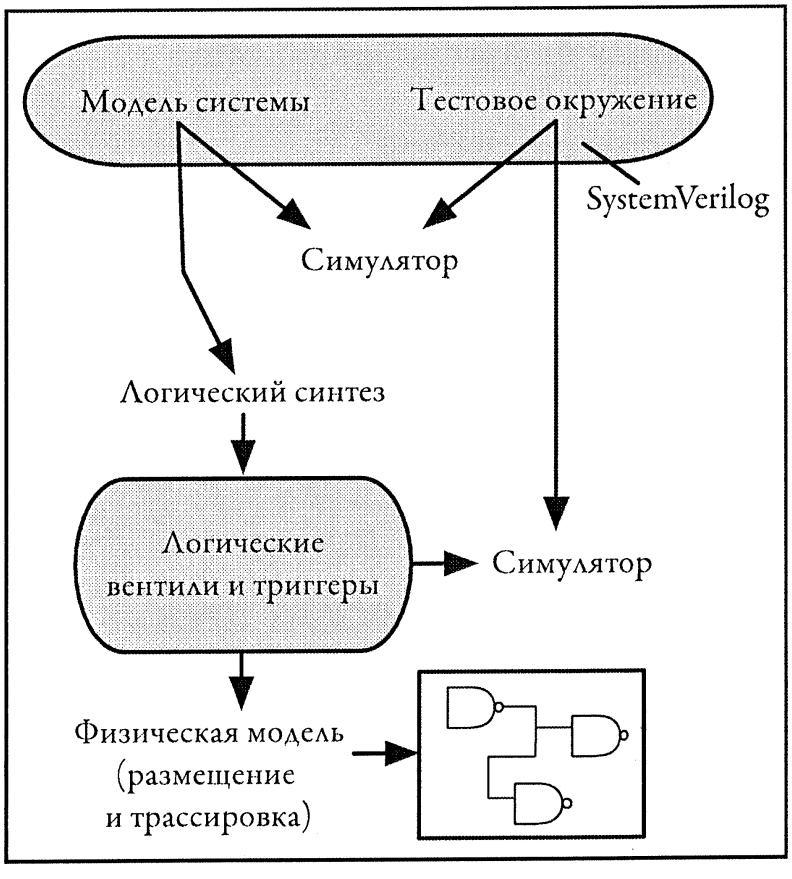
Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

# Контекст: проектирование на уровне регистровых передач

Сейчас производятся цифровые системы с миллиардами транзисторов на кристалле. Любитель, конечно, может в качестве спецификации (для реализации на макетной плате) нарисовать несколько логических вентилей и соединить их проводами, но для коммерческих проектов это древняя история.

Современные системы специфицируются на языках описания аппаратуры, таких как SystemVerilog. Языки этого типа позволяют описывать аппаратуру в терминах ее функциональности. Система автоматизированного проектирования (САПР) получает описание аппаратуры на входе и предоставляет средства для автоматизированного (иногда автоматического) формирования детальной модели[5].




На рисунке справа показана упрощенная схема проектирования на уровне регистровых передач[6]. Модель, задающая функциональность системы, представляется на языке описания аппаратуры, например SystemVerilog (сверху). Для

[5] Имеется в виду модель, используемая для производства микросхем: топология интегральной схемы или прошивка ПЛИС.

[6] Подробно модели уровня регистровых передач рассматриваются в частях I и II (главах 2–6).

того чтобы проверить функциональную корректность модели, симулятор исполняет ее вместе с тестовым окружением. Тестовое окружение – это программа на языке описания аппаратуры, предназначенная для тестирования разрабатываемой системы.

Затем по модели синтезируется логическая схема. САПР автоматически определяет триггеры и вентили, необходимые для реализации модели, представленной на языке описания аппаратуры. Как правило, инструменты логического синтеза оптимизируют схему с учетом заданных ограничений, таких как занимаемая площадь, задержка распространения сигналов и потребляемая мощность. После этого начинается процесс физического проектирования, ориентированный на создание интегральной схемы или ПЛИС. Процесс называется *физическим*, поскольку его результатом является физическая реализация системы[7]. В случае интегральной схемы необходимо задать место расположения каждого логического вентиля и соединить их между собой (эта процедура называется размещением и трассировкой). Все это показано на рисунке. Схожие действия осуществляются, если конечной целью является ПЛИС.

Несмотря на то что это сильно упрощенный взгляд на процесс проектирования интегральных схем, общая суть понятна: в процессе задействуется множество инструментов, да и сам по себе процесс сложен. Без обеспечиваемой этими инструментами продуктивности было бы невозможно думать о создании систем, состоящих из миллиардов транзисторов! Отправной точкой всего процесса является язык описания аппаратуры, например SystemVerilog, – именно на нем описывается модель системы и тестовое окружение для ее верификации.

Проектирование цифровой системы сводится к использованию тех или иных абстракций, и разные части книги посвящены моделированию разных аспектов системы и соответствующим абстракциям.

- Сначала дается введение в языки описания аппаратуры и событийное моделирование.
- Затем, в первой части, описывается моделирование на уровне регистровых передач: комбинационные схемы, триггеры и конечные автоматы.
- Во второй части уровень проектирования повышается до вычислительных блоков – конечных автоматов с трактом данных. Мы называем их «аппаратными потоками».
- В третьей части рассматривается разработка тестового окружения, позволяющего верифицировать модель. Сюда же включены сведения об утверждениях и функциональном покрытии.
- В последней части обсуждаются детали, которым не нашлось места в других разделах книги.

Даже все эти части не охватывает язык целиком. С некоторыми деталями лучше знакомиться по справочному руководству.

---

[7] На самом деле результатом физического проектирования интегральной схемы является *топология* – описание (например, в формате GDSII) пространственно-геометрического расположения элементов схемы и соединений между ними. По такому описанию могут быть построены фотошаблоны, применяемые для производства микросхем.

# Благодарности

Автор выражает благодарность всем студентам, которым пришлось иметь дело с ранними вариантами этой книги. Ваши полезные, порой болезненные замечания позволили улучшить объяснения и примеры. Профессор Билл Нейс (Bill Nace), с которым я имел удовольствие совместно преподавать в течение многих семестров, дал подробный отзыв о книге, доходя иногда до указания замеченных опечаток. Я благодарю также Габриэля Сомло (Gabriel Somlo) за корректуру и Дебру Острандер Виейра (Debra Ostrander Vieira) за дизайн обложки и подготовку к печати. Наконец, я признателен всем своим учителям, прошлым и нынешним, благодаря которым смог написать эту книгу.

# Глава 1

# Введение

*Симулятором*[8] называется программа, которая предсказывает, как состояние физической системы изменяется со *временем*. Симулятор погоды предсказывает погоду в будущий момент времени в зависимости от текущей погоды. Компьютерная игра SimCity™ – это симулятор, предсказывающий развитие города в зависимости от текущей ситуации и действий, например капиталовложений в строительство дорог и другой инфраструктуры. Написанная вами программа для моделирования скорости брошенного тела – тоже симулятор; она вычисляет новое значение скорости в зависимости от предыдущего, ускорения свободного падения, трения и времени, прошедшего с момента начала падения. Общим для всех этих симуляторов является тот факт, что они непосредственно моделируют время.

SystemVerilog – язык описания и симулятор электронной цифровой аппаратуры. Он позволяет *моделировать* цифровую систему, например соединенные между собой логические вентили, и сообщает, как в системе распространяются значения в зависимости от времени. Он помогает определить, правильно ли система реализует свою функциональность и обеспечивает ли заданную производительность.

Модель проектируемой системы представляется на специальном *языке моделирования*. В языке имеются средства описания элементов модели и средства, позволяющие их использовать для построения больших систем. Например, язык SystemVerilog позволяет моделировать такие элементы, как логические вентили. Основные аспекты SystemVerilog, благодаря которым на нем можно описывать цифровую аппаратуру, – моделирование времени, функциональности и соединений. Мы уже отметили, что время – фундаментальная концепция любого симулятора, а стало быть, и языка моделирования. Межсоединения позволяют связывать вентили проводами (wires)[9]: когда изменяется выход одного вентиля, новое значение распространяется по проводам и попадает на входы других вентилей. Так физически устроены цифровые схемы, и язык

---

[8] Термин *simulator* переводится как *система (имитационного) моделирования*. Мы будем использовать более короткое и привычное разработчикам аппаратуры слово *симулятор*.

[9] Речь, очевидно, идет не о физических проводах, а о логической абстракции – переменных специального вида.

призван это отражать. SystemVerilog позволяет выразить куда более сложные и абстрактные конструкции, чем логические вентили, однако мы начнем с моделей уровня вентилей, поскольку с их помощью можно проиллюстрировать базовые возможности языка.

Язык SystemVerilog отличается от традиционных языков программирования. Предполагается, что читатель знаком с каким-нибудь языком программирования; в этой главе мы постараемся научить вас думать в терминах цифровой аппаратуры и объясним, какие языковые средства необходимы для описания и тестирования моделей аппаратуры.

# 1.1. Приступая к работе

Основным структурным элементом в языке SystemVerilog является *модуль*. У каждого модуля имеется интерфейс – входные и выходные порты, через которые осуществляется взаимодействие с другими модулями, – а также описание его содержимого. Модуль представляет собой блок, который можно описать, либо задав его внутреннюю структуру (например, указав, из каких логических вентилей он состоит), либо определив его поведение – так же, как в программах (в этом случае фокус смещается на функциональность модуля, а не на его реализацию с помощью вентилей). Модули соединяются проводами, благодаря чему они могут взаимодействовать и образуют более крупные системы.

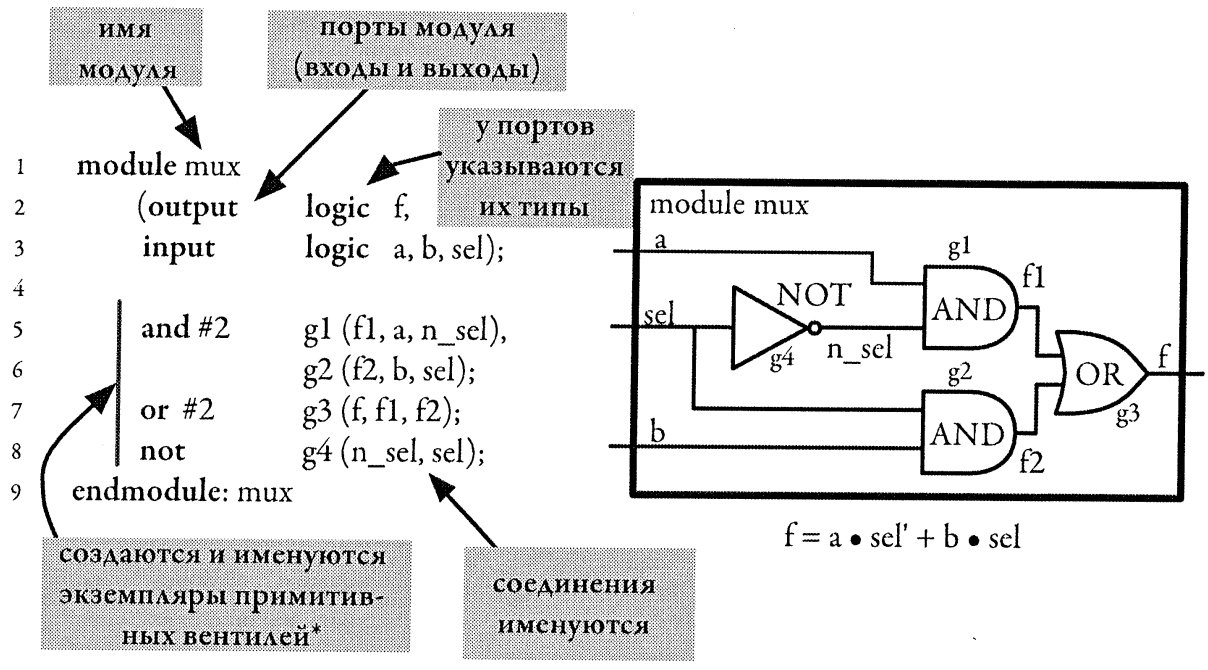
## 1.1.1. Структурное описание

Начнем с простой комбинационной схемы – двухвходного мультиплексора (2:1). В примере 1.1 приведена схема из логических вентилей, реализуемая ей булева функция и описание на языке SystemVerilog. Схема выбирает, какой из двух входов (`a` или `b`) определяет значение на выходе `f`. Если значение сигнала выбора (`sel`) равно TRUE, то на выход `f` подается значение `b`. Если же значение `sel` равно FALSE, то на выход `f` подается значение `a`.

В левой части рисунка показано *определение* SystemVerilog-модуля, соответствующего схеме справа. В данном случае модуль называется `mux`. Определение любого модуля начинается ключевым словом `module`, за которым следует имя модуля, а заканчивается ключевым словом `endmodule`. В строках 2 и 3 определения указаны имена и типы входных и выходных портов. Все входы и выходы имеют тип `logic`, который мы опишем ниже.

В строках 5 и 6 создаются *экземпляры*[10] двух вентилей AND. В строке 7 добавляется вентиль OR, а в строке 8 – вентиль NOT. Всем экземплярам вентилей даются имена (от `g1` до `g4`) в соответствии с тем, как они названы в схеме. Запись #2 у некоторых вентилей означает, что задержка распространения сигнала от входов до выхода составляет две единицы времени. Для вентиля NOT

[10] Создание экземпляра (instance) модуля также называют инстанцированием (instantiation).




* Примитивными называются вентили, реализующие простейшие булевы функции: NOT (НЕ, отрицание), AND (И, конъюнкция), NAND (И-НЕ, штрих Шеффера), OR (ИЛИ, дизъюнкция), NOR (ИЛИ-НЕ, стрелка Пирса), XOR (исключающее ИЛИ, сумма по модулю 2), XNOR (исключающее ИЛИ-НЕ, эквивалентность).

**Пример 1.1.** Модуль и соответствующая ему логическая схема

задержка не указана и, следовательно, равна 0. Имена в скобках в строках 5–8 обозначают соединения входов и выходов вентилей. Первое имя соответствует выходу, остальные – входам. Имена соединений и экземпляров вентилей указаны на рисунке, чтобы пояснить соответствие между логической схемой и эквивалентным описанием на SystemVerilog. Номера строк не являются частью описания, они включены только для удобства. Ключевые слова языка выделены полужирным шрифтом.

Несмотря на простоту, этот пример иллюстрирует несколько важных особенностей SystemVerilog.

- Модули – основные структурные элементы языка. В определении модуля описываются порты и внутренняя функциональность; из модулей можно конструировать более сложные системы. В примере показан простой мультиплексор, но в виде модуля можно описать и целый компьютер.
- Примитивные вентили – в языке предопределены такие вентили, как AND, NAND, OR, NOR, NOT и XOR. При создании экземпляра вентиля первое имя в скобках – выход, остальные – входы. У примитивных вентилей может быть несколько входов, они перечисляются в списке портов через запятую.
- Создание экземпляров – эту процедуру можно рассматривать как помещение нового элемента на макетную плату. В данном модуле созданы экземпляры четырех примитивных вентилей. В каждой строке описывается новый экземпляр вентиля определенного типа, и каждый

экземпляр добавляет в модуль новую логику. Система становится физически больше, потому что для реализации каждого вентиля необходимы транзисторы.

- Соединения – вентили соединены проводами с другими вентилями и с портами содержащего их модуля. В данном случае соединения названы a, b, sel, n_sel, f1, f2 и f. Тем самым моделируются электрические соединения между вентилями (модулями).
- Сокрытие информации – экземпляр модуля можно создать в других модулях. При этом внутреннее устройство модуля может быть неизвестно, известны лишь имена и типы портов. Потенциально сложная внутренняя структура и имена внутренних сигналов скрыты от пользователя модуля.

## 1.1.2. Как интерпретируется описание модуля

У человека, знакомого с языками программирования, который никогда не видел языка описания аппаратуры, сразу возникает вопрос: «Как этот модуль исполняется?» Здесь нет ни привычных циклов for, ни операторов if. Программист, пишущий на C, спросит, где функция main. Для ответа на эти вопросы нужно рассмотреть несколько моментов.

Во-первых, при соединении вентилей (а также модулей, как мы скоро увидим) мы делаем только одно – задаем именованные соединения компонентов. Создание экземпляра – это добавление в модуль еще одного компонента. Значения передаются по соединениям с выхода одного вентиля на вход другого. Определять компоненты и соединения между ними можно в любом порядке. Таким образом, оба варианта модуля mux, показанные в примере 1.2, определяют ту же самую логику, что и модуль из примера 1.1.

```
1 module mux
2   (output logic f,
3    input logic a, b, sel);
4
5    and #2 g1 (f1, a, n_sel),
6              g2 (f2, b, sel);
7    not       g4 (n_sel, sel);
8    or #2   g3 (f, f1, f2);
9 endmodule: mux
```

```
1 module mux
2   (output logic f,
3    input logic a, b, sel);
4
5    or #2   g3 (f, f1, f2);
6    not       g4 (n_sel, sel);
7    and #2 g1 (f1, a, n_sel),
8              g2 (f2, b, sel);
9 endmodule: mux
```

**Пример 1.2.** Два эквивалентных модуля

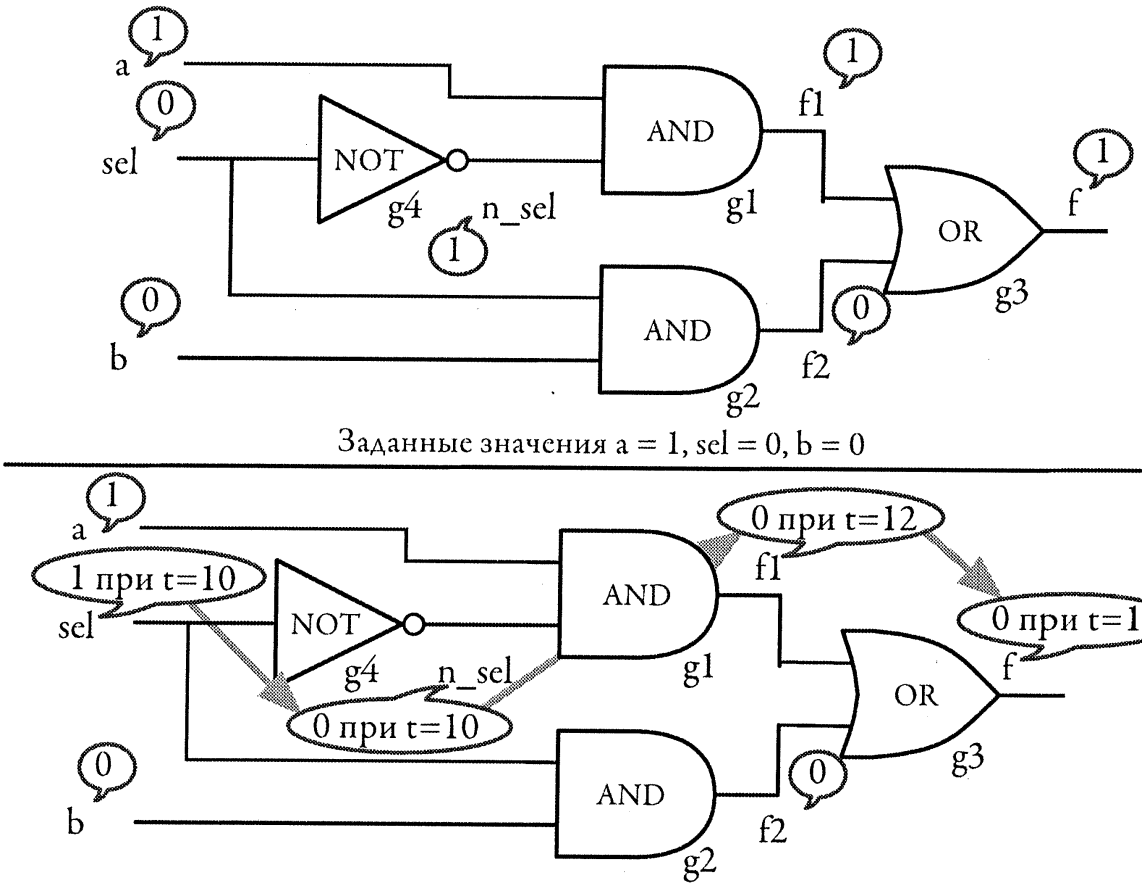



**Рис. 1.1.** Распространение значений во времени

Во-вторых, нужно понимать, что модуль является исполняемым, но *модель исполнения* не такая, к какой мы привыкли в языках программирования. В SystemVerilog при изменении значения входа вентиля симулятор перевычисляет значение его выхода. Если значение изменилось, симулятор распространяет это изменение на соединенные с ним вентили, возможно, с задержкой во времени. Исполняя модуль таким способом, симулятор моделирует распространение электрических сигналов между соединенными между собой компонентами. Это модель исполнения на уровне вентилей.

Поток значений от входа вентиля к его выходу, затем на вход следующего вентиля и т. д. не описывается вызовами функций, как можно было бы подумать, т. е. строки 5–8 в примерах выше не вызовы функций. Еще раз повторим, что эти строки описывают экземпляры вентилей и соединения между ними. Когда значение входа вентиля изменяется, симулятор вычисляет значение выхода и смотрит, изменилось ли оно. Если да, измененное значение распространяется по соединениям на входы других вентилей.

Рассмотрим, как значения распространяются в логической схеме на рис. 1.1. Предположим, что в течение длительного времени входы схемы не менялись и имеют следующие значения: `a=1`, `b=0` и `sel=0`. Легко видеть, что `n_sel=1`, `f1=1` и, следовательно, `f=1`, как показано на верхней диаграмме. Если в момент 10 значение `sel` станет 1, некоторые значения в схеме изменятся, как показано

на второй диаграмме. Значение n_sel станет равным 0 в тот же момент времени, поскольку у вентиля NOT нет задержки. В момент 12, из-за задержки в вентиле AND, значение f1 изменится на 0, поскольку один из его входов (n_sel) теперь имеет нулевое значение. Наконец, в момент 14, из-за задержки в вентиле OR, значение f станет равным 0.

Отметим, что порядок изменения значений не обязан совпадать с порядком создания экземпляров вентилей. В описании экземпляров (строки 5–8 в обоих модулях в примере 1.2) определяются только их имена и соединения. Потоком распространения значений от выходов к входам по соединениям управляет симулятор.

## 1.2. Моделирование цифровых систем

Для простой схемы, показанной выше, понять, как распространяются значения, легко, но для более сложных моделей необходим симулятор. Как уже отмечалось, SystemVerilog позволяет не только создать модель цифровой системы, но и устанавливать значения ее входов и наблюдать за тем, какие значения получаются на выходах. Это помогает понять, правильно ли работает модель. Часть описания, отвечающая за установку входных значений и наблюдение за выходными, называется *тестовым окружением*.

Пример 1.3 основан на примере 1.1. В него добавлено тестовое окружение и внесены некоторые изменения для целей иллюстрации: убраны порты, а переменные сделаны внутренними переменными модуля (см. строку 2). Строки 4–7 совпадают со строками 5–8 из примера 1.1. Тестовое окружение для мультиплексора описано в строках 9–18; оно начинается с оператора initial.

Оператор initial (строка 9) – это *процедурный* оператор, с которого начинается исполнение при запуске симулятора. Здесь используется термин «процедурный», поскольку можно считать, что операторы внутри скобок begin-end исполняются процедурно, т. е. друг за другом от начала к концу. (Именно такая модель исполнения применяется в традиционных языках программирования.) Есть и отличие – в некоторых операторах задана временна́я задержка (см. строки 16 и 17). Знак # с числом после него означает задержку заданной длительности перед исполнением оператора.

```
1  module muxSim; // из примера 1.1
2  logic a, b, sel, n_sel, f1, f2, f;
3
4    and #2 g1 (f1, a, n_sel),
5           g2 (f2, b, sel);
6    or #2  g3 (f, f1, f2);
7    not    g4 (n_sel, sel);
8
9    initial begin
10     $monitor ($time,
11     " a=%b b=%b sel=%b n_sel=%b f1=%b f2=%b f=%b",
12     a, b, sel, n_sel, f1, f2, f);
13     a = 0;
14     b = 0;
15     sel = 0;
16     #12 a = 1;
17     #6 $finish;
18   end
19 endmodule: muxSim
```

**Пример 1.3.** Модуль muxSim

Рассмотрим этот пример по шагам, чтобы лучше понять смысл операторов и логику работы симулятора. Оператор initial (его часто называют блоком initial[11]) начинает исполняться в момент 0. Сначала исполняется процедура $monitor, которая сообщает симулятору, какую информацию выводить на консоль при изменении значений переменных (ниже мы рассмотрим его более подробно). Затем исполняются строки 13–15, в которых a, b и sel присваивается значение 0. Дойдя до строки 16, симулятор видит #12 и понимает, что должен подождать 12 единиц времени.

В этой точке (время по-прежнему 0) исполнение блока initial прекращается, и симулятор запоминает, где остановился. Он знает, что a, b и sel только что стали равны 0, и исполняет вентили модели, входы которых связаны с этими переменными (эти вентили определены в строках 4, 5 и 7). Поскольку задержка распространения значений через вентиль AND равна 2, f1 и f2 изменят значения в момент 2. Далее, поскольку эти значения подаются на вход вентиля OR, задержка которого тоже равна 2, значение выхода f будет установлено в момент 4. В этот момент все значения распространились через вентили, и до момента 12 можно ничего не делать.

После этого симулятор переводит время на момент 12[12], на который запланировано возобновление исполнения блока initial. Блок initial «просыпается» и продолжает работу с того места, в котором был приостановлен (строка 16). Симулятор исполняет строку 16 – присваивает a значение 1 – и переходит к строке 17. В строке 17 сказано, что нужно подождать еще 6 единиц времени – до момента 18 (12 + 6). Исполнение блока initial приостанавливается, и симулятор запоминает, где остановился. Поскольку значение a изменилось, перевычисляется значение выхода вентиля g1, у которого a является входом. Поскольку значение n_sel также равно 1, спустя 2 единицы времени (в момент 14) выход f1 становится равным 1, а f получает значение 1 еще через 2 единицы времени (в момент 16). Теперь все значения распространились через вентили, и до момента 18 можно ничего не делать.

В момент 18 блок initial возобновляет исполнение и вызывает процедуру $finish, которая завершает симуляцию.

## 1.2.1. Замечания по поводу симуляции

При обсуждении блока initial мы лишь мельком упомянули процедуру $monitor. В ней задается список наблюдаемых переменных (строка 12). Если в процессе симуляции какая-нибудь из них изменится, то $monitor выведет на консоль текущее время ($time в строке 10), а затем значения переменных в формате, заданном в строке 11. В нашем примере вывод выглядит, как показано на рис. 1.2.

---

[11] За ключевым словом initial может следовать любой оператор; часто это *блок* – последовательность операторов, заключенная в скобки begin-end.

[12] Важно понимать, что симулятор физически не ждет, когда истечет заданный промежуток времени, – он просто инкрементирует показание модельных часов.

Строка управления выводом очень похожа на форматную строку в языке программирования C; "a=%b" означает, что нужно вывести "a=", а затем – значение a (первая переменная в строке 12) в двоичном формате (как того требует спецификатор %b).

Интересное свойство процедуры $monitor заключается в том, что это последнее действие, исполняемое симулятором перед обновлением времени. Таким образом, на консоль выводятся значения, получившиеся в результате *всех* изменений, сделанных в текущий момент времени, а не значения после первого изменения. Например, в момент 0 всем трем переменным a, b и sel присваивается значение 0, поэтому для них печатаются нули (что мы и видим в первой строке). Для n_sel печатается значение 1. Это связано с тем, что вентиль NOT, формирующий n_sel из sel, имеет нулевую задержку, т. е. значение n_sel также устанавливается в момент 0.

```
2    2 a=0 b=0 sel=0 n_sel=1 f1=0 f2=0 f=x
3    4 a=0 b=0 sel=0 n_sel=1 f1=0 f2=0 f=0
4   12 a=1 b=0 sel=0 n_sel=1 f1=0 f2=0 f=0
5   14 a=1 b=0 sel=0 n_sel=1 f1=1 f2=0 f=0
6   16 a=1 b=0 sel=0 n_sel=1 f1=1 f2=0 f=1
7   $finish at simulation time                18
```

**Рис. 1.2.** Результаты симуляции

Еще одно замечание – в первой строке на рис. 1.2 для переменных f1, f2 и f на консоль выводится значение x. Это не ошибка, так симулятор дает нам знать, что значение переменной еще неизвестно. Напомним, что тип всех переменных в этом примере – logic. Такие переменные могут принимать одно из четырех значений: 0, 1, x или z. Что такое 0 и 1, должно быть понятно; x означает, что значение неизвестно симулятору; z – высокий импеданс (пока можете считать, что такое значение не может появиться на выходе вентиля).

При запуске симулятора всем переменным типа logic присваивается значение x. Чтобы переменную можно было использовать, ей необходимо присвоить начальное значение. В нашем примере в момент 0 значения входов еще не дошли до выходов f1, f2 и f.

Наконец, отметим, что даже в процедурных операторах в блоке initial можно задавать время – в языках программирования такое невозможно! Суть в том, что временны́е задержки, указанные у процедурных операторов и вентилей, позволяют планировать присваивания новых значений и исполнение действий в будущем (как, например, возобновление исполнения блока initial). SystemVerilog – параллельный язык, допускающий исполнение нескольких операций в одно и то же время.

## 1.2.2. Что делает симулятор?

Из нашего обсуждения стало понятно, как симулятор отслеживает время и изменяет значения в нужные моменты при исполнении вентильной модели и блока initial с задержками в процедурных операторах (например, #2). В этом разделе мы более подробно опишем, как работает дискретно-событийное моделирование. Это позволит лучше разобраться в языке SystemVerilog и понять, как работает симулятор.

Как показано на рис. 1.3, внутренняя структура симулятора отражает его способность отслеживать время и планировать изменение значений на определенные моменты времени в будущем, а не сразу, как в обычном языке программирования. Время внутри симулятора называют *виртуальным*, или *модельным*. Употребляя слово «время», мы, как правило, имеем в виду именно виртуальное время. Изменение значения переменной в определенный момент времени называется *событием обновления*. Симулятор хранит события обновления в списке, упорядоченном по времени их возникновения. Все события, которые должны произойти в один и тот же момент времени (например, $t_i$), хранятся вместе, как показано в левой части рисунка.

В момент запуска симулятор обнуляет время, а всем переменным присваивает значение x. После этого он в цикле *выбирает* события обновления, запланированные на текущий момент времени, *обновляет* внутреннее состояние модели (состоянием вентиля, например, является значение его выхода) и *распространяет* новое состояние, следуя вдоль указанных в описании соединений к входам других вентилей. Далее симулятор *исполняет* соответствующие вентильные модели и проверяет, изменились ли значения их выходов. Если да, новые значения добавляются в список событий как новые события обновления (говорят, что симулятор *планирует* события). Описанный процесс продолжается до тех пор, пока список событий не станет пустым. Одна итерация этого циклического процесса (loop) называется *циклом симуляции* (*simulation cycle*).

Хотя это и не вся функциональность симулятора, сказанного вполне достаточно, чтобы понять, как он работает и как создает впечатление, будто разные действия происходят в одно и то же виртуальное время. Одно дополнение к рис. 1.3 состоит в том, что симулятор поддерживает события двух типов: *событие обновления* задает новое значение выхода вентиля или переменной для присваивания в определенный момент времени; *событие исполнения* говорит,

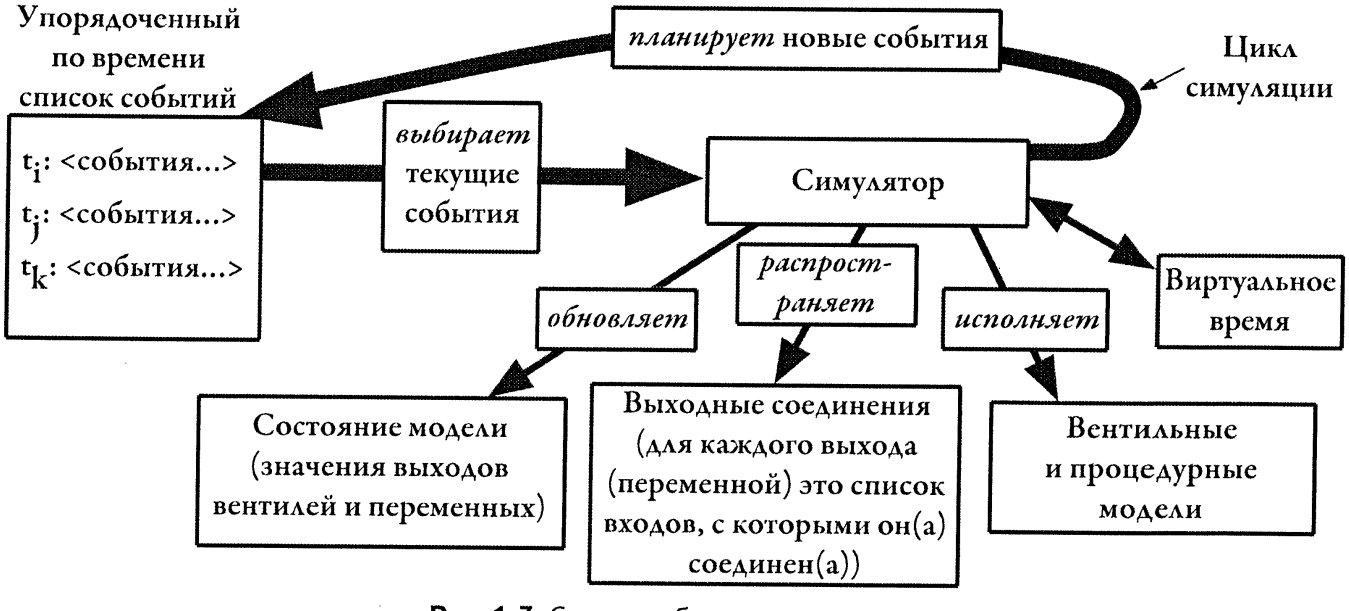



**Рис. 1.3.** Схема работы симулятора

что некая процедурная модель, например блок initial, должна начать исполняться или возобновить исполнение в определенный момент времени.

## 1.2.3. Более подробно о симуляции

Чтобы объяснить, как работает симулятор, обратимся к примеру 1.3 (номера строк в блоке initial изменены). Действия симулятора показаны на рис. 1.4–1.6. При запуске симулятор *планирует* исполнение всех блоков initial и always на момент времени 0. В нашем примере есть только один блок – это блок initial. На рис. 1.4 видно, что в начале симуляции все значения (состояние модели) неизвестны (равны x) и в списке событий на момент 0 запланировано событие исполнения блока initial. Симулятор обнуляет время и выбирает все события, запланированные на момент 0. В нашем случае выбирается только одно событие – событие исполнения блока initial, – и симулятор начинает исполнять этот блок. В результате исполнения строк 3–6 переменным a, b и sel присваивается значение 0; в строке 6 исполнение блока initial приостанавливается, а возобновление планируется на момент 12. Состояние в этой точке показано на логической схеме в средней части рис. 1.4 (время 0, конец цикла симуляции 1).

```
1   initial begin
2     ...
3     a = 0;
4     b = 0;
5     sel = 0;
6     #12 a = 1;
7     #6 $finish;
8   end
```

После этого симулятор в произвольном порядке распространяет значения a, b и sel до вентилей, с которыми они соединены. Начнем с распространения значения a до вентиля g1. Исполняя модель вентиля AND, мы видим, что выход f1 станет равным 0 через две единицы времени. Поэтому симулятор планирует событие обновления для f1 (присваивание значения 0) на момент 2. Аналогично значение b распространяется до g2; f2 вычисляется по двум входам: b=0 и sel=0. Событие обновления f2 (присваивание значения 0) планируется на момент 2. Наконец, значение sel распространяется до g4 и g2. На значение f2 это не влияет, а выход g4 (сигнал n_sel) получит значение 1; это обновление планируется на момент 0. Отметим, что n_sel изменяется не сразу, несмотря на нулевую задержку вентиля! Вместо этого соответствующее событие помещается в конец списка событий, запланированных на текущее время. Результаты распространения значений, исполнения вентильных моделей и планирования событий показаны справа средней части рис. 1.4 – в список событий добавлены новые события обновления и исполнения.

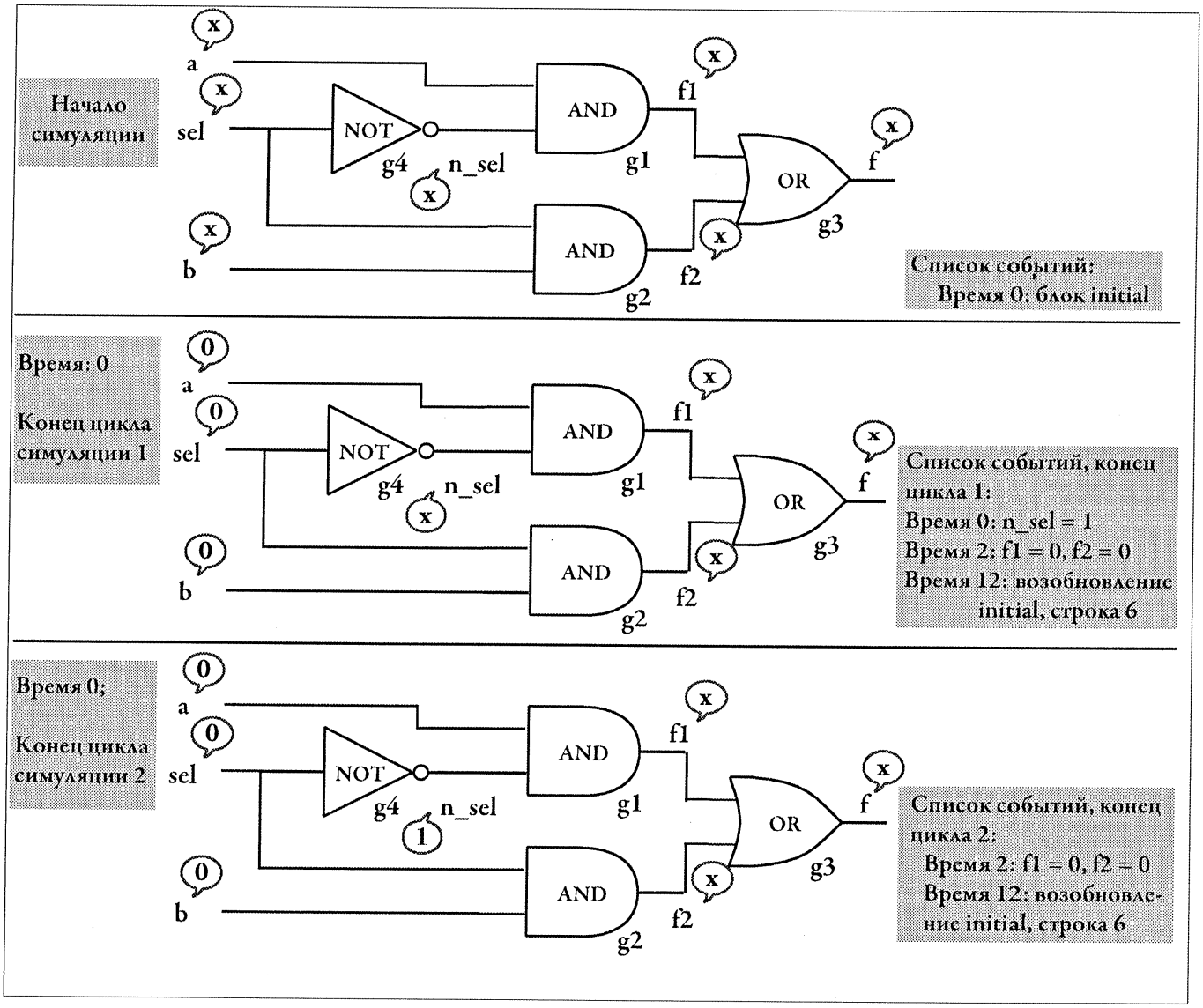



**Рис. 1.4.** Пошаговая схема работы симулятора (начало)

В этой точке симулятор видит, что больше делать нечего (все выбранные события обработаны); цикл симуляции завершается.

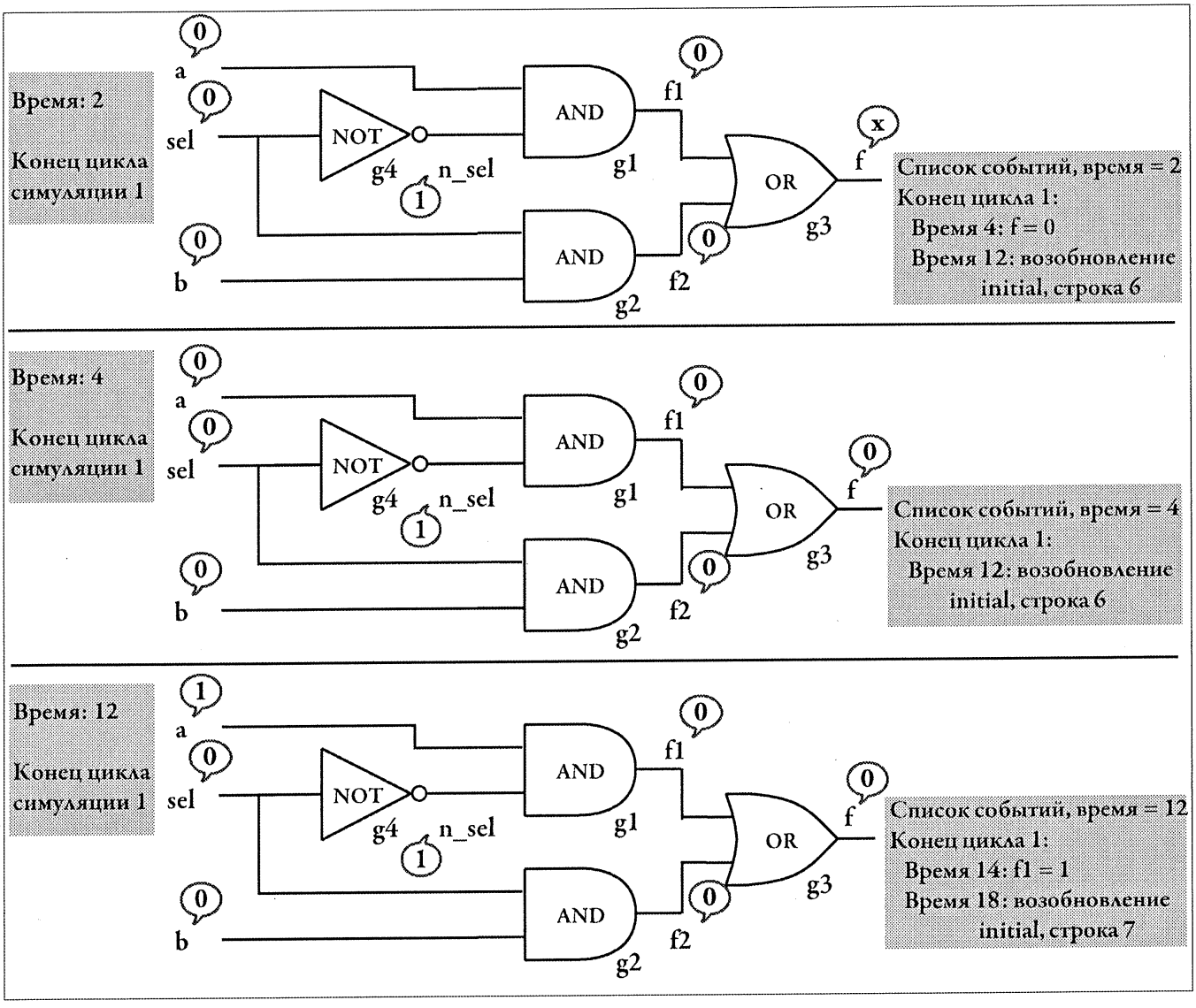



**Рис. 1.5.** Пошаговая схема работы симулятора (продолжение)

Далее симулятор переходит к циклу симуляции 2 для момента времени 0. Симулятор знает, что для текущего момента времени есть еще одно событие – обновить значение выхода вентиля `g4`: `n_sel` присваивается 1, как показано в нижней части рисунка. Новое значение `n_sel` распространяется до вентиля `g1`, но не изменяет запланированного значения выхода `f1`, поэтому новое событие обновления не планируется. Цикл симуляции завершается. В нижней части рисунка показано состояние схемы и списка событий в конце цикла 2 для момента 0.

В начале следующего цикла симулятор видит, что на момент 0 не запланировано никаких событий – процедура `$monitor` выводит на консоль первую строку, показанную на рис. 1.2. Отметим, что `n_sel` имеет значение 1 – оно было присвоено в момент 0. Значения `f1` и `f2` все еще неизвестны (равны `x`); они будут присвоены только в момент 2. Зная, что следующее событие в списке запланировано на момент 2, симулятор переводит время на две единицы вперед; на этот момент запланированы два события: обнуление `f1` и обнуление `f2`.

Симулятор выбирает эти события из списка и обновляет значения f1 и f2, после чего в произвольном порядке распространяет их до входов других вентилей, в данном случае вентиля g3. Это показано в верхней части рис. 1.5. Поскольку значения f1 и f2 равны 0, через две единицы времени f примет значение 0. Таким образом, на момент 4 (текущий момент 2 + 2 единицы) планируется событие обнуления f. Больше событий для момента 2 не осталось, и $monitor выводит на консоль вторую строку, показанную на рис. 1.2. В верхней части рис. 1.5 показано состояние схемы и списка событий в конце цикла 1 для момента 2.

Симулятор начинает следующий цикл: переводит время на 4 и записывает в f значение 0. Поскольку делать больше нечего (значений, подлежащих распространению, не осталось), $monitor выводит на консоль третью строку, показанную на рис. 1.2. В средней части рис. 1.5 показано состояние схемы и списка событий в конце цикла 1 для момента 4.

После этого время переводится на 12. Событий обновления на этот момент не запланировано, но есть событие исполнение блока initial. Исполнение возобновляется со строки 6, и a устанавливается в 1. Далее симулятор видит #6 и помещает событие исполнения блока initial в список, планируя его на момент 18. Цикл симуляции не закончен – необходимо распространить измененное состояние, получившееся в результате исполнения блока initial. В данном случае (см. нижнюю часть рис. 1.5) новое значение a распространяется до вентиля g1; модель этого вентиля исполняется. Поскольку теперь оба входа a и n_sel равны 1, через две единицы времени (на момент 14) планируется обновление значения выхода f (оно должно стать 1); это событие добавляется в список. Так как больше в момент 12 делать нечего, на консоль выводится четвертая строка, показанная на рис. 1.2; мы видим, что значение a равно 1, но значение f1 еще не изменилось. В нижней части рис. 1.5 показано состояние схемы и списка событий в конце цикла 1 для момента 12.

На рис. 1.6 показаны результаты следующего цикла симуляции, когда в момент 14 значение f1 становится равным 1. Из-за этого на момент 16 планируется обновление: присваивание f значения 1. Наконец, в момент 16 выход f принимает значение 1, а в момент 18 возобновляется исполнение блока initial; вызывается оператор $finish, и симуляция останавливается.

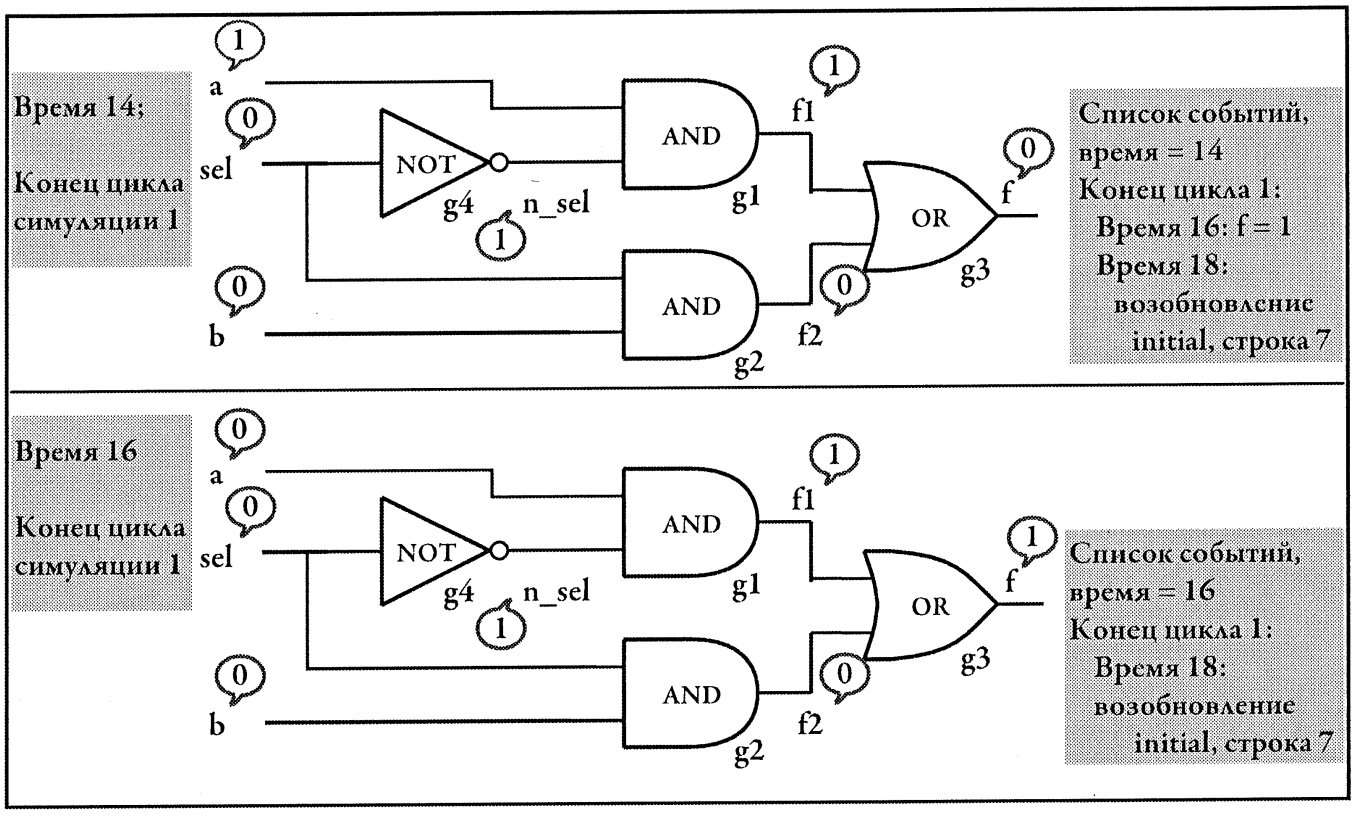



**Рис. 1.6.** Пошаговая схема работы симулятора (завершение)

Вы, вероятно, обратили внимание, что время симуляции измеряется в абстрактных «единицах времени». С единицами времени можно ассоциировать *временну́ю шкалу* и точность округления, однако в этой книге физические единицы не используются.

## 1.2.4. Модели исполнения SystemVerilog

В предыдущем разделе мы детально разобрали, как обрабатываются модели вентилей и процедурные операторы в SystemVerilog: как инкрементируется время, изменяются входные значения, отслеживаются обновления внутренних и выходных значений.

Для понимания SystemVerilog-описаний важно понимать, что в языке есть две модели исполнения: одна для вентилей, вторая для процедурных операторов. *Модель исполнения* определяет, как новые значения вычисляются на основе старых и как продвигается время.

- *Процедурная модель исполнения* была проиллюстрирована на примере блока `initial`. Операторы исполняются последовательно, пока не встретится маркер `#`; когда это происходит, исполнение приостанавливается на указанное время. В список событий добавляется соответствующее *событие исполнения*. Когда наступит время, исполнение модели возобновится с прерванного места.
- *Модель исполнения вентиля* была проиллюстрирована на примере примитивных вентилей. Если вход вентиля меняет значение, вычисляется реализуемая вентилем функция, в результате чего может измениться выходное

значение. Если оно действительно изменилось, соответствующее *событие обновления* добавляется в список событий (возможно, с задержкой). Когда наступит время, значение выхода будет обновлено, и новое значение распространится до вентилей, с которыми он соединен, после чего будут вычисляться их функции. Это продолжается до тех пор, пока все выходы не получат окончательные значения. Тем самым моделируются потоки данных в комбинационных схемах: когда изменяется вход, перевычисляется выход.

Вместе эти модели исполнения позволяют моделировать параллельные действия, описываемые вентилями и процедурными операторами. Симулятор управляет продвижением времени, распространением обновленных значений и исполнением процедурных операторов.

### 1.2.5. Зачем все это?

Для чего это нужно? Язык SystemVerilog позволяет описывать цифровые системы, состоящие из миллионов параллельно работающих компонентов. Используя симулятор, мы можем понять, правильно ли спроектирована модель с точки зрения функциональности и временны́х характеристик (timing). Это крайне важно в тех областях, где требуется с первой попытки получить правильную реализацию, заказную микросхему или ПЛИС.

## 1.3. Иерархия модулей

Язык SystemVerilog был бы не очень полезен, если его возможности ограничивались добавлением в модуль вентилей и их соединением. Важно, что модель описывается в виде модуля, в котором создаются экземпляры других модулей. Такая организация называется *иерархией модулей*. Для построения модулей можно использовать не только примитивные вентили (AND, OR и другие), но и более сложные модули, например рассмотренный выше `mux`.

Как это делается, показано в примере 1.4 и на рис. 1.7. Здесь мы возвращаемся к первоначальному варианту модуля `mux` из примера 1.1, продублировав его в строках 10–18. На примере модуля `two_bit_mux` (строки 1–8) показано, как с помощью модуля `mux` построить 2-битный мультиплексор 2:1, принимающий два 2-битных входа и выбирающий один из них для формирования 2-битного выхода. Это почти ничего не стоит. Давайте сначала познакомимся с новыми языковыми средствами.

До сих пор мы имели дело только с 1-битными переменными, называемыми также *скалярами*. В SystemVerilog можно определять переменные практически любого размера, указав размер в битах; такие многобитные переменные называются *векторами*. Например, в строке 2 определена 2-битная переменная (выход) `f`; нотация `[1:0]` означает, что это 2-битный *вектор*, в котором крайний левый бит имеет номер 1, а крайний правый – 0. Входы `a` и `b` также являются 2-битными векторами; 1-битный вход `sel` определяет, какой из двух векторов, `a` или `b`, будет подан на выход.

В строках 6 и 7 создаются два экземпляра модуля mux с именами b0 и b1 (соответствуют битам 0 и 1). Таким образом, модуль two_bit_mux *состоит* из двух копий модуля mux. В скобках указаны входные и выходные соединения экземпляров модуля. Порядок перечисления соединений важен – он должен соответствовать порядку в определении модуля mux.

```
1   module two_bit_mux
2    (output logic [1:0] f,
3     input logic [1:0]   a, b,
4     input logic         sel);
5
6     mux b0 (f[0], a[0], b[0], sel);
7     mux b1 (f[1], a[1], b[1], sel);
8   endmodule: two_bit_mux
9
10  module mux
11    (output logic f,
12    input logic   a, b, sel);
13
14    and #2 g1 (f1, a, n_sel),
15         g2 (f2, b, sel);
16    or  #2 g3 (f, f1, f2);
17    not g4 (n_sel, sel);
18  endmodule: mux
```

**Пример 1.4.** Модуль two_bit_mux

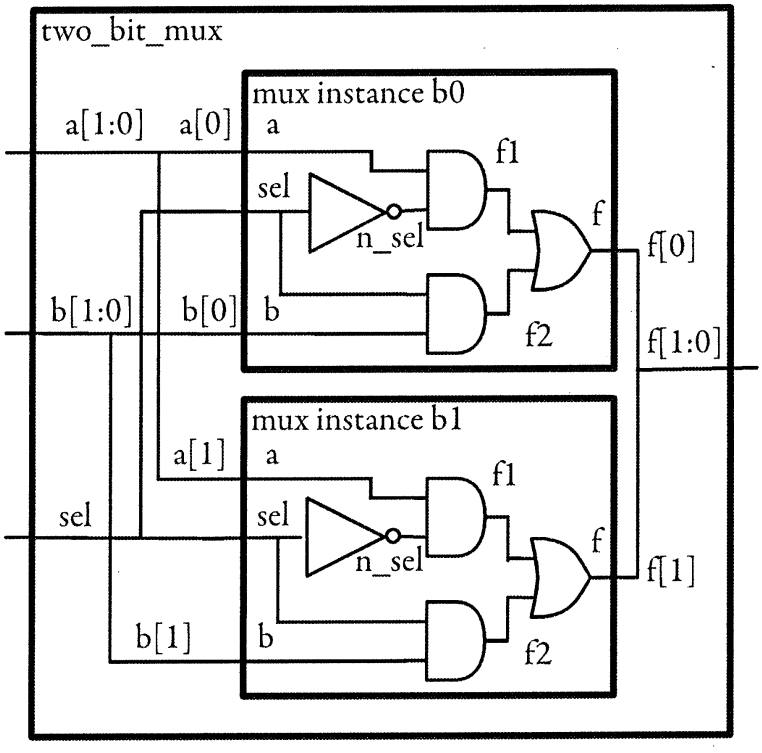



**Рис. 1.7.** Иерархическая логическая схема модуля two_bit_mux

Нужно пояснить использованную здесь языковую конструкцию. Запись f[0], где f – вектор, например 2-битный, называется *выбором бита*; конструкция позволяет выбрать один бит вектора. В строке 6 сказано, что бит 0 вектора f соединен с первым портом экземпляра b0 модуля mux. Далее в этой же строке говорится, что биты a[0], b[0] и sel также соединены с портами b0. Вспоминая определение модуля mux, мы понимаем, что f[0], a[0], b[0] и sel соединены соответственно с выходом f и входами a, b и sel. Важно, что соединение осуществляется с портами экземпляра; при создании экземпляра создается *копия* модуля, которой дается имя. Его внутренние логические вентили и переменные полностью независимы от вентилей и переменных в любом другом экземпляре того же модуля, например b1.

На рис. 1.7 показано физическое представление модуля из примера 1.4. Здесь можно видеть два экземпляра модуля mux внутри модуля two_bit_mux. На схеме также показаны имена переменных. Например, в строке 7 примера 1.4 создается экземпляр b1 модуля mux и производится соединение портов. В частности, порт a[1] модуля two_bit_mux соединяется с портом a экземпляра b1 модуля mux. Остальные порты соединяются аналогично. Вместе два модуля mux обеспечивают требуемую функциональность: подачу вектора a или b на выход f модуля two-bit-mux.

Сделаем несколько важных замечаний.

- С модулями можно делать две вещи: определять их и создавать их экземпляры. В строках 1–8 и 10–18 показаны определения модулей. В строках 6 и 7 внутри модуля two_bit_mux создаются два экземпляра модуля mux, определенного в строках 10–18. Таким образом, модуль two_bit_mux состоит из двух экземпляров модуля mux.
- Создание экземпляров порождает копии. На рис. 1.7 мы видим две копии модуля mux, т. е. всего 4 вентиля AND, 2 вентиля OR и 2 вентиля NOT. Экземпляры делают разные вещи, поскольку их порты соединены с разными переменными.
- Написать и протестировать модуль mux нужно только один раз; после этого его можно использовать в более крупных моделях.
- У каждого модуля свое пространство имен. Это означает, что переменные, такие как f, a, b и sel, объявленные в модуле, видны только в этом модуле[13]. Поскольку при создании экземпляра порождается копия модуля, у каждого экземпляра есть свои собственные переменные f, a, b и sel. Таким образом, sel в модуле two_bit_mux и sel в каждом экземпляре модуля mux суть разные переменные. Они логически и электрически эквивалентны, только потому что в модуле two_bit_mux соединены с переменной, которая по стечению обстоятельств тоже называется sel. По сути, sel в модуле two_bit_mux – это провод, который соединяет вход этого модуля с входами обоих экземпляров модуля mux. Эти соединения показаны на рис. 1.7.

## 1.4. Тестовое окружение для модуля mux

Спроектировав модуль, например mux, неплохо было бы проверить, что все сделано правильно. Для этого инженеры используют тестовое окружение (testbench). Если бы мы создали физический прототип системы, то поместили бы его на испытательный стенд, а к его входам и выходам подключили бы генераторы сигналов и осциллографы. В виртуальном мире моделирования мы пишем модель такого стенда, соединяем ее с тестируемой моделью (Design Under Test – DUT) и запускаем симуляцию, по результатам которой судим о правильности модели. Модуль, который осуществляет тестирование, так и называется – *тестовое окружение*.

### 1.4.1. Простой пример

На рис. 1.8 показана схема тестового окружения для модуля mux, а в примере 1.5 – ее описание на SystemVerilog. Тестовое окружение – это внешний модуль (muxTester), внутри которого создается экземпляр тестируемого модуля (mux). На схеме показаны только переменные, объявленные в строках 2 и 3

[13] Это не совсем так: на каждый именованный объект SystemVerilog можно сослаться, используя полное (иерархическое) имя, например two_bit_mux.b0.a.

описания muxTester: 3-битная переменная count (слева) и 1-битная (скалярная) переменная muxOut (справа). Тестируется все тот же модуль mux, который рассматривался на протяжении всей главы; его экземпляр создан в строке 5 модуля muxTester и назван dut. Выход f соединен с переменной muxOut, а три бита переменной count соединены с тремя входами mux.

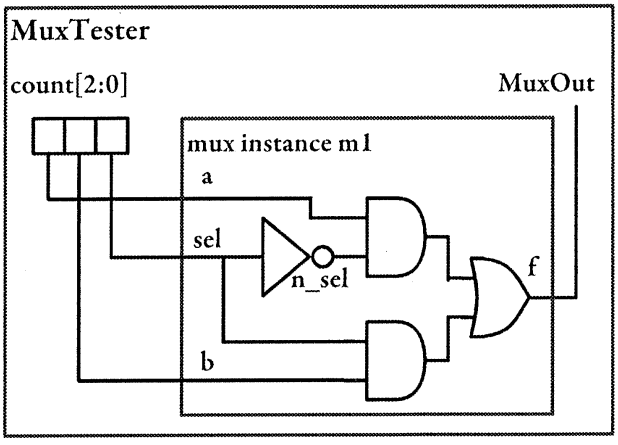



**Рис. 1.8.** Тестовое окружение для модуля mux

Чтобы разобраться в коде тестового окружения, нам понадобятся две новые конструкции SystemVerilog. *Константы с указанным размером* (*sized constants*) задаются в виде 3'b101. Так задается 3-битная константа с двоичным (b – binary) значением 101, т. е. 5. Можно также использовать десятичные (d – decimal) и шестнадцатеричные (h – hexadecimal) константы[14]. Как насчет основания 13? Забудьте об этом.

В блоке initial модуля muxTester описана функциональность, необходимая для тестирования модуля mux. Первым идет процедура $monitor, которая выводит сообщение на консоль всякий раз, когда изменяется значение count или muxOut. Затем следует цикл for. Мы еще не встречались с этой языковой конструкцией, но она очень похожа на циклы for в языках программирования. Цикл работает следующим образом. Переменная count инициализируется значением 0. Затем, до тех пор пока значение count не станет равным 3'b111, исполняется тело цикла (в нашем случае оно состоит только из оператора #10), после чего count инкрементируется. После обновления счетчика мы снова переходим на начало цикла и проверяем, совпадает ли значение count с 3'b111. При совпадении цикл завершается; блок initial ждет еще 10 единиц времени, после чего вызывает оператор $finish (который останавливает симуляцию).

Важно понимать, что происходит в теле цикла, когда исполняется оператор #10. Как мы уже знаем, #10 приостанавливает исполнение блока initial на 10 единиц времени. Пока он приостановлен, новое значение переменной count, полученное при инициализации или инкременте, распространяется через порты мультиплексора mux до вентилей внутри него. Симулятор моделирует работу вентилей с учетом задержек, и спустя четыре единицы времени, если выход f изменился, переменной muxOut присваивается новое значение с выхода мультиплексора.

Результат симуляции показан в строках 27–39 примера 1.5. Это вывод процедуры $monitor. В моменты времени, кратные 10, значение count обновляется. В моменты времени, оканчивающиеся на 4 (например, 34), muxOut получает новое значение.

---

[14] В языке SystemVerilog можно задавать константы и в восьмеричной системе счисления (o – octal).

Смысл цикла for в этом примере состоит в том, чтобы подать на входы mux все возможные комбинации значений. Глядя на консоль с результатами, можно сказать, правильно работает система или нет. Это стандартный подход к тестированию комбинационных схем – подать на входы все возможные комбинации значений и проверить значения выходов. Чтобы использовать рассмотренное тестовое окружение для верификации других комбинационных модулей с одним выходом, нужно только изменить размер переменной count и создать экземпляр другого модуля.

```
1 module muxTester;
2   logic [2:0] count;
3   logic muxOut;
4
5   mux dut (muxOut, count[2], count[1], count[0]);
6
7   initial begin
8     $monitor ($time,
9     " a b sel = %b, muxOut = %b", count, muxOut);
10
11    for (count = 0; count != 3'b111; count++)
12      #10;
13
14    #10 $finish;
15  end
16 endmodule: muxTester
17
18 module mux
19  (output logic f,
20   input  logic a, b, sel);
21
22  and #2 g1 (f1, a, n_sel),
23         g2 (f2, b, sel);
24  or  #2 g3 (f, f1, f2);
25  not    g4 (n_sel, sel);
26 endmodule: mux // вывод тестового окружения показан ниже
27                0 a b sel = 000, muxOut = x
28                4 a b sel = 000, muxOut = 0
29               10 a b sel = 001, muxOut = 0
30               20 a b sel = 010, muxOut = 0
31               30 a b sel = 011, muxOut = 0
32               34 a b sel = 011, muxOut = 1
33               40 a b sel = 100, muxOut = 1
34               50 a b sel = 101, muxOut = 1
35               54 a b sel = 101, muxOut = 0
36               60 a b sel = 110, muxOut = 0
37               64 a b sel = 110, muxOut = 1
38               70 a b sel = 111, muxOut = 1
39 $finish at simulation time          80
```

**Пример 1.5.** Тестовое окружение для модуля mux

Таким образом, можно легко отделить описание модели от средств тестирования. Воздействия на тестируемую модель и ее мониторинг осуществляются через входные и выходные порты; по модели синтезируется логическая схема с помощью других инструментов САПР. Важно, что для симуляции и синтеза используется одно и то же описание; вся функциональность, связанная с тестированием, инкапсулирована в отдельном модуле (в нашем случае в модуле muxTester).

## 1.4.2. Более интеллектуальное тестовое окружение

Цикл for в тестовом окружении можно написать по-другому: вычислить ожидаемое значение выхода mux и сравнить его с фактическим. В конце концов, процедурные операторы в блоке initial очень похожи на имеющиеся в языках программирования – с их помощью можно вычислить результат. Ниже приведен пример такого цикла for:

```
1   for (count = 0; count != 3'b111; count++) begin
2     #10;
3     if (count[0]) // если sel равно TRUE
4       if (muxOut != count[1])// если muxOut != b
5           $display("oops: a b sel = %b, muxOut = %b", count, muxOut);
6       else if (muxOut != count[2]) // если muxOut != a
7           $display("oops: a b sel = %b, muxOut = %b", count, muxOut);
8   end
```

Если эти 8 строк подставить в пример 1.5 вместо строк 8–12, сообщение будет выведено на консоль только в случае, когда значение muxOut неправильно. В этом цикле после задержки в 10 единиц времени исполняется оператор if (строка 3). У оператора if в SystemVerilog следующий синтаксис:

```
if (expression) thenStatement;
```

Если значение выражения expression равно TRUE (точнее, отлично от 0), то исполняется оператор thenStatement. У оператора может присутствовать и ветвь else, исполняемая в противном случае. Оператор if в строке 3 проверяет, равно ли TRUE значение count[0], соединенное с sel. Если да, на выходе mux должно быть значение входа b (count[1]); это проверяется в строке 4. Если muxOut не совпадает с b, то исполняется thenStatement. В данном случае вызывается процедура $display (строка 5); это оператор вывода на консоль, который мы обсудим ниже. Отметим, что else в строке 6 образует пару с if в строке 4; такие пары должны быть вам знакомы по языкам программирования.

До сих пор не встречавшаяся нам процедура $display похожа на процедуры вывода в языках программирования; она работает почти так же, как уже знакомая нам процедура $monitor. Отличие в том, что $display выводит данные на консоль в момент вызова; симулятор не ждет конца цикла симуляции, как в случае с $monitor[15]. В нашем примере значения count и muxOut печатаются, если

[15] Другое отличие состоит в том, что процедура $display выводит данные один раз, а $monitor – в конце каждого цикла симуляции.

произошла ошибка. Строки 6–7 соответствуют случаю, когда на выходе mux должно быть значение a. Процедура $display неоценима при отладке!

На самом деле приведенный код чересчур громоздкий. Как и во многих языках программирования, в SystemVerilog есть *условное выражение*, которым здесь можно воспользоваться. Оно имеет следующую форму:

```
(expression1) ? expression2 : expression3
```

Смысл такой: если значение выражения expression1 равно TRUE, то значением всего выражения будет значение expression2; в противном случае – значение expression3. Интересно, что условное выражение имеет ту же функциональность, что и мультиплексор.

Перепишем цикл for, используя условное выражение для вычисления ожидаемого результата mux:

```
1   for (count = 0; count != 3'b111; count++)
2     #10 if ( (count[0])? count[1] : count[2]) != muxOut)
3       $display("oops: a b sel = %b, muxOut = %b", count, muxOut);
```

Если эти три строки подставить вместо строк 8–12 в примере 1.5, то, как и раньше, сообщение будет выведено на консоль, только если mux выдает неправильное значение. Внутри цикла мы по-прежнему ждем в течение 10 единиц времени, а затем исполняем оператор if, проверяющий правильность выходного значения.

Условие оператора if выглядит так:

```
1   ((count[0])? count[1] : count[2]) != muxOut)
```

В нем проверяется, что значение выражения слева не равно значению выражения справа (muxOut). Выражение слева – условное выражение, описывающее функцию мультиплексора: если значение бита count[0] (соединенного с sel) равно TRUE, значением будет count[1] (b); в противном случае – count[2] (a). Таким образом, результат, count[1] или count[2], сравнивается с muxOut. При расхождении значений $display выводит сообщение на консоль.

Из этого раздела следует вынести два урока. Первый – это базовые принципы разработки тестовых окружений для комбинационных схем. Мы показали, что блоки initial и процедурные операторы могут быть использованы не только для описания функций тестового окружения (например, подачи значений на входы схемы), но и для спецификации функциональности схемы, что позволяет использовать тестовое окружение для проверки правильности модели.

Второй урок – описание функциональности мультиплексора с помощью оператора if-else или условного выражения. Ниже мы будем использовать подобные конструкции для описания цифровых систем. Для преобразования процедурных операторов в реализующую их схему применяются инструменты логического синтеза. Никто не проектирует большие системы без поддержки со стороны САПР, в т. ч. инструментов синтеза!

## 1.5. Резюме

Эта глава задумана как краткое введение в язык. Многие детали опущены – мы хотели, чтобы читатель почувствовал, что требуется от языка, используемого для описания цифровых систем и тестовых окружений. Основные особенности языка были показаны на примерах. Отметим наиболее важные из них.

- Время и параллельные действия. Фундаментальное требование, предъявляемое ко всем симуляторам и используемым в них языкам, – способность моделировать физическое время. Время в симуляторе называется виртуальным. Поскольку симулятор управляет временем, он может создать иллюзию одновременного исполнения нескольких действий. Это нужно, поскольку реальные электронные системы работают параллельно.
- Модули, экземпляры и иерархия. Модуль – базовая единица проектирования в языке. Модули могут представлять как простые логические функции, так и сложные системы. Модули можно копировать (создавать экземпляры); это позволяет строить системы из подсистем и дает возможность управлять сложностью модели, особенно если та состоит из миллионов логических вентилей. Никто не создает «плоские» модели (т. е. модели, в которых только один модуль) из миллионов вентилей NAND[16]; всегда используется иерархия: небольшие части проектируются, тестируются, а затем компонуются в более крупные подсистемы. Такой подход называется *компонентным* (*design by composition*) – система создается путем выбора и соединения компонентов.
- Структурные и процедурные модели. Структурное моделирование есть способ проектирования больших систем путем композиции ранее определенных модулей и вентилей. Процедурное моделирование использовалось для описания тестового окружения. Процедурные модели исполняются почти так же, как обычные программы; основное отличие состоит в том, что в них можно управлять временем. Ниже мы увидим, что оба подхода позволяют представлять функциональность больших систем. По абстрактному описанию может быть синтезирована логическая схема.

В последующих главах мы подробнее изучим язык и его приложения, сохраняя верность подходу на основе примеров.

## 1.6. Задачи и упражнения

**1.1.** Напишите и упростите логическое выражение, описывающее выход следующего SystemVerilog-модуля.

---

[16] Напомним, что функция NAND (штрих Шеффера) образует полную систему – через нее можно выразить любую булеву функцию. Другими словами, всякую булеву функцию можно реализовать в виде комбинационной схемы, состоящей только из вентилей NAND.

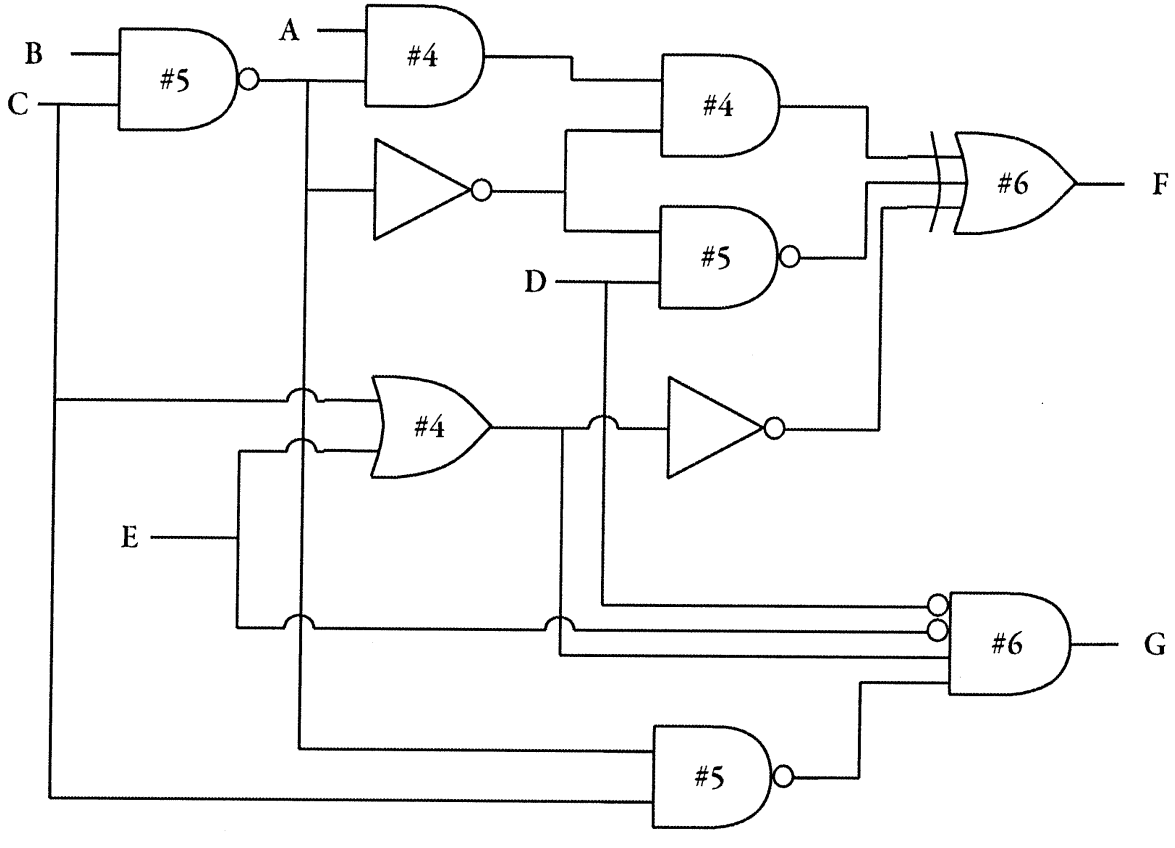
```
1 module trueBoole
2 (output logic f,
3    input logic a, b, c);
4
5   nor (f, f1, f2, f5);

6   or (f2, f3, f4, f5);
7   not (f1, a);
8   xor (f3, a, f1);
9   and (f4, f3, c, a, b),
10      (f5, a, c);
11 endmodule: trueBoole
```

**1.2.** Одна из функций тестового окружения комбинационной схемы – генерация всех возможных входных паттернов (комбинаций из нулей и единиц). Напишите тестовое окружение, которое делает это для модели с четырьмя входами. Каждый паттерн должен держаться в течение одной единицы времени. Ниже показан заголовок модуля. Считайте, что выходы `a-d` тестового окружения соединены с входами тестируемой модели.

```
1   module fourBitTest
2     (output logic a, b, c, d);
```

**1.3.** Напишите на языке SystemVerilog структурный модуль, описывающий следующую схему.

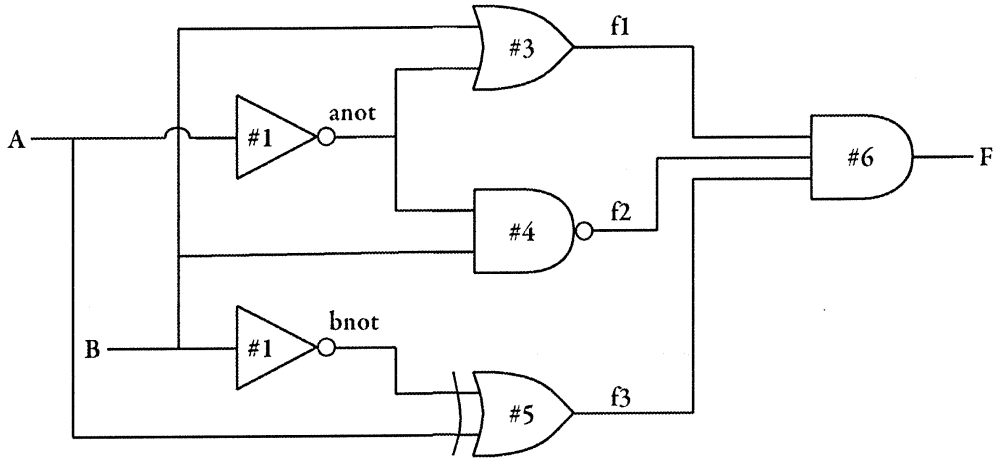



**1.4.** Обобщите решение задачи 1.2 для тестирования модуля из задачи 1.3.

**1.5.** Как работает вентиль NOR (используемый в SystemVerilog) с двумя входами для 4-значных логических значений? Напишите модуль и тестовое окружение для получения 4-значной таблицы истинности. Выведите на консоль каждую строку таблицы с помощью процедуры `$display`.

**1.6.** Показанная ниже схема управляется следующим блоком `initial`:

```
1  logic A, B;
2  initial begin
3     A = 0;
4     B = 1;
5     #15 A = 1;
6     B = 0;
7     #15 $finish;
8  end
```




Нарисуйте временну́ю диаграмму этой схемы, начиная с момента 0 и до установления стационарных значений всех сигналов.

**1.7.** Полным сумматором называется логическая схема, которая складывает два входных бита и бит переноса и формирует сумму и выходной бит переноса. Сумма получается путем применения операции XOR (исключающее ИЛИ) ко всем трем битам, а выходной бит переноса равен TRUE, если не менее двух входных битов равны 1.

**A)** Напишите модуль `fullAdd` с тремя входами и двумя выходами, используя примитивные вентили.

**B)** Создайте 4 экземпляра этого модуля внутри модуля полного 4-битного сумматора. Входами этого модуля должны быть 4-битные векторы `a` и `b` и скаляр `carryIn`. На выходе должны получиться 4-битная сумма и скаляр `carryOut`. Соедините порты экземпляров таким образом, чтобы выходной бит переноса каждого каскада был соединен с входным битом переноса следующего каскада[17].

**C)** Обобщите решение задачи 1.2 так, чтобы тестовое окружение порождало все возможные комбинации значений девяти входных битов. Проверьте правильность работы полного сумматора.

[17] Выходной бит переноса последнего каскада соединяется с выходом `carryOut`.

# Часть I

# МОДЕЛИ УРОВНЯ РЕГИСТРОВЫХ ПЕРЕДАЧ

# Глава 2

# Комбинационные схемы

Отличительная характеристика комбинационных схем состоит в том, что значения их выходов можно задать с помощью булевых функций, зависящих от значений входов (и только от них). Логические функции есть абстракное описание электронных схем, в которых входы преобразуются в выходы непосредственно, без использования элементов памяти. Каждому входному паттерну (комбинации нулей и единиц, задающей значения входов) из множества I соответствует один и только один выходной паттерн из множества O.

Математически комбинационная схема описывается функцией вида:

$$F: I \rightarrow O.$$

В реальной комбинационной схеме сигналы распространяются с задержкой, поэтому значение функции появляется не сразу, а только тогда, когда входные значения полностью распространяются через вентили схемы до ее выходов.

## 2.1. Моделирование комбинационных схем

Логическую функцию можно представить в виде формулы, таблицы истинности с $2^n$ строками, где $n$ – число входов, списка минтермов[18], списка макстермов[19] или модели на языке описания аппаратуры, например SystemVerilog.

В современном подходе к проектированию аппаратуры моделирование и синтез имеют первостепенное значение. В описаниях аппаратуры часто используются абстрактные процедурные конструкции, что дает возможность инструментам синтеза производить оптимизацию схем с учетом таких ограничений, как занимаемая площадь, задержка распространения сигналов и потребляемая мощность. Хотя на SystemVerilog можно описать комбинационную схему путем соединения примитивных логических вентилей (AND, OR, NOT и т. п.), как было описано во введении, в этой книге мы не будем прибегать к такому подходу – для проектирования больших и сложных систем применяются инструменты синтеза, оперирующие с моделями более высокого уровня абстракции.

---

[18] В виде дизъюнктивной нормальной формы (ДНФ); минтерм – конъюнкция литералов.

[19] В виде конъюнктивной нормальной формы (КНФ); макстерм – дизъюнкция литералов.

## 2.1.1. Операторы assign и always_comb

SystemVerilog предлагает два подхода к процедурному описанию комбинационных схем: операторы `always_comb` и `assign`. Оба показаны в примере 2.1, где приведена модель простого сумматора-вычитателя: в зависимости от входа `select_plus` производится либо сложение `a+b` (если `select_plus` имеет значение TRUE), либо вычитание `a-b`. Результат выдается на выходе `result`.

```
1 module sum_and_dif_A
2   (output logic [3:0] result,
3    input logic [3:0] a, b,
4    input select_plus);
5
6   always_comb
7     if (select_plus)
8         result = a + b;
9     else result = a - b;
10 endmodule: sum_and_dif_A
11
12 module sum_and_dif_B
13   (output logic [3:0] result,
14    input logic [3:0] a, b,
15    input logic select_plus);
16
17   assign result = (select_plus) ? a + b : a - b;
18 endmodule: sum_and_dif_B
```

**Пример 2.1.** Операторы always_comb и assign

Оператор `always_comb` показан в строках 6–9. Можно считать, что это бесконечный цикл. В начале цикла мы ждем, когда изменится значение одной или нескольких переменных, встречающихся в правых частях операторов присваивания (в данном случае `a` или `b`) или в условиях (в данном случае `select_plus`). Как только это произойдет, исполняется оператор, записанный после ключевого слова `always_comb` (в данном случае оператор `if-else`), и вычисляются новые значения выходов (в данном случае `result`). Поскольку `always_comb` работает как цикл, он ожидает следующего изменения значений входов, после чего исполняется снова.

Оператор `assign` функционально похож на `always_comb` и часто применяется для однострочных логических выражений. В строке 17 он ожидает изменения любой из переменных, встречающихся в правой части (в данном случае `a`, `b` или `select_plus`); когда это происходит, оператор исполняется и ожидает следующего изменения. Здесь в правой части использовано *условное выражение* для вычисления результата. Если значение `select_plus` равно TRUE, то значение выражения будет определяться выражением после знака ? (`a+b`); в противном случае – выражением после знака : (`a-b`). Результирующее значение присваивается переменной `result`.

Оба модуля описывают эквивалентные комбинационные схемы, они взаимозаменяемы. Когда же предпочесть `assign`, а когда `always_comb`? Обычно опера-

тор assign используется для простых однострочных выражений, как показано выше. В операторе always_comb можно использовать другие процедурные операторы (например, case), чтобы описать более сложное поведение.

В операторе assign можно через запятую указать несколько присваиваний. Предположим, что требуется описать модуль, который определяет, равно ли нулю значение его входа (см. пример 2.2). Здесь мы видим два присваивания, разделенных запятой: первое описывает выход neq, а второе – выход eq. В строке 6 говорится, что eq равно TRUE, когда value равно 0, в строке 5 neq определяется как дополнение (отрицание) eq. Мы поставили neq на первое место, чтобы подчеркнуть важный момент: операторы assign и входящие в них присваивания исполняются не в процедурном порядке (т. е. сначала первый, потом второй и т. д.). На самом деле каждое присваивание задает отдельную функцию, вычисляемую при изменении значения хотя бы одной из ее переменных. Выражения для neq и eq можно записать в любом порядке, без изменения смысла кода.

```
1  module compare
2    (output logic eq, neq,
3     input logic [3:0] value);
4
5    assign neq = ~eq,
6    eq = (value == 0);
7  endmodule: compare
```

**Пример 2.2.** Несколько операторов assign

```
1  module add_sub_compare
2    (output logic [3:0] result,
3     input logic [3:0] a, b,
4     output logic neq, eq,
5     input logic plus_minus);
6
7    sum_and_dif_A alu (result, a, b, plus_minus);
8    compare       c (eq, neq, result);
9  endmodule: add_sub_compare
```

**Пример 2.3.** Создание экземпляра модуля

Для комбинационных схем можно задать задержки распространения сигналов, однако инструменты синтеза их игнорируют. Вместо этого инструменту указывается максимальная задержка системы в целом, выраженная в терминах тактовой частоты, а он синтезирует схему с учетом этого ограничения.

Рассмотрим модель, состоящую из сумматора-вычитателя, такого как в модуле sum_and_dif_A (пример 2.1), на выходе которого стоит компаратор, такой как в модуле compare (пример 2.2), определяющий, равен ли результат нулю. Этот новый модуль add_sub_compare показан в примере 2.3 и на рис. 2.1. В строках 7 и 8 создаются экземпляры модулей: сначала указывается имя модуля, экземпляр которого создается, а за ним имя экземпляра и список портов. В строке 7 создается экземпляр модуля sum_and_dif_A с именем alu. Порты result, a, b и plus_minus модуля add_sub_compare соединяются с четырьмя портами модуля

sum_and_dif_A. Отметим, что порты перечисляются в порядке, определенном в модуле sum_and_dif_A (строки 2–4 в примере 2.1).

В этом примере выход result модуля sum_and_dif_A соединен с входом модуля compare (т. е. result является не только выходом add_sub_compare, но и входом compare), а выходы модуля compare, eq и neq, соединены с выходами add_sub_compare. Переменную result можно интерпретировать как провод, соединяющий выход одного модуля с входом другого.

На рис. 2.1 показана структура модуля add_sub_compare. Функциональность модулей представлена «облаками», а границы – прямоугольниками. Рисунок показывает, как результаты одного модуля подаются на вход следующего, а также то, что имена соединений могут различаться на разных уровнях иерархии. Например, в модуле add_sub_compare сигнал выбора операции (сложение или вычитание) называется plus_minus, тогда как в модуле sum_and_dif_A он называется select_plus. Это вызвано тем, что при создании экземпляра модуля в строке 7 указано, что порт plus_minus соединен с select_plus, а значит, их электрические (и логические) значения одинаковы.

Из рассмотренных примеров следует вынести следующие уроки.

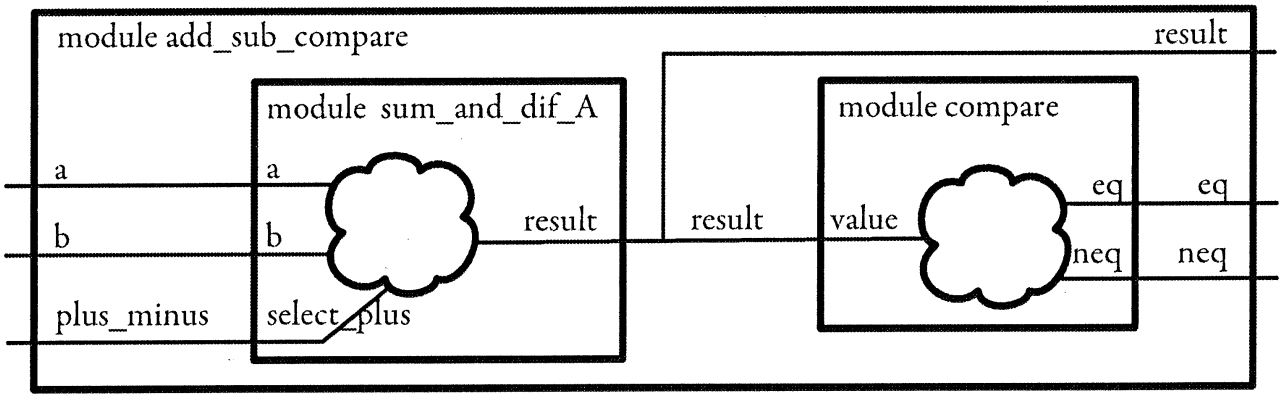



**Рис. 2.1.** Структура модуля: создание экземпляров и именование сигналов

- Совсем необязательно описывать схему с помощью примитивных вентилей, описанных во введении. Моделирование на уровне вентилей полезно при изучении цифровой схемотехники, однако современные инструменты синтеза сами разбираются с низкоуровневыми деталями и строят функционально корректные схемы[20], удовлетворяющие ограничениям по занимаемой площади, энергопотреблению и задержке распространения сигналов.
- Каждый оператор always_comb и каждое присваивание в операторе assign можно рассматривать как виртуальный модуль в том смысле, что это са-

[20] Инструмент синтеза по описанию аппаратуры строит логическую схему, функционально эквивалентную этому описанию; за корректность исходного описания он ответственности не несет.

модостаточная комбинационная схема. Представьте, что вы придумали свой логический вентиль и нарисовали его графическое обозначение вместо always_comb или assign, сохранив входы и выходы. Мы использовали для этой цели изображение облака.

- В примере 2.3 и на рис. 2.1 показана иерархическая структура модуля add_sub_compare. Теперь в других частях проекта можно создать экземпляр этого модуля. На рис. 2.1 также показано, что имена, определенные в модуле, видны только внутри самого модуля; все детали внутреннего устройства модуля скрыты от его пользователя (создателя экземпляра).

## 2.1.2. Вы уверены, что это комбинационные схемы?

Говоря об операторах always_comb и assign из примера 2.1, важно отметить, что в них ничего не запоминается; это функции без состояния (или без памяти). Иначе говоря, хотя a, b и select_plus могут храниться где-то в регистрах объемлющей системы, эти места являются внешними по отношению к операторам always_comb и assign. Семантика always_comb и assign такова, что при изменении значений входов a, b или select_plus сразу же перевычисляется значение выхода result. Старое значение result не участвует в вычислении, оно нигде не запоминается. Это определяющая характеристика комбинационных схем; always_comb и assign ей соответствуют.

Инструмент синтеза анализирует содержимое always_comb и assign, проверяя, что они действительно не имеют состояния. На основе анализа делается вывод о возможности реализации в виде комбинационной схемы. Если оператор предполагает сохранение некоторых значений между его исполнениями, схема комбинационной не является. В этом случае инструмент может выдать предупреждение или сообщение об ошибке. Если он продолжает работу, значит, для обеспечения требуемой функциональности были добавлены защелки (элементы памяти).

Рассмотрим модуль из примера 2.4, не являющийся комбинационным. Вход hold сигнализирует, нужно ли запоминать предыдущее значение sum. Если hold установлен (имеет значение 1), то новое значение sum не вычисляется, а на выходе держится старое, запомненное ранее. Если же hold сброшен (имеет значение 0), то при изменении значения входа a или b исполняется тело цикла и значение sum обновляется. Такая схема не является комбинационной: в ней нужно запомнить значение sum между исполнениями always_comb. Говоря о физической реализации, если при изменении значения входа a или b установлен сигнал hold, то на выходе устанавливается предыдущее значение sum, извлеченное из памяти. Это не комбинационное поведение!

```
1  module notCombinational
2  (input logic [3:0] a, b,
3     output logic [3:0] sum,
4     input logic hold);
5
6    always_comb
7    if (~hold)
8        sum = a + b;
9  endmodule: notCombinational
```

**Пример 2.4.** Некорректное описание

Для реализации модуля notCombinational инструмент синтеза должен был бы использовать защелки, управляемые сигналом hold. Большинство инструментов предупреждает, что у модели есть состояние и что для реализации понадобятся защелки. (В примере 2.4 следовало бы использовать оператор always_latch вместо always_comb.)

Приведем несколько рекомендаций по описанию комбинационных схем, т. е. схем, не имеющих состояния.

- Проверьте, что модель исполняется при изменении значения любого входа. Ведь если изменяется вход, может измениться и выход. Применение операторов assign или always_comb гарантирует это.
- Проверьте, что выход (или выходы) оператора always_comb или assign всегда перевычисляется при изменении значений входов. В примере 2.4 sum не перевычисляется, если установлен hold, – это нарушение данного правила. Дело в том, что у оператора if в строке 7 нет ветви else. Оператор assign гарантирует, что значение выхода обновляется, но при использовании always_comb об этом должен позаботиться разработчик.
- Проверьте, что у оператора always_comb нет побочных эффектов; например, он не должен увеличивать счетчик.

Еще раз: фундаментальная характеристика комбинационной схемы – отсутствие состояния. При любом изменении на входе необходимо перевычислять значение выхода, причем в вычислении должны участвовать только текущие значения входов. Это в точности соответствует семантике операторов always_comb и assign.

# 2.2. Использование операторов assign и always_comb

## 2.2.1. Оператор always_comb

Из ранее сказанного понятно, что оператор always_comb – один из столпов моделирования комбинационных схем. Этот оператор может встречаться в определении модулей и имеет следующий вид:

```
1   always_comb statement;
```

где statement – оператор, например a = b + c. В этом случае строка

```
1   always_comb a = b + c;
```

означает, что при изменении значения переменной b или c (или обеих сразу) симулятор исполнит оператор, вычисляющий новое значение a, после чего станет ожидать следующего изменения b или c. Если новое значение a отличается от старого, симулятор распространяет его до других моделей, как описано в разделе 1.2.2. Таким образом, оператор always_comb всегда готов к изменению значений входов и, когда это происходит, исполняет свое тело. Если бы мы синтезировали модуль, содержащий только этот оператор, в предположении что a, b и c – скаляры, то получили бы полусумматор.

Как правило, для моделирования поведения нужно нечто большее, чем просто a = b + c. Если для описания требуется задействовать несколько операторов, они заключаются в скобки begin и end[21], например:

```
1   always_comb begin
2     sum = b + c;
3     dif = b - c;
4   end
```

В этом случае всякий раз, когда b или c изменяется, вычисляются новые значения sum и dif (если новые значения отличаются от старых, они распространяются). Этот пример также демонстрирует, что в операторе always_comb можно сформировать несколько выходов для одних и тех же входов; все они вычисляются вместе при изменении хотя бы одного из входов.

В блоке begin-end можно использовать и более сложные процедурные операторы, по сравнению с sum = b + c; ниже мы увидим, что для описания логических функций часто применяются операторы if, if-else и case.

Когда оператор always_comb компилируется с остальной SystemVerilog-моделью, проверяется, что нет других операторов, которые присваивали бы значения выходам этого always_comb. Если требуется смоделировать ситуацию, когда несколько выходов формируют значение (drive) одной переменной, как это бывает в шинах, нужно использовать операторы assign или обычные блоки always. Эти вопросы будут обсуждаться ниже.

В начале симуляции все операторы always_comb исполняются хотя бы раз в момент времени 0 после исполнения всех блоков initial и always, например являющихся частью тестового окружения. Таким образом, с самого начала значения выходов блоков always_comb согласованы с другими начальными значениями.

## 2.2.2. Оператор assign

Оператор assign, также известный как оператор *непрерывного присваивания*, – это вторая конструкция, применяемая для описания комбинационных схем. Его можно использовать в любом месте модуля, как и некоторые другие конструкции, с которыми мы познакомимся позже. Оператор имеет вид:

```
1   assign statement;
```

где statement – присваивание. Например, если statement – это a = b + c, оператор assign будет выглядеть так:

```
1   assign a = b + c;
```

Оператор работает следующим образом: всякий раз, когда изменяется значение какой-нибудь переменной, стоящей в правой части присваивания (в данном случае b или c), вычисляется значение всего выражения, и значение

[21] Поэтому операторы always, always_comb и initial часто называют блоками.

переменной в левой части обновляется. После этого оператор ожидает изменения значений переменных в правой части. Название *непрерывное присваивание* говорит о том, что операция исполняется постоянно (непрерывно) – так же, как оператор always_comb, который постоянно ожидает изменения значений входных переменных. Оператор не нужно вызывать при каждом изменении входов – переменная в левой части (в данном случае a) будет обновляться автоматически.

Выражение в правой части присваивания может быть сколь угодно сложным: в нем может быть несколько операторов и скобки; разрешается вызывать написанные на SystemVerilog функции, о чем пойдет речь ниже. В следующем примере показано, как в рамках одного оператора assign записать несколько присваиваний. Каждое присваивание рассматривается как отдельный оператор, который постоянно ожидает изменения переменных в правой части и исполняется, когда это происходит.

```
1   assign a = b,                  // a будет иметь то же значение, что и b
2   muxOut = (select)? in1 : in0, // ( )? : - условная операция.
                                      Если значение select равно TRUE,
                                      значением выражения будет значение in1;
                                      иначе - in0.
                                      Таким образом, если select установлен,
                                      выходу muxOut присваивается значение входа in1;
                                      иначе - in0.
3   f = (f1 & f2) | (f3 & f4),     // f - логическое ИЛИ двух булевых произведений (И)
                                      f1 & f2 и f3 & f4.
4   eq = (value == 3);             // значение истинности выражения (value == 3),
                                      TRUE (1) или FALSE (0), присваивается eq
```

Переменная в левой части непрерывного присваивания может стоять в левой части другого непрерывного присваивания. Какое значение получается в результате, мы обсудим ниже. Обычно при проектировании логических схем такой подход не применяется, поскольку ведет к усложненному коду, однако некоторые конструкции, в т. ч. шины, часто описывают подобным образом.

В правой части разрешается использовать любые операции SystemVerilog.

## 2.2.3. Процедурное моделирование с помощью if и case

Чтобы описать логику модели, внутри always_comb можно использовать некоторые процедурные операторы. В языке SystemVerilog есть много процедурных операторов, встречающихся в традиционных языках программирования (см. главу 11), но лишь некоторые из них могут применяться для описания комбинационных схем. Мы поговорим о наиболее распространенных: if, if-else и case.

Пусть f, a, b и c – 1-битные переменные. Рассмотрим, как на SystemVerilog записывается модель, соответствующая следующей формуле булевой алгебры:

```
f = (a • b) + (b • c) + (a • c)
```

В SystemVerilog используется другая нотация; эквивалентное выражение выглядит так:

```
1   assign f = (a & b) | (b & c) | (a & c);
```

Его можно записать также с использованием операторов if и if-else, как показано в примере 2.5. Оба модуля в этом примере имеют ту же функциональность, что и оператор assign, приведенный выше.

Рассмотрим, как работает оператор if в SystemVerilog. За ключевым словом if в круглых скобках записывается условие:

```
1   if (conditional_expression) statement;
```

В строке 7 примера 2.5 условие имеет вид конъюнкции (И) входов a и b (a & b). Если условие истинно (имеет значение TRUE), то исполняется оператор, следующий за if (в данном случае это f = 1). Иначе говоря, если a и b имеют значение 1, то исполняется присваивание f = 1.

```
1  module if1
2    (input logic a, b, c,
3     output logic f);
4
5    always_comb begin
6      f = 0;
7      if (a & b) f = 1;
8      if (c & (a ^ b)) f = 1;
9    end
10 endmodule: if1
11
12 module if2
13   (input logic a, b, c,
14    output logic f);
15
16   always_comb begin
17     if (a & b) f = 1;
18     else if (c & a ^ b)
19       f = 1;
20     else f = 0;
21   end
22 endmodule: if2
```

**Пример 2.5.** Оператор if

Оператор if в строке 8 читается аналогично, нужно только знать, что знаком ^ обозначается исключающее ИЛИ (XOR). Таким образом, в строке 8 говорится, что если конъюнкция c с результатом a XOR b имеет значение TRUE, то также исполняется присваивание f = 1.

Совокупное действие блока always_comb в модуле if1 можно описать следующим образом: при изменении значения a, b или c установить f в 0; затем, если условие хотя бы в одном операторе if (строки 7 и 8) имеет значение TRUE, установить f в 1. Разумеется, если условие в обоих операторах if имеет значение FALSE, f останется равным 0. Убедитесь, что это действительно эквивалентно показанной выше формуле булевой алгебры.

В модуле if2 из примера 2.5 продемонстрирована альтернативная реализация с использованием операторов if-else. Общая форма оператора if-else такова:

```
1   if (conditional_expression) statement;
2   else statement;
```

Как и в языках программирования, если условие conditional_expression ложно (имеет значение FALSE), то исполняется оператор, следующий за ключевым словом else, а не тот, что следует за условием.

Сделаем замечание, касающееся двух блоков always_comb в примере 2.5. При использовании оператора always_comb нужно помнить о следующей особенности: каждое исполнение оператора должно завершаться присваиванием в выходную переменную (вторая рекомендация из раздела 2.1.2). В нашем случае это f. Уверены ли вы, что ей всегда присваивается значение в строках 16–21? Оказывается, что да. В другом блоке always_comb, в строках 5–9, тот факт, что f

получает значение, очевиден. Популярный подход к написанию блока always_comb – присвоить выходной переменной значение 0 или 1, а затем написать процедурные операторы, которые, если нужно, присвоят другое значение. Выбирайте тот способ, который кажется вам наиболее ясным.

В примере 2.6 показано, как описать то же поведение, что и в примере 2.5, с помощью оператора case. Оператор case в SystemVerilog похож на операторы case и switch в языках программирования: в зависимости от значения выражения исполняется один из нескольких операторов. В данном случае выражение, указанное в круглых скобках после ключевого слова case (строка 6), определяет, какой из вариантов (строки 7–14) выбрать для исполнения.

```
1 module basicCase
2   (input logic a, b, c,
3    output logic f);
4
5   always_comb
6     case ({a, b, c})
7       3'b000: f = 0;
8       3'b001: f = 0;
9       3'b010: f = 0;
10      3'b011: f = 1;
11      3'b100: f = 0;
12      3'b101: f = 1;
13      3'b110: f = 1;
14      3'b111: f = 1;
15    endcase
16 endmodule: basicCase
```

**Пример 2.6.** Некорректное описание

Чтобы разобраться в этом примере, нам понадобится ввести две новые конструкции. Первая – это конкатенация. В SystemVerilog переменные с известной разрядностью (sized variables) можно конкатенировать (сцеплять) для образования новых переменных. Оператор конкатенации {} показан в строке 6. Оператору конкатенации передан список переменных, разделенных запятой. Смысл в том, чтобы сделать новую переменную, составленную из a, b и c. Так, если a и b равны 1, а c – 0, то переменная {a,b,c} имеет двоичное значение 110.

Кроме того, в SystemVerilog есть способ определять константы с заданной разрядностью (sized constants). Он демонстрируется в строках 7–14. Конструкция 3'b110 в строке 13 определяет 3-битную константу, значение которой в двоичной записи есть 110 (то есть 6). Можно записать ее в виде 3'd6 (3-битная константа с десятичным значением 6). (Первое число – размер в битах, второе – значение. Буквы b и d задают основание системы счисления. Также допустима буква h – шестнадцатеричная система[22].)

Оператор case находит первый подходящий вариант для значения выражения case[23]. Затем исполняется оператор, стоящий справа, после чего исполнение продолжается с оператора, следующего за endcase. Смысл приведенного оператора always_comb состоит в том, что при изменении значения любой из переменных a, b или c исполняется оператор case, который устанавливает f в 1 или 0, после чего always_comb ожидает следующего изменения значений входов. Поскольку мы перечислили все возможные значения 3-битной переменной {a, b, c}, выход f обновляется при каждом исполнении цикла; значит, инструмент синтеза будет считать это описанием комбинационной схемы. Ключевые

[22] И буква o – восьмеричная система.

[23] Пока можно считать, два значения подходят (match) друг другу, если они совпадают.

слова begin-end здесь не нужны, поскольку вся конструкция case-endcase в строках 6–15 – это один оператор.

Подход, продемонстрированный в примере 2.6, – распространенный метод описания логических функций. По сути, мы задали таблицу истинности. Код можно сократить. Следующие фрагменты эквивалентны примеру 2.6.

```
1   always_comb
2     case ({a, b, c})
3       3'b000: f = 0;
4       3'b001: f = 0;
5       3'b010: f = 0;
6       3'b100: f = 0;
7       default: f = 1; // оператор по умолчанию
8     endcase
```

Ключевое слово default говорит, что если значение выражения case не совпадает ни с одним из вариантов, то исполняется оператор, указанный справа от него (оператор по умолчанию).

```
1   always_comb
2     case ({a, b, c})
3       3'b000,
4       3'b001: f = 0; // исполнить этот оператор, если выражение case
                       // имеет значение 3'b000 или 3'b001
5       3'b010: f = 0;
6       3'b100: f = 0;
7       default: f = 1;
8     endcase
```

В этом фрагменте первые два варианта разделены запятой. Это означает, что при совпадении значения выражения с любым из них исполняется соответствующий оператор. В следующих разделах мы увидим, что существуют и другие способы описания логических функций с помощью оператора case.

### 2.2.4. Задание несущественных комбинаций с помощью unique case

Часто функции задаются не полностью, а частично, – разработчику безразлично, каким будет значение выхода для некоторых комбинаций значений входов. Такие комбинации называются *несущественными* (*don't cares*). Как правило, несущественные комбинации – это те комбинации, которые не могут возникать, поэтому инструмент синтеза может сам сопоставить им то значение функции, 1 или 0, которое даст оптимальную реализацию.

Сейчас, когда в процессе синтеза применяются высокоэффективные оптимизации, мало кто специфицирует несущественные комбинации. Кроме того, из соображений безопасности разработчик может предусмотреть специальный сигнал ошибки, устанавливаемый при возникновении какой-либо несущественной комбинации и сообщающий, что что-то пошло не так.

```
1 module alu
2   (input logic [7:0] a, b,
3    output logic [7:0] result,
4    input logic [2:0] op);
5
6   always_comb // oops!
7     case (op)
8       3'b100: result = a + b;
9       3'b010: result = a - b;
10      3'b001: result = a & b;
11      3'b110: result = a | b;
12      3'b011: result = a ^ b;
13    endcase
14 endmodule: alu
```

**Пример 2.7.** Простое АЛУ

```
1 module aluUnique
2   (input logic [7:0] a, b,
3    output logic [7:0] result,
4    input logic [2:0] op);
5
6   always_comb
7     unique case (op)
8       3'b100: result = a + b;
9       3'b010: result = a - b;
10      3'b001: result = a & b;
11      3'b110: result = a | b;
12      3'b011: result = a ^ b;
13    endcase
14 endmodule: aluUnique
```

**Пример 2.8.** Оператор unique case

Мы все же включили этот материал в книгу – в некоторых ситуациях такая оптимизация может быть полезна. Рассмотрим модель простого арифметико-логического устройства (АЛУ), которое может производить пять операций над двумя 8-битными значениями и выдает 8-битный результат (см. пример 2.7). Эти пять операций перечислены в строках 8–12 оператора case: сложение, вычитание, побитовое И, побитовое ИЛИ и побитовое исключающее ИЛИ. Например, если op равно 3'b001 (строка 10), то вычисляется побитовое И значений входов a и b.

Все бы хорошо, только схема, описанная в примере 2.7, не комбинационная! Поскольку op – 3-битная переменная, у нее существует восемь возможных значений, а мы перечислили только пять из них. Инструмент синтеза проанализирует это описание и увидит, что если op имеет значение 3'b111, 3'b000 или 3'b101, то значение result – результат предыдущего исполнения цикла.

Чтобы исправить ошибку, мы могли бы добавить вариант default, как было показано выше. Другой способ – воспользоваться оператором unique case. Если перед ключевым словом case стоит unique, тем самым утверждается, что возможно совпадение *с одним и только одним* из перечисленных вариантов. Для этого в примере 2.8 нужно изменить строку 7 на unique case (op). Все остальное остается прежним.

Инструмент синтеза, видя оператор unique case, понимает, что должен рассматривать только значения, перечисленные в описании: значение переменной op должно совпасть с одним и только одним вариантом; все остальные значения невозможны, а значит, несущественны.

Что произойдет в процессе симуляции, если в другой части модели имеется ошибка и op принимает значение, совпадающее с одной из пропущенных комбинаций: 3'b111, 3'b000 или 3'b101? В этом случае симулятор сообщит об ошибке и прекратит исполнение. Таким образом, unique case автоматически проверяет, что на вход op подаются только допустимые значения, а заодно сообщает инструменту синтеза, что все остальные комбинации несущественны. Использование unique case наделяет симулятор и инструмент синтеза одинаковым пониманием проектируемой схемы. Мы еще вернемся к этому оператору в главе 11.

## 2.2.5. Упрощение спецификации с помощью ? и casez

Иногда строка, столбец или блок карты Карно[24] содержит одно и то же значение. Этим фактом можно воспользоваться, чтобы упростить спецификацию. В примере 2.9 и на рис. 2.2 показана простая карта Карно. Для ее представления удобно использовать специальный подстановочный символ (wildcard) ? (вместо ? можно подставить как 1, так и 0). Чтобы знак ? считался допустимым в вариантах case, нужно использовать ключевое слово casez. В строке 7 нашего примера говорится, что если вход a установлен, то при любых значениях остальных входов значением f будет 0. Этот вариант соответствует нижней строке карты Карно на рис. 2.2. Остальные варианты соответствуют другим областям карты Карно.

```
1  module simpleKmap
2   (input logic a, b, c,
3    output logic f);
4
5   always_comb
6     casez ({a, b, c})
7       3'b1??: f = 0;
8       3'b01?: f = 1;
9       3'b000: f = 1;
10      3'b?01: f = 0;
11    endcase
12 endmodule: simpleKmap
```

**Пример 2.9.** Использование casez и ?

Отметим, что варианты в строках 7 и 10 перекрываются. Некоторые значения входа {a, b, c}, например 3'b101, соответствуют обеим строкам. В этом нет ничего плохого, особенно если учесть, что эти строки представляют два простых импликанта[25], на которых функция равна нулю, и что эти импликанты на карте Карно пересекаются.

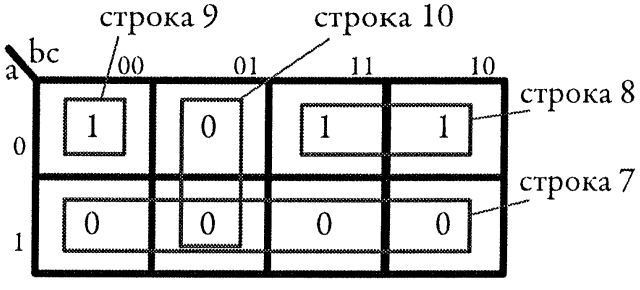



**Рис. 2.2.** Карта Карно

По определению оператор case (или casez, как в данном примере) исполняет оператор первого подходящего варианта и никакой другой. Для значения 3'b101 это 3'b1?? (строка 7). Вариант в строке 10 исполняется, только когда значение входа равно 3'b001, несмотря на то что ему также соответствует значение 3'b101.

Еще одно предостережение, касающееся casez. Определение casez гласит, что любой бит, имеющий значение z, трактуется как несущественный при сравне-

---

[24] Напомним, что *картой Карно* называется способ представления таблицы истинности булевой функции, используемый для минимизации ее ДНФ или КНФ. Множество переменных разбивается на два подмножества; составляется матрица (карта Карно), строки которой соответствуют значениям переменных из первого подмножества, столбцы – значениям переменных из второго подмножества. Существенно, что соседние строки (столбцы) отличаются значением только одной переменной, при этом первая(ый) и последняя(ий) строки (столбцы) считаются соседними (обычно используется двоичный отраженный код Грея). Суть подхода состоит в покрытии клеток карты минимальным числом прямоугольных областей, целиком состоящих из 1 (минимизация ДНФ) или 0 (минимизация КНФ).

[25] Здесь под *простым импликантом* понимается прямоугольная область карты Карно, целиком состоящая из 0 или 1, которая не может быть увеличена.

нии с вариантами. Поэтому если в модели могут встретиться значения с высоким импедансом, например вследствие использования тристабильных драйверов (drivers), результаты симуляции casez могут отличаться от ожидаемых.

## 2.2.6. Моделирование с учетом уровней сигналов

До сих пор мы не уделяли особого внимания именованию переменных. В реальных моделях сигналам обычно сопоставляют *активный уровень* (*assertion level*), указывающий, какое значение, 0 или 1, говорит, что сигнал установлен (условие выполнено). Допустим, что мы проектируем датчик, определяющий, открыта дверь или закрыта. Можно при открытой двери выдавать логическую единицу, а можно наоборот – логический ноль. Как спроектировать схему, решает разработчик; оба варианта допустимы.

Предположим, что мы завели переменную doorOpen. Как обозначить, какое значение, 0 или 1, соответствует открытой двери? Обычно для этой цели в имя переменной добавляют специальный индикатор. Например, имя doorOpen_h говорит, что значение 1 соответствует открытой двери; высокий (h – high) уровень сигнала является активным (asserted high). Мы могли бы назвать сигнал doorOpen_l, если бы схема была сконструирована так, что при открытой двери выдавалось значение 0; активным является низкий (l – low) уровень сигнала (asserted low).

Стандартов именования переменных нет – обычно члены команды используют некоторое соглашение. Чтобы показать, что сигнал doorOpen имеет высокий активный уровень, его можно назвать doorOpen_h, doorOpen_H или просто doorOpen. По умолчанию, если суффикс не указан, предполагается высокий активный уровень. Для сигнала с низким активным уровнем подошло бы имя doorOpen_l, doorOpen_L, doorOpenL, doorOpenB или doorOpenN[26].

Для чего нужны соглашения об именовании? Чтобы было проще читать код. Симулятор и инструмент синтеза воспринимают имя переменной как единое целое, об этих соглашениях они ничего не знают. Рассмотрим следующие фрагменты кода, в которых используются скалярные переменные doorOpen (высокий активный уровень) и theCatIsOut_L (низкий активный уровень).

```
1   if (doorOpen)       f = 1; // если дверь открыта, установить f в 1
2   if (~theCatIsOut_L) g = 1; // если кошка вышла, установить g в 1
```

Отметим, что в обоих комментариях спрашивается, установлен сигнал или нет, а не какое значение он имеет, 1 или 0. Конечно, в строке 1 можно спросить «верно ли, что значение doorOpen равно 1?», а в строке 2 – «верно ли, что значение theCatIsOut_L равно 0?». Но вопрос о том, чему конкретно равно значение, часто вносит путаницу.

Если условие (выражение в скобках в операторе if), использующее сигнал с высоким активным уровнем, не содержит в начале операции дополнения (~),

[26] *N* – от англ. *not* (*не*) или *negated* (*с отрицанием*); *B* – от англ. *bar* (*черта*) или *barred* (*с чертой*) – распространенное обозначение отрицания.

значит, вы спрашиваете, установлен ли сигнал. Тот же вопрос задается, если в условии используется сигнал с низким активным уровнем и в начале стоит операция дополнения. Короче говоря, чтобы спросить, установлен ли сигнал, наличие тильды (~) должно соотноситься с активным уровнем сигнала: низкий – есть тильда, высокий – нет.

А как быть, если мы хотим узнать, что сигнал не установлен? В этом случае наличие тильды и активный уровень сигнала должны находиться в обратном соответствии, например:

```
1   if (~doorOpen)     p = 1; // если дверь не открыта, установить p в 1
2   if (theCatIsOut_L) q = 1; // если кошка не вышла, установить q в 1
```

В условиях можно комбинировать сигналы разных активных уровней, например:

```
1   if (doorOpen & ~theCatIsOut_L) t = 1;
```

Это означает, что если дверь открыта и кошка вышла, нужно установить t в 1. Или позвонить в пожарную часть. При сопоставлении активных уровней сигналов с тильдами все становится понятным! Следите за тильдами!

## 2.2.7. Оператор priority case

Еще одна возможность, связанная с оператором case, – добавить ключевое слово priority, как показано в примере 2.10. Напомним, что оператор case ищет первый подходящий вариант для значения выражения. Как только вариант найден, исполняется оператор справа от двоеточия, после чего исполнение продолжается с оператора, следующего за endcase. В примере 2.10 если составной вход {a, b} имеет значение 2'b11, то на выходе будет 0 (поскольку первый подходящий вариант находится в строке 7). Вариант в строке 9 тоже подходит, но он не будет исполнен, поскольку не является первым.

Отметим, что если бы мы переставили первый и третий варианты:

```
1   always_comb
2     priority casez ({a, b})
3       2'b?1: out = 1'b1; // изменено
4       2'b?0: out = 1'b0;
5       2'b1?: out = 1'b0; // изменено
6     endcase
```

то при том же значении {a, b}, 2'b11, мы получили бы на выходе 1.

В обоих случаях результат симуляции не зависит от наличия слова priority. Зачем же нужен оператор priority case? Слово priority сообщает инструменту синтеза, что имеется, *по крайней мере, один* подходящий вариант (возможно, несколько). Отсутствующие варианты считаются несущественными.

Рассмотрим арбитр на основе приоритетов, показанный в примере 2.11. Идея в следующем. Есть несколько устройств, которые независимо друг от друга посылают запросы на обслуживание, но в каждый момент времени

```
1 module priEncode
2   (input logic r0, r1, r2,
3    output logic [2:0] gnt);
4   logic [2:0] req;
5
6   assign req = {r2, r1, r0};
7
8   always_comb begin
9     gnt = 0;
10    priority casez (1'b1)
11      req[0]: gnt[0] = 1;
12      req[1]: gnt[1] = 1;
13      req[2]: gnt[2] = 1;
14    endcase
15  end
16 endmodule: priEncode
```

**Пример 2.11.** Арбитр на основе приоритетов

может быть обслужено только одно устройство. В данном примере имеются три линии r0–r2, по которым посылаются запросы. Выходом арбитра является 3-битная переменная, в которой установлен только один бит (разрешение на обслуживание соответствующего устройства). Предполагается, что всегда выставлен хотя бы один запрос (что отражено ключевым словом priority); соответственно, модель каждый раз устанавливает один и только один бит разрешения. Линия r0 имеет наивысший приоритет: даже если запрос на ней выставлен вместе с другими запросами, исполнен будет только первый вариант оператора case, и в переменной gnt будет установлен бит в позиции 0.

Логическая оптимизация, проистекающая из знания того, что *хотя бы на одной* линии будет выставлен запрос, состоит в том, что ситуация {r2, r1, r0} == 3'b000 невозможна, а значит, комбинация значений 3'b000 является несущественной для определяемой функции.

## 2.3. Основы разработки тестового окружения

Для тестирования реальных схем используются испытательные стенды. Они состоят из генераторов входных сигналов и осциллографов, отображающих выходные сигналы. Также используются логические анализаторы – своего рода осциллографы, предназначенные для анализа цифровых схем. Наши же «схемы» – это модели, описанные на языке SystemVerilog; вполне естественно, что и тестовое окружение разрабатывается на том же языке.

Основные задачи тестового окружения – моделирование окружения, в котором будет работать схема, и отладка модели схемы. Моделирование окружения состоит в описании компонентов, непосредственно соединенных с тестируемой моделью и, возможно, каких-то еще. Тестовое окружение не обязано описывать соседние компоненты точно; с задачей могут справиться и абстрактные модели. Большая часть средств отладки (например, печать промежуточных значений) находится в тестовом окружении, но некоторые могут быть включены в тестируемую модель.

Отметим также, что по описанию тестового окружения схема не синтезируется. Таким образом, нет смысла применять к нему правила написания синтезируемых моделей, сформулированные в разделе 2.1.2.

В этом разделе обсуждаются только вопросы, касающиеся разработки тестовых окружений для комбинационных схем.

## 2.3.1. Отладочная печать

Одна из функций тестового окружения – вывод сообщений об ошибках или просто текущих значений переменных. В табл. 2.1 описаны три основные процедуры печати.

**Таблица 2.1.** Операторы вывода на консоль

| Процедура | Когда осуществляется печать | Для чего используется |
|---|---|---|
| `$monitor (...)` | Процедура `$monitor` работает параллельно с тестируемой моделью и выводит сообщение всякий раз, когда изменяется значение одной из переменных, указанных в аргументах. Таким образом гарантируется, что все значения согласованы с моментом времени, указанным в сообщении | Процедура `$monitor` дает мгновенный снимок значений в конце цикла симуляции. Активным может быть только один монитор. Вызов `$monitor` останавливает предыдущий монитор и запускает новый |
| `$display (...)` | Процедура `$display` похожа на процедуры вывода в языках программирования. Она вызывается при исполнении части модели (например, блока `always_comb`), внутри которой находится вызов. Печатаются текущие значения переменных, указанных в качестве аргументов. Отметим, что значения одной переменной, напечатанные операторами `$display` и `$monitor`, могут отличаться. Кроме того, комбинационные блоки могут исполняться несколько раз в течение одного цикла симуляции (это связано с импульсными помехами на входах); в таком случае процедура `$display` выдает несколько сообщений, возможно, с разными значениями | Процедура `$display` применяется для вывода информации о состоянии модели во время исполнения. Она может сообщить о том, что достигнута определенная точка кода, и вывести значения переменных в этой точке |
| `$strobe (...)` | Процедура `$strobe` отличается от `$display` только временем, когда осуществляется вывод. Она исполняется в конце текущего цикла симуляции и выводит значения переменных в этот момент | Процедура `$strobe` используется так же, как `$display`, – для вывода информации о состоянии модели в процессе исполнения |

Все эти процедуры могут использоваться внутри блоков `initial` и `always`. Перечисленные в скобках аргументы передаются симулятору, который решает, какие значения выводить и когда.

Обычно процедуры вывода на консоль вызываются следующим образом:

```
1  $display (<print_string>, <list_of_variables>);
```

Здесь `<print_string>` – заключенная в кавычки строка, которая может содержать управляющие последовательности символов. Управляющая последовательность начинается с % и содержит букву: `d` означает, что значение нужно напечатать в десятичном виде, `h` – в шестнадцатеричном, `b` – в двоичном. Для каждой управляющей последовательности в строке `<print_string>` в списке `<list_of_variables>` должна присутствовать переменная (или выражение); ее значение печатается в соответствующем формате.

```
1 module if2
2   (input logic a, b, c,
3    output logic f);
4
5   always_comb begin
6     if (a & b) f = 1;
7   else if (c & a ^ b)
8         f = 1;
9     else f = 0;
10  end
11 endmodule: if2
12
13 module top;
14  logic [2:0] count;
15  logic result;
16
17  if2 dut (count[2], count[1], count[0], result);
18
19  initial begin
20    $monitor ($time, " abc=%b, f=%b",
21            count, result);
22    for (count = 0; count != 3'b111; count++)
23      #1;
24    #1 $finish;
25  end
26 endmodule: top
```

**Пример 2.12.** Простое тестовое окружение

В процедуре `$monitor` для вывода на консоль текущего времени используется конструкция `$time` (см., например, строку 20 в примере 2.12). `$time` – это системная функция, возвращающая текущее время. Процедуры `$display` и `$strobe` также могут выводить текущее время:

```
1    $display ("At time %d, value=%h",
$time, value);
```

В печатаемую строку вместо `%d` and `%h` подставляется соответственно время и значение `value`. Если время занимает не больше шести цифр, можно указать формат `%6d`, – вывод станет компактнее.

Кроме того, имеются возможности для вывода данных в файлы (для их последующего анализа).

## 2.3.2. Основы тестирования комбинационных схем

На рис. 2.12 показан прямолинейный подход к созданию тестового окружения для комбинационной схемы. Он хорош для совсем простых схем; мы приводим его только для того, чтобы познакомиться с основами разработки тестовых окружений.

В этом примере используются те же операторы `if`, что и в примере 2.5. Тестируемая модель (Design Under Test – DUT) – это модуль `if2`, а его тестовое окружение – модуль `top`. Модуль `if2` имеет три 1-битных входа и один 1-битный выход. Экземпляр модуля `if2` создается в строке 17 модуля `top`; его порты соединяются с переменными, определенными в модуле `top`. В данном случае три бита `count` соединены с входами `a`, `b` и `c`, а выход `f` – с переменной `result`.

Поведение тестового окружения описывается в блоке `initial` (строки 19–25). Сначала в строках 20–21 настраивается монитор, после чего в строках 22–23 идет цикл `for`, задающий действия тестового окружения. По завершении цикла вызывается системная процедура `$finish`, останавливающая симуляцию.

Вызов системной процедуры `$monitor` сообщает симулятору, за какими переменными следить и что выводить на консоль, когда значение хотя бы одной из них меняется. В данном случае монитор следит за переменными `count` и `result`. Как только значение любой из них меняется, печатается текущее время (`$time`)

и строка "abc=%b, f=%b", в которую вместо первой управляющей последовательности %b подставляется значение count, а вместо второй – result. Этот оператор работает параллельно с остальной частью модели; он играет роль пробника в логическом анализаторе. Его не нужно вызывать явно.

Затем исполняется цикл for в строках 22–23. Работает он так же, как циклы for в языках программирования: счетчик count обнуляется; далее, пока его значение не станет равным двоичному числу 3'b111, исполняется тело цикла. В данном случае цикл приостанавливается (строка 23) на одну единицу времени, а затем возобновляет работу – исполняет операцию count++, увеличивающую значение count на 1. Когда значение станет равным 3'b111, цикл завершается, блок initial приостанавливается еще на одну единицу времени (строка 24), после чего симуляция останавливается в результате вызова $finish.

Важно понимать, что происходит, когда блок initial приостанавливается на одну единицу времени в строках 23 и 24. В течение этого времени новое значение count распространяется до входов модуля if2. Блок always_comb видит, что, по крайней мере, один из входов изменился, поэтому исполняет свое тело, в результате чего может измениться значение выхода f.

Одновременно с исполнением этих моделей исполняется процедура $monitor, которая выводит на консоль значения переменных, указанных при ее вызове (count и result), каждый раз, как они изменяются. Результаты вывода показаны на рис. 2.3.

```
1        0 abc=000, f=0
2        1 abc=001, f=0
3        2 abc=010, f=1
4        3 abc=011, f=1
5        4 abc=100, f=0
6        5 abc=101, f=1
7        6 abc=110, f=1
8        7 abc=111, f=1
```

**Рис. 2.3.** Результаты симуляции модели из примера 2.12

Сделаем несколько замечаний о примере 2.12.

- Тестовое окружение подает все возможные комбинации значений на входы тестируемой модели. Таким образом, модуль if2 протестирован исчерпывающим образом. В том, что он работает правильно, можно убедиться, посмотрев на консоль.
- Поведение тестового окружения инкапсулировано в блоке initial – той части SystemVerilog-кода, которая не задействуется в логическом синтезе. В нем могут встречаться задержки (#1) и другие процедурные операторы, не допустимые в описании синтезируемых моделей.
- В описании тестируемой модели, модуля if2, нет отладочного кода, а есть лишь синтезируемые операторы, определяющие ее поведение.

```
1 module fourBitAdder
2   (input logic [3:0] a, b,
3    output logic [3:0] sum,
4    input logic cIn,
5    output logic cOut);
6   logic [2:0] c;
7
8   adder b0 (a[0], b[0], cIn, sum[0], c[0]);
9   adder b1 (a[1], b[1], c[0], sum[1], c[1]);
10  adder b2 (a[2], b[2], c[1], sum[2], c[2]);
11  adder b3 (a[3], b[3], c[2], sum[3], cOut);
```

```
12 endmodule: fourBitAdder
13
14 module adder
15   (input logic a, b, cI,
16    output logic s, cO);
17
18   assign s = a ^ b ^ cI,
19   cO = (a&b) | (a&cI) | (b&cI);
20 endmodule: adder
21
22 module test;
23   logic [3:0] s;
24   logic cOut;
25   logic [9:0] count;
26
27   fourBitAdder
28       add0 (count[7:4], count[3:0], s, count[8], cOut);
29
30   initial begin
31     for (count = 0; count <= 10'h200; count++)
32       #1 if ({cOut, s} != (count[7:4] + count[3:0] + count[8]))
33         $display ("oops! %d != %d + %d + %d",
34           {cOut, s}, count[7:4], count[3:0], count[8]);
35     $finish;
36   end
37 endmodule: test
```

**Пример 2.13.** Тестирование 4-битного сумматора

- Для тех, кто уже немного знаком с языком SystemVerilog, отметим, что в примере не используется конструкция `program`. Хотя для описания тестовых окружений применяются программы, мы еще не готовы к знакомству со всеми тонкостями этой конструкции. Пока нам достаточно описания тестового окружения в виде модуля.

## 2.3.3. Более сложная система

Рассмотренное выше тестовое окружение подавало на вход модели все возможные комбинации значений трех входов, а процедура `$monitor` вывела на консоль полученные результаты. Вместо этого тестовое окружение может вычислить ожидаемый результат и сравнить его с фактическим. Этот подход полезен, если существует простое описание поведения модели, без дублирования реализации.

Рассмотрим пример 2.13, где 4-битный сумматор реализован с помощью четырех экземпляров полного сумматора, который, в свою очередь, описан с помощью операторов `assign`. Сначала рассмотрим модуль полного сумматора (строки 14–20). Модуль получает на вход два бита, подлежащих сложению (`a` и `b`), а также бит переноса (`cI`). На выходе получается 1-битная сумма (`s`) и бит переноса для следующего каскада сложения (`cO`). Поведение сумматора реализовано с помощью операторов `assign`; сумма равна результату операции

xor над всеми тремя входами, а перенос имеет место, если значения хотя бы двух входов равны 1.

Нам нужен 4-битный сумматор: у модуля fourBitAdder два 4-битных входа (a и b); он складывает их друг с другом и с битом переноса (cIn), выдавая на выходе 4-битную сумму (sum) и один бит переноса (cOut). Модуль fourBitAdder создает четыре экземпляра модуля adder (строки 8–11) и соединяет отдельные биты a, b и sum, а также вход cIn и выход cOut. 3-битная переменная c играет роль проводов между каскадами сумматора (экземплярами модуля adder): каждый провод (бит переменной c) соединяет выходной бит переноса одного каскада с входным битом переноса следующего каскада. В строке 9, где создается экземпляр b1, мы видим, что порты модуля соединяются с битами a[1], b[1], c[0] (выходной бит переноса каскада b0), sum[1] и c[1] (входной бит переноса каскада b2). Выходной бит переноса сумматора старших битов (экземпляр b3) соединяется с выходом cOut модуля fourBitAdder.

Внутри модуля test используется 10-битная переменная count, предназначенная для подачи данных на входы модуля fourBitAdder. Значение count последовательно изменяется от 0 до 511, а когда становится равным 512 (шестнадцатеричное 200), цикл завершается. В строке 27 создается экземпляр модуля fourBitAdder: входы a и b задаются как count[7:4] и count[3:0] соответственно; в качестве cIn указывается count[8]. Конструкция count[7:4] *выбирает часть* count, т. е. диапазон соседних битов, а конкретно – четыре бита с седьмого по четвертый. Таким образом, все три входа, a, b и cIn, модуля fourBitAdder определяются одной переменной count.

Что проверяет тестовое окружение? Модуль test тестирует модуль fourBitAdder, подавая на вход все возможные комбинации значений двух 4-битных слагаемых и бита переноса – всего 512 комбинаций. Отличие этого тестового окружения от предыдущего состоит в том, что вместо процедуры $monitor, выводящей полученные результаты, здесь используется процедура $display, выводящая сообщения об ошибках (текст в двойных кавычках, строка 33). В строках 33–34 печатается значение результата (конкатенации выходного бита переноса и суммы), а также значения входов: a, b и cIn.

Условие оператора if в строке 32 нуждается в пояснении. В нем производится сравнение результата работы сумматора (левая часть операции !=) с ответом, которого ожидает тестовое окружение (правая часть). Слева находится 5-битный результат, образованный конкатенацией выходного бита переноса и 4-битной суммы, а справа – 5-битный результат, вычисленный тестовым окружением. Если эти значения не совпадают, то процедура $display выводит на консоль сообщение об ошибке и цикл продолжается. Когда count принимает значение 512, симуляция останавливается.

Отметим несколько важных моментов.

- В примере показывается, что для тестирования модели можно использовать альтернативный способ описания функциональности. В данном случае сумма count[7:4] + count[3:0] + count[8] сравнивается с результа-

том работы более детальной модели, при создании которой могли быть допущены ошибки. Отметим, что не у всякой модели есть простая альтернативная реализация.

- Перебор значений одной переменной (счетчика) и выбор ее частей позволяют простым образом сгенерировать все возможные комбинации значений входов комбинационной схемы.
- Задержка #1 в строке 32 необходима для того, чтобы значения, полученные в результате выбора частей `count`, распространились от входов модуля `fourBitAdder` до его выходов. Каждый раз, когда блок `initial` достигает #1, его исполнение приостанавливается (в это время происходит распространение сигналов). К концу цикла симуляции устанавливаются окончательные значения переменных, и значения `s` и `cOut` можно сравнивать с ожидаемыми. В следующий момент блок `initial` возобновляет исполнение и проверяет условие оператора `if` (строка 32), используя значения, только что полученные симулятором.

## 2.4. Параметризованные модули

На SystemVerilog можно писать запутанные модели, подверженные ошибкам со стороны разработчиков. Задача SystemVerilog – предоставить средства для уменьшения числа ошибок. Одно из таких средств мы уже рассмотрели – создание экземпляров модулей (что позволяет повторно использовать один раз написанные модули для построения сложных систем). Другое средство – параметризованные модули.

Определение модуля может содержать *параметры* – константы, используемые при компиляции описания для симуляции или синтеза. При создании экземпляра модуля параметры можно переопределять, что дает дополнительные возможности повторного использования уже протестированных модулей.

```
1 module aluParam
2   #(parameter W = 8)
3   (input logic [W-1:0] a, b,
4    output logic [W-1:0] result,
5    input logic [2:0] op);
6
7   always_comb
8     unique case (op)
9       3'b100: result = a + b;
10      3'b010: result = a - b;
11      3'b001: result = a & b;
12      3'b110: result = a | b;
13      3'b011: result = a ^ b;
14    endcase
15 endmodule: aluParam
```

**Пример 2.14.** Параметризация АЛУ

Рассмотрим модель из примера 2.8. Это 8-битное АЛУ, реализующее пять операций. Если бы мы захотели разработать АЛУ с теми же самыми операциями, но большей шириной операндов, то могли бы отредактировать код, изменив разрядность входов и выхода. Во избежание ошибок лучше сделать ширину портов параметром, как показано в примере 2.14. По сравнению с предыдущим примером изменились только строки 2–4. В строке 2 определяется параметр `W`, задающий разрядность портов данных (по умолчанию `W` равен 8). В строках 3 и 4 указывается номер самого левого бита – не в виде числа, а как `W-1` (в данном случае это 7). Таким образом, `a` и `b` определены

как 8-битные векторы с нумерацией битов [7:0]. Модуль оказывается таким же, как в примере 2.8.

Как это помогает использовать модуль повторно? Дело в том, что при создании экземпляра модуля значение параметра можно переопределить. Рассмотрим следующие два примера, которые могут встретиться в описании на SystemVerilog:

```
1   aluParam #(12) a1 (s, t, u, oper);
2   aluParam a2 (e, f, g, op);
```

В обеих строках создается экземпляр одного и того же модуля aluParam из примера 2.14. Экземпляр a1, созданный в строке 1, является 12-битным АЛУ: конструкция #(12) означает, что параметр переопределяется – ему присваивается значение 12. В экземпляре, созданном в строке 2, параметр не переопределяется – используется значение по умолчанию; таким образом, это 8-битное АЛУ.

Предположим, что мы проектируем тракт данных (datapath) системы. В этом тракте могут присутствовать разные компоненты, например регистры. Можно параметризовать разрядность тракта данных и использовать введенный параметр для задания разрядности АЛУ, как показано в примере 2.15. В строке 2 определен параметр W; каждая переменная, соединенная с портами АЛУ, имеет именно такую разрядность. Когда в строке 7 создается экземпляр модуля aluParam из примера 2.14, параметр W используется для переопределения параметра создаваемого экземпляра. Точно так же при создании экземпляра модуля datapath мы можем задать его разрядность – это значение будет распространено на экземпляры, создаваемые внутри datapath. Многоточия в строках 3 и 6 означают, что соответствующие части описания были для краткости опущены.

```
1 module datapath
2    #(parameter W = 8)
3   (... входы и выходы);
4   logic [W-1:0] a, b, result;
5   logic [2:0] op;
6     ...
7   aluParam #(W) a3 (a, b, result, op);
8 endmodule: datapath
```

**Пример 2.15.** Параметризованный модуль datapath

Число параметров в определении модуля может быть произвольным, например:

```
1   module paramDemo
2     #(parameter A = 1,
3                 B = 75,
4       parameter [3:0] C = 3'b010);
...
```

В данном случае модуль имеет три параметра, каждый из которых имеет значение по умолчанию. При создании экземпляра модуля параметры можно переопределить:

```
1   paramDemo #(7, 13, 3'b110) p1 (... порты);
```

В экземпляре p1 модуля paramDemo параметр A имеет значение 7, B – 13, а C – 3'b110. Отметим, что, как и порты модуля, параметры должны указываться в том порядке, в котором они определены в модуле. Заглавные буквы по соглашению применяются для обозначения констант.

Локальные параметры (константы) модуля можно определять с помощью ключевого слова localparam. Такие параметры перечисляются после объявления портов, например:

```
1   module Lparam
2     #(...    ) // параметры
3     (...   ); // порты
4     localparam Q = 17;
5   ...
```

Здесь определена константа Q. Ее можно использовать внутри модуля, но нельзя переопределить при создании экземпляра (если только ее значение не зависит от одного из параметров, заданных в строке 2), например:

```
1   module anotherParam
2     #(parameter R = 3) // параметры
3     (...         );     // порты
4     localparam Q = R + 17;
5   ...
```

В этом случае значение Q зависит от значения параметра R. Значение может быть изменено путем переопределения R при создании экземпляра модуля.

## 2.5. Спецификация портов

Порты обеспечивают соединения между парами элементов: один элемент пары находится внутри экземпляра модуля; другой принадлежит модулю, создающему экземпляр. Синтаксис SystemVerilog допускает два способа соединения портов: *упорядоченные соединения портов* и *именованные соединения портов*. При создании экземпляра модуля следует выбрать какой-то один способ (смешивать их нельзя).

До сих пор нам встречались только упорядоченные соединения портов. В этом способе при создании экземпляра указывается список имен, известных создающему модулю. Элементы перечисляются в том порядке, в каком соответствующие им элементы определены в создаваемом модуле. Это еще раз показано в примере 2.16. Здесь мы видим четыре модуля. В строках 1–6 определен простой мультиплексор, аналогичный приведенному во введении; у него есть четыре порта: f, m0, m1 и sel (именно в таком порядке). В модуле orderedPort (строки 8–12) создается экземпляр модуля mux; как мы видим, порядок соединений (строка 11) соответствует порядку определения портов в модуле (строки 2–3). Таким образом, между собой соединены result и f, in0 и m0, in1 и m1, sel и sel. Ничего нового, по сравнению с предыдущими примерами, здесь нет.

В именованных соединениях портов указываются пары имен: первое имя задает элемент создаваемого экземпляра; второе – элемент модуля, в котором экземпляр создается. Этот способ также показан в примере 2.16. В определении модуля byNamePort (строки 14–18) при создании экземпляра модуля mux используются именованные соединения портов (строка 17). Первый элемент списка соединений имеет вид .m0(i0) – переменная i0 в создающем модуле (byNamePort) соединяется с переменной m0 в создаваемом экземпляре модуля mux. Вообще, в конструкции .x(y) переменная y определена в создающем модуле, а переменная x – в модуле, экземпляр которого создается. Отметим, что поскольку в соединении указываются имена обоих портов, перечислять соединения можно в любом порядке. Легко видеть, что порядок портов в строке 17 не совпадает с порядком их определения в модуле mux.

Если имена всех портов в создающем и создаваемом модулях совпадают, то для задания соединений можно использовать нотацию .*. Это показано в строке 23. Конструкция интерпретируется следующим образом: просматриваются порты в определении создаваемого модуля (в данном случае mux); порты соединяются с переменными текущего модуля (allNamesMatch), имеющими такие же имена. Соединения устанавливаются автоматически – выписывать их явно не нужно. Меньше возможностей опечататься!

Иногда совпадают не все имена, а только некоторые. В этом случае соединения задаются, как показано в модуле someNamesMatch (строки 26–30). Здесь совпадают все имена, кроме sel в модуле mux и select в модуле someNamesMatch. В строке 29 показано, что это соединение устанавливается явно; следующая конструкция .* говорит, как соединять остальные порты. Наконец, если все имена совпадают, то можно использовать подход, показанный в модуле namesMatchDocumenting (строки 32–36), где имена портов перечислены в произвольном порядке, – это документирует модуль лучше, чем .*.

```
1  module mux
2    (output f,
3     input m0, m1, sel);
4
5    assign f = (sel) ? m1 : m0;
6  endmodule: mux
7
8  module orderedPort;
9    logic sel, in1, in0, result,
10   ...
11   mux a (result, in0, in1, sel);
12 endmodule: orderedPort
13
14 module byNamePort;
15   logic s, i1, i0, out;
16   ...
17   mux b (.m0(i0), .sel(s), .m1(i1), .f(out));
18 endmodule: byNamePort
19
20 module allNamesMatch21 logic m0, m1, sel, f;
22   ...
23   mux c (.*);
24 endmodule: allNamesMatch
25
26 module someNamesMatch
27   logic m0, m1, select, f;
28   ...
29   mux d (.sel(select), .*);
30 endmodule: someNamesMatch
31
32 module namesMatchDocumenting
33   logic m0, m1, select, f;
34   ...
35   mux e (.sel, .f, .m0, .m1);
36 endmodule: namesMatchDocumenting
```

**Пример 2.16.** Примеры соединений портов

Все модули в примере 2.16 – orderedPort, byNamePort, allNamesMatch, someNamesMatch и namesMatchDocumenting – соединены с модулем mux корректно. Какой вариант выбрать, зависит от соглашения об именовании, принятого вами или командой, в которой вы работаете. В больших проектах многие переменные на разных уровнях создания экземпляров часто имеют одинаковые имена. Действительно, какой смысл изобретать разные имена для одной и той же сущности? В таком случае можно пользоваться именованными соединениями портов и подстановочным символом .*. В больших моделях, где у модулей много портов, конструкция .* может уменьшить число опечаток.

## 2.6. Основные типы данных

До сих пор все переменные в примерах имели тип logic. В языке SystemVerilog существуют и другие типы данных, некоторые из них мы опишем в этом разделе. Наибольший интерес для нас сейчас представляют целочисленные типы, их варианты с 2- и 4-значными «битами»[27].

### 2.6.1. Двух- и четырехзначные «биты»

Тип logic благодаря своей 4-значной природе широко применяется при проектировании на уровне регистровых передач. Значениями битов являются: 0 и 1, представляющие FALSE и TRUE, x, представляющее неизвестное значение, и z, представляющее высокий импеданс.

Неизвестное значение x полезно при отладке модели. При запуске симуляции биты переменных 4-значных типов (за исключением типа time, см. табл. 2.2) получают значение x. Значение остается таковым, пока не произойдет одно из двух: первое – биту присваивается значение 0, 1 или z (например, в блоке initial); второе – бит, являющийся выходом блока always_comb, получает значение при первом исполнении блока.

Значение высокого импеданса z моделирует ситуацию, когда на соединении не устанавливается никакое значение. Это случается, когда несколько блоков always или операторов assign формируют (drive) значение на одном и том же соединении (проводе); так описываются шины. Если ни один из драйверов (формирователей) шины не устанавливает значение на ней, образуется значение z (высокий импеданс). Входы модуля, не соединенные ни с чем, также имеют значение z. Логически z на входе модуля обрабатывается как x. Использование значений высокого импеданса обсуждается в разделе 2.7.

С электрической точки зрения значения 0, 1 и z представляют три различные ситуации: напряжение уровня логического нуля, напряжение уровня логической единицы и отсутствие напряжения. Измеряя напряжение в физиче-

[27] Строго говоря, биты (от англ. *bits* – *bi*nary digi*ts*) могут быть только двузначными. Здесь слово «бит» используется как синоним слову «разряд».

ской схеме, можно понять, какая из ситуаций имеет место. Значение x не имеет электрического аналога; оно говорит, что симулятор просто не знает значения.

4-значные типы полезны, но они не так эффективны, как 2-значные типы, – симулятору нужны дополнительные биты для представления значений x и z. При проектировании крупных систем на более абстрактном уровне разработчики часто предпочитают 2-значные типы (см., например, тип bit в табл. 2.2).

## 2.6.2. Целочисленные типы данных

В табл. 2.2 перечислены целочисленные типы данных языка SystemVerilog.

**Таблица 2.2.** Целочисленные типы данных

| Тип данных | 2 или 4 значения бита | Размер в битах | Знаковый или беззнаковый по умолчанию | Значение в момент начала симуляции | Прочие свойства |
|---|---|---|---|---|---|
| shortint | 2 | 16 | Знаковый | 0 | То же, что и short в C[1] |
| int | 2 | 32 | Знаковый | 0 | То же, что и int в C |
| longint | 2 | 64 | Знаковый | 0 | То же, что и long в C |
| bit | 2 | Определяется пользователем | Беззнаковый | 0 | |
| byte | 2 | 8 | Знаковый | 0 | То же, что и signed bit[7:0] |
| logic | 4 | Определяется пользователем | Беззнаковый | X | |
| reg | 4 | Определяется пользователем | Беззнаковый | X | Тип Verilog, замещенный в SystemVerilog типом logic |
| integer | 4 | 32 | Знаковый | X | Отличается от int в C (четырехзначные биты) |
| time | 4 | 64 | Только беззнаковый | 0 | Не используется в моделях, но может использоваться в тестовых окружениях |

**Примечание.** В языке C разрядность целочисленных типов данных не определена жестко.

Основными типами данных, используемыми для моделирования систем на языке SystemVerilog, являются bit и logic. Разница между ними только в том, что тип bit подразумевает 2-значную логику, а тип logic – 4-значную (что менее эффективно при симуляции). Тип byte – это, по сути, знаковый битовый вектор размера 8.

Первые три типа данных (shortint, int, longint) имеют аналоги в языке C (они указаны в последнем столбце). Эти типы были добавлены в язык, чтобы иметь возможность вызывать из SystemVerilog функции, написанные на C, и наоборот.

Тип time часто используется в тестовых окружениях. Это беззнаковый тип (вам не удастся изобрести машину времени и отправиться в прошлое!). Значение типа time можно вывести с помощью процедур $display и $monitor. Такое значение может представлять промежуток времени, занимаемый тем или иным вычислением.

Все типы данных, кроме time, могут быть как знаковыми, так беззнаковыми; для всех типов определены значения по умолчанию. Симулятор и инструмент синтеза представляют отрицательные значения знаковых типов в дополнительном коде. Для модификации типа нужно написать перед его именем ключевое слово signed или unsigned, например:

```
signed bit [9:0] a; // 10-битная знаковая переменная с 2-значными битами
unsigned byte b;    // 8-битная беззнаковая переменная с 2-значными битами
```

По определению дополнительного кода, старший бит $n$-битного знакового вектора имеет значение $-2^{(n-1)}$ и задает знак числа (1 – отрицательное число, 0 – положительное число). Старший бит $n$-битного беззнакового вектора имеет значение $2^{(n-1)}$. Как и в языках программирования, признак signed/unsigned влияет на результат некоторых операций сравнения.

Приведем полезную информацию о целочисленных типах данных.

- Значения между типами logic и bit преобразуются автоматически, без выдачи предупреждений. Рассмотрим следующую ситуацию:

```
logic [3:0] a;
bit [3:0] b;
```

- Если a имеет значение 4'b10xz и в блоке initial (или always) исполняется оператор b = a, то b будет присвоено значение 4'b1000; b – переменная типа bit с 2-значными битами, которые не могут принимать значения x и z. Правило преобразования 4-значных бит в 2-значные гласит, что 1 всегда остается 1, а все остальное преобразуется в 0.
- Пусть один вектор присваивается другому вектору, имеющему больший размер. Результат зависит от того, являются векторы знаковыми или нет. Если векторы беззнаковые, значение дополняется слева нулевыми битами; если же векторы знаковые, в дополнительные биты копируется знаковый (старший) бит. См. пример ниже.

```
logic [3:0] smallLogic;
logic [5:0] bigLogic;
...
  smallLogic = 4'b1000;
  bigLogic = smallLogic; // получится 6'b001000

signed logic [3:0] smallSignedLogic;
signed logic [5:0] bigSignedLogic;
...
  smallSignedLogic = 4'b1000;
  bigSignedLogic = smallSignedLogic; // получится 6'b111000, потому что знак
                     распространяется (копируется) в два дополнительных бита
  smallSignedLogic = 4'b0100;
  bigSignedLogic = smallSignedLogic; // получится 6'b000100, потому что знак
                     распространяется (копируется) в два дополнительных бита
```

### 2.6.3. Перечисления и определения типов

Перечисления позволяют вводить в SystemVerilog-описания именованные константы. По синтаксису и семантике они очень похожи на перечисления, встречающиеся в языках программирования. Использование имен вместо значений уменьшает вероятность внесения ошибок в описания.

Вот пример простого перечисления в SystemVerilog:

```
1   enum {CHEVY, FORD, TOYOTA} car1, car2;
```

Здесь определены две переменные, car1 и car2, которые могут принимать три значения: CHEVY, FORD и TOYOTA. Переменные строго типизированы в том смысле, что могут принимать только эти значения и сравниваться только с переменными этого же типа. Ниже приведены примеры использования этих переменных, которые можно встретить в блоках initial и always:

```
1   car1 = FORD;
2   car2 = TOYOTA;
3   if (car1 == car2) q = 3;
4   car1 = car2;
5   car1 = 5;      // ошибка: несоответствие типов
6   if (car1 != 5) ... ; // ошибка: несоответствие типов
```

По умолчанию элементы перечисления enum имеют тип int и принимают последовательные значения начиная с 0. Можно указать любой целочисленный тип из упомянутых в разделе 2.6.2. Кроме того, можно явно задать значения всех или некоторых констант при соблюдении изложенных ниже ограничений.

Рассмотрим модуль datapath с блоком always_comb, моделирующим АЛУ, как показано в строках 4–11 примера 2.17 (этот блок always_comb скопирован из примера 2.8). Внутри модуля определена переменная op 3-битного перечислимого типа, задающего имена реализованных в АЛУ операций. В операторе case используются не 3-битные значения, а имена, указанные в перечислении. Где-то в другом месте переменной op могло быть присвоено значение – одна из пяти именованных констант (в силу ограничений на тип). Как обычно, как только изменяется значение op, исполняется блок always_comb (строка 4), и результат перевычисляется.

```
1 module datapath_enum;
2   enum logic [2:0] {ADD, SUB, AND, OR, XOR} op;
3   ...
4   always_comb
5     unique case (op)
6       ADD: result = a + b;
7       SUB: result = a - b;
8       AND: result = a & b;
9       OR:  result = a | b;
10      XOR: result = a ^ b;
11    endcase
12 endmodule: datapath_enum
```

**Пример 2.17.** Использование перечислимого типа в описании АЛУ

Это описание годится для ранних стадий проекта, когда фактические значения констант еще не определены. Допустим, что на каком-то этапе коды операций были согласованы и стали такими, какими были в примере 2.8. Тогда объявление примет вид:

```
enum logic [2:0]
  {ADD= 3'b100,
  SUB= 3'b010,
  AND= 3'b001,
  OR= 3'b110,
  XOR= 3'b011} op;
```

Преимущество такого использования enum заключается в том, что значения задаются в одном месте. Там, где эти значения используются, например в операторах if-else и case, не нужно их дублировать (что чревато ошибками), достаточно просто написать соответствующие имена.

Значения переменных перечислимых типов можно выводить в виде имен. Например, следующая процедура $display выведет на консоль "AND", если op имеет значение 3'b001:

```
$display("The value of op is %s." op.name);
```

Отметим, что в этом операторе используется управляющая последовательность %s: печатается строка op.name – имя, соответствующее значению переменной op.

В примере показано, как назначать значения константам. Если значения задаются явно только для некоторых констант, остальные получают последующие значения. В этом случае нужно следить за тем, чтобы не появились дубликаты. Рассмотрим, к примеру, такое определение:

```
enum bit[3:0]
  {CHEVY=3,
   FORD,
   TOYOTA=6} car1, car2;
```

Здесь все в порядке: константе FORD автоматически назначается значение 4 (это значение следует за значением 3, явно заданным для константы CHEVY). Однако если бы у константы TOYOTA было указано значение 4, при компиляции была бы выдана ошибка: FORD и TOYOTA имеют одинаковые значения.

В перечислении могут использоваться значения с x и z, при условии что указанный тип это допускает:

```
enum logic[2:0] {s1 = 3'b000, s2=3'b100, s3=3'bxx1} s_value; // OK:
                        переменные типа logic имеют 4-значные биты
enum bit[2:0] {s1 = 3'b000, s2=3'b100, s3=3'bxx1} s_value; // Ошибка:
                        переменные типа bit имеют 2-значные биты
```

Пример 2.17 показывает, как использовать enum внутри модуля. Часто enum встречается в определении типов. *Определение типа* (ключевое слово typedef)

вводит новый тип данных, который можно использовать при определении переменных и портов.

Рассмотрим то же самое АЛУ, используемое как часть процессора (пример 2.18). В строках 1–5 определен новый тип aluInst_t, который теперь можно использовать при объявлении переменных, принимающих перечисленные значения (обычно имена типов имеют суффикс _t). Поскольку aluInst_t – тип данных, его можно использовать в спецификациях выходных и входных портов (как это делается в строках 8 и 15), а также в объявлениях переменных (как это делается в строке 30).

Итак, в примере 2.18 показан модуль instructionDecoder (строки 7–10) с выходом decode типа aluInst_t. В модуле top этот выход соединен с переменной iDecode (строка 29), также имеющей тип aluInst_t. Переменная iDecode соединена с четвертым входом модуля alu (входом op), также имеющим тип aluInst_t. В вариантах case, как и раньше, используются элементы перечисления, описывающие, какую операцию следует исполнить для вычисления результата.

```
1  typedef enum logic [2:0] {ADD=3'b100,
2    SUB= 3'b010,
3    AND= 3'b001,
4    OR=  3'b110,
5    XOR= 3'b011} aluInst_t;
6
7  module instructionDecoder
8    (output aluInst_t decode,
9     ...);
10 endmodule: instructionDecoder
11
12 module alu
13   (input logic [7:0] a, b,
14    output logic [7:0] result,
15    input aluInst_t op);
16
17   always_comb
18     unique case (op)
19       ADD: result = a + b;
20       SUB: result = a - b;
21       AND: result = a & b;
22       OR:  result = a | b;
23       XOR: result = a ^ b;
24     endcase
25 endmodule: alu
26
27 module top;
28   logic [7:0] inA, inB, aluOut;
29   aluInst_t iDecode;
30   ...
31   instructionDecode id (iDecode, …);
32   alu a1 (inA, inB, aluOut, iDecode);
33 endmodule: top
```

**Пример 2.18.** Использование typedef и enum

Возникает вопрос, к чему вся эта суета с определением перечислений и типов, ведь и прежние версии модуля alu прекрасно работали. Причина в том, что такой подход позволяет строже контролировать определение и использование констант в большом описании. Разработчик вкладывает в модель свои знания, которые компилятор впоследствии проверяет. В данном случае знания заключаются в том, что АЛУ реализует ровно пять команд, определенных выше. Любой контроль такого рода позволяет сократить число ошибок кодирования, что уменьшает время проектирования и отладки системы. Кроме того, поскольку константы определяются в одном месте, их проще изменить, если мы надумаем модифицировать АЛУ или расширить его функциональность.

Несмотря на все языковые абстракции, окончательная реализация предстает в виде проводов, логических вентилей и регистров. Однако модель гораздо проще описывать и поддерживать.

## 2.6.4. Тестирование комбинационных схем с перечислениями

Для тестирования комбинационной схемы нужно подать все возможные комбинации значений входов и проверить правильность значений выходов.

```
1  typedef enum bit [2:0] {
2    ADD= 3'b100,
3    SUB= 3'b010,
4    AND= 3'b001,
5    OR=  3'b110,
6    XOR= 3'b011} aluFun_t;
7
8  module combTestWithEnums;
9    aluFun_t op;
10   bit [7:0] a, b, result;
11
12   aluWithEnums dut (.*);
13
14   initial begin
15     $monitor ("%h = %h %s %h",
16               result, a, op.name, b);
17     for (op = op.first; 1; op = op.next) begin
18       a = 8'h35; b = 8'h15;
19       #1 if (op == op.last) break;
20     end
21   end
22 endmodule: combTestWithEnums
23
24 module aluWithEnums
25   (input bit [7:0] a, b,
26    output bit [7:0] result,
27    input aluFun_t op);
28
29   always_comb
30     unique case (op)
31       ADD: result = a + b;
32       SUB: result = a - b;
33       AND: result = a & b;
34       OR:  result = a | b;
35       XOR: result = a ^ b;
36     endcase
37 endmodule: aluWithEnums
```

**Пример 2.19.** Элементы перечисления в тестовом окружении

В схеме возможны несущественные комбинации, которые описывают с помощью оператора `unique case`. Как уже отмечалось, ключевое слово `unique` означает, что всегда при исполнении оператора `case` подходящим будет один и только один вариант. Инструмент синтеза может оптимизировать логику, предполагая, что неуказанные варианты никогда не встретятся.

В такой ситуации не имеет смысла проверять все результаты комбинационной схемы. Если мы тестируем модель частично определенной логической схемы, мы не знаем, какими должны быть результаты для несущественных комбинаций. Как протестировать схему только на значениях, указанных в перечислении?

Рассмотрим пример 2.19, где в вариантах оператора `unique case` используются элементы перечисления. В строках 1–6 определен тип `aluFun_t`, перечисляющий имена операций АЛУ (ADD, SUB, AND, OR, XOR). Далее в операторе `unique case`, реализующем функции АЛУ (строки 29–36), эти имена используются в вариантах `case`. Как сделать тестовое окружение, подающее на вход АЛУ указанные операции и опускающее несущественные комбинации?

У переменных перечислимых типов есть несколько полезных методов.

- `name` – возвращает строку, которую можно вывести на консоль процедурой `$monitor` или `$display`. В строках 15–16 показано, как напечатать имя элемента перечисления, хранящегося в переменной `op`.
- `first` – возвращает значение первого элемента перечисления (в данном случае значение 3'b100, сопоставленное элементу ADD).

- next – возвращает значение следующего элемента перечисления. Если текущий элемент является последним, возвращается значение первого элемента. Методу можно передать целое неотрицательное число n (по умолчанию n = 1) – будет возвращено значение элемента, отстоящего от текущего на n.
- last – возвращает значение последнего элемента перечисления (в данном случае значение 3'b011, сопоставленное элементу XOR).

Понятно, что делает цикл for в тестовом окружении combTestWithEnums (строки 8–22 после настройки монитора). В качестве операции (op) берется первый элемент перечисления; операнды a и b получают значения (в данном случае это константы); за время #1 эти значения распространяются по модели. По истечении задержки проверяется, является ли текущее значение op последним. Если да, мы выходим из цикла с помощью оператора break. Если нет, цикл продолжается – переменной op присваивается значение следующего элемента перечисления. При таком подходе на блок always_comb в АЛУ подаются только допустимые значения входов. Отметим, что во вложенном цикле for можно было бы перебирать значения входов a и b. Ниже приведен вывод на консоль процедуры $monitor:

```
1   4a = 35 ADD 15
2   20 = 35 SUB 15
3   15 = 35 AND 15
4   35 = 35 OR 15
5   20 = 35 XOR 15
```

Обратите внимание, что для печати op.name используется управляющая последовательность %s, предназначенная для строк.

У перечислений есть и другие методы.

- prev – возвращает значение предыдущего элемента перечисления. Если текущий элемент является первым, возвращается значение последнего элемента. Как и в случае next, методу можно передать целое неотрицательное число n (по умолчанию n = 1) – будет возвращено значение элемента, отстоящего от текущего на n.
- num – возвращает число элементов перечисления (в нашем примере 5).

## 2.6.5. Структуры

Сделаем еще одну вещь в примере 2.19 – модифицируем модуль alu, как показано в примере 2.20. Новый модуль принимает команду, включающую код операции и два операнда (см. рис. 2.4). Модуль из примера 2.19 принимает то же самое, но по отдельности, в виде входов a, b и op. Считаем, что дешифратор команд выдает команду в нужном нам формате.

Первое, что приходит на ум, – определить переменную command, включающую 3-битное поле кода операции и два 8-битных поля операндов. Всего получается 19 бит, так что определение выглядит так:

```
1   logic [18:0] command;
```

Чтобы выделить из команды код операции и операнды, воспользуемся операцией выбора части:

```
2   command[18:16]      // код операции
3   command[15:8]       // операнд inA
4   command[7:0]        // операнд inB
```

```
1  typedef enum logic [2:0] {ADD=3'b100,
2    SUB= 3'b010,
3    AND= 3'b001,
4    OR=  3'b110,
5    XOR= 3'b011} aluInst_t;
6
7  typedef struct packed {
8    aluInst_t oper;
9    logic [7:0] inA, inB;
10 } command_t;
11
12 module instructionDecoder
13   (output command_t inst,
14    ...);
15 endmodule: instructionDecoder
16
17 module alu
18   (input command_t c,
19    output logic [7:0] result);
20
21   always_comb
22     unique case (c.oper)
23       ADD: result = c.inA + c.inB;
24       SUB: result = c.inA - c.inB;
25       AND: result = c.inA & c.inB;
26       OR:  result = c.inA | c.inB;
27       XOR: result = c.inA ^ c.inB;
28     endcase
29 endmodule: alu
30
31 module top;
32   command_t iCommand;
33   logic [7:0] aluOut;
34   ...
35   instructionDecode id (iCommand, ...);
36   alu a1 (iCommand, aluOut);
37 endmodule: top
```

**Пример 2.20.** Использование typedef and struct

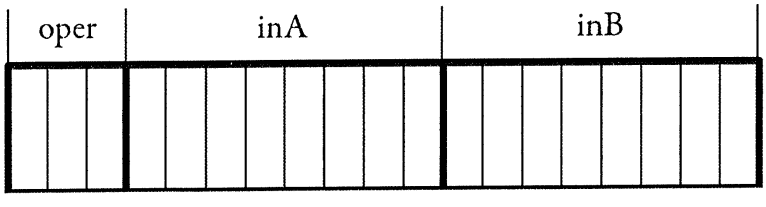



**Рис. 2.4.** Тип command_t – упакованная структура

Например, для сложения inA и inB мы написали бы:

```
5   result = command[15:8] + command[7:0];
```

Этот подход работает, но лучше использовать структурный тип данных (*struct*), позволяющий собрать команду из отдельных частей; структуры знакомы нам по языкам программирования. Тип struct группирует переменные, представляющие разные аспекты одной сущности. В данном случае сущностью является команда АЛУ, а ее аспектами – код операции и два элемента данных.

В примере 2.20 показан структурный тип, описывающий команду (строки 7–10); он проиллюстрирован на рис. 2.4. Структура состоит из поля oper типа aluInst_t и двух 8-битных операндов inA и inB. В строке 7 эта структура определена как новый тип command_t. Структура *упакована* (packed), т. е. представляет собой непрерывное слово (битовый вектор): ее поля вплотную примыкают друг к другу и расположены в порядке, заданном в определении типа. На рис. 2.4 показано, как переменная типа command_t размещается в памяти.

В примере 2.20 есть и другие изменения, обусловленные определением нового типа в строке 7. У модуля alu теперь два порта: первый (строка 18) – команда, описываемая в виде структуры (мы назвали этот порт c); второй – выход result. Чтобы сослаться на любой элемент структуры, используется нотация

с точкой: c.oper, c.inA и c.inB обозначают три поля команды. В соответствии с этим в строке 23 сложение записывается в виде:

```
1   result = c.inA + c.inB;
```

Изменения внесены также в модуль instructionDecoder, выход которого имеет тип command_t, и в модуль top, в котором экземпляр id модуля instructionDecoder соединяется с экземпляром a1 модуля alu с помощью соединения iCommand типа command_t.

Есть несколько причин для использования такого стиля при спецификации сложных систем. Во-первых, код становится более понятным для автора и других членов команды. Теперь абстракцией является команда АЛУ, а не ее отдельные элементы. Таким образом, структуры делают описание более абстрактным, сохраняя при этом его явность и точность. Во-вторых, при таком подходе уменьшается число ошибок кодирования – разрядность элементов задается в одном месте (в определении типа). Для обращения к полям структуры используется точка – при разумном выборе имен это понятнее и менее чревато ошибками.

## 2.7. Множественные драйверы

Модели комбинационных схем, рассмотренные в предыдущих частях этой главы, имеют общую черту: их функция описывается одним блоком always_comb или одним присваиванием assign. Говорят, что always_comb и assign являются *драйверами* (*drivers*), или *формирователями*, соответствующих выходов. С точки зрения физической реализации это означает, что выходной вентиль подсхемы, описанной в always_comb или assign, является единственным источником сигнала для провода (wire), передающего выходное значение. В большинстве случаев у каждой функции есть только один драйвер. Одна из особенностей блока always_comb в том и состоит, что его выход не может формироваться несколькими источниками (включая другие always_comb). Это проверяется компилятором и поощряет хорошее кодирование.

Однако бывают ситуации, когда функции разделяются и выходные соединения формируются несколькими блоками. Это характерно для шин и вообще соединений. Шина может представлять собой длинный провод, драйверы которого распределены физически; например, процессор и устройство ввода-вывода обмениваются данными с помощью шины на печатной плате, как это показано на рис. 2.5. Драйверы такой шины реализуются в виде отдельных микросхем, быть может, изготавливаемых разными производителями. Имеет смысл описывать логику драйверов отдельно, а затем соединять воедино. При проектировании таких систем нужно следить за тем, чтобы разные сигналы не подавались на шину одновременно!

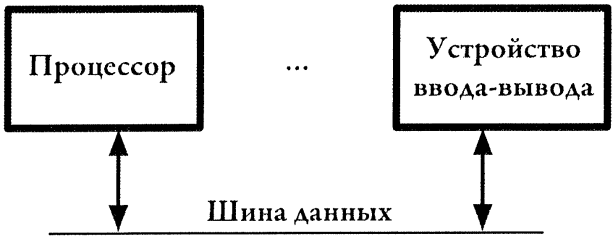



**Рис. 2.5.** Шина, соединяющая процессор и устройство ввода-вывода

При моделировании шин используют операторы `assign`, примитивные вентили и блоки `always` (но не `always_comb`). В следующих разделах рассматриваются способы моделирования шин и соединений.

## 2.7.1. Цепи

*Цепи* (*nets*) применяются для моделирования электрических соединений. Существует несколько типов цепей с разными электрическими свойствами (см. раздел 12.2). Все цепи в SystemVerilog основаны на 4-значной логике, однако физически они могут иметь либо два состояния (0, 1), либо три (0, 1, z). Объявление `wire` используется для цепей с двумя состояниями, а объявление `tri` – для цепей с несколькими драйверами, в которых возможны три состояния. Симулятор интерпретирует `tri` и `wire` одинаково; разница в том, что `tri` говорит разработчику, что моделируется тристабильная (tri-state) шина.

На рис. 2.6 показано, что происходит при формировании нескольких значений на цепях типа `wire` и `tri`. Рассмотрим пример, когда два оператора `assign` являются драйверами одной и той же цепи (это значит, что их левые части совпадают):

```
1   tri a;
2   assign a = top;
3   assign a = side;
```

| wire/tri | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | x | x | 0 |
| **1** | x | 1 | x | 1 |
| **x** | x | x | x | x |
| **z** | 0 | 1 | x | z |

**Рис. 2.6.** Формирование нескольких значений

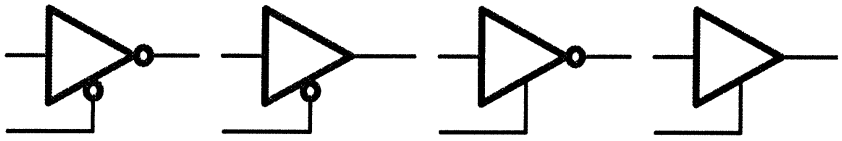
В таблице показано, как разрешается эта ситуация. Предположим, что первый оператор `assign` представлен строкой сверху, а второй – столбцом слева. Если оба оператора формируют 0, то на цепи будет 0. Если же один формирует 0, а другой – 1, то на цепи будет x. Как уже говорилось, x означает, что значение не определено – симулятор не знает, какое оно: 0, 1 или z.

Значение в цепи типа `tri` определяется следующим образом. Если x формируется на одном или нескольких выходах, соединенных с цепью, то в результате получится x. Если один или несколько драйверов устанавливают z, то любое значение — 0, 1 или x — заместит его (x вообще замещает любое значение). Правило работает и в случае, когда драйверов больше двух.

При отладке модели в симуляторе полезно знать, что происходит, когда в строке 2 рассматриваемого примера формируется 1, а в строке 3 – 0. Симулятор сообщит о значении x на линии шины, и вы поймете, что в модели имеется проблема. Отметим, что если такая ситуация произойдет в реальности, два драйвера вступят в конфликт, который, скорее всего, закончится тем, что линия выступит в роли предохранителя и расплавится – это не есть хорошо!

Микросхема превратится в поджаренный тост! В реальных схемах применяются специальные тристабильные вентили, которые либо устанавливают на линии 0 или 1, либо «формируют» z (выход, по сути, отсоединяется, и напряжение не подается на линию).

На рис. 2.7 показаны логические схемы тристабильных драйверов – инверторов и буферов. Сбоку у каждого драйвера имеется сигнал разрешения: если он установлен, драйверу разрешается формировать значение на шине; в противном случае выход отсоединяется, что при симуляции выглядит как формирование z.



Тристабильные инвертор и буфер с низким активным уровнем сигнала разрешения

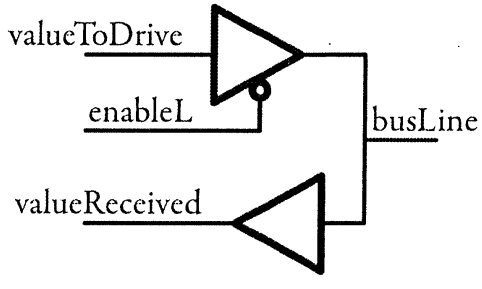
Тристабильные инвертор и буфер с высоким активным уровнем сигнала разрешения

**Рис. 2.7.** Логические схемы тристабильных драйверов

## 2.7.2. Тристабильные приемопередатчики

Воспользуемся операторами `assign` для проектирования шинных приемопередатчиков (см. пример 2.21). Обычно приемником шины является буфер; он воспроизводит значение на шине и усиливает сигнал для использования внутри подключенного к шине модуля. Таким образом, драйверы шины отвечают за передачу сигнала на интерфейсы (обеспечивая нужную для этого мощность); приемники усиливают полученный сигнал, делая возможным его внутреннее использование. В данном примере `valueReceived` – внутренний усиленный сигнал. Передатчик устроен, как описано выше. Если сигнал `enableL` установлен, то значение `valueToDrive` подается на линию `busLine`; в противном случае на выходе передатчика формируется z. В примере 2.21 приведена схема приемопередатчика вместе с его описанием на SystemVerilog.

```
1  module busTransceiver
2    (input logic enableL, valueToDrive,
3     output logic valueReceived,
4     inout logic busLine);
5
6    assign
7      valueReceived = busLine,
8      busLine = (~enableL) ? valueToDrive : 1'bz;
9  endmodule: busTransceiver
```




**Пример 2.21.** Шинный приемопередатчик и его логическая схема

В этом примере нам встретились две новые конструкции. Во-первых, это новый тип порта – `inout` (строка 4). До сих пор в наших модулях были только входные и выходные порты. В случае линий шин один и тот же порт иногда яв-

ляется входным (в момент приема), а иногда – выходным (в момент передачи). Тип inout позволяет описать такое поведение.

В строке 8 показано, как описывается тристабильный драйвер. Если сигнал разрешения уставлен, на линии busLine формируется значение valueToDrive; в противном случае – z. В данном примере enableL является сигналом с низким активным уровнем: если значение enableL равно 0, то ~enableL есть TRUE, и на линии busLine окажется valueToDrive.

Обычно шина представляется вектором, разрядность которого задается параметром в определении модуля, как показано в примере 2.22.

```
1 module busTransceiverParam
2   #(parameter w = 1)
3   (input logic enableL,
4         logic [w-1:0] valueToDrive,
5    output logic [w-1:0] valueReceived,
6    inout logic [w-1:0] busLine);
7
8   assign
9     valueReceived = busLine,
10    busLine = (~enableL) ? valueToDrive : 'bz;
11 endmodule: busTransceiverParam
```

**Пример 2.22.** Приемопередатчик многоразрядной шины

Запись 'bz в строке 10 говорит о том, что busLine имеет значение z, если enableL не установлен. Ранее в записи констант перед b стояло число, задающее разрядность; в данном случае константа безразмерна. Вне зависимости от разрядности busLine все биты будут иметь значение z.

Когда создается экземпляр модуля busTranceiverParam, он соединяется с линией, имеющей тип tri, что иллюстрируется ниже. В модуле system создаются шинные приемопередатчики для процессора (processor) и устройства ввода-вывода (ioGadget). В объявлении шины busLine указан тип tri; шина соединена с портами типа inout приемопередатчиков.

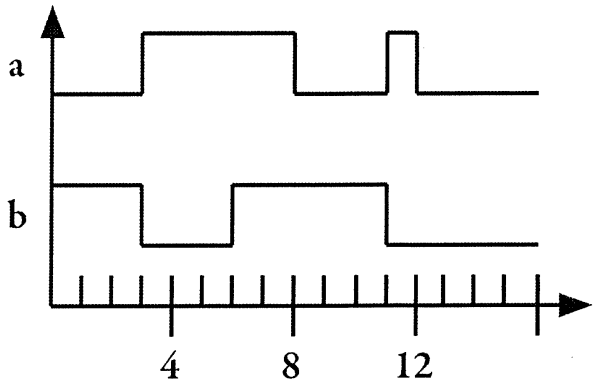
```
1   module system;
2     tri [31:0] busLine;
3     ...
4     busTranceiverParam #(32) processor (pEnable, pValueSend, pValueRecv, busLine);
5     busTranceiverParam #(32) ioGadget (ioEnable, ioValueSend, ioValueRecv, busLine);
6     ...
7   endmodule: system
```

## 2.8. Задачи и упражнения

В ответах на вопросы о проектировании комбинационных схем используйте операторы assign и/или always_ comb.

**2.1. Требуемая форма сигнала.** Напишите блок initial, который будет генерировать сигнал показанной ниже формы.

**2.2. Двоично-десятичный сумматор.** Напишите на языке SystemVerilog модуль, вычисляющий сумму двух чисел (a и b), представленных в двоично-десятичном коде[28], и входного бита переноса (carryIn). Ниже приведен заголовок модуля:




```
1   module bcdAdd
2     (input logic [3:0] a, b,
3      input logic carryIn,
4      output logic [3:0] sum,
5      output logic carryOut);
```

Выход sum – сумма в двоично-десятичном коде; carryOut – выходной бит переноса. Разработайте тестовое окружение и продемонстрируйте корректность сумматора.

**2.3. Параметризованное АЛУ.** АЛУ реализует следующие функции: ADD, AND, OR и XOR. Функция выбирается 2-битным входом fn: значение 0 соответствует ADD, значение 1 – AND и т. д. У АЛУ имеется входной бит переноса (cIn) и выходной бит переноса (cOut). Специальные биты N, Z и V показывают, является ли результат отрицательным (используется дополнительный код), является ли результат нулем и произошло ли переполнение. Входами являются a и b, выходом – result. Разработайте и протестируйте модуль АЛУ, используя перечислимый тип и параметр разрядности width.

**2.4. Какой порядковый номер дня относительно начала года?** Странно, но люди часто задают мне этот вопрос; нужно спроектировать схему для ответа на него.

**A)** Напишите SystemVerilog-модуль, вычисляющий номер дня в году; например, 1 февраля – 32-й день. На первом этапе проигнорируйте високосные годы. Заголовок модуля приведен ниже:

```
1   module dayOfYrCalc
2   (input logic [5:0] dayOfMonth,
3    input logic [3:0] month,
4    output logic [8:0] dayOfYear);
```

**B)** Теперь учтем високосные годы. Сначала определим (неправильно) високосный год как делящийся нацело на 4. Перепишите модуль так, чтобы он правильно вычислял номер дня при таком определении високосности. Заголовок нового модуля показан ниже. Подсказка: подумайте, как осуществить деление на степень основания системы счисления. Не используйте оператор деления.

---

[28] Двоично-десятичный код (Binary-Coded Decimal – BCD) – форма представления чисел, в которой каждая десятичная цифра записывается в виде двоичного кода. Обычно используется естественная кодировка (BCD 8421): 0 – 0000; 1 – 0001, ..., 9 – 1001 (остальные комбинации запрещены).

```
module dayOfYrCalc
  (input logic [5:0] dayOfMonth,
   input logic [3:0] month,
   input logic [10:0] year,
   output logic [8:0] dayOfYear);
```

**C)** Ну и наконец, более сложное определение. Год называется високосным, если он делится нацело на 4, но не делится на 100, а если делится, то делится и на 400. Таким образом, год 1900 не високосный, а год 2000 високосный. Измените модуль из части B, принимая в расчет вышесказанное. Напишите тестовое окружение, проверяющее годы с 1600 по 2020.

Прежде чем бросаться писать «программерский» код с операторами деления и вычисления остатка (в SystemVerilog есть и такие), примите во внимание, что они необязательно комбинационные, и результат синтеза может быть «кривым». Не используйте эти операторы в своем коде.

**2.5. Категории символов ASCII.** Напишите и испытайте в симуляторе модуль, принимающий 7-битный ASCII-символ и выдающий следующие выходы:

- `is_cap` – TRUE, если это заглавная буква (A–Z);
- `is_lc` – строчная буква (a–z);
- `is_num` – цифра (0–9);
- `is_printable` – печатный символ (от пробела до ~ включительно).

Вот заголовок этого модуля:

```
module charType
(input logic [6:0] code,
 output logic is_cap, is_lc, is_num, is_printable);
```

Разработайте модуль и тестовое окружение для его проверки. Тестовое окружение должно вывести на консоль таблицу, в которой для каждого 7-битного символа есть строка следующего вида:

| code | is_cap | is_lc | is_num | is_printable |
|---|---|---|---|---|
| code=00: | 0 | 0 | 0 | 0 |

**2.6. Автомат, выдающий сдачу.** (Интересно, что эта задача была опубликована в номере Wall Street Journal от 9 ноября 2011 года в статье о студентах, стремящихся выбрать профилирующий предмет полегче. Задача не такая уж и простая.)

Мы все привыкли к звону монет в торговом автомате. А как автомат определяет, сколько сдачи дать (и надо ли ее давать вообще)? В этом и состоит суть задачи – реализовать комбинационную часть схемы, отвечающей за выдачу сдачи.

Предположим, что где-то внутри автомата имеется ящик, заполненный монетами по 5, 10 и 25 центов. Предположим также, что нам известны цена товара и сумма, уплаченная покупателем. Требуется спроектировать комбинационную схему, которая делает две вещи:

- определяет, достаточно ли в ящике денег, чтобы выдать сдачу;
- определяет две монеты наивысшего достоинства, которые должны быть выданы в составе сдачи.

Если быть более точным, требуется по значениям указанных ниже входов генерировать значения указанных ниже выходов. У схемы 5 многобитных входов и 6 многобитных выходов, а именно:

- `Cost [3:0]`: 4-битный вход (беззнаковое целое), представляющий стоимость покупаемого товара в 5-центовиках. Суммы, не кратные 5 центам, не допускаются! Максимальная стоимость равна 15*5 = 75 центам;
- `Paid [3:0]`: 4-битный вход (беззнаковое целое), представляющий денежную сумму (число 5-центовиков), уплаченную покупателем (суммы, не кратные 5 центам, также не допускаются). Максимальная сумма равна 15*5 = 75 центам;
- `Quarters [1:0]`: 2-битный вход (беззнаковое целое), представляющий число 25-центовиков (quarters), доступных для сдачи;
- `Dimes [1:0]`: 2-битный вход (беззнаковое целое), представляющий число 10-центовиков (dimes), доступных для сдачи;
- `Nickels [1:0]`: 2-битный вход (беззнаковое целое), представляющий число 5-центовиков (nickels), доступных для сдачи;
- `FirstCoin[2:0]`: 3-битный выход, показывающий достоинство (в 5-центовиках) первой возвращенной монеты. Он может принимать значения 3'b001, 3'b010 и 3'b101 (5 центов, 10 центов и 25 центов), а также 3'b000 (монеты нет). Значение выхода должно соответствовать монете старшего достоинства из имеющихся в ящике и подлежащих выдаче. Например, если покупателю нужно вернуть 15 центов, выход `FirstCoin` должен иметь значение 3'b010 – 10 центов. Конечно, так будет, только если 10-центовики есть в наличии (см. вход `Dimes` выше). Если же их нет, значение `FirstCoin` должно соответствовать 5 центам (при наличии 5-центовиков);
- `SecondCoin[2:0]`: 3-битный выход, показывающий достоинство (в 5-центовиках) второй возвращенной монеты. Он, как и выход `FirstCoin`, может принимать значения 3'b001, 3'b010, 3'b101 и 3'b000. Значение должно соответствовать монете старшего достоинства, подлежащей выдаче (с учетом первой монеты). Например, если покупателю нужно вернуть 20 центов, значение `FirstCoin` должно соответствовать 10 центам (при наличии 10-центовиков), значение `SecondCoin` – тоже (если их не менее двух). Если же таковых монет нет, оба значения должны соответствовать 5 центам (при наличии 5-центовиков) (понятно, что в этом случае покупатель не получит свою сдачу полностью[29]);
- `ExactAmount`: 1-битный выход, показывающий, что уплаченная сумма в точности равна цене товара, поэтому сдача не нужна (т. е. `Paid == Cost`, и ни одно из чисел не равно 0);

[29] Предполагается, что сдача состоит из двух или менее монет.

- NotEnoughChange: 1-битный выход, показывающий, что требуется выдать сдачу, но у нас нет монет нужного достоинства (отметим, что речь идет не о выдаче двух 5-центовиков вместо одного 10-центовика. Проблема в том, что мы остались должны покупателю – он может начать колотить по автомату!);
- CoughUpMore: 1-битный выход, показывающий, что значение Paid меньше значения Cost;
- Remaining[3:0]: 4-битный выход, показывающий, сколько 5-центовиков мы остались должны покупателю (Paid - Cost - FirstCoin - SecondCoin).

Внимательный читатель уже, наверное, сосчитал, что на вход подается 14 бит, а на выходе выдается 13. Это означает, что таблица истинности содержит 27 столбцов и 16 384 строки. Не бросайтесь писать заявление об отказе от курса! Нужно придумать, как разбить модель на меньшие части, которые были бы обозримы. Потом можно будет составить из них единое целое.

Существует несколько способов разбить большую задачу на подзадачи; по мере накопления опыта вы научитесь отличать хорошие от плохих. Модель аппаратуры разбивается на модули, создаются экземпляры модулей и соединяются между собой; выходы одних модулей подаются на входы других.

Такой подход часто называют функциональным разбиением – выделяются основные функциональные элементы системы, которые разбиваются на меньшие части, проектируемые независимо. Например, для того чтобы определить первую монету сдачи, нужно сначала узнать общую сумму сдачи. Соответствующий комбинационный модуль просто вычисляет разность Change=Paid-Cost. Вот так внезапно восемь входных битов были перемещены в часть модели.

**Как подступиться?** Сначала нарисуйте диаграмму из модулей, реализующих простые функции. Такая схема, состоящая из основных комбинационных блоков (always_comb и assign), будет служить планом проекта.

Затем приступайте к кодированию модулей. Вычисление в каждом модуле должно быть описано оператором always_comb или assign (возможно, несколькими).

Проверьте свою модель в симуляторе (не выводите на консоль всю таблицу истинности!). Разработайте тестовое окружение.

**Примечание.** В SystemVerilog обычно используются беззнаковые числа. Если вы напишете if (a>=b)..., ветвь then будет исполнена, когда a>=b (в предположении, что a и b – беззнаковые числа). Например, 1111 больше 0101 (15 больше 5). Это относится ко всем операторам <, >, <=, >=, ==, !=.

Нужно понимать, что если вычислить c = a - b, а затем написать if (c>=0) ..., то условие всегда будет истинным. Напротив, в if (c<0) ... условие всегда будет ложным. Это правильно: c – беззнаковое число, оно всегда больше или равно 0! Не делайте так: используйте операторы вида if (a>b)... или if (a==b). Также отметим, что в if (c>0)... ветвь then будет исполнена при любом значении c, кроме 0.

# Глава 3

# Конечные автоматы

В проектировании цифровых систем конечные автоматы (Finite State Machine – FSM) – следующий уровень абстракции, расположенный выше комбинационных схем. Как уже было отмечено, комбинационные схемы характеризуются тем, что значения их выходов задаются с помощью булевых функций, зависящих лишь от значений входов, т. е. комбинационные схемы не содержат памяти. А вот при проектировании на уровне конечных автоматов в цифровую систему вводятся последовательные элементы: триггеры и защелки. Они служат для запоминания ключевой информации о функционировании системы – ее *состояния*.

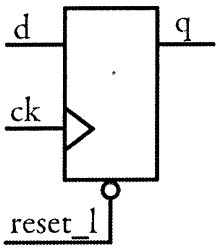
При совместном использовании комбинационных и последовательных элементов появляется возможность описания конечных автоматов. Изменение состояния автоматов синхронизируется специальным входным сигналом, называемым тактовым сигналом; смена состояния происходит по переднему или заднему фронту тактового сигнала[30]. Системы, в которых все действия синхронизируются тактовым сигналом, называются *синхронными*. У конечных автоматов есть входы и выходы. Значения выходов определяются функцией, зависящей от текущего состояния и, возможно, значений входов. Отметим, что для каждого состояния определено множество допустимых следующих состояний; следующее состояние определяется функцией, зависящей от текущего состояния и значений входов.

В рамках этой главы сначала будет рассмотрен D-триггер – последовательный элемент, широко применяемый при проектировании конечных автоматов. Затем мы обратимся к языку SystemVerilog как средству описания, симуляции и тестирования конечных автоматов. Далее поговорим о логической оптимизации, точнее о кодировании состояний; разные способы кодирования приводят к разным схемотехническим реализациям автомата, из которых можно выбрать наиболее подходящую. Мы рассмотрим два стиля описания конечных автоматов: *явный* и *неявный*. Явный стиль используется для описания синтезируемых моделей аппаратуры, а *неявный* – для разработки тестовых окружений. В заключение будет показана важность использования неблокирующего параллельного оператора присваивания (<=) для обновления состояния.

---

[30] *Передний фронт* (*positive edge*, *posedge*) – смена значения с 0 на 1; *задний фронт* (*negative edge*, *negedge*) – смена значения с 1 на 0.

```
1   module dFF
2     (output logic q,
3      input logic d, ck, reset_l);
4
5     always_ff @(posedge ck, negedge reset_l)
6     if (~reset_l)
7         q <= 0;
8     else q <= d;
9   endmodule: dFF
```




**Пример 3.1.** Базовый D-триггер

# 3.1. D-триггер

D-триггер – широко используемый в синхронных системах запоминающий элемент, который хранит один бит состояния и обновляет его значение по переднему или заднему фронту тактового сигнала (clock), используя значение информационного входа. Как правило, он имеет также входной сигнал сброса (reset) и/или предустановки (preset), что позволяет инициализировать состояние в 0 или 1 при включении питания или сбросе системы вне зависимости от тактового сигнала.

В примере 3.1 приведена схема D-триггера и его описание на SystemVerilog: `q` – состояние триггера; `d` – информационный вход; `ck` – тактовый сигнал; `reset_l` – сигнал сброса (с низким активным уровнем). Для моделирования D-триггера в SystemVerilog используется оператор `always_ff`. Это еще одна версия блока always, но если `always_comb` использовался для моделирования комбинационных схем, то `always_ff` *необходим для указания инструменту синтеза* определить D-триггер (или, как мы увидим далее, вектор D-триггеров). Блок `always_ff` реализован как бесконечный цикл, который ждет (знак @ в строке 5) *изменения значения* любой из переменных, указанных в скобках. В данном случае ожидается либо передний фронт `ck`, либо задний фронт `reset_l`. Если изменение значения хотя бы одной переменной удовлетворяет ожидаемым условиям, то `always_ff` переходит к исполнению операторов внутри цикла, затем сохраняет текущие значения `ck` и `reset_l`, после чего опять ожидает изменения значений этих переменных.

```
1 module reg8
2   (output logic [7:0] q,
3    input logic  [7:0] d,
4    input logic        ck, reset_l);
5
6   always_ff @(posedge ck, negedge reset_l)
7     if (~reset_l)
8         q <= 0;
9     else q <= d;
10 endmodule: reg8
```

**Пример 3.2.** 8-битный регистр

```
1 module register
2   #(parameter W = 8)
3   (output logic [W-1:0] q,
4    input logic  [W-1:0] d,
5    input logic          ck, reset_l);
6
7   always_ff @(posedge ck, negedge reset_l)
8     if (~reset_l)
9         q <= 0;
10    else q <= d;
11 endmodule: register
```

**Пример 3.3.** Параметризованный регистр

Рассмотрим ситуацию, когда значение reset_l изменяется с 1 на 0 (т. е. задний фронт). По данному фронту оператор @ переходит к исполнению операторов внутри цикла always_ff: происходит установка q в 0 (строка 7). А вот если бы имел место передний фронт ck, а reset_l, наоборот, был бы сброшен, то исполнилась бы ветвь else в строке 8, и значение q стало бы равным значению d. В обоих случаях по завершении операторов происходит переход в состояние ожидания следующего изменения значений переменных.

```
1  module datapath;
2     ...
3     register #(32) regA (Q, D, clock, r_l);
4  endmodule: datapath
```

**Пример 3.4.** Создание экземпляра регистра с указанием значения параметра

Когда reset_l установлен, значение q остается равным 0 даже при возникновении переднего фронта ck. При сбросе reset_l q сохраняет значение 0 до ближайшего фронта, в момент которого d будет имеет значение 1. Это стандартное поведение D-триггера с асинхронным сбросом, работающего по переднему фронту.

## 3.1.1. Смотри, куда ступаешь

Возможно, вы обратили внимание на фразу «необходим для указания инструменту синтеза», которую мы употребили при описании конструкций SystemVerilog. Рассмотрим подробно эту ситуацию. Для начала заметим, что инструмент синтеза знает, что для хранения значения q необходим триггер (об этом говорит ключевое слово always_ff), у которого есть требующие подключения входные и выходные соединения. Необходимо понимать, что имена ck, reset_l и d не являются особенными именами, используемыми только для тактового сигнала, сигнала сброса и сигнала данных: каждое из них можно заменить любым другим именем. Однако можно сказать, что у инструмента синтеза есть шаблон для разбора такого рода описания. Он знает, что нужно использовать D-триггер с асинхронным сбросом и что reset_l – это сигнал сброса, потому что по его заднему фронту триггер устанавливается в 0 (строка 7), а если этот сигнал отличен от 0, то в q запоминается значение входного сигнала d. Таким образом, d должен быть входом d триггера. Все эти вещи автоматически определяются, что помогает описывать триггеры приведенным выше образом.

Если бы строка 6 включала posedge reset_l (вместо negedge), а строка 7 оставалась без изменений, то многие инструменты синтеза не знали бы, что делать! Но если в строке 6 используется posedge reset_h, а в строке 7 – if (reset_h), то инструмент синтеза может сделать вывод: это асинхронный сброс с высоким активным уровнем.

Рассмотрим оставшийся код примера 3.1. Присваивание значения переменной состояния в строках 8 и 9 производится с помощью *неблокирующего присваивания*, иногда называемого *параллельным*, обозначаемого символом <=. Оно отличается от обычного (*блокирующего*) присваивания, используемого в операторе

always_comb, поддержкой моделирования следующей ситуации: в результате прихода даже одного фронта тактового сигнала может произойти изменение состояния многих (возможно, тысяч!) триггеров. И все они должны измениться ровно в один и тот же момент времени, ни одно состояние не может измениться раньше другого. Оператор <= моделирует такое одновременное присваивание всем битам состояния: симулятор гарантирует, что состояние тех тысяч триггеров изменится строго одновременно. Как мы увидим ниже, использование оператора = в этой ситуации могло бы привести к некорректному обновлению состояния.

Итак, при проектировании D-триггеров предлагается следовать примеру 3.1.

## 3.1.2. Вариации на тему

Иногда требуется использовать триггеры для хранения многобитной информации (т. е. вектора); для этого используется вектор триггеров, который называется *регистром*, его модель приведена в примере 3.2. От примера 3.1 она отличается спецификацией разрядности в строках 2 и 3. В данном случае получаются восемь D-триггеров с общим тактовым сигналом и сигналом сброса, т. е. 8-битный регистр.

Как показано в разделе 3.2.4, оператор parameter можно использовать для задания констант, которые получают значения в момент создания экземпляра. Так, в примере 3.3 показано, как параметризуется разрядность регистра. В строке 2 определяется параметр W, которому присваивается значение по умолчанию 8. Затем W используется в спецификациях разрядности в строках 3 и 4. При создании экземпляра этого модуля в строке 3 примера 3.4 параметр можно переопределить, присвоив ему другое значение. В данном случае конструкция #(32) говорит, что в этом экземпляре параметр W будет иметь значение 3, т. е. мы включаем в модель 32-битный регистр с указанными соединениями. Если при создании экземпляра модуля register не используется конструкция #(...), разрядность регистра будет такой, какой она задана по умолчанию. Оператор parameter, таким образом, уменьшает вероятность ошибки при создании нескольких вариантов часто используемых модулей.

## 3.1.3. Тестовое окружение для D-триггера

Для тестирования последовательной системы типа D-триггера или регистра необходимо более сложное тестовое окружение, чем для комбинационных схем. Дело в том, что теперь у схемы есть память, и, помимо этого, приходится иметь дело с фронтами сигнала сброса и тактового сигнала.

В примере 3.5 показано тестовое окружение для D-триггера из примера 3.1, а в примере 3.6 – результаты его симуляции, включая диаграмму сигналов. Сначала в описании тестового окружения определяются все переменные (строка 2). Поскольку их имена совпадают с именами в модуле dFF, то при создании экземпляра dFF его соединения могут быть описаны конструкцией .*, как это показано в строке 4. Конструкция .* означает, что сначала в описании мо-

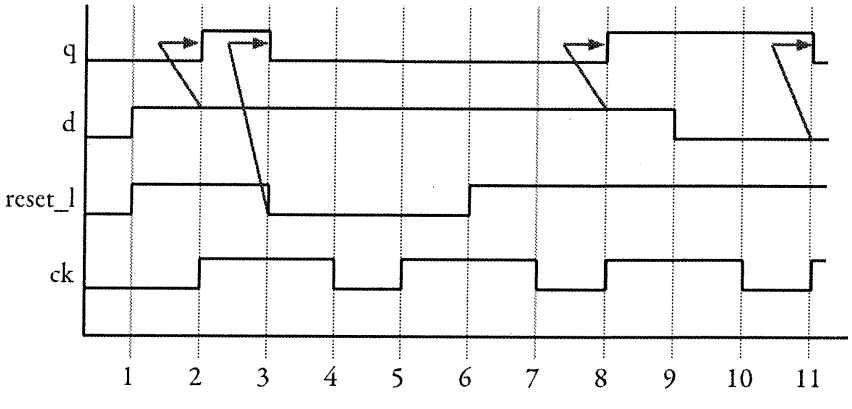
дуля dFF необходимо найти имена всех портов, а затем соединить эти порты с одноименными переменными в модуле testDff. (см. также раздел 2.5).

В блоке initial настраивается процедура $monitor, которая выводит на консоль время и все переменные в те моменты времени, когда хотя бы одна из них изменяется. В строках 11–13 всем входам dFF присваивается значение 0. В строке 14 симуляция приостанавливается на одну единицу времени. В конце момента времени 0 выводится строка 1 результата: мы видим, что значение q равно 0 (это вызвано установкой reset_l). В момент времени 1 (строка 14) сигнал reset_l сбрасывается, а d становится равным 1, что мы и видим в строке 2. Затем в момент времени 2 (строка 15) сигнал ck устанавливается (возникает передний фронт), что приводит к загрузке в q значения 1; это отражено в строке 3 результата и во временно́й диаграмме.

```
1  module testDff;
2    logic q, d, ck, reset_l;
3
4    dFF q0(.*); // из примера 3.1
5
6    initial begin
7      $monitor($time,
8        " d=%b, q=%b, ck=%b, reset_l=%b",
9        d, q, ck, reset_l);
10
11     ck = 0;
12     reset_l = 0;
13     d = 0;
14     #1 reset_l = 1; d = 1;
15     #1 ck = 1;
16     #1 reset_l = 0;
17     #1 ck = 0;
18     #1 ck = 1;
19     #1 reset_l = 1;
20     #1 ck = 0;
21     #1 ck = 1;
22     #1 d = 0;
23     #1 ck = 0;
24     #1 ck = 1;
25   end
26 endmodule: testDff
```

**Пример 3.5.** Тестирование D-триггера

```
1  0 d=0, q=0, ck=0, reset_l=0
2  1 d=1, q=0, ck=0, reset_l=1
3  2 d=1, q=1, ck=1, reset_l=1
4  3 d=1, q=0, ck=1, reset_l=0
5  4 d=1, q=0, ck=0, reset_l=0
6  5 d=1, q=0, ck=1, reset_l=0
7  6 d=1, q=0, ck=1, reset_l=1
8  7 d=1, q=0, ck=0, reset_l=1
9  8 d=1, q=1, ck=1, reset_l=1
10 9 d=0, q=1, ck=1, reset_l=1
11 10 d=0, q=1, ck=0, reset_l=1
12 11 d=0, q=0, ck=1, reset_l=1
```




**Пример 3.6.** Результат симуляции примера 3.5

В момент времени 3 устанавливается сигнал сброса (строка 16). На фоне сброса в момент времени 5 возникает передний фронт тактового сигнала (строки 17–18): видно, что триггер остается сброшенным. На следующем такте в q загружается значение 1, а еще через такт – 0.

## 3.2. Основы проектирования конечных автоматов

Если бы проектирование конечных автоматов сводилось только к заданию регистров, жизнь была бы гораздо проще. Однако этот процесс состоит из нескольких шагов, которые мы кратко опишем. Конечный автомат – это последовательная система, действия которой производятся в строго определенные моменты времени: по фронту тактового сигнала. Одни действия зависят от значений, хранящихся в системе, другие исполняются безусловно. Примером последовательной системы может служить светофор, а состоянием в этом случае являются комбинации светящихся лампочек, обозначающие, кому можно ехать, а кому переходить улицу. Некоторые из этих состояний последовательные и безусловные, например желтый всегда сменяется красным, а некоторые – условные, как, например, активация датчика левого поворота.

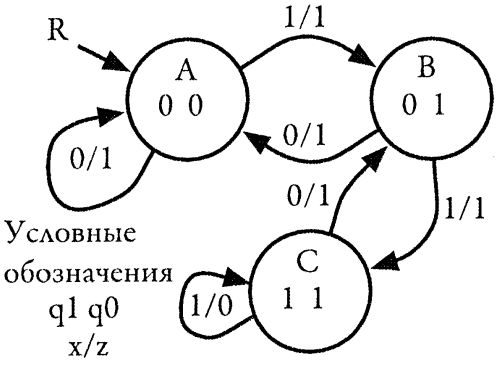



**Рис. 3.1.** Диаграмма состояний

Последовательные системы часто описывают с помощью диаграмм состояний, как показано на рис. 3.1. Каждый круг на этой диаграмме соответствует состоянию системы, которые обозначаются буквами от A до C. Стрелки, помеченные значениями входов и выходов, указывают на следующее состояние системы. В данном случае мы имеем один вход x и один выход z, что можно записать в виде x/z. Если система находится в состоянии A, то при x, равном 1, на ее выходе будет тоже 1, а ее следующим состоянием будет B. Если система получит на входе 0, будучи в состоянии A, то ее следующее состояние не изменится – стрелка ведет обратно в состояние A.

Проектируемые нами цифровые системы переходят в следующее состояние по переднему (или заднему) фронту тактового сигнала, подаваемого на триггеры. Если установлен сигнал сброса, то система переходит в состояние R.

Чтобы спроектировать такую систему с тремя состояниями, нужно иметь возможность сохранять текущее состояние системы. Для этого нужно, по меньшей мере, два триггера, которые мы назовем q1 и q0. Как показано на рисунке, когда система находится в состоянии A, триггерам присваиваются значения

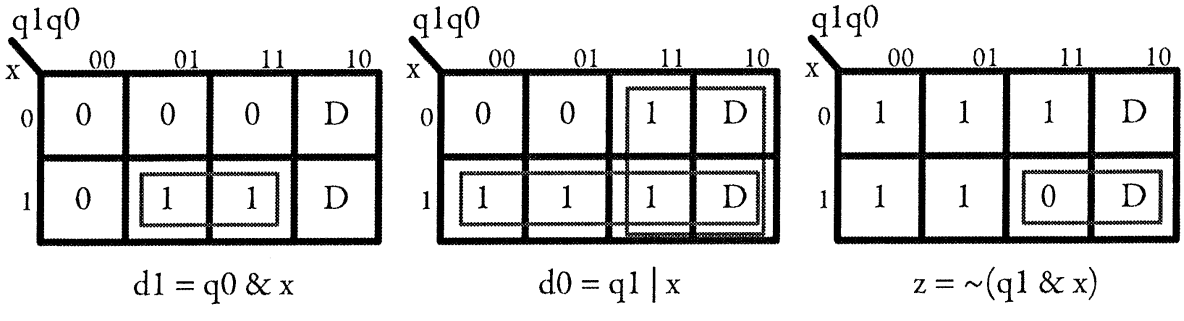



**Рис. 3.2.** Карты Карно

q1 = 0 и q0 = 0. В состоянии B триггеры имеют значения 2'b01, а в состоянии C – значения 2'b11. Это называется *присваиванием значений состояниям*.

При выбранном способе кодирования состояний проектирование системы сводится к разработке комбинационной схемы: следующее состояние является функцией текущего состояния и входов. Поскольку мы будем использовать D-триггеры с выходами q1 и q0, значения d1 и d0 будут следующими состояниями соответствующих триггеров. Для определения d1 и d0 можно использовать карты Карно, как показано на рис. 3.2. Карта для выхода z (функции текущего состояния и входа) также изображена на рисунке. По этому рисунку можно сгенерировать логическую схему, показанную на рис. 3.3.

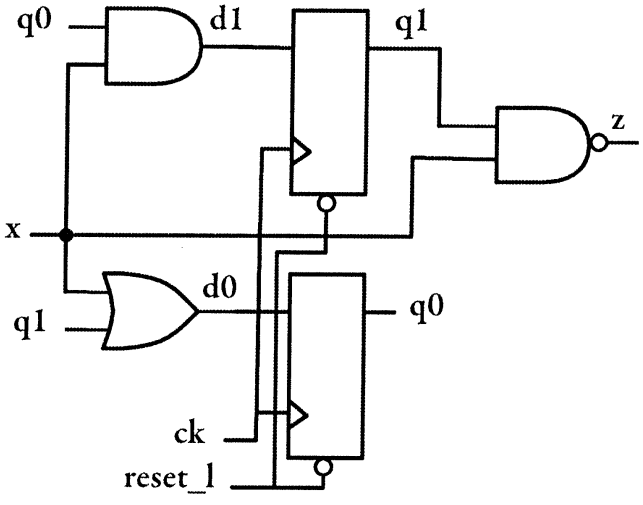



**Рис. 3.3.** Логическая схема конечного автомата

```
1 module FSM
2   (input logic x, ck, reset_l,
3    output logic z);
4   logic q1, q0;
5
6   always_ff @(posedge ck, negedge reset_l) begin
7     if (~reset_l)
8         {q1, q0} <= 2'b00;
9     else begin
10        q0 <= q1 | x;
11        q1 <= q0 & x;
12    end
13  end
14
15  assign z = ~(x & q1);
16 endmodule: FSM
```

**Пример 3.7.** Конечный автомат

## 3.2.1. Описание на SystemVerilog

Имея только логическую схему, изображенную на рис. 3.3, можно сконструировать описание на SystemVerilog (по сути, выполнить обратное проектирование); это описание приведено в примере 3.7. Входами конечного автомата являются x, ck и reset_l, выходом – z. В строке 4 определены 1-битные переменные q1 и q0, которые являются переменными состояния. Далее мы видим два блока кода: always_ff в строках 6–13 и assign в строке 15. Оператор assign из них самый простой: выход z получается применением операции NAND к входу x и q1.

Блок always_ff похож на те, что мы видели ранее при рассмотрении триггеров. Сигнал сброса является асинхронным: по заднему фронту reset_l блок always_ff исполняется, и состояние {q1, q0} устанавливается в 2'b00. Фактически последнее является присваиванием значения состояния данного автомата после сброса. Использование конкатенации в левой части оператора присваивания (в строке 8) – конструкция, которая нам раньше не встречалась при рассмотрении языка SystemVerilog. Левая часть представляет собой конкатенацию двух скалярных переменных, т. е. 2-битный вектор. Правая часть – также 2-битный вектор. А в результате 2-битный вектор загружается в две конкатенированные скалярные переменные.

Продолжим рассмотрение блока always_ff. Если несколько фронтов тактового сигнала придет в течение времени, когда reset_l равно 0, внутренний цикл исполнится, но он лишь оставит состояние равным 2'b00. Однако первый же фронт тактового сигнала после возврата reset_l в 1 приведет к исполнению блока else внутри always_ff и, соответственно, загрузке в триггеры следующего состояния. Присваивания в строках 10 и 11 написаны в соответствии с рис. 3.3. В рассматриваемом always_ff изменение состояния синхронизировано с тактовым сигналом ck, а сброс является асинхронным относительно того же ck. Это соответствует функциональности логической схемы на рис. 3.3.

Как и в случае комбинационных схем, always_ff и отдельные строки оператора assign можно представлять в виде виртуальных модулей. В строках 6–13 примера 3.7 находится блок always_ff, отвечающий за переход в следующее состояние и логику работы триггера, а в строке 15 находится оператор assign, отвечающий за выходное значение. При разработке описаний на SystemVerilog всегда помните о том, что рассмотренные операторы занимают место на интегральной схеме или на печатной плате. Как говорилось раньше, описание на SystemVerilog – это не обычная программа: каждому из рассмотренных элементов соответствует схема, которая *непрерывно* выполняет свою функцию одновременно с другими схемами.

Представить работу модуля FSM можно также следующим образом: когда приходит передний фронт тактового сигнала, в триггеры загружаются новые значения q1 и q0. Затем значение q1 доходит до оператора assign, он генерирует новое значение z. Между этим моментом и следующим тактом x может измениться; если это случится, то assign пересчитает значение выхода z.

## 3.2.2. Неблокирующие (параллельные) присваивания

В примере 3.8 показана типичная ошибка при описании обновления переменных состояния. Единственное различие между этим примером и примером 3.7 находится в строках 10–11: в примере 3.8 неправильно используются операторы присваивания =. Здесь необходимо неблокирующее присваивание <=, поскольку оно говорит, что оба оператора (присваивание значений q0 и q1 в строках 10 и 11) должны исполняться одновременно. Ведь и в физической схеме (рис. 3.3) эти присваивания происходят в одно и то же время.

```
1 module FSM_wrong // Ошибочный!
2   (input logic x, ck, reset_l,
3    output logic z);
4   logic q1, q0;
5
6   always_ff @(posedge ck, negedge reset_l) begin
7     if (~reset_l)
8         {q1, q0} <= 2'b00;
9     else begin
10        q0 = q1 | x;    // неправильно!
11        q1 = q0 & x;    // неправильно!
12    end
13  end
14
15  assign z = ~(x & q1);
16 endmodule: FSM_wrong
```

**Пример 3.8.** Некорректный модуль из примера 3.7

В качестве синонима неблокирующее присваивание иногда называют *параллельным*. Обновление значений левых частей всех неблокирующих присваиваний происходит в один и тот же момент времени, то есть параллельно. Такие присваивания производятся синхронно, по одному и тому же фронту тактового сигнала; таким образом, не следует считать, что на рис. 3.3 q1 обновится раньше q0 и влияет на значение d0 (а значит, и q0). Всякий раз, когда вы видите параллельное присваивание, вспомните, что именованная переменная в левой части будет сохранена в триггере (или в регистре, если это вектор). Для обычного блокирующего присваивания (=) это не так.

В других языках описания аппаратуры неблокирующая операция присваивания может называться по-другому, например *буферизованное обновление*, но и здесь речь идет о том, что несколько значений изменяется в точности одновременно, так что нельзя сказать, что какое-либо присваивание произошло раньше других.

Отметим, что если бы в примере 3.7 использовались блокирующие присваивания для обновления значений переменных q0 и q1, то результат работы модуля получился бы некорректным. Так, в строке 11 q1 получила бы значение (q0 & x), где q0 – значение, вычисленное в предыдущей строке 10. На логической схеме на рис. 3.3 это выглядело бы так, будто тактовый сигнал привел к появлению нового значения q0 на выходе триггера, и затем оно появилось бы на входе d1 триггера q1 еще до того, как тот же самый фронт тактового сигнала дошел до q1. Однако имеющаяся логическая схема утверждает совершенно другое: q0 и q1 обновляются одновременно по фронту тактового сигнала, новые значения q0 и q1 независимы и никак не влияют друг на друга. Такая ситуация моделируется как раз неблокирующим присваиванием <=. Отметим, что параллельное присваивание обрабатывается приведенным выше образом не только в пределах одного и того же блока always_ff, но и во *всех* блоках always_ff данной модели, зависящих от одного и того же фронта тактового сиг-

нала. Таким образом, если бы модель содержала тысячи таких операторов присваивания (и, стало быть, тысячи триггеров!), то симулятор исполнил бы их так, как будто все они происходят одновременно. В реальной аппаратуре такие присваивания происходят одновременно, и SystemVerilog моделирует именно такую реальную аппаратуру.

Теперь рассмотрим пример 3.9, который отличается от примера 3.7 только переставленными местами строками 10 и 11. Поскольку в обоих случаях используется неблокирующее присваивание <=, то поведение в обоих примерах в точности совпадает. Этот факт можно понимать следующим образом: правые части всех неблокирующих операторов присваивания вычисляются до обновления их левых частей. Поэтому и получается, что все присваивания происходят одновременно, что, как уже было сказано, и происходит в реальной аппаратуре.

```
1 module FSM_alternate
2   (input logic x, ck, reset_l,
3    output logic z);
4   logic q1, q0;
5
6   always_ff @(posedge ck, negedge reset_l) begin
7     if (~reset_l)
8         {q1, q0} <= 2'b00;
9     else begin
10        q1 <= q0 & x;   // строки переставлены местами
11        q0 <= q1 | x;
12    end
13  end
14
15  assign z = ~(x & q1);
16 endmodule: FSM_alternate
```

**Пример 3.9.** То же поведение, что и в примере 3.7

Даже если переписать этот пример таким образом, чтобы переменные q1 и q0 были в разных модулях, то при условии что для обновления значений обеих переменных используется один и тот же фронт одного и того же тактового сигнала, такое описание все равно будет корректным.

## 3.2.3. Другой взгляд на = и <=

При обсуждении симулятора в разделе 1.2 рассматривались только блокирующие вычисления и события обновления значений. Они возникали в результате вычислений выходных значений вентилей, а также исполнения блокирующих присваиваний (=) и оператора #delay в процедурных блоках. В этом разделе мы будем называть их *обычными* (*regular*) событиями. Неблокирующими событиями называются события обновления значений, возникающие в результате исполнения неблокирующего присваивания, как, например, в строках 10 и 11 примера 3.9. Эти события хранятся отдельно от обычных.

Симулятор в каждый момент времени выбирает только обычные события, оставляя неблокирующие в списке событий. Обработка обычных событий может привести к другим обычным событиям для того же момента времени (так может получиться при исполнении событий, между которыми отсутствует какая-либо задержка). Эти новые обычные события будут выбраны и исполнены в следующем цикле симуляции. Если какое-то из тех событий приводит к неблокирующему событию обновления (присваиванию оператором <=), то это неблокирующее событие будет сохранено отдельно, вместе с остальными неблокирующими событиями, и не будет выбрано в следующем цикле симуляции вместе с обычными событиями.

После того как все обычные события для текущего момента времени обработаны, симулятор выбирает неблокирующие события обновления. Они используются для обновления переменных состояния (триггеров и регистров). Таким образом, все правые части неблокирующих операторов присваивания вычисляются до фактического обновления состояния (а значения левых частей хранятся в составе неблокирующих обновлений). Отметим, что все неблокирующие обновления совершаются атомарно – вы не сможете увидеть ни одного отдельного обновления, только все одновременно. Поэтому невозможно состояние гонки, когда выходное значение одного триггера подается на вход d другого триггера до прихода тактового сигнала.

В результате неблокирующих обновлений могут быть сгенерированы дополнительные обычные события для текущего момента времени (например, перевычисление выходного значения конечного автомата, на вход которого подается переменная состояния, является обычным событием).

Если в один и тот же момент времени запланированы два или более неблокирующих обновления одной и той же переменной, будет исполнено только последнее из них.

В примере 3.10 показана иллюстрация отличия блокирующего (=) и неблокирующего (<=) присваиваний. Модули ff_A и ff_B различаются только в строках 7 и 18: присваивание значения переменной a осуществляется либо оператором =, либо оператором <=. На рис. 3.4 приведены логические схемы, соответствующие этим описаниям. В модуле ff_A значение b&c загружается в переменную a с помощью блокирующего присваивания. Затем в следующей строке (8) переменная q получает это значение посредством неблокирующего присваивания. Таким образом, значение, которое будет иметь q после прихода фронта тактового сигнала, – это конъюнкция текущих значений b и c.

```
 1 module ff_A
 2   (output logic q,
 3    input logic b, c, ck);
 4   logic a;
 5
 6   always_ff (posedge ck) begin
 7     a = b & c;
 8     q <= a;
 9   end
10 endmodule: ff_A
11
12 module ff_B
13   (output logic q,
14    input logic b, c, ck);
15   logic a;
16
17   always_ff (posedge ck) begin
18     a <= b & c;
19     q <= a;
20   end
21 endmodule: ff_B
```

**Пример 3.10.** Использование операторов = и <=

В модуле ff_B значение b&c загружается в переменную a с помощью неблокирующего присваивания в строке 18. Поэтому a трактуется как триггер; новое значение a появится одновременно со всеми остальными неблокирующими присваиваниями. (Напомним, что до обновления левой части неблокирующего присваивания вычисляются все правые части. Благодаря этому обновления оказываются одновременными.) В строке 19 присваивание также неблокирующее. Поскольку значение a в правой части этого присваивания еще не обновлено предыдущей строкой, используется то значение, которое было загружено в a по предыдущему фронту тактового сигнала. Поэтому в схеме на рис. 3.4 присутствует дополнительный триггер. Присваивания в строках 18 и 19 исполняются одновременно, и значение, загруженное в q, является конъюнкцией не текущих значений входных значений, а входных значений в момент прихода предыдущего фронта тактового сигнала.

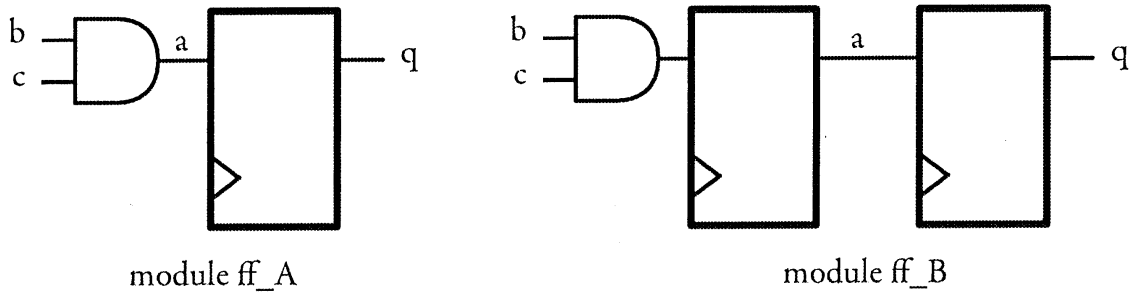



**Рис. 3.4.** Логические схемы модулей из примера 3.10

Какую разницу может сделать всего лишь один символ в спецификации! Общим правилом тут является использование = при проектировании комбинационных схем и использование <= для присваивания значений переменным состояния при проектировании последовательных схем.

Пример 3.11 также помогает разобраться в работе операторов неблокирующего присваивания. В блоке обновления состояния always_ff (строки 13–17) есть одно неблокирующее присваивание переменной a, за которым идет вызов процедуры $display, выводящий на консоль текущее время и значения a, b и c. Поскольку неблокирующее присваивание не приводит к немедленному обновлению левых частей операторов присваиваний, то $display не покажет значение, только что присвоенное a. В блоке initial процедура $monitor настаивает вывод той же строки (с единственным отличием в первой букве строки). После инициализации переменных в строках 7–8 блок initial приостанавливается на одну единицу времени. Так как это конец момента времени 0, то $monitor выводит сообщение (см. строку 20 после слова endmodule), из которого видно, что в конце этого момента времени значение a еще неизвестно (x). Действительно, этой переменной еще не было присвоено значение.

Затем время сдвигается на 1 единицу вперед, и ck устанавливается в 1 (строка 9). Это порождает событие вычисления значения для блока always_ff, по-

скольку он ждет прихода переднего фронта ck. Блок исполняется и записывает в a значение 1 с помощью неблокирующего присваивания. Далее (строка 15) процедура $display выводит результат на консоль (строка 21). В момент времени 1 a по-прежнему равно x. Как и раньше, это объясняется тем, что неблокирующие присваивания не приводят к обновлению значений своих левых частей до тех пор, пока не будут вычислены все правые части всех неблокирующих присваиваний. Когда блок always_ff заканчивает исполнение, симулятор не видит необработанных обычных событий, поэтому переходит к неблокирующим обновлениям (в данном случае значения переменной a). После этого никаких событий для обработки в данный момент времени уже нет, и $monitor выводит на консоль сообщение (строка 22), из которого видно, что в конце момента времени 1 переменная a действительно получила значение 1.

```
1  module DvsM;
2    logic a, b, c, ck;
3
4    initial begin
5      $monitor ("M(%1d): a=%b, b=%b, c=%b",
6              $stime, a, b, c);
7      ck = 0;
8      b = 1; c = 1;
9      #1 ck = 1;
10     #1 $finish;
11   end
12
13   always_ff @(posedge ck) begin
14     a <= b & c;
15     $display ("D(%1d): a=%b, b=%b, c=%b",
16             $stime, a, b, c);
17   end
18 endmodule: DvsM
19 /* вывод результатов на консоль
20 M(0): a=x, b=1, c=1
21 D(1): a=x, b=1, c=1
22 M(1): a=1, b=1, c=1
23 */
```

**Пример 3.11.** Сравнение процедур $display и $monitor

## 3.2.4. Наглядное изображение диаграмм состояний

У авторов учебников и разработчиков разное понимание того, как надо изображать диаграммы состояний; они расходятся по всем пунктам. Свои рекомендации мы представили на рис. 3.5.

- *Состояние сброса (начальное состояние).* Одно состояние всегда помечается как начальное, с использованием буквы R и стрелки, указывающей на это состояние.
- *Имя состояния.* Каждому состоянию присваивается уникальное осмысленное имя. Кодировка состояний обычно не показывается (она задается в модели на SystemVerilog).
- *Условие перехода.* Условие изображается рядом со стрелкой, ведущей в следующее состояние, в которое автомат переходит, если это выражение истинно.
- *Выходные значения.* Список значений выходов показывается либо рядом со стрелкой перехода (для автоматов Мили), либо в самом состоянии (для автоматов Мура). В списке присутствуют только установленные выходы: диаграмму проще читать, когда она не обременена длинным списком всех выходов (включая те, значения которых не обновились). Иногда в список включают несущественные выходы.

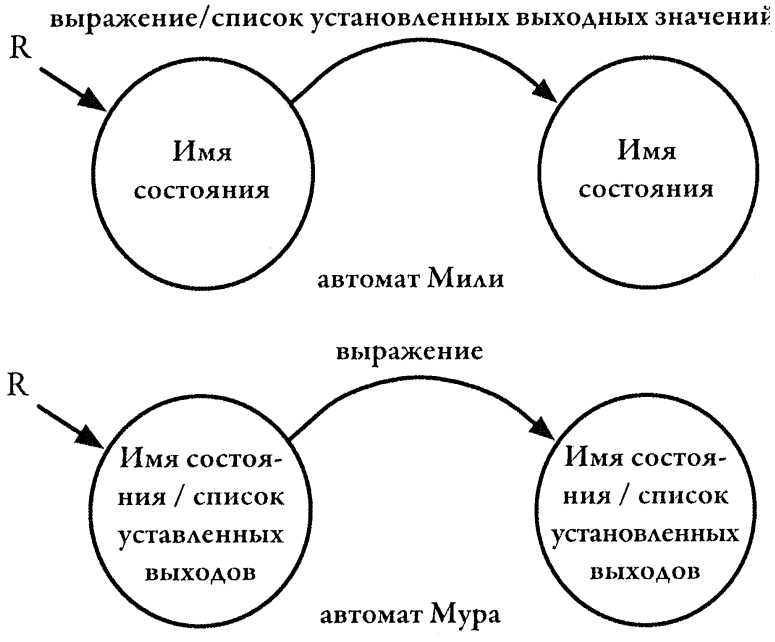



**Рис. 3.5.** Рекомендации по изображению диаграмм состояний

У каждой рекомендации есть исключения: некоторые отхождения от приведенных правил могут встретиться и в этой книге.

## 3.2.5. Формальное определение

С формальной точки зрения конечный автомат – это кортеж (X, Z, S, δ, λ, C, R), где:

- X – множество комбинаций значений входов. В примере 3.9 X = {1, 0};
- Z – множество комбинаций значений выходов. В нашем примере Z = {1, 0};
- S – множество состояний, включая состояние сброса (начальное состояние) $s_0 \in$ S. В нашем примере S(q1, q0) = {00, 01, 11} и $s_0$ = 00;
- δ – функция перехода (δ: X × S → S). В нашем примере ей соответствует комбинационная схема с двумя выходами: d1 = q0 & X и d0 = q1 | X;
- λ – функция выхода (λ: X × S → Z для автоматов Мили и λ: S → Z для автоматов Мура). В нашем примере используется автомат Мили: Z = ~(q1 & x));
- C – тактовый сигнал (или, как мы обсудим ниже, домен синхронизации);
- R – сигнал сброса, по которому автомат переходит в состояние $s_0$. Сброс является асинхронным (не нужно ждать фронта тактового сигнала – сброс происходит немедленно).

Некоторые элементы этого кортежа показаны на рис. 3.6 – абстрактном представлении автомата на рис. 3.3. Для разработки SystemVerilog-модели, соответствующей рис. 3.6, нужны две комбинационные схемы, определяемые с помощью `always_comb` или `assign`, и последовательное обновление, задаваемое с помощью `always_ff`.

Как уже отмечалось, когда присваивания в переменные состояния сделаны и их значения определены, дело остается за комбинационными схемами. Это так, однако добавление состояния кардинально меняет способ понимания, моделирования, реализации и использования систем. Далее в этой главе мы сосредоточимся на моделировании.

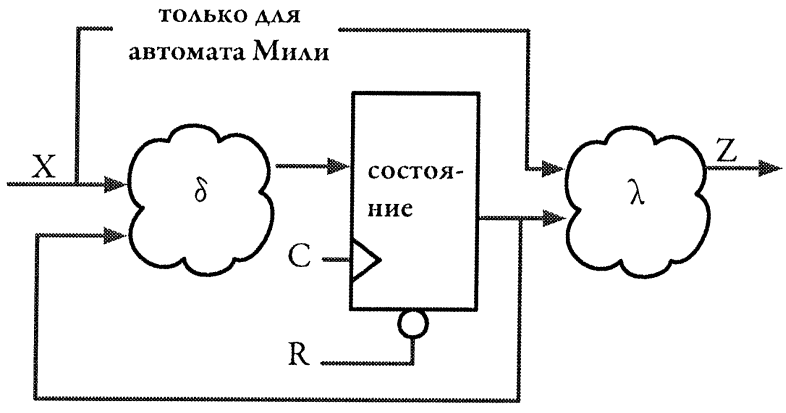



**Рис. 3.6.** Абстрактное представление конечного автомата

## 3.3. Явный стиль описания конечных автоматов

Предыдущий пример на SystemVerilog был написан на основе уже реализованной схемы на уровне вентилей, чтобы было проще их сравнить и разобраться в разных вопросах, связанных с моделированием конечных автоматов: параллельном присваивании, запуске по фронту тактового сигнала и асинхронном сбросе. Однако если в технологическом процессе проектирования используются инструменты синтеза, никто не занимается начальным проектированием до уровня вентилей. Как и в главе, посвященной проектированию комбинационных схем, обратимся к языковым абстракциям, подаваемым на вход инструмента синтеза.

Часто проектирование начинается с диаграммы состояний, как это показано на рис. 3.7 (мы уже ее использовали в качестве примера). На этом рисунке изображены две диаграммы: слева показана исходная диаграмма, а справа – перерисованная в соответствии с рекомендациями из раздела 3.2.4. Следуя этой диаграмме, мы создаем модель для каждой из трех основных частей, реализующих идеи раздела 3.2.5: вычисление значений выходов, вычисление следующего состояния и обновление состояния. Эти три части показаны в примере 3.12. Блок `always_ff` в строках 7–16 описывает обновление состояния и вычисление следующего состояния. Блок `always_comb` в строках 17–20 описывает вычисление значения выхода. Такой стиль задания конечного автомата назы-

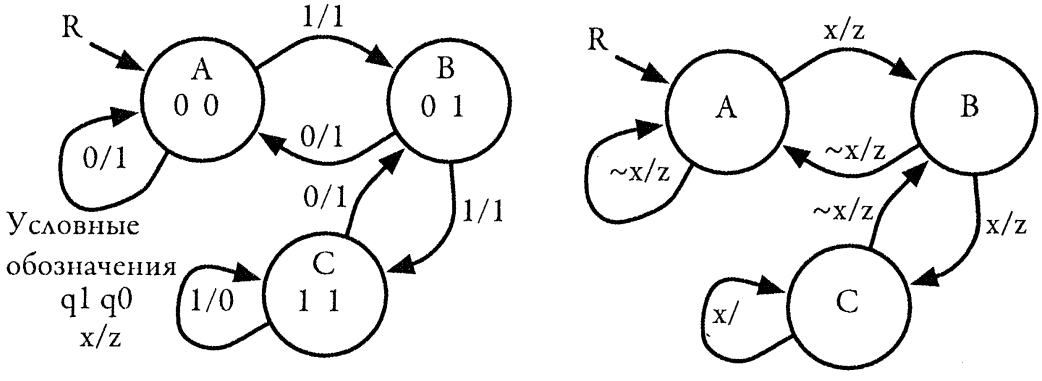



**Рис. 3.7.** Копия и модификация рис. 3.1

вается *явным*: явным образом перечисляются все состояния, задаются присваивания в переменную состояния, определяются функции перехода и выхода. *Неявный* стиль описания конечных автоматов рассматривается в разделе 7.3.3. Обратимся к примеру 3.12.

Для начала обсудим перечислимые типы, с которыми мы уже сталкивались в разделе 2.6.3. Конструкция enum позволяет ввести в модель именованные константы и объявить переменные для работы с ними. В данном случае в строке 5 определяется перечислимый тип с элементами A, B и C и объявляется переменная state. Для переменных типа enum осуществляется контроль типов: им можно присваивать только константы, заданные в определении типов, и сравнивать только с переменными этого же типа или с константами. Отметим, что нас практически не интересуют значения элементов перечислений. Тем не менее если значения явным образом не заданы, по умолчанию это 32-битные целые числа (типа int), нумеруемые с 0. Например, значениям A–C соответствуют числа из диапазона 0–2. Как мы увидим ниже, базовый тип элементов и их значения можно переопределить.

```
1 module FSMbehavior
2 (input logic x, ck, r_l,
3  output logic z);
4
5 enum {A, B, C} state; // значения не такие,
6                       // как на рис. 3.7
7 always_ff @(posedge ck, negedge r_l) begin
8   if (~r_l)
9       state <= A;
10  else case (state)
11      A: state <= (x) ? B : A;
12      B: state <= (x) ? C : A;
13      C: state <= (x) ? C : B;
14      default: state <= A;
15  endcase
16 end
17 always_comb begin
18  z = 1'b1;
19  if (state == C) z = ~x;
20 end
21 endmodule: FSMbehavior
```

**Пример 3.12.** Модель, соответствующая рис. 3.7

Теперь рассмотрим блок always_ff. Если сигнал сброса установлен (строки 8–9 примера 3.12), то state присваивается значение A. Если сигнал сброса не уставлен, то следующее состояние определяется оператором case, находящимся в ветви else. Вообще, описание на SystemVerilog может быть создано на основе только диаграммы состояний. Рассмотрим состояние A на рис. 3.7 и в строке 11 примера 3.12. Диаграмма состояний говорит, что если мы находимся в состоянии A и на вход x поступает 1, то следующим состоянием будет B, а иначе следующим состоянием будет A. Это соответствует строке 11 кода. Отметим, что в оператор case была добавлена ветвь default, которая обрабатывает ситуацию, когда state принимает значение, отличное от A, B или C: следующим состоянием автомата станет начальное (A). Как и всегда, при обновлении текущего состояния мы пользуемся неблокирующим оператором присваивания (<=). Инструмент синтеза оптимизирует комбинационную схему в правой части операторов (строки 11–14) и присоединит их выходные значения к входам D триггеров состояния.

Оператор always_comb, описывающий вычисление значения выхода, также можно написать на основе диаграммы состояний. Заметим, что в обоих состояниях A и B выходное значение всегда равно 1, а в состоянии C выходное зна-

чение будет являться дополнением входного значения. Эти требования к работе модуля реализованы в строках 17–20 примера 3.12, которые заодно иллюстрируют, что переменную state можно сравнивать с константой, определенной в enum. В более сложных случаях функция выхода записывается с помощью оператора case.

В альтернативном варианте этой модели (см. пример 3.13) все три части конечного автомата (обновление регистра состояния, вычисление следующего состояния и вычисление значения выхода) полностью разделены. Обновление состояния описано в строках 6–8 и напоминает спецификацию регистра (и это не случайно!). Строки 10–16 – описание вычисления следующего состояния с помощью оператора always_comb. Наконец, строка 18 – это описание вычисления значения выхода с помощью оператора assign. При таком стиле проектирования явным образом записываются все основные блоки рис. 3.6. Многие считают, что такой способ описания конечного автомата понятнее, чем тот, что использовался в примере 3.12. С другой стороны, код в примере 3.12 более эффективен с точки зрения симулятора, он исполняется быстрее кода из примера 3.13. Как мы увидим далее, код в примере 3.12 производит вычисление следующего состояния только в случае прихода фронта тактового сигнала, а не при любом изменении переменной state или x. А вообще, оба варианта логически эквивалентны. Можете выбрать тот, который кажется вам понятнее.

```
1  module FSMalternate
2    (input logic x, ck, r_l,
3    output logic z);
4
5    enum {A, B, C} state, nextState;
6    always_ff @(posedge ck, negedge r_l)
7      if (~r_l) state <= A;
8      else state <= nextState;
9
10   always_comb
11     case (state)
12       A: nextState = (x) ? B : A;
13       B: nextState = (x) ? C : A;
14       C: nextState = (x) ? C : B;
15       default: nextState = A;
16     endcase
17
18   assign z = (state == C) ? ~x : 1'b1;
19 endmodule: FSMalternate
```

**Пример 3.13.** Три части конечного автомата

Вычисление следующего состояния и вычисление значения выходов часто объединяют в одном блоке always_comb. В этом случае вся комбинационная часть, относящаяся к автомату, оказывается в одном месте. Если объединить строки 10–18 примера 3.13 в один блок always_comb, то получится:

```
1   always_comb begin
2     z = 1'b1;
3     case (state)
4       A: nextState = (x) ? B : A;
5       B: nextState = (x) ? C : A;
6       C: begin
7            nextState = (x) ? C : B;
8            z = ~x;
9         end
10      default: nextState = A;
11    endcase
12  end
```

При таком подходе в ветви состояния C внутри пары begin-end находится вычисление и следующего состояния, и выходного значения. Поскольку для всех остальных состояний значением выхода z будет 1, то проще и понятнее вынести один оператор присваивания в самое начало блока always_comb (строка 2).

Если система задается автоматом Мили (в котором значения выходов зависят от состояния и значений входов), можно объединить вычисление выходных значений с вычислением следующего состояния (как в примере выше), но не следует включать вычисление выходных значений в always_ff, где происходит обновление состояния. В этом случае значение выходов будут пересчитываться только по фронту тактового сигнала, что не совсем корректно: в автомате Мили при изменении значений входов должны меняться значения выходов.

Если проектируется автомат Мура (в котором значения выходов зависят только от состояния), вычисление выходных значений может быть объединено с обновлением состояния.

# 3.4. Логическая оптимизация

В спецификацию конечного автомата входит описание вычисления выходных значений, вычисления следующего состояния, а также присваивания значений переменным состояния. На основе этой спецификации инструмент синтеза может создать много альтернативных вариантов реализаций, оптимизированных по разным критериям.

## 3.4.1. Кодирование состояний

В примере 3.12 мы использовали перечисление, чтобы задать множество констант – имен состояний. Такая мнемоника делает описание аппаратуры проще для создания и отладки. К тому же перечисления обеспечивают контроль типов.

В примере встречается определение переменной перечислимого типа:

```
1   enum {A, B, C} state;
```

где константы находятся внутри фигурных скобок, а имя декларируемой переменной (в общем случае их может быть несколько) находится в конце выражения. В данном случае конструкция enum определяет константы – элементы перечисления – как 32-битные целые числа (типа int), принимающие значения от 0 до 2. Однако в объявлении enum можно указать другой тип (см. раздел 2.6.3) и явно задать значения элементов перечисления, например:

```
2   enum logic [2:0] {A=3'b001, B=3'b010, C=3'b100} state;
```

Если подставить это перечисление в строку 5 примера 3.13, то получится унитарное кодирование (когда каждое состояние кодируется одним триггером и этот триггер активен, только когда автомат находится в этом состоянии). А можно определить перечисление так:

### 3.4.3. Так ли вам нужны несущественные элементы?

Различие между примерами 3.12 и 3.14 состоит в учете в целях оптимизации несущественных элементов множества состояний. В примере 3.12 в операторе case имеется ветвь default (строка 14), которая определяет поведение в том случае, когда текущее состояние не совпадает ни с одним из элементов перечисления: следующим будет состояние сброса. Таким образом, в коде реализации явно описано, что ситуация некорректного значения состояния приводит к переходу в состояние сброса. В примере 3.14, наоборот, ничего не говорится о том, каким будет следующее состояние, если текущее состояние не совпадает ни с одним из элементов перечисления, – следующее состояние должен определить оптимизатор, встроенный в инструмент синтеза. Вполне может оказаться, что это будет еще одно некоторое неопределенное состояние.

```
1  unique case (state)
2    A: state <= (x) ? B : A;
3    B: state <= (x) ? C : A;
4    C: state <= (x) ? C : B;
5  endcase
```

**Пример 3.14.** Оператор unique case для обновления состояния

В связи с этим возникает вопрос, когда и где следует использовать несущественные элементы. Обычно это решение принимает команда разработчиков. Когда выясняется, что некоторые ошибки проще обнаружить и обработать в другой части модели, несущественные элементы можно использовать в некоторых местах для оптимизации логики. В таких случаях в систему можно включить состояние ошибки или сигнализирующее об ошибочной ситуации выходное значение, которые указывают другим ее частям, что есть некая проблема.

## 3.5. Тестовые окружения для конечных автоматов

Тестовые окружения для конечных автоматов сложнее тестовых окружений для комбинационных схем. Они рассматриваются в главе 7.

Для студентов, выполняющих небольшие проекты, наиболее актуальны разделы 7.1 и 7.3, посвященные тестированию систем на основе конечных автоматов.

Инженерам-разработчикам, работающим над крупными проектами в группе верификации, рекомендуем прочитать также раздел 7.2.

Разница между двумя подходами заключается в том, что SystemVerilog включает конструкцию program для написания программ тестового окружения. Программы похожи на модули, но обладают средствами, полезными для верификации крупных систем. Студент, работающий над простым проектом, может обойтись модулями.

## 3.6. Задачи и упражнения

**3.1.** Напишите и промоделируйте в симуляторе 8-битный регистр с таким заголовком модуля:

```
3   enum logic [2:0] {A=3'b110, B=3'b101, C=3'b011} state;
```

т. е. используя метод инверсного унитарного кодирования (one-cold-one encoding). (Текущее состояние автомата идентифицируется нулевым значением триггера. Такое кодирование популярно у любителей пива.)

Для кодирования состояний можно использовать и привычный 2-битный код:

```
4   enum logic [1:0] {A=3'b11, B=3'b10, C=3'b01} state;
```

В целом такой подход к разработке описания полезен тем, что изменение всего одной строки кода дает совершенно разные реализации. В определении перечислимого типа содержится как разрядность, так и значения.

Обычно при перечислении имен состояний каждому из них сопоставляется определенное значение. Однако можно задать лишь некоторые из них и поручить компилятору задать остальные. Он возьмет последнее заданное значение, а следующему имени сопоставит инкрементированное значение. Отметим, что если значение, которое хочет выбрать компилятор, уже было назначено какому-то элементу перечисления, то произойдет ошибка компиляции.

В целях отладки можно вывести имена элементов перечисления на консоль. Вызовем процедуру `$monitor`:

```
5   $monitor ("At time %d the state changed to %s", $time, state.name);
```

На месте %s будут напечатаны имена элементов перечисления (а не значения), что задано с помощью `state.name`.

## 3.4.2. Комбинационные схемы для функций перехода и выхода

В разделе 2.2.4 был введен оператор `unique case`, который может, очевидно, пригодиться и здесь. Этот оператор означает, что ожидается выбор только лишь одной-единственной ветви. Рассмотрим оператор `case` из примера 3.13 (строки 11–16). Он скопирован ниже с заменой ключевого слова `case` на `unique case` в строке 1, а также из него была удалена ветвь `default`.

У `unique case` есть две особенности.

1. Поскольку при каждом исполнении данного оператора `case` происходит выбор единственной ветви, то в случае, когда по значению переменной состояния state не может быть выбрано ни одной ветви, должна произойти ошибка симуляции. Можно считать это утверждением, которое симулятор проверяет при каждом исполнении `unique case`.
2. Инструмент синтеза понимает, что в каждый момент времени одна из трех ветвей обязательно будет выбрана для исполнения. Неиспользуемые значения перечислимого типа будут рассматриваться как несущественные.

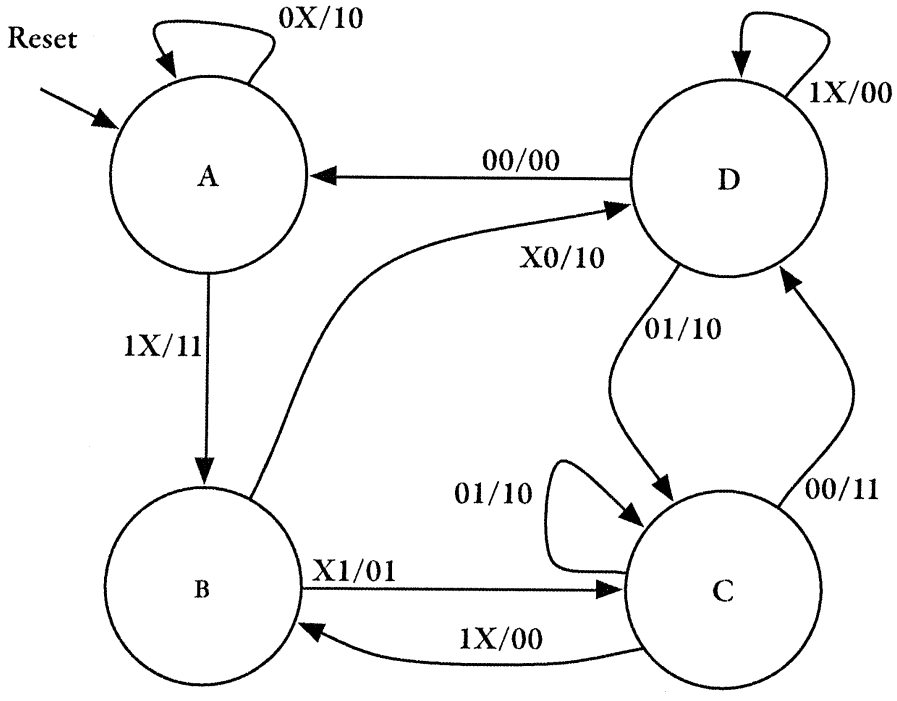
Таким образом, перечисление и оператор `unique case` дополняют друг друга при присваивании значений состояниям и передаче симулятору подсказок о том, как можно оптимизировать реализацию.

```
module regWithBenefits
  #(parameter W = 8)
  (input logic [W-1:0] d,
   input logic rstN, ck, clr, ld, shl, shIn,
   output logic [W-1:0] q);
```

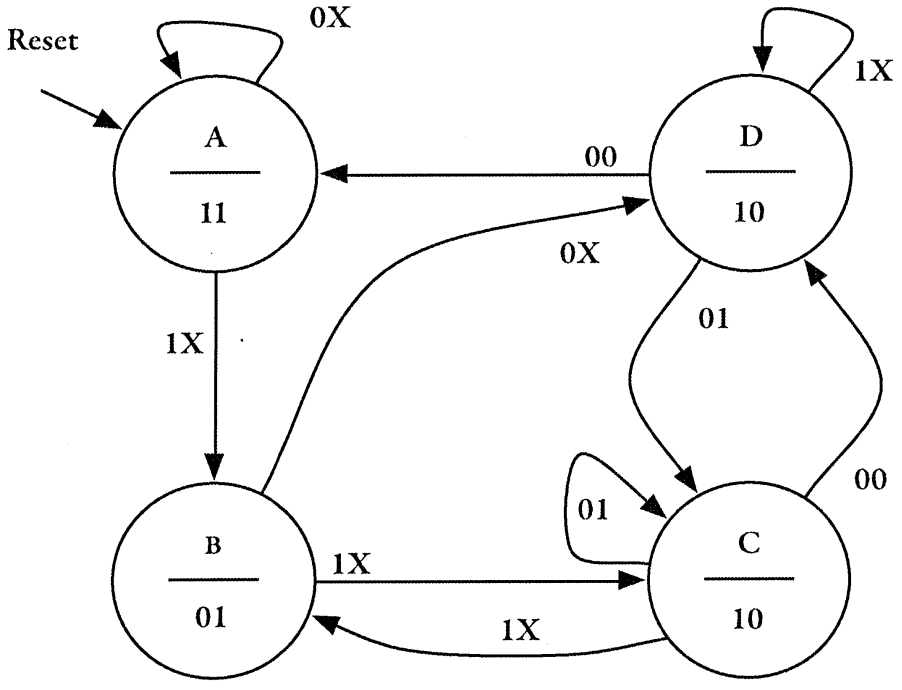
Здесь rstN – это сигнал асинхронного сброса, а clr – сигнал синхронного сброса. Работа происходит по переднему фронту тактового сигнала ck. По сигналу ld в регистр должно быть загружено значение d. shl сдвигает содержимое регистра влево на одну позицию, а самым младшим битом становится shIn. Если установлен более чем один управляющий сигнал, то предполагаются следующие приоритеты: clr, ld и shl.

**3.2.** Напишите на SystemVerilog модуль для конечного автомата, определенного следующей диаграммой состояний. Порядок входов: i и j. Порядок выходов: x и y. Для определения имен состояний воспользуйтесь перечислимым типом. Определите конечный автомат явным образом. Напишите для своего модуля тестовое окружение. Автомат тестового окружения задайте неявно. Заголовок модуля приведен ниже:

```
module FSMprob
  (input logic clk, rstN, i, j,
   output logic x, y);
```

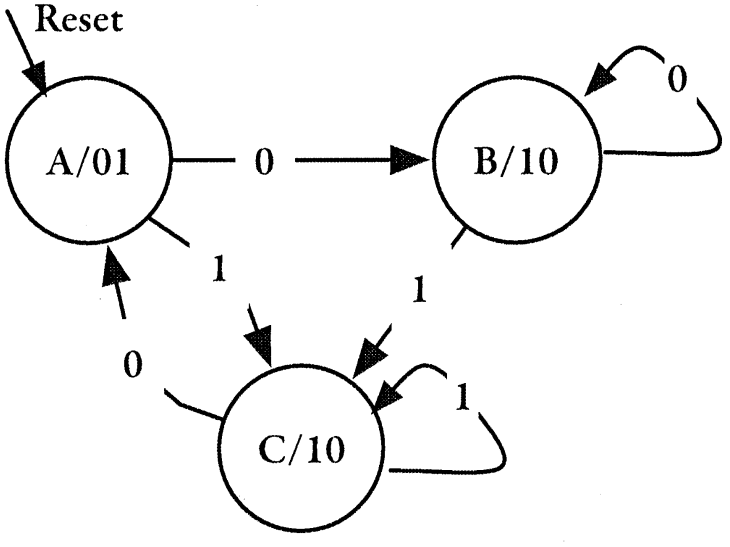



**3.3.** Сделайте то же самое для следующей диаграммы состояний.




**3.4 A.** В следующей диаграмме состояний имеются три состояния (A–C), один вход (x) и два выхода (yz). Задайте для этого конечного автомата числовые значения состояний с использованием кодирования на основе выходных значений. Запишите выражения для выходных значений и следующего состояния в виде сокращенной дизъюнктивной нормальной формы. Используйте несущественные комбинации, если таковые имеются. Затем напишите на SystemVerilog модель, соответствующую этой диаграмме состояний, с таким объявлением модуля:

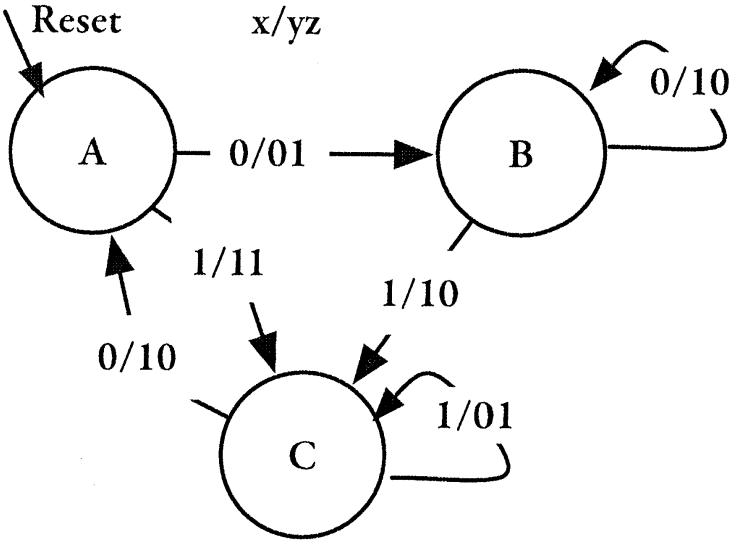
```
1   module p3_4
2     (input logic x, clk, reset,
3      output logic y, z);
```




**B.** Напишите тестовое окружение, используя неявный стиль описания конечного автомата. Необходимо обойти все допустимые состояния и пройтись по всем дугам. Чтобы показать, что модель работает, включите в тестовое окружение вывод на консоль текущего состояния вместе со значениями входов и выходов. Ваш модуль должен создать экземпляр модуля `p3_4` внутри модуля тестового окружения `testbench`.

**3.5 A.** В следующей диаграмме состояний три состояния (A–C), один вход (x) и два выхода (yz). Задайте для этого конечного автомата числовые значения состояний с использованием метода унитарного кодирования. Запишите выражения для выходных значений и следующего состояния в виде сокращенной дизъюнктивной нормальной формы. Используйте несущественные комбинации, если таковые имеются. Затем напишите на SystemVerilog модель, соответствующую этой диаграмме состояний, с таким заголовком модуля:

```
1  module p3_5
2    (input logic x, clk, reset,
3     output logic y, z);
```




**B.** Напишите тестовое окружение, используя неявный стиль описания конечного автомата. Необходимо обойти все допустимые состояния и пройтись по всем дугам. Чтобы показать, что модель работает, включите в тестовое окружение вывод на консоль текущего состояния вместе со значением входов и выходов. Ваш модуль должен создать экземпляр модуля p3_5 внутри модуля тестового окружения testbench.

# Глава 4

# Предположение о синхронности

В примерах моделей, рассмотренных ранее, никак не использовалось понятие времени (timing). Наиболее близкое к тому, что имеется в виду, – описание способа формирования тактового сигнала в тестовом окружении. Но даже там задержки задавались в произвольных единицах; время использовалось для продвижения модели вперед в процессе тестирования, а не для описания временны́х ограничений.

В современном процессе разработки ИС временны́е требования учитываются в оптимизациях, осуществляемых инструментами проектирования, и проверяются инструментами анализа. В этой главе описана методология проектирования синхронных систем, нацеленная на удовлетворение временны́х требований.

## 4.1. Основные предположения: доверяй, но проверяй

Проектирование синхронных цифровых систем основано на методологии, согласно которой один сигнал (обычно называемый *тактовым*) синхронизирует работу всей системы. Именно по фронту этого сигнала происходят все изменения состояния. *Предположение о синхронности системы* состоит в том, что цифровую систему можно описать на этом уровне абстракции и что такую систему можно физически реализовать.

Что понимается под «этим уровнем абстракции»? Представленный выше процесс проектирования на уровне регистровых передач основан на диаграммах состояний и булевых функциях: и то, и другое – абстракции. Например, мы постулировали, что все триггеры меняют свое состояние одновременно по фронту тактового сигнала – именно в этом смысл неблокирующего присваивания (<=), с помощью которого описывается обновление состояния. Это абстракция: в реальных электрических схемах состояние не меняется мгновенно. Имеются вариации в распространении электрических сигналов по проводникам внутри ИС, в т. ч. и тактового сигнала. Характеристики транзисторов, реализующих активную логику, также могут варьироваться, поэтому задержки распространения сигналов могут различаться. Наконец, не надо забывать

о температуре и других физических параметрах, влияющих на временны́е характеристики.

Методология проектирования синхронных систем требует, чтобы соблюдалось следующее условие: очередные значения битов состояния, распространяемые через комбинационные схемы, должны стабилизироваться и принять окончательные значения до их подачи на входы триггеров по фронту тактового сигнала.

Процесс проектирования включает несколько этапов (инструментов САПР), на которых строится модель физического уровня, позволяющая оценить временны́е характеристики. Имеется в виду следующее. По описанию на SystemVerilog инструмент синтеза создает модель из библиотечных элементов (зачастую это простые вентили и триггеры), реализующую заданную логику. Далее следует физическое проектирование: определяется положение каждого элемента ИС или ПЛИС – этот этап называется размещением; после чего задаются соединения между элементами – этот этап называется трассировкой (разводкой). Теперь мы знаем нагрузку каждого вентиля (число вентилей, соединенных с его выходом), а также сопротивление и емкость проводов, соединяющих вентили, – можно рассчитать временны́е характеристики.

Вопрос в том, сможет ли система работать на заданной тактовой частоте. Если нет, применяются оптимизации, повышающие производительность системы; после чего заново вычисляются временны́е характеристики. Итеративная процедура, нацеленная на удовлетворение временны́х требований, называется *временно́й оптимизацией* (*timing closure*).

Таким образом, мы описали модель, используя синхронную парадигму, и верим, что инструменты САПР позаботятся об удовлетворении временны́х требований, т. е. осуществят временну́ю оптимизацию. В этом разделе мы рассмотрим некоторые предположения, используемые при проектировании синхронных систем на языке типа SystemVerilog.

## 4.2. Предположения о временны́х характеристиках

Основной элемент памяти синхронных систем – D-триггер, работающий по фронту тактового сигнала. Какой фронт использовать, передний или задний, – деталь реализации, согласуемая между разработчиками.

### 4.2.1. Временные характеристики D-триггера

Временны́е характеристики D-триггера привязаны к фронту тактового сигнала и входу D. Они показаны на рис. 4.1. Основная проблема состоит в том, что тактовый сигнал и сигнал D не могут изменяться одновременно (хотя бы приблизительно). Как показано на рисунке, D должен оставаться стабильным (не изменяться) в течение некоторого времени до фронта тактового сигнала ($T_{setUp}$ – *время предустановки*) и некоторого времени после фронта ($T_{hold}$ – *время удержания*). Эти временны́е требования должны выполняться, иначе состояние

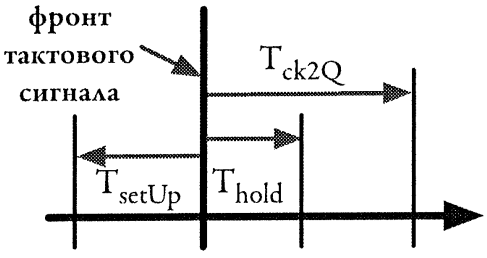



**Рис. 4.1.** Время предустановки и время удержания

триггера будет осциллировать; такая ситуация – когда в течение некоторого времени триггер не находится ни в одном из стабильных состояний – называется *метастабильностью*. На рисунке также показано время от фронта тактового сигнала до момента появления на выходе Q нового значения ($T_{ck2Q}$ – *время распространения*).

Теперь рассмотрим, как временны́е характеристики триггера проявляются в более крупной системе, показанной на рис. 4.2. В верхней части рисунка мы видим два триггера, FF-A и FF-B, между которыми расположена комбинационная схема. На входы комбинационной схемы поступают значения от FF-A и других триггеров системы (включая FF-B). Выходы схемы соединены с FF-B и другими триггерами. Для простоты мы показали только соединения с FF-A и FF-B.

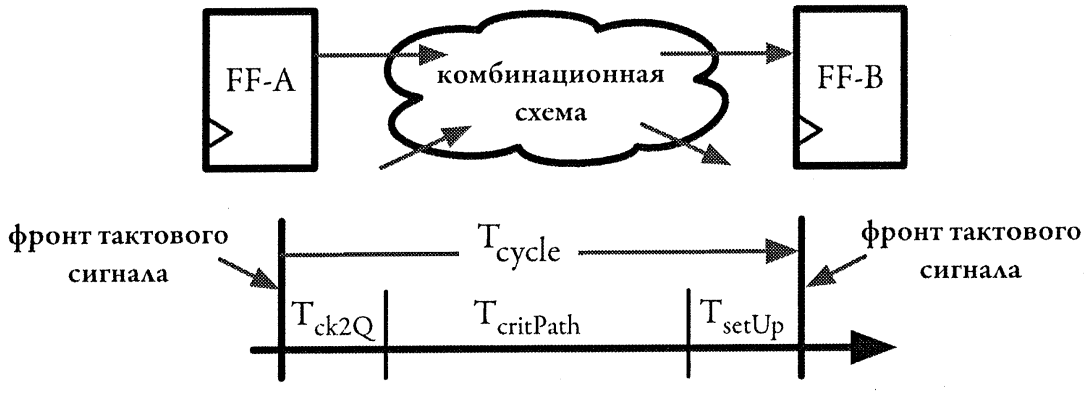



**Рис. 4.2.** Распространение значений в рамках одного такта

На рисунке в абстрактном виде отражены уже встречавшиеся нам элементы. Комбинационная схема описывается операторами `always_comb` и `assign`; их выходы подаются на входы D-триггеров. Триггеры моделируются запускаемыми по фронту неблокирующими присваиваниями, описанными в блоках `always_ff`. Нужно ответить на поставленный в начале главы вопрос: выполнены ли временны́е требования?

В нижней части рис. 4.2 на временно́й оси показаны события, происходящие между двумя соседними фронтами тактового сигнала. Этот промежуток времени называется *периодом тактового сигнала*; на рисунке он обозначен как $T_{cycle}$. Задержка на самом длинном пути в комбинационной схеме – *критическом пути* – обозначена как $T_{critPath}$. Чтобы система работала корректно, после возникновения фронта значения должны распространиться на выходы триггеров ($T_{ck2Q}$), затем – через комбинационную схему ($T_{critPath}$) на входы триггеров, при этом до возникновения следующего фронта должна пройти предустановка ($T_{setUp}$). Все эти времена показаны на рисунке. Таким образом, должно выполняться неравенство:

$$T_{cycle} > T_{ck2Q} + T_{critPath} + T_{setUp}.$$

Мы называем его *ограничением на период тактового сигнала* (*clock cycle timing relation*). Иначе говоря, для любого пути в комбинационной схеме, соединяющего пару триггеров, задержка (включая $T_{ck2Q}$ и $T_{setUp}$) должна быть меньше $T_{cycle}$. Если это не так, то может иметь место некорректная смена состояния и даже переход автомата в неопределенное состояние.

При разработке синхронных систем предполагается, что временна́я оптимизация обеспечит выполнение ограничения на период тактового сигнала. Пока временны́е требования не удовлетворены, комбинационная схема будет модифицироваться; цель модификаций – уменьшить длину критического пути.

## 4.2.2. Расфазировка тактового сигнала

В предыдущем разделе мы считали время между соседними фронтами тактового сигнала ($T_{cycle}$) одинаковым во всей модели. На самом деле это не так. Задержки на соединениях между логическими вентилями могут варьироваться из-за особенностей процесса производства; то же самое справедливо в отношении задержек на линиях тактовых сигналов.

На рис. 4.3 показано, что время прихода фронта тактового сигнала к разным триггерам может варьироваться. В верхней части показано формирование фронта тактовым генератором. Фронт распространяется до всех триггеров модели, но до одних он доходит раньше, а до других позже. На рисунке показано самое раннее и самое позднее время прибытия. Отметим, что тактовый сигнал один и тот же. Разница вызвана задержками распространения сигнала по проводам. Таким образом, разные триггеры срабатывают в разные моменты времени. Максимальная разница во времени (разность между самым поздним и самым ранним моментами времени) называется величиной *расфазировки тактового сигнала* (*clock skew*).

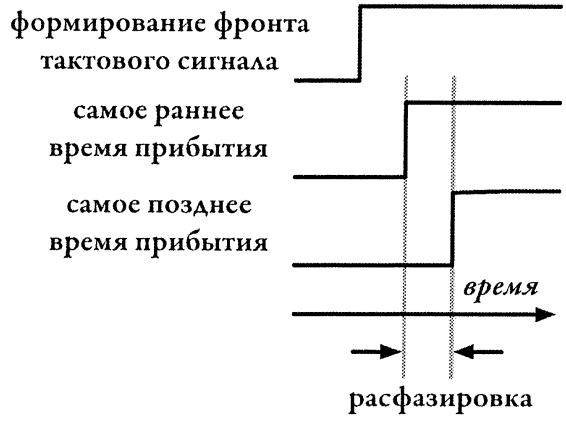



**Рис. 4.3.** Расфазировка тактового сигнала

Рассмотрим ситуацию, когда триггер с самым поздним временем прибытия фронта тактового сигнала, скажем FF-B, выдает значение, которое распространяется через комбинационную схему до триггера с самым ранним временем прибытия, скажем FF-A. Значение на выходе Q триггера FF-B выдается на время расфазировки позже, чем значение на выходе Q триггера FF-A. Соответственно, время, отводимое на вычисления, уменьшается.

Эта ситуация показана на рис. 4.4. По сравнению с рис. 4.2 здесь добавлена величина расфазировки ($T_{skew}$). Так как время распространения сигналов в триггере $T_{ck2Q}$ постоянно, как и время предустановки $T_{setUp}$, то время $T_{skew}$ можно «выкроить» только из допустимого времени прохождения сигнала по критическому пути.

Тот факт, что фронт тактового сигнала не прибывает одновременно ко всем триггерам, важен. Это означает, что даже если мы будем прилежно использо-

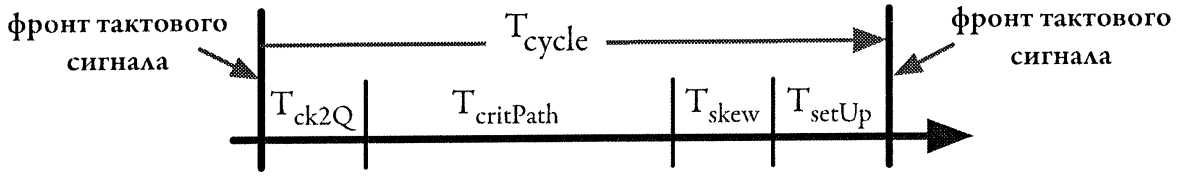



**Рис. 4.4.** Учет расфазировки в ограничении на период тактового сигнала

вать неблокирующие присваивания (<=) и симулятор будет обновлять триггеры в одно и то же время, в реальности все будет иначе.

С учетом расфазировки ограничение на период тактового сигнала выглядит следующим образом:

$$T_{cycle} > T_{ck2Q} + T_{critPath} + T_{setUp} + T_{skew}.$$

Это ограничение точнее представленного в разделе 4.2.1, поскольку учитывает не только характеристики триггеров, но и параметры распространения тактового сигнала.

Даже при расфазировке тактового сигнала можно положиться на временну́ю оптимизацию и продолжать проектирование, используя абстракции уровня регистровых передач.

## 4.2.3. Нарушение ограничения на время удержания

Нарушение ограничения на время удержания может вызываться расфазировкой тактового сигнала и короткими задержками в комбинационной схеме между триггерами. Рассмотрим систему на рис. 4.5: для триггеров FF-A и FF-B имеет место расфазировка; комбинационная схема между ними обладает собственной задержкой. В настоящий момент нас интересует наименьшее время прохождения сигнала через комбинационную схему, так называемая *задержка отклика* (*contamination delay*). Поскольку выход схемы соединен с входом D триггера FF-B, значение следующего состояния «загрязнено».

На рис. 4.6 проиллюстрировано, как эта ситуация может привести к нарушению удержания. Здесь показана временна́я диаграмма в окрестности фронта тактового сигнала. Из-за расфазировки этот фронт сначала прибывает на триг-

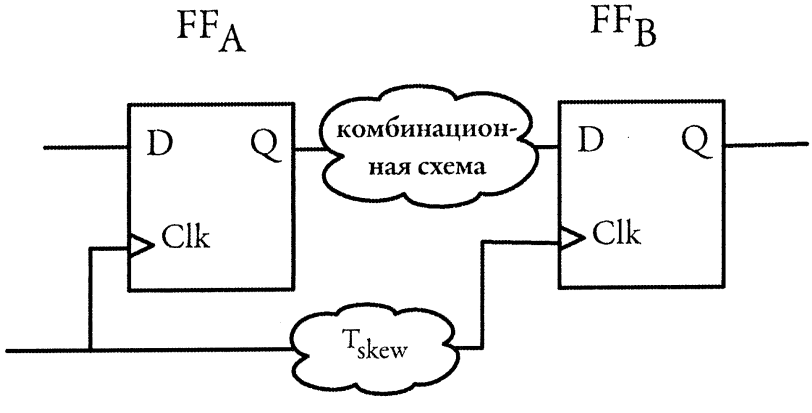



**Рис. 4.5.** Система с расфазировкой тактового сигнала

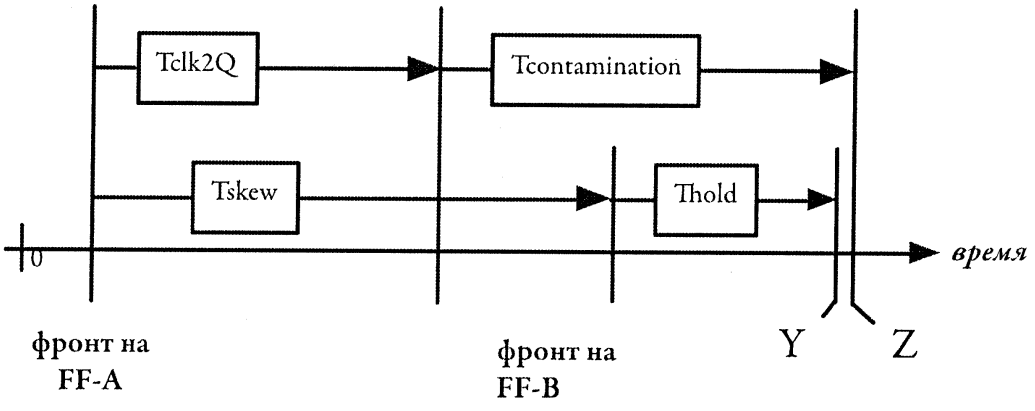



**Рис. 4.6.** Временна́я диаграмма, демонстрирующая нарушение удержания

гер FF-A (слева), а немного позже – на триггер FF-B (справа). После прибытия фронта вход D триггера FF-B должен оставаться постоянным в течение времени удержания; окончание этого промежутка обозначено на диаграмме буквой Y.

На триггер FF-A фронт прибыл раньше; значение его выхода Q изменится в момент $T_{ck2Q}$. Считая от этого момента, вход D триггера FF-B изменится не раньше, чем через время $T_{contamination}$; этот момент обозначен на диаграмме буквой Z. В том виде, как показано на рисунке, система будет работать правильно – значение входа FF-B не меняется в течение времени удержания (Z > Y). Однако если бы время $T_{skew}$ оказалось больше, момент Y переместился бы правее Z, и вход D триггера FF-B изменился бы до истечения времени удержания. Вот вам и нарушение.

## 4.3. Домены синхронизации

Во всех системах, которые мы рассматривали до сих пор, предполагалось наличие одного тактового генератора. Для всех элементов таких систем должно выполняться ограничение на период тактового сигнала. Назовем систему с одним тактовым сигналом, внутри которой это ограничение соблюдается, *доменом синхронизации* (*clock domain*). Домен синхронизации включает:

- триггеры, синхронизируемые одним и тем же тактовым сигналом (для которых соблюдается ограничение на период тактового сигнала);
- выходы комбинационных схем (Z на рис. 3.6), входы которых поступают исключительно с этих триггеров.

Таким образом, домен синхронизации охватывает состояние, обновляемое по общему тактовому сигналу, и выходы, порождаемые этим состоянием (и только им). Отметим, что именно так определен домен C на рис. 3.6.

### 4.3.1. Что ограничивает размер домена синхронизации?

Рассмотрим временну́ю диаграмму на рис. 4.4. $T_{ck2Q}$ и $T_{setUp}$ – небольшие константы, определяемые реализацией используемых триггеров. Для простоты будем считать, что время $T_{critPath}$ прохождения сигнала через комбинационную схему постоянно. Это вполне разумное предположение, если считать, что си-

стема строится из большого числа конечных автоматов[31]. Для вычисления состояний и значений выходов нужно некоторое время, но это время практически не меняется при добавлении новых автоматов и соединении некоторых их выходов со входами других автоматов.

Если при добавлении новых автоматов время $T_{critPath}$ сохраняется, то время $T_{skew}$ увеличивается, поскольку триггеры все дальше разносятся друг от друга. В какой-то момент величина расфазировки $T_{skew}$ начинает играть такую же роль, что и $T_{critPath}$. Это приводит к тому, что укрупнение системы становится возможным только при уменьшении тактовой частоты. То есть чем больше система, тем меньше выигрыш от ее усложнения. В такой ситуации имеет смысл использовать несколько доменов синхронизации: в каждом из них вычисления производятся на меньшей физической площади, что уменьшает расфазировку и позволяет увеличить тактовую частоту. Оборотная сторона нескольких доменов – необходимость их синхронизации, о чем мы поговорим в следующем разделе.

## 4.3.2. Междоменные сигналы

Как быть с сигналами, не входящими в домен синхронизации? Предположим, кто-то нажимает кнопку, которая формирует сигнал, подаваемый на вход комбинационной схемы; этот кто-то не является частью домена синхронизации. Вы пытались когда-нибудь нажать кнопку, скажем, в лифте синхронно с тактовым генератором системы, для которой сигнал от кнопки является входом? (Нет, конечно! Вы даже не знаете, на какой частоте работает система, да и в любом случае не смогли бы синхронизировать свой палец с генератором, работающим на частоте в несколько мегагерц!)

Сигналы, поступающие извне домена синхронизации, являются *асинхронными* по отношению к этому домену. Их значения изменяются по фронту какого-то тактового сигнала, но в текущем домене этот сигнал тактовым не является. В этом случае нет смысла проверять величину периода тактового сигнала и оптимизировать временны́е характеристики.

Представьте границу домена синхронизации: все триггеры внутри границы управляются одним тактовым сигналом; триггеры за пределами границы – другими. Например, триггер FF-X со своим выходным сигналом (см. рис. 4.7) асинхронен по отношению к показанному домену – он управляется другим тактовым генератором. Суть проблемы состоит в следующем. Асинхронный входной сигнал может изменяться в любое время, и может так получиться, что значение следующего состояния будет устанавливаться на входе триггера FF-B, в то время когда требуется, чтобы оно было стабильным[32]. Из-за рас-

---

[31] Предполагается, что конечные автоматы, используемые для построения системы, имеют сопоставимую сложность, в том смысле, что длины критических путей в их схемах примерно одинаковы. Таким образом, можно описать сколь угодно сложное поведение, сохраняя время $T_{critPath}$.

[32] То есть во время установки или удержания.


**Рис. 4.7.** Граница домена синхронизации

согласования временны́х характеристик система может легко перейти в некорректное состояние.

Частичное решение этой проблемы – поставить *синхронизаторы* на все входные сигналы, приходящие из других доменов, как показано на рис. 4.8. Здесь триггеры FF-A, FF-B и FF-Xsync управляются одним и тем же тактовым сигналом, а триггер FF-X расположен за границей домена синхронизации. Как только асинхронный входной сигнал загружается в триггер FF-Xsync по фронту тактового сигнала домена синхронизации, он становится частью домена и учитывается в расчете критического пути и временны́х характеристик.

При таком подходе все равно остается проблема – входной сигнал домена синхронизации (от FF-X к FF-Xsync) может поступить одновременно с фронтом тактового сигнала FF-Xsync, что приведет к осцилляции триггера-синхронизатора. В этом случае для стабилизации значения понадобится время, большее $T_{ck2Q}$. Для устранения такой возможности применяют разные приемы: ставят два триггера-синхронизатора или используют триггеры, которые не так чувствительны к нарушению времен предустановки и удержания.

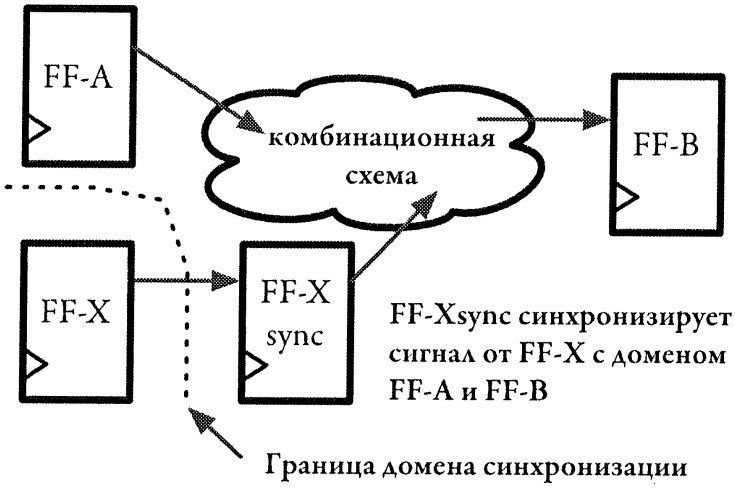



**Рис. 4.8.** Использование триггера-синхронизатора

## 4.4. Логическая оптимизация: коррекция временны́х характеристик

Учитывая, что единственным временны́м параметром, передаваемым инструменту синтеза, является период тактового сигнала, возникает вопрос: обладает ли разработчик средствами контроля над логической оптимизацией. Ответ утвердительный. Некоторые из этих средств мы уже рассмотрели: распределение состояний и оптимизация несущественных комбинаций. Еще один вид оптимизации называется *коррекцией временны́х параметров* (*retiming*).

*Коррекция временны́х параметров* позволяет перенести часть логики через границу такта, сделав ее выходной. Рассмотрим схему на рис. 4.9. Это стандартное представление конечного автомата, в котором есть схема определения следующего состояния, а выход (Z) напрямую формируется одним из триггеров. Комбинационная схема, вычисляющая Z, показана в нижней части рисунка: у нее два входа (i и j), поступающих от комбинационной схемы сверху. Предположим, что при проверке задержки критического пути выяснилось, что сигнал на вход D триггера FF-B поступает слишком поздно – нарушается ограничение на время предустановки.

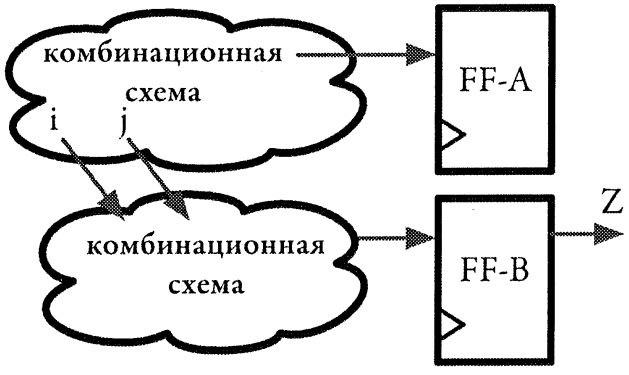



**Рис. 4.9.** Схема вычисления следующего состояния автомата и его выхода

Коррекция временны́х параметров позволяет разработчику вынести часть логики вычисления Z за триггер. Это показано на рис. 4.10. Что мы сделали? Два значения (i и j), первоначально подаваемых в нижнюю комбинационную схему для вычисления Z, теперь хранятся в триггерах FF-B и FF-C (был добавлен еще один триггер). Выходы обоих триггеров подаются на комбинационную схему вычисления Z. Сама схема не изменилась. Отличие от исходного варианта состоит в том, что Z формируется чуть позже относительно фронта тактового сигнала. Говоря точнее, появилась комбинационная задержка на выходе: значение Z появляется не через время $T_{ck2Q}$ после фронта, а через время $T_{ck2Q}+T_{outputComb}$.

Вы как разработчик должны решить, допустимо ли это. Например, такой подход не годится, если значение Z используется для вычисления следующего состояния в петле обратной связи! Если же Z является выходом (и только вы-

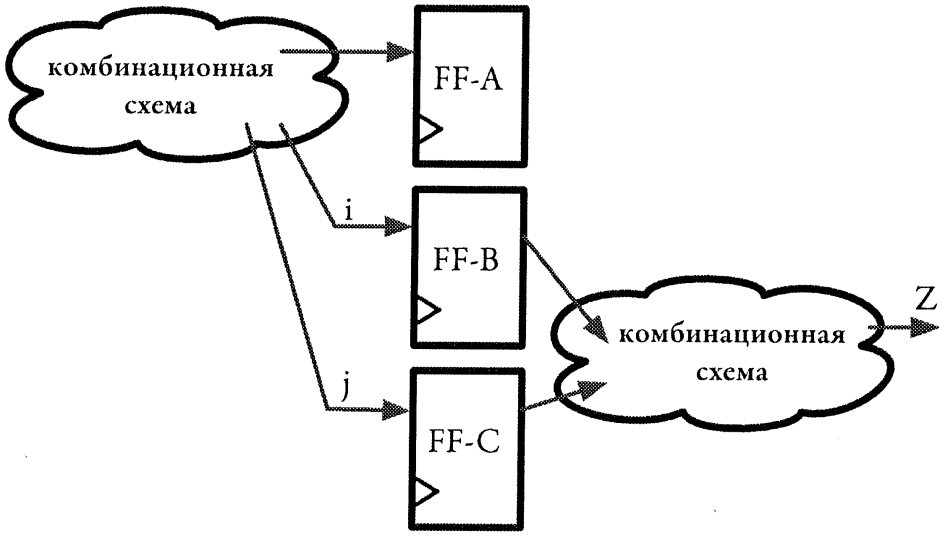



**Рис. 4.10.** Модель после коррекции временны́х параметров

ходом), все может сложиться удачно. В модель, помимо прочего, был добавлен триггер. Для хранения промежуточных значений может потребоваться несколько триггеров. Не займут ли они слишком много места? Это тоже вопрос к разработчику. В заключение отметим, что некоторые инструменты синтеза могут выполнять оптимизацию типа описанной выше.

## 4.5. Правила проектирования синхронных систем

Проектирование цифровых систем – весьма сложное дело. Чтобы гарантировать правильность синхронизации в крупной системе, работающей на высокой частоте, нужно придерживаться определенных правил. Эти правила были разработаны, чтобы упростить временну́ю оптимизацию и лучше координировать работу разработчиков в большом коллективе.

Вот эти правила:

1) в пределах домена синхронизации все изменения состояния происходят по одному фронту одного тактового сигнала. По определению;
2) в рамках одного такта состояние изменяется только один раз. Неоднократное изменение нарушает правило 1;
3) сигналы, поступающие извне домена, перед использованием должны быть синхронизированы с локальным тактовым генератором. Только в этом случае можно проводить временно́й анализ;
4) сигналы сброса триггеров не используются для смены состояния. Они служат для приведения системы в начальное состояние (состояние после включения питания) и не являются частью нормальной процедуры смены состояния;
5) значения тактовых сигналов не используются в логических схемах, разве что в буферах для их усиления.

Основная мотивация для первых четырех правил – возможность проверки ограничения тактового интервала. Состояние и входы должны изменяться

только по фронту тактового сигнала – синхронно. Последнее правило иногда нарушается, например при проектировании маломощных устройств: в самом деле, один из способов уменьшить энергопотребление системы – перевести ее в спящий режим, т. е. отключить подачу тактового сигнала в большинство подсистем!

Будет полезным рассмотреть и другие причины, по которым правило 4 включено в этот список. Правило 4 запрещает производить изменения состояния по изменению значения сигнала, не являющегося тактовым. Рассмотрим схему счетчика от 0 до 5 на рис. 4.11. В этом примере имеется 3-битный регистр состояния ({C, B, A}), а логика определения следующего состояния реализуется функцией

```
nextState = (current_state + 1) mod 6
```

так что следующее состояние всегда совпадает со следующим значением счетчика. При таком подходе можно легко проверить ограничение на период тактового сигнала.

Во второй схеме (см. рис. 4.12) показан счетчик со вспомогательным вентилем NAND, определяющим, что счетчик досчитал до 6 и потому должен быть сброшен в 0; сброс состояния осуществляется асинхронным сигналом `resetFF_L`. Изъян этой модели состоит в том, что изменение состояния с 6 на 0 управляется данными (а не тактовым сигналом): тот факт, что B и C равны 1, служит основанием для изменения состояния (путем сброса триггеров). Чтобы понять, в чем потенциальная проблема, рассмотрим ситуацию, когда на выходе вентиля NAND возможны импульсные помехи (glitches). При изменении значения счетчика с 3 на 4 бит B выключается, а C включается. Если C включится раньше, чем B выключится (из-за разного времени распространения сигналов, задержек в проводах и $T_{ck2Q}$), на выходе вентиля NAND может возникнуть помеха достаточно большая, чтобы вызвать сброс системы (возможно, она

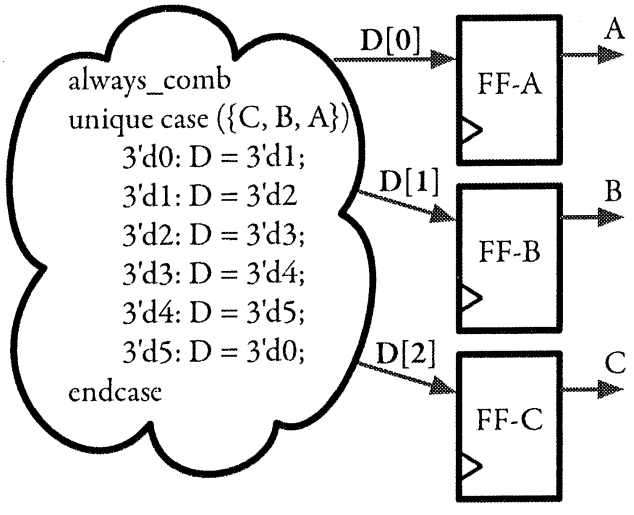



**Рис. 4.11.** Счетчик от 0 до 5

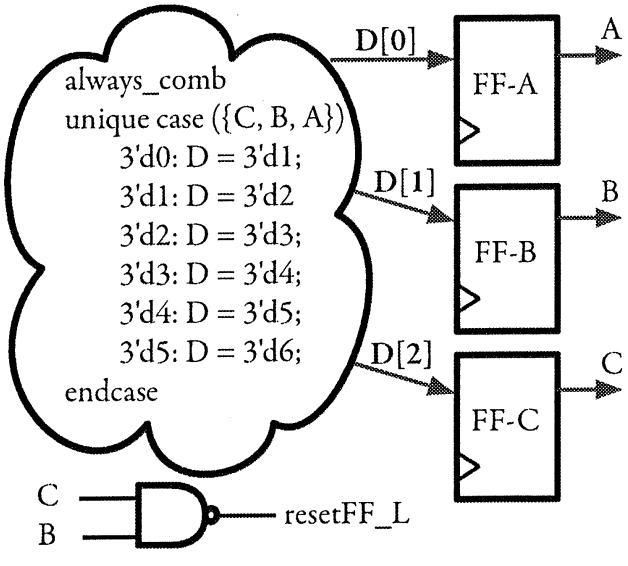



**Рис. 4.12.** Некорректная модель

приведет к сбросу лишь части триггеров). Так что при определенных условиях такая схема может работать некорректно.

Не делайте так!

У этой модели есть еще один изъян: состояние изменяется дважды в течение одного такта, что нарушает правило 2. При проверке ограничения на период тактового сигнала нужно учитывать это дополнительное изменение.

Правило 5 касается размещения на линии тактового сигнала логики, отвечающей за *стробирование тактового сигнала* (*clock gating*). На рис. 4.13 приведена одна из проблем, возникающих в этом случае. Здесь нормальный тактовый сигнал подается на вход вентиля AND; другой вход этого вентиля (`stop_ck_L`), будучи установлен, останавливает тактовый сигнал. Рассмотрим логическую схему и временну́ю диаграмму, показанные на рисунке. Если на входе `stop_ck_L` возникнет импульсная помеха, она может перейти на выход `gated_clock`. Проблема в том, что триггер сработает дважды за такт. Точнее, в него может быть загружено некорректное значение следующего состояния (поскольку оно не стабилизировалось); это значение распространится через логику определения следующего состояния, приводя к неправильному поведению.

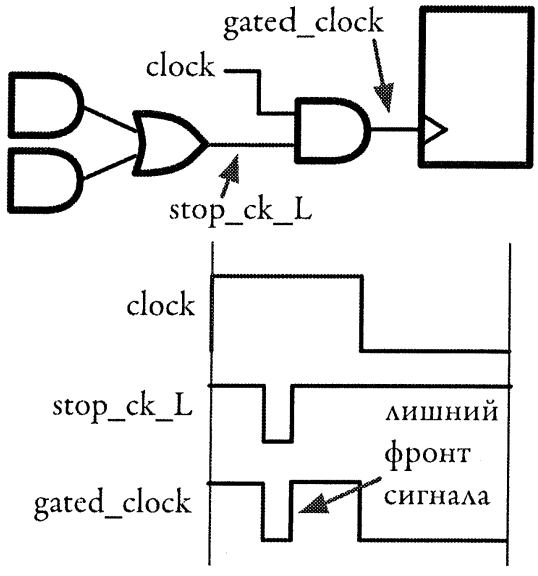



**Рис. 4.13.** Стробирование тактового сигнала

Стробирование тактового сигнала применяется как метод энергосбережения. При таком подходе нужно аккуратно проектировать схему, чтобы избежать импульсных помех. Для этого сигнал `stop_ck_L` часто делают выходом регистра состояния.

Отметим, что проблемы, описанные в этом разделе, вызываются импульсными помехами в комбинационных схемах. Такое случается постоянно и не является причиной для беспокойства – в конечном итоге выходы принимают стационарные значения. Однако если существует возможность, что помеха вызовет изменение состояния триггера, могут возникнуть серьезные проблемы. Неудивительно, что в синхронных системах линии тактового сигнала и сигнала сброса проектируются так, чтобы помех не было. Это специальные сигналы, которым на всем протяжении всего процесса проектирования уделяется особое внимание.

Не удивляйте своих коллег, нарушая эти правила. Выискивание таких нарушений может повлечь срыв сроков выпуска продукта, а упущенный в результате задержки проекта доход может стоить вам работы.

# Часть II

# АППАРАТНЫЕ ПОТОКИ

# Глава 5

# Аппаратные потоки (конечные автоматы с трактом данных)

Иногда конечные автоматы используются сами по себе, но чаще соединяются с трактом данных, обрабатывающим информацию. *Трактом данных* называется часть цифровой системы, реализующая вычисления. Значения переменных хранятся в регистрах и в памяти; после их чтения они преобразуются комбинационными схемами тракта данных, например АЛУ, а результаты загружаются в регистры по следующему фронту тактового сигнала. Такая пересылка информации называется *регистровой передачей*. Последовательностью регистровых передач, совершаемых трактом данных, управляет конечный автомат (*устройство управления*). Вместе их часто называют *конечным автоматом с трактом данных* (*FSM with datapath – FSM-D*), или *аппаратным потоком*. В этой книге термины «конечный автомат с трактом данных», «аппаратный поток» и просто «поток» считаются синонимами. Если вдруг нам потребуется упомянуть о программных потоках, мы отметим, что потоки программные.

## 5.1. Аппаратные потоки

Аппаратный поток описывает повторяющуюся последовательность вычислительных шагов. Он синхронизируется с остальной частью системы для получения входных данных, их обработки и записи результатов. В потоке описаны логика управления и логика тракта данных; имеются входы и выходы, связывающие его с внешним миром, будь то другой поток, физический датчик или привод.

Аппаратные потоки похожи на программные потоки, знакомые нам по языкам параллельного программирования; основное отличие состоит в том, что аппаратные потоки всегда активны. Напомним, что в языках описания аппаратуры, таких как SystemVerilog, блоки `always` и присваивания `assign` определяют схемотехнические элементы (вентили, провода, транзисторы), которые занимают место в ИС или ПЛИС и постоянно функционируют. Аналогично обстоит дело с аппаратными

потоками: переход, пусть даже в состояние, совпадающее с текущим, производится по каждому фронту тактового сигнала. Программные же потоки, в отличие от аппаратных, активируются, только когда готовы входные данные и когда операционная система выделяет процессорное время для их исполнения. Аппаратные потоки имеют свою собственную логику, управляющую их исполнением.

В следующем разделе мы реализуем поток с простой функциональностью.

## 5.1.1. Иллюстративный пример

Спроектируем аппаратный поток, вычисляющий сумму чисел, подаваемых на вход на нескольких последовательных тактах. Поток должен знать, когда выполнять суммирование, т. е. когда на входе находятся значимые (valid) данные. После завершения суммирования он должен сообщить о том, что результат вычислен и другой поток может его получить.

На рис. 5.1 поток изображен абстрактно, в виде облака (детали будут добавлены ниже). Внутреннее устройство скрыто, но входные и выходные порты определены:

- `inA` – входной 16-битный вектор, используемый для подачи чисел, подлежащих суммированию. В процессе вычисления значения появляются на каждом фронте тактового сигнала;
- `go_l` – входной 1-битный сигнал, сообщающий о начале вычисления. В момент установки `go_l` на входе `inA` находится первое число суммируемой последовательности;
- `sum` – выходной 16-битный вектор, используемый для выдачи вычисленной частичной суммы. Если вычисление начато и значение входа `inA` равно 0, значит, последовательность закончилась;
- `done` – выходной 1-битный сигнал, сообщающий другим частям системы, что сумма вычислена и результат можно сохранить. В момент установки `done` на выходе `sum` выдается сумма всех чисел, поданных на вход `inA` с момента установки `go_l`; на входе `inA` в этот момент находится 0 – индикатор конца последовательности;
- `reset_l` и `ck` – специальные входные сигналы: первый сбрасывает поток в состояние ожидания `go_l`; второй переводит поток в следующее состояние.

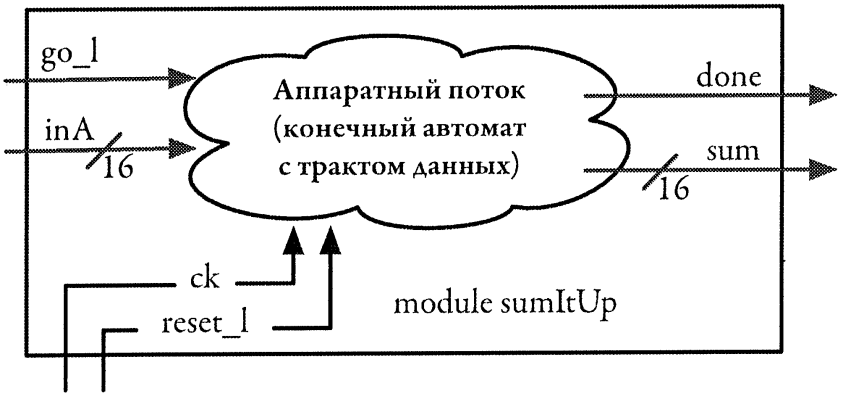



**Рис. 5.1.** Абстрактное представление аппаратного потока

Это очень простой пример, который тем не менее иллюстрирует основные идеи, связанные с аппаратными потоками. Поток можно рассматривать как реализацию повторяющейся последовательности вычислительных шагов (состояний). В нашем примере процесс реализует последовательное суммирование чисел, подаваемых на вход. Определен *протокол* – соглашение о том, как процесс узнает, что пора начать работу, и как он сообщает другим частям системы, что результат получен. В нашем примере для этого используются сигналы `go_l` и `done`.

Поток можно также рассматривать как модуль, предоставляющий услугу (сервис) для остальной части системы. В данном случае услугой является суммирование чисел: вычисление начинается по запросу (установка `go_l`) и завершается, когда на вход подается 0. В общем случае установка сигналов вроде `go_l` и `done` может трактоваться по-разному: запуск действия (как в данном случае) или запрос на обработку; завершение услуги (как в данном случае) или готовность к вычислениям. Вкладываемая семантика определяется разработчиком – как ему удобнее.

## 5.1.2. Временна́я диаграмма работы потока

На рис. 5.2 показана временна́я диаграмма работы системы, дающая более детальное представление о ней. Мы видим, как посылается потоку один пакет данных: передача начинается со значения 7 на входе `inA` (в момент, когда установлен сигнал `go_l`), а завершается значением 0 (в этот момент вычисленное значение `sum` равно 10).

Активным является передний фронт тактового сигнала; моменты его прихода обозначены на диаграмме буквами от A до E. Видно, что `go_l` устанавливается по фронту B; в этот момент на входе `inA` находится значение 7. Это значение

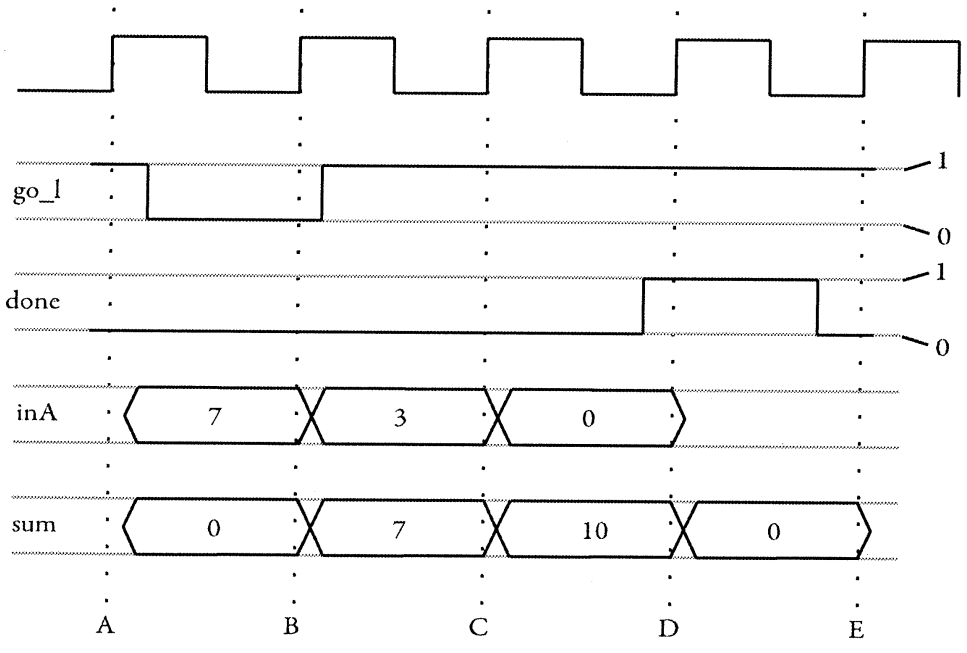



**Рис. 5.2.** Временна́я диаграмма работы потока

прибавляется к значению регистра результата (выход sum), в котором сейчас находится 0; после прихода фронта значение регистра становится равным 7. Это типичная временна́я диаграмма: входные значения присутствуют к моменту прихода фронта тактового сигнала; состояние потока, например значение sum, изменяется только после фронта. Предполагается, что входные данные генерируются системой с тем же тактовым сигналом; время, требуемое для появления новых значений после прохождения фронта, на диаграмме несколько преувеличено.

В момент прихода следующего фронта (C) значение входа inA равно 3, а значение выхода sum – 7. После прохождения фронта сумма изменяется на 10. Согласно спецификации, система продолжает работать, пока на входе не появится 0. Когда это происходит, устанавливается сигнал done (фронт D), а на следующем такте выход sum сбрасывается в 0 (поток готов к суммированию входных чисел). Если какая-то часть системы (скажем, другой поток) хочет использовать результат суммирования, она должна наблюдать за сигналом done, используя его как индикатор для копирования значения sum в свой собственный регистр.

Показанная на диаграмме суммируемая последовательность состоит всего из двух значений. Если бы она была длиннее, в вычислении появились бы дополнительные изменения состояния, аналогичные тому, что происходит по фронту C.

Отметим, что в начале и в конце вычисления значение выхода sum равно 0 – он очищается при установке done и по сигналу сброса. Отметим также, что go_l можно установить повторно по фронту E – к этому моменту исполнено последнее действие предыдущего вычисления (очистка sum).

## 5.1.3. Тракт данных потока

Заполним деталями облако на рис. 5.1; начнем с тракта данных. Тракт данных реализует регистровые передачи, используемые потоком в вычислениях. *Регистровая передача* – это базовый вычислительный шаг, состоящий из считывания значений регистров, их подачи на комбинационную схему (например, сумматор) и установки полученных значений на входы регистров. По фронту тактового сигнала проверяется, разрешена ли запись в регистр: если да, результат регистровой передачи записывается. В рассматриваемом примере можно выделить две регистровые передачи:

```
sum = 0 и
sum = inA + sum.
```

Действительно, вычисление в том и состоит, чтобы сначала загрузить в sum 0, а затем по каждому фронту прибавлять к sum значение входа inA.

На рис. 5.3 показан тракт данных потока из нашего примера. Для хранения суммы используется регистр sum. Выход регистра – это один из выходов потока и один из входов сумматора (другим входом является inA). Выход сумматора (addOut) подается на вход регистра sum. Если запись в регистр разрешена, по сле-

дующему фронту в регистр загружается сумма его текущего значения и значения входа. Эта операция реализует требуемое вычисление `sum = inA + sum`.

Как разрешить запись в регистр? Установив сигнал `ld_l` или `cl_l`: в первом случае регистр сохранит значение своего входа; во втором сбросит значение в 0.

Рассмотрим фрагменты описания тракта данных на SystemVerilog. Вот код, относящийся к сумматору:

```
1   assign addOut = inA + sum,
```

Имена, используемые в операторе `assign`, взяты прямо из рис. 5.3. Ниже описана логика работы регистра `sum`:

```
1   always_ff @(posedge ck, negedge reset_l)
2     begin: reg_sum
3       if (~reset_l) sum <= 0;
4       else if (~ld_l) sum <= addOut;
5       else if (~cl_l) sum <= 0;
6     end: reg_sum
```

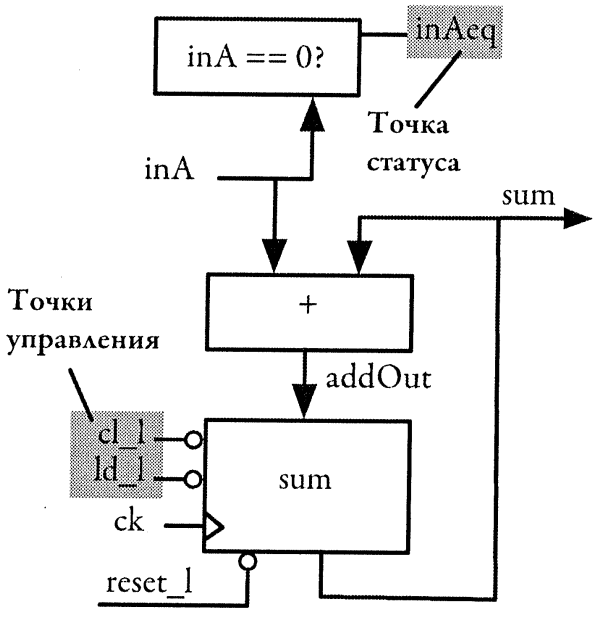



**Рис. 5.3.** Тракт данных аппаратного потока

Можно видеть, как `ld_l` и `cl_l` используются для загрузки или очистки регистра `sum`. Эти сигналы называются *точками управления* (*control points*). Если установлен один из них, исполняется соответствующее действие. Если установлены оба, в регистр загружается входное значение – условие на `ld_l` проверяется первым (строка 4). Если не установлен ни один из этих сигналов, ничего не произойдет – значение в регистре `sum` не изменится. Разумеется, если установлен `reset_l`, регистр сразу обнуляется.

В этом тракте данных есть только две точки управления, но в более сложных могут быть и другие, например выбор функции АЛУ или порта мультиплексора. Важно, что все они соединены с конечным автоматом, управляющим порядком осуществления регистровых передач (в определенных состояниях автомата должны происходить определенные регистровые передачи).

Также в тракте данных проверяется, равно ли значение входа нулю. Это показано в верхней части рис. 5.3. Здесь `inA` подается на вход комбинационной схемы, осуществляющей сравнение. Если значение равно 0, устанавливается сигнал `inAeq`, называемый *точкой статуса* (*status point*). Точки статуса, как правило, являются выходами комбинационных схем и сообщают ту или иную информацию о ходе вычисления (в данном случае – равно значение `inA` нулю или нет). Используя эту информацию, управляющий вычислением автомат меняет состояние, инициирует регистровые передачи и устанавливает значения выходов. В нашем примере если значение `inA` равно 0, значит, вычисление нужно завершать – это важное для потока решение.

И последнее замечание относительно фрагмента кода для регистра sum. В строке 3 описан *асинхронный сброс*, а в строке 5 – *синхронная очистка*. Хотя конечное значение в sum одно и то же – 0, попадает оно туда по-разному. Асинхронный сброс происходит всякий раз при установке сигнала сброса; нулевое значение остается в регистре до тех пор, пока сигнал сброса установлен, и по меньшей мере до очередного фронта тактового сигнала. Синхронная очистка происходит по фронту тактового сигнала (в предположении, что установлен только сигнал cl_l). Эта операция осуществляется синхронно с прочими обновлениями состояния. Сигнал сброса не должен использоваться для управления вычислением. У него специальная функция – перевод системы в то состояние, в котором она находилась после включения питания (см. раздел 3.2).

## 5.1.4. Диаграмма состояний

На рис. 5.4 показана диаграмма состояний потока. В ней два состояния, которые можно определить так: sA – ожидание установки go_l; sB – ожидание появления 0 на входе inA. Начальным состоянием (состоянием сброса) является sA. Отметим, что входами автомата, показанного на диаграмме, являются точка статуса inAeq и вход go_l; выходами – точки управления ld_l и cl_l, а также выход done.

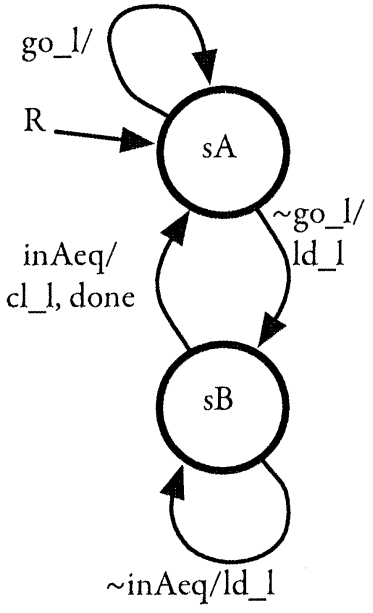



**Рис. 5.4.** Диаграмма состояний

Если в состоянии sA установлен сигнал go_l, поток устанавливает ld_l, что приводит к загрузке в регистр sum первого числа последовательности, и переходит в состояние sB. Эти действия происходят по фронту B (см. рис. 5.2). Если в состоянии sB значение сигнала inAeq равно FALSE, в регистр sum загружается выходное значение сумматора (сигнал ld_l все еще установлен). Наконец, если значение входа inA равно 0 (значение сигнала inAeq равно TRUE), поток возвращается в состояние sA: регистр sum очищается (путем установки cl_l); устанавливается сигнал done. Эти действия происходят по фронту D (см. рис. 5.2).

Сделаем несколько замечаний о том, как интерпретировать диаграмму состояний. Мы придерживаемся следующих соглашений о пометке дуг:

- если входной сигнал не указан (перед косой чертой), значит, его значение несущественно;
- если выходной сигнал указан (после косой черты), он устанавливается; если нет – не устанавливается.

Так, в состоянии sA значение сигнала inAeq несущественно – соответствующей пометки нет ни на одной из дуг, исходящих из sA. Кроме того, в состоянии sA сигналы cl_l и done никогда не устанавливаются; ld_l устанавливается (получает значение 0), но только если установлен go_l.

Пометка go_l на петле в состоянии sA говорит о том, что если значение go_l равно 1, то состояние не меняется (следующим состоянием будет sA). Пометка ~go_l/ld_l на другой дуге говорит о том, что если сигнал go_l установлен (значение go_l равно 0 или, что то же самое, значение ~go_l равно 1), то ld_l должен быть установлен (обнулен). Это описано во втором пункте списка выше.

### 5.1.5. Совмещение конечного автомата и тракта данных

На рис. 5.5 показана вся система (названная sumItUp): тракт данных и реализация конечного автомата в виде логической схемы (ранее не приводилась). Чтобы диаграмма легче читалась, некоторые одноименные сигналы не были соединены линиями. Пунктирная линия вокруг конечного автомата и тракта данных очерчивает содержимое облака на рис. 5.1 и выделяет входы и выходы потока – они станут портами модуля в SystemVerilog-описании.

Внимательно посмотрев на реализацию потока внутри пунктирной линии, мы видим, что конечный автомат находится слева, а тракт данных – справа. Выходы автомата связаны с точками управления (cl_l и ld_l) и системным выходом (done). На вход автомату подаются точка статуса (inAeq) и системный вход (go_l). Автомат управляет последовательностью регистровых передач – он реагирует на значения системного входа и точки статуса и формирует значения точек управления и системного выхода.

Обычно при проектировании аппаратного потока сначала занимаются трактом данных. В процессе этой работы определяются точки управления и статуса, о которых должен знать конечный автомат.

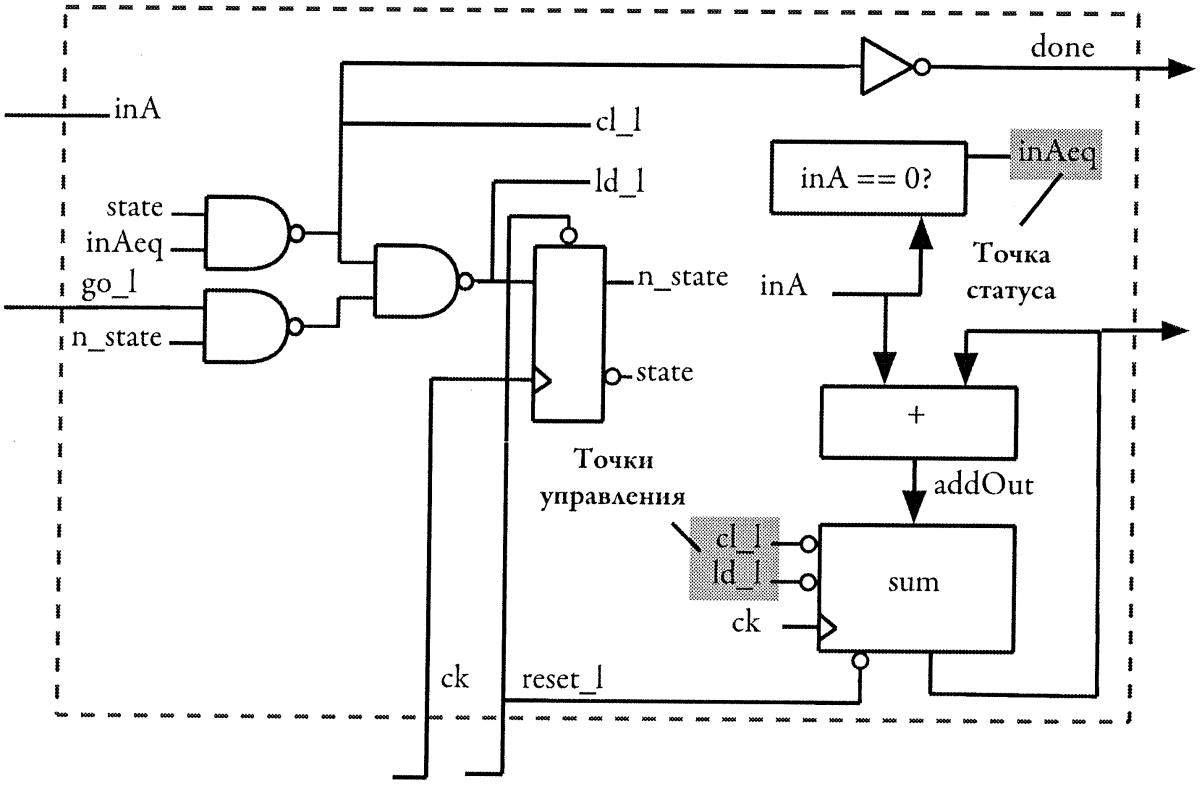



**Рис. 5.5.** Подробное представление аппаратного потока sumItUp

## 5.1.6. Описание на SystemVerilog

Полное описание аппаратного потока sumItUp на языке SystemVerilog приведено в примере 5.1. Некоторые части нам уже знакомы: регистр sum (строки 24–29) и сумматор (строка 31).

Порты модуля совпадают с портами потока, показанного на рис. 5.1 и 5.5.

Конечный автомат системы описан в блоке always_ff (строки 12–22). Описание соответствует диаграмме состояний на рис. 5.4. Например, строки 17–18 говорят, что если автомат находится в состоянии sA и не установлен сигнал go_l или автомат находится в состоянии sB и установлен сигнал inAeq, то следующим будет состояние sA. Это соответствует дугам, входящим в состоянии sA. Аналогично строки 19–20 говорят, при каком условии следующим состоянием будет sB. Установка значений выходов ld_l, cl_l и done (строки 32–35) также соответствует диаграмме состояний.

Обратите внимание, что используется два блока always_ff: первый описывает конечный автомат, второй – регистр тракта данных. Они соответствуют *состоянию управления* (состояние автомата) и *состоянию данных* (значение регистра). Для всех обновлений состояния используются неблокирующие присваивания (<=), поскольку все триггеры (как в автомате, так и в тракте данных) работают синхронно. Таким образом, состояние управления и состояние данных изменяются одновременно.

Интересно рассмотреть схемотехническую реализацию автомата, показанную на рис. 5.5. Здесь используются вентили nand (всякий знакомый с правилами де Моргана понимает, что эти вентили реализуют сумму произведений[33]). Обратите внимание: при сбросе триггер получает значение 1, а сигнал state, снимаемый с инверсного выхода, – значение 0, соответствующее элементу перечисления sA. Все дополнения были вычислены инструментом синтеза при построении реализации конечного автомата. (А вы, наверное, подумали, что автор спроектировал все это сам!) Инструмент синтеза мог инвертировать входы (inAeq и go_l)[34], а мог использовать инвертированную переменную состояния (n_state). В последнем случае не потребовалась дополнительная логика.

```
1 module sumItUp
2   (input logic ck, reset_l, go_l,
3   input logic [15:0] inA,
4   output logic done,
5    output logic [15:0] sum);
6
7   logic ld_l, cl_l, inAeq;
8   logic [15:0] addOut;
9
10  enum bit {sA, sB} state;
11
```

[33] ~(~(n_state & go_l) & ~(state & inAeq)) = (n_state & go_l) | (state & inAeq).

[34] (n_state & go_l) = ~(state | ~go_l).

```
12  always_ff @(posedge ck, negedge reset_l)
13    begin: st_machine
14    if (~reset_l)
15      state <= sA;
16    else begin
17      if (((state == sA) & go_l) | ((state == sB) & inAeq))
18        state <= sA;
19      if (((state == sA) & ~go_l) | ((state == sB) & ~inAeq))
20        state <= sB;
21    end
22  end: st_machine
23
24  always_ff @(posedge ck, negedge reset_l)
25    begin: reg_sum
26      if (~reset_l) sum <= 0;
27      else if (~ld_l) sum <= addOut;
28      else if (~cl_l) sum <= 0;
29  end: reg_sum
30
31  assign addOut = inA + sum,
32         ld_l = ~(((state == sA) & ~go_l) |
33              ((state == sB) & ~inAeq)),
34         cl_l = ~((state == sB) & inAeq),
35         done = (state == sB) & inAeq,
36         inAeq = inA == 0;
37 endmodule: sumItUp
```

**Пример 5.1.** Поток sumItUp – конечный автомат с трактом данных

Важно, что внутренняя реализация потока скрыта. Кто бы, например, мог подумать, что сигнал сброса соединен с установочным входом триггера, хранящего состояние автомата? Такие детали нужно знать только разработчику, занимающемуся временно́й оптимизацией. Остальным достаточно знать входы и выходы потока, а также протокол, описывающий порядок взаимодействия с ним, т. е. то, что показано на рис. 5.1.

## 5.1.7. Формальное определение

Формальное описание потока (конечного автомата с трактом данных) расширяет описание конечного автомата: добавляются переменные (в т. ч. массивы), операции и протокол ввода-вывода. Итак, поток описывается кортежем следующего вида (X, Z, S, δ, λ, V, O, P, C, R).

Его элементы иллюстрируются на примере, рассмотренном выше.

- X – множество скалярных и векторных входов. В нашем примере X = {go_l, inA}.
- Z – множество скалярных и векторных выходов. В нашем примере Z = {done, sum}.
- S – множество состояний, в котором выделено состояние сброса (начальное состояние) $s_0 \in S$. В нашем примере S= {0, 1} и $s_0 = 0$ (состояние sA).

- δ – функция перехода ($\delta: X \times S \rightarrow S$)[35]. В нашем примере ей соответствует комбинационная схема, выход которой подается на триггер (см. рис. 5.5), а также строки 17–20 примера 5.1.
- λ – функция выхода ($\lambda: X \times S \rightarrow Z$ для автоматов Мили и $\lambda: S \rightarrow Z$ для автоматов Мура)[36]. В нашем примере используется автомат Мили. Функция выхода вычисляет значения `cl_l`, `ld_l` и `done`; ей соответствуют строки 32–35 примера 5.1.
- V – множество переменных (включая векторы и массивы), используемых в тракте данных. В нашем примере V = {`sum`}. Обновление состояния данных показано в строках 24–29 примера 5.1.
- O – множество операций, используемых в тракте данных. В нашем примере O = {+, ==}. Операции задействованы в строках 31 и 36 примера 5.1.
- P – протокол ввода-вывода. В нашем примере он отражает отношения между установкой `go_l` и последовательностью значений `inA`, установкой `done` и значением `sum`. Часто описывается словесно или в виде временны́х диаграмм, как на рис. 5.2.
- C – тактовый сигнал (точнее, домен синхронизации).
- R – сигнал сброса, по которому поток переходит в начальное состояние. В нашем примере начальное состояние – $s_0$ (состояние управления) и `sum` == 0 (состояние данных).

## 5.2. Временны́е характеристики автоматов Мура и Мили

Важным аспектом проектирования систем является разработка протоколов взаимодействия между потоками. В протоколе, определенном для потока `sumItUp`, требуется автомат Мили – на рис. 5.6 объяснено, почему.

В левой части рисунка изображена часть диаграммы состояний: на ней показаны переходы, исходящие из состояния `sA`. Пока автомат находится в состоянии `sA`, значение выхода `ld_l` определяется значением входа `go_l`. Это определяется протоколом: если установлен сигнал `go_l`, то на линии `inA` находится первое из суммируемых чисел; чтобы это значение было учтено, установка `go_l` должна вызывать установку `ld_l`. В правой части рисунка показана временна́я диаграмма сигнала «`inA` Мили»; она совпадает с диаграммой сигнала `inA` на рис. 5.2 – при переходе из `sA` в `sB` сигнал имеет значение 7. Это наводит на мысль, что `inA` соединен – в данном случае посредством сумматора – с входом регистра `sum`. Таким образом, если в состоянии `sA` установлен сигнал `go_l`, это приводит к установке `ld_l` и загрузке значения в регистр `sum`. Временна́я диаграмма с таким соотношением между входом и выходом предполагает реализацию в виде автомата Мили.

---

[35] Здесь есть математическая неточность: вместо множества входов (X) здесь должно быть множество значений (еще точнее – функций означивания) входов ($Dom_X$).

[36] См. предыдущее замечание в отношении X и Z.

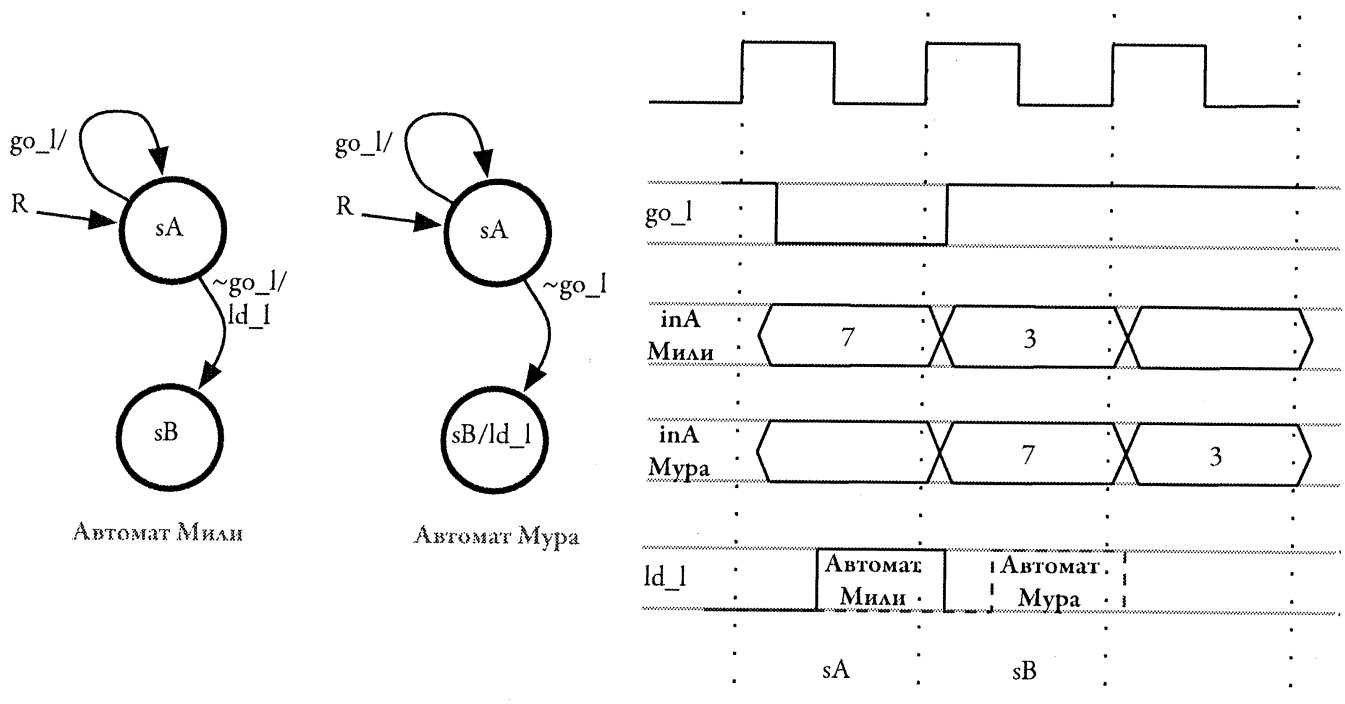



**Рис. 5.6.** Сравнение временны́х диаграмм автоматов Мили и Мура: связь между установкой go_l и подачей значения на inA

На рисунке не показана связь между нулевым значением входа inA и установкой выхода done (см. рис. 5.2). Это тоже связь, характерная для автоматов Мили.

Правая диаграмма состояний на рис. 5.6, а также строки временно́й диаграммы для сигналов «inA Мура» и ld_l показывают, как бы выглядел протокол, если бы мы решили реализовать поток в виде автомата Мура. Переход из sA в sB по-прежнему вызывается установкой go_l, однако выходной сигнал ld_l, разрешающий загрузку значения inA, устанавливается только на следующем такте, т. е. когда автомат уже находится в состоянии sB. Это показано серой пунктирной линией в нижней строке временно́й диаграммы. Таким образом, измененный протокол предполагает, что значения на входе inA начинают подаваться не одновременно с установкой go_l, а с задержкой в один такт. Как правило, протокол описывается в спецификации.

Интересно, что связь {go_l – inA}, характерную для автоматов Мили, можно реализовать с помощью автомата Мура. Для этого понадобилось бы слегка изменить тракт данных. В состоянии sA нужно загружать в регистр sum значение inA (непосредственно, без сложения с предыдущим значением sum), а в состоянии sB – сумму значений inA и sum. Связь {inA – done} все-таки требует автомата Мили.

## 5.3. Компоненты тракта данных

Как вы, наверное, заметили, проектируя тракт данных потока sumItUp, мы изобрели несколько полезных компонентов: сумматор, регистр и устройство сравнения. Как правило, каждая команда разработчиков имеет в своем распоряжении библиотеку готовых и протестированных компонентов, которые

задействуются в проекте по мере необходимости. Иногда, стремясь оптимизировать тракт данных, команды создают свои собственные компоненты.

В этом разделе мы рассмотрим основные типы компонентов тракта данных и их разновидности. Упор делается на компоненты общего вида; для специфических вещей придется разрабатывать собственные компоненты.

## 5.3.1. Комбинационные элементы

Регистровые передачи заключаются в преобразовании значений, хранящихся в одном или нескольких регистрах, и загрузке результатов преобразований в другие регистры по следующему фронту тактового сигнала. Преобразования реализуются с помощью комбинационных схем. Как мы узнали в главе 2, для описания таких схем в SystemVerilog используются операторы `assign` и `always_comb`.

Рассмотрим сумматор из предыдущего раздела:

```
1   assign addOut = inA + sum,
```

По этому описанию строится сумматор той же разрядности, что у операндов.

Если нужно реализовать две функции, скажем сумматор и вычитатель, это можно сделать с помощью условного оператора:

```
1   assign fnOut = (addSub) ? a + b : a - b;
```

Здесь, если `addSub` равно TRUE, результатом будет `a+b`, иначе `a-b`. Переменная `addSub`, таким образом, является точкой управления этого функционального блока (а также вышестоящего тракта данных).

Как правило, в тракте данных больше операций; для их реализации используется АЛУ (см. пример 5.2).

```
1  typedef enum logic [2:0] {ADD=3'b100,
2    SUB=3'b010, AND=3'b001, OR=3'b110,
3    XOR=3'b011} aluInst_t;
4
5  module alu
6    (input logic [7:0] a, b,
7     output logic [7:0] result,
8     input aluInst_t op);
9
10   always_comb
11     unique case (op)
12       ADD: result = a + b;
13       SUB: result = a - b;
14       AND: result = a & b;
15       OR: result = a | b;
16       XOR: result = a ^ b;
17     endcase
18   end
19 endmodule: alu
```

**Пример 5.2.** Арифметико-логическое устройство

АЛУ из примера имеет управляющий вход (`op`), позволяющий выбрать операцию, применяемую к двум операндам (`a` и `b`). Множество операций задано в перечислимом типе `aluInst_t` (для именования типа используется конструкция `typedef`). Итак, вход `op` имеет тип `aluInst_t` (строка 8); для каждого значения этого входа в операторе `case` реализована соответствующая операция. Какую операцию исполнять в тот или иной момент времени, решает конечный автомат, управляющий трактом данных. Таким образом, `op` – многобитная точка управления.

Меняя значения элементов перечисления и состав реализуемых функций, можно спроектировать различные АЛУ (при расширении множества функций потребуется увеличение разрядности управляющего входа).

Отметим, что поскольку компоненты тракта данных являются комбинационными схемами, при их проектировании должны соблюдаться правила из раздела 2.1.2.

## 5.3.2. Регистры

Результат регистровой передачи сохраняется в регистре. Регистр может быть совсем простым, как этот:

```
1   bit [9:0] a, b;
2   always_ff (posedge ck)
3       a <= b;
```

А может иметь функцию сброса, как этот:

```
1   bit [16:0] q, d;
2   always_ff (posedge ck or negedge reset)
3     if (~reset) q <= 0;
4     else q <= d;
```

Обычно регистр не загружается по каждому фронту тактового сигнала – у него есть сигнал разрешения загрузки:

```
1   always_ff (posedge ck)
2     if (~load_l) a <= b;
```

Загрузка осуществляется, только если установлен сигнал load_l. Этому фрагменту кода соответствует логическая схема на рис. 5.7. Если установлен load_l, порт 0 мультиплексора соединяется с входом триггера, и значение b становится следующим состоянием. Если load_l не установлен, в триггер загружается его текущее значение.

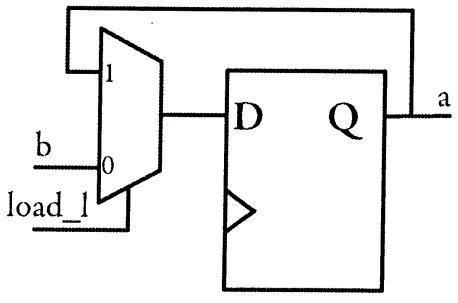



**Рис. 5.7.** 1-битный регистр с сигналом разрешения загрузки

Регистры часто реализуют функции над своим содержимым: входное значение берется из регистра, в него же загружается результат. Например, у регистров могут быть такие функции, как инкремент, декремент, сдвиг влево, сдвиг вправо, очистка, загрузка, асинхронный сброс. Что из этого нужно, решает разработчик.

Ниже продублирован пример из предыдущего раздела, иллюстрирующий регистр, с которым можно производить три операции: загрузку данных, синхронную очистку и асинхронный сброс.

```
1   always_ff @(posedge ck, negedge reset_l)
2     begin: reg_sum
3       if (~reset_l) sum <= 0;
4       else if (~ld_l) sum <= addOut;
5       else if (~cl_l) sum <= 0;
6     end: reg_sum
```

```
1 module register
2   #(parameter w = 1)
3   (input logic ck, reset_l, load, incr,
4    input logic [w-1:0] dInput,
5    output logic [w-1:0] sum);
6
7   always_ff @(posedge ck, negedge reset_l)
8     begin
9     if (~reset_l) sum <= 0;
10    else if (load) sum <= dInput;
11    else if (incr) sum <= sum + 1;
12  end
13 endmodule: register
```

**Пример 5.3.** Регистр с функциями загрузки и инкремента

Нормальными операциями регистра являются загрузка и очистка – в том смысле, что если система работает, как задумано, задействуются только эти две операции (и, конечно, отсутствие операции). Сброс используется только при включении питания.

В примере 5.3 показан регистр с функциями загрузки и инкремента (строки 10–11). Модуль параметризован шириной и имеет две точки управления: `load` и `incr`.

Сдвиговые регистры – еще одна разновидность регистров, часто используемых в трактах данных. В примере 5.4 приведен 8-битный регистр со сдвигом вправо. Здесь `q` – внутреннее состояние, в которое загружается значение `d`, если установлен сигнал `ld` (строка 9). Если установлен сигнал `sh` (строки 10–11), значение `q` сдвигается на одну позицию вправо, при этом в старший бит записывается 0 (строка 11). У этого регистра две точки управления (`ld` и `sh`) и одна точка статуса (`lowBit`).

```
1 module sr
2   (input ck, ld, sh,
3    input [7:0] d,
4    output lowBit);
5   logic [7:0] q;
6
7   assign lowBit = q[0];
8   always_ff @(posedge ck)
9     if (ld) q <= d;
10    else if (sh)
11      q <= q >> 1;
12 endmodule: sr
```

**Пример 5.4.** Сдвиговый регистр

Поясним, что >> – это оператор сдвига вправо: левый операнд сдвигается вправо на число позиций, указанное в правом операнде; старшие биты заполняются нулями. В нашем примере производится сдвиг значения `q` на один бит вправо; в старший бит помещается 0. Значение младшего бита при сдвиге вправо теряется. Результат загружается обратно в `q`, как показано в строке 11. Величина сдвига необязательно должна быть равной 1. Так, следующий код

```
1   bit [11:0] q, r;
2   r <= q >> 3;
```

сдвигает значение `q` на 3 позиции вправо, вставляя в старшие биты 0, и загружает результат в `r`. Значение `q` при этом не изменяется.

Отметим, что в одних примерах точками управления являются сигналы с низким активным уровнем, а в других – с высоким. Решение о выборе активных уровней сигналов принимает разработчик библиотечных модулей.

## 5.3.3. Дешифраторы

Дешифратор (иногда называемый демультиплексором) получает на входе n-битное значение, которое используется для выбора одного из $2^n$ выходов. Дешифраторы часто встречаются в трактах данных – с их помощью выбирается одна

из нескольких функций. В примере 5.5 показан дешифратор 3-8, который по значению 3-битного входа устанавливает один из 8 выходов. Вход sel интерпретируется как 3-битное число, принимающее значения от 0 до 7. В выходном 8-битном векторе устанавливается бит с этим номером. Так, если значение sel равно 5, на выходе получается 8'b0010_0000: установлен 5-й бит справа (нумерация начинается с 0). В приведенном примере выходные сигналы имеют высокий активный уровень. Могут быть дешифраторы с низким активным уровнем выходных сигналов (все биты, кроме выбранного, имеют значение 1). Очевидно, что модуль преобразует двоично-кодированное число в одно единичное представление (установлен только один бит).

```
1 module decoder
2   (input logic [2:0] sel,
3    output logic [7:0] oneHot);
4
5   always_comb
6     case (sel)
7       0: oneHot = 8'b0000_0001;
8       1: oneHot = 8'b0000_0010;
9       2: oneHot = 8'b0000_0100;
10      3: oneHot = 8'b0000_1000;
11      4: oneHot = 8'b0001_0000;
12      5: oneHot = 8'b0010_0000;
13      6: oneHot = 8'b0100_0000;
14      7: oneHot = 8'b1000_0000;
15    endcase
16 endmodule: decoder
```

**Пример 5.5.** Дешифратор 3-8

На рис. 5.8 показано, как использовать дешифратор 2-4 для выбора одного из четырех регистров для загрузки данных. Выходом конечного автомата является 2-битный сигнал sel; в зависимости от его значения дешифратор выдает высокий уровень на одной из линий d0–d3, что, в свою очередь, разрешает загрузку в один из регистров a–d на очередном фронте тактового сигнала. Отметим, что каждый раз разрешается загрузка только в один регистр, причем эта загрузка всегда происходит.

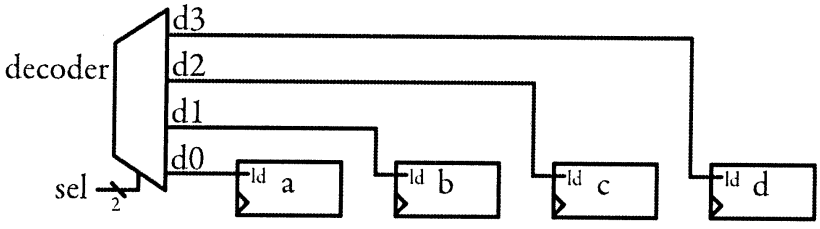



**Рис. 5.8.** Дешифратор для выбора одного из четырех регистров

В примере 5.6 показана частичная реализация логики, представленной на рис. 5.8. Здесь описаны дешифратор 2-4 и регистр с сигналом разрешения загрузки. При создании экземпляра дешифратора в строке 26 объявленная в модуле дешифратора векторная переменная (oneHot[3:0]) соединяется с конкатенированными скалярами ({d3, d2, d1, d0}).

```
1 module decode2to4
2   (input logic [1:0] sel,
3    output logic [3:0] oneHot);
4
5   always_comb
6     case (sel)
7       0: oneHot = 4'b0001;
8       1: oneHot = 4'b0010;
```

```
9         2: oneHot = 4'b0100;
10        3: oneHot = 4'b1000;
11      endcase
12 endmodule: decode2to4
13
14 module reg16
15   (input logic ck, load,
16    input logic [15:0] dInput,
17    output logic [15:0] qOutput);
18
19   always_ff @(posedge ck)
20     if (load) qOutput <= dInput;
21 endmodule: reg16
22
23 module partialDataPath;
24 ...
25   logic d0, d1, d2, d3;
26   decode2to4 dec (sel, {d3, d2, d1, d0});
27
28   reg16 a (.load(d0), ...);
29   reg16 b (.load(d1), ...);
30   reg16 c (.load(d2), ...);
31   reg16 d (.load(d3), ...);
32 endmodule: partialDataPath
```

**Пример 5.6.** Использование дешифратора

### 5.3.4. Шины

Шиной называется средство передачи информации, разделяемое между несколькими компонентами. Другими словами, в разные моменты времени по шине может передаваться информация, сформированная разными источниками. К шине может быть подключено несколько передатчиков и несколько приемников; выбор отправителя и адресата для каждой передачи осуществляется управляющим автоматом.

Основные функции в системах с шинами реализуются в шинных приемниках и передатчиках. Иногда приемник и передатчик объединяются в одно устройство – приемопередатчик. Важно, что в каждый момент времени только один драйвер (передатчик) может формировать данные на шине. Соответственно, должна быть возможность отключения функции формирования данных на выходе передатчика. Для этого используются буферы с тристабильными выходами.

В SystemVerilog тристабильными считаются вентили, которые «формируют» значения 0, 1 и z. Термин «формировать» в этом контексте – оксюморон: значение z означает, что выход отключен (не сформирован). Реальный физический вентиль формирует на линии шины либо 0, либо 1. Если драйвер выключен, ситуация выглядит так, как будто соединения с шиной нет. Точнее, соединение имеет такое высокое сопротивление, что можно считать, что его нет.

У тристабильных вентилей имеется отдельная линия разрешения, позволяющая включать и выключать драйвер; это точка управления, формируемая конечным автоматом. Приведем пример:

```
1   assign busLine = (~enableL) ? valueToDrive : 1'bz;
```

Если сигнал enableL установлен, на шине busLine формируется значение valueToDrive; в противном случае – значение z. Предположим, что к шине подключено несколько передатчиков; в каждый момент времени активным может быть только один из них. Часто, чтобы активировать тот или иной передатчик, управляющий автомат выдает номер; номер передается дешифратору, который устанавливает один из сигналов разрешения.

Тристабильные шинные приемники, передатчики и приемопередатчики рассматриваются в разделе 2.7.

## 5.3.5. Модули памяти

В трактах данных часто используются регистры и небольшая рабочая память. В SystemVerilog массив памяти описывается следующим образом:

```
1   bit [7:0] m [256];
2   logic [7:0] n [255:0];
```

Числа в первой паре квадратных скобок определяют разрядность элемента массива, а во второй – число элементов. В обеих строках определяется массив из 256 8-битных векторов, однако элементы массива m имеют тип bit, а элементы массива n – logic. Допустимы многомерные массивы.

В примере 5.7 показана работа с массивом памяти. Модуль параметризован разрядностью и числом элементов памяти (строки 2–4). Обратите внимание на системную функцию $clog2, возвращающую ширину адреса, достаточную для заданного числа элементов. Имя функции – сокращение от «ceiling of the base 2 log» (наименьшее целое, не меньшее значения двоичного логарифма). Таким образом, мы получаем корректную разрядность, даже если число элементов не является точной степенью двойки. Параметры используются при определении входов модуля и его выхода, а также для объявления массива (строка 11).

Функциональность модуля памяти описана в строках 13–15. Как видно из строки 13, память всегда читается: на выходе dataOut устанавливается ее содержимое по заданному адресу. Тактовый сигнал и сигнал разрешения записи в память (строка 8) используются для загрузки в па-

```
1  module mem
2    #(parameter
3        Dwidth = 8,
4        Words = 256,
5        Awidth = $clog2(Words))
6    (input bit [Dwidth-1:0] dataIn,
7     input bit [Awidth-1:0] address,
8     input bit clock, wEnable,
9     output bit [Dwidth-1:0] dataOut);
10
11   bit [Dwidth-1:0] m [Words];
12
13   assign dataOut = m[address];
14   always_ff @(posedge clock)
15     if (wEnable) m[address] <= dataIn;
16 endmodule: mem
```

**Пример 5.7.** Простой массив памяти

```
1 module busmem
2   #(parameter DW = 8,
3         W = 256,
4         AW = $clog2(W))
5   (input bit [AW-1:0] addr,
6    input logic re, we, clk,
7    inout tri [DW-1:0] data);
8
9   logic [DW-1:0] m [W];
10
11  assign data = (re) ? m[addr] : 'bz;
12
13  always @(posedge clk)
14    if (we) begin
15      m[addr] <= data;
16    end
17 endmodule: busmem
```

**Пример 5.8.** Память с разделяемым портом данных

мять нового значения (загрузка происходит по фронту тактового сигнала). Эта функциональность показана в строках 14–15.

Многие модули памяти имеют двунаправленный порт для чтения из памяти и записи в нее. В этом случае к памяти добавляется шинный приемопередатчик, как показано в примере 5.8. Поскольку порт служит как для чтения, так и для записи, он объявлен как `inout` (строка 7); в объявлении указан тип `tri`, так как используется тристабильная логика.

Для чтения данных из памяти нужно установить сигнал разрешения чтения (`re`). Когда сигнал установлен, тристабильный драйвер (строка 11) записывает в порт значение, хранящееся в памяти по заданному адресу. Для записи данных в память нужно установить сигнал разрешения записи (`we`). Как и в предыдущем примере, переданное в порт значение записывается в память по фронту тактового сигнала. В этом примере `re` и `we` являются точками управления модуля памяти.

Когда создается экземпляр модуля `busmem` (строке 2 ниже), его порт соединяется с цепью типа `tri`; для формирования значений этой цепи используется тристабильный драйвер:

```
1   tri [DW:0] dataline;
2   busmem b (.data(dataline), .we(we), …);
3   assign dataline = (we)? valueToWrite : 1'bz;
```

Точки управления `re` и `we` не могут быть установлены одновременно. Проблема, препятствующая одновременной установке, в самом описании модуля не видна. Она возникает при создании экземпляра: если, например, используется фрагмент кода, показанный выше, то при одновременной установке `re` и `we` значение порта данных формируется как в строке 3, так и в строке 11 примера 5.8. Держите под рукой телефон пожарной части.

В массиве типа `logic` все биты имеют начальное значение `x`, в массиве типа `bit` – 0. Чтобы инициализировать их другими значениями, есть две возможности. Во-первых, можно воспользоваться циклом `for` в блоке `initial`, например:

```
1   logic [15:0] mem [512];
2   initial begin
3     for (int i; i<512; i++)
4       mem[i] = i;
```

Здесь каждый элемент памяти (массива) инициализируется своим адресом (индексом). Разумеется, можно использовать другие значения.

Во-вторых, память можно загрузить из внешнего файла с помощью системных процедур `$readmemb` и `$readmemh`. Обе функции имеют одинаковый формат вызова:

```
$readmemh (filename, memName);
```

Функция читает числа в шестнадцатеричном формате из файла с именем `filename` и загружает их в массив `memName`. В третьем и четвертом аргументах можно указать начальный и конечный индексы инициализируемой части массива. Если начальный индекс не указан, загрузка начинается с младшего индекса. В файле могут присутствовать только числа и разделители: пробелы, табуляции и символы новой строки. Функция `$readmemb` делает то же самое, но требует, чтобы числа были представлены в двоичном формате.

## 5.4. Тестовые окружения для аппаратных потоков

Тема разработки тестовых окружений для аппаратных потоков гораздо обширнее того, что мы обсуждали в контексте тестирования комбинационных схем. Созданию тестовых окружений посвящена глава 7, к которой мы отсылаем читателя.

Для студентов, выполняющих небольшие проекты, наиболее актуальны разделы 7.1, 7.3 и 7.4, посвященные тестированию аппаратных потоков.

Инженерам-разработчикам, работающим над крупными проектами в группах верификации, рекомендуем также прочитать раздел 7.2.

Разница в следующем. В индустриальных проектах тестовые окружения обычно имеют вид программ (для этого используется конструкция `program`). Программы очень похожи на модули, но предоставляют средства, полезные для проектирования и верификации сложных систем. Студенту, работающему над простым проектом, для начала будет вполне достаточно модулей. Просто используйте в примерах module вместо program.

## 5.5. Задачи и упражнения

**5.1. Слишком много единиц.** 5-битные последовательности (пятерки) принимаются по одному биту за такт; каждая пятерка представляет десятичную цифру в формате «2 из 5» (ровно два бита имеют значение 1 – см. таблицу); задача – определить, является ли пятерка допустимой. Биты начинают приниматься сразу после сброса. Выход `valid` устанавливается тогда и только тогда, когда на входе появляется последний бит допустимой пятерки. Следующий входной бит считается первым битом очередной пятерки (т. е. пятерки поступают непрерывно).

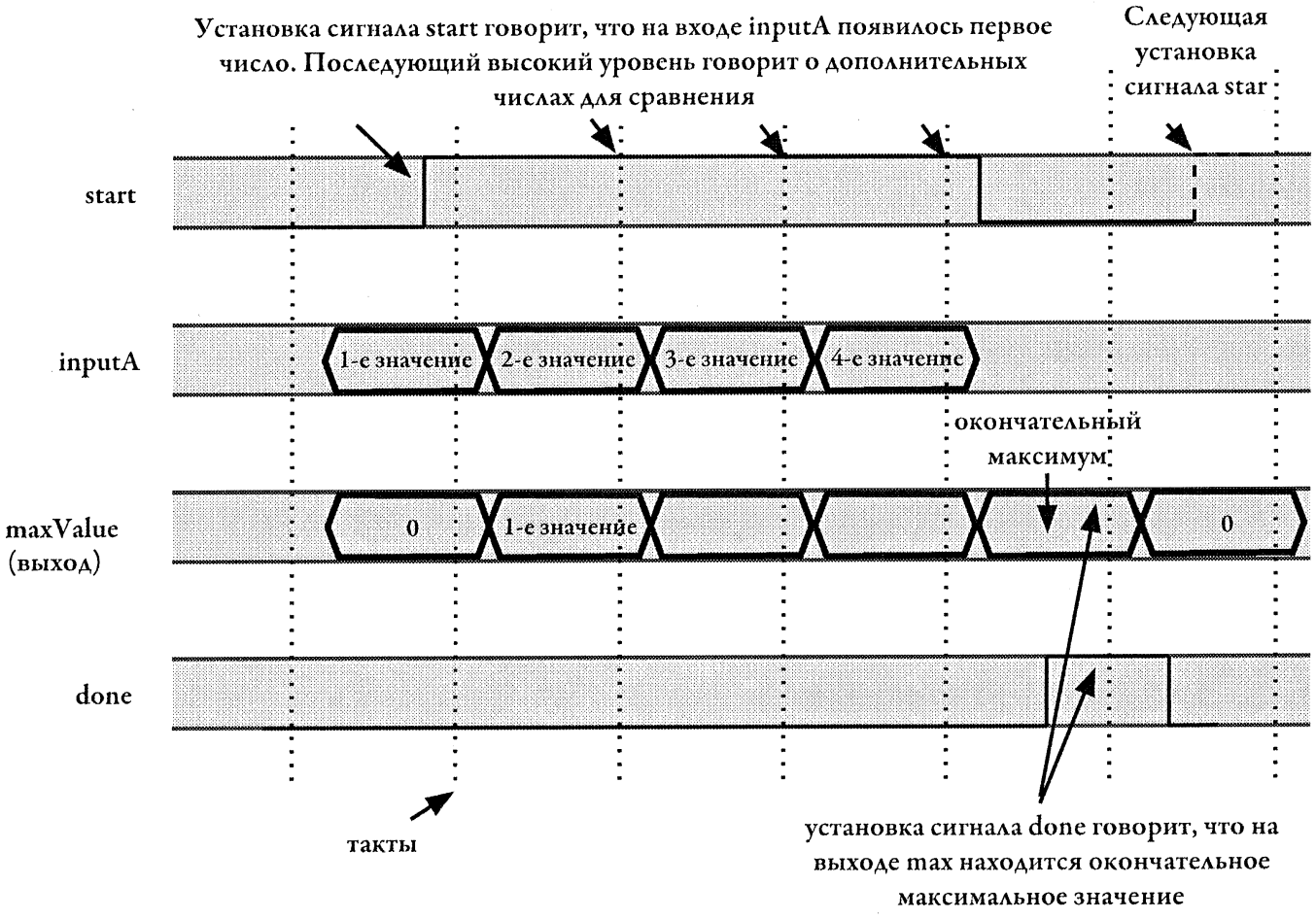
В своей модели используйте два счетчика: один – для отслеживания номера бита внутри пятерки; другой – для подсчета числа единиц, встретившихся в пятерке.

| 2-of-5 |
|---|
| 00011 |
| 00101 |
| 00110 |
| 01010 |
| 01100 |
| 01001 |
| 11000 |
| 10100 |
| 10010 |
| 10001 |

Нарисуйте диаграмму состояний. Выявите точки управления и точки статуса. Напишите и протестируйте код на SystemVerilog (вход назовите `in`, выход – `valid`). Сравните полученную модель с моделью без тракта данных, в которой вся логика представлена конечным автоматом.

**5.2. Где максимум?** Спроектируйте и реализуйте аппаратный поток для нахождения максимального из чисел, подаваемых на вход (числа беззнаковые). Ниже приведено определение портов:

```
1  module findMax
2    (input logic start,
3     input logic [7:0] inputA,
4     output logic done,
5     output logic [7:0] maxValue,
6     input logic clk, rst);
```




Сигнал `start` – это управляющий сигнал, сообщающий потоку, когда начинать прием входных данных (поток ждет установки `start`). Одновременно с установкой `start` на входе `inputA` появляется первое число. Пока `start` остается установленным, на вход поступают новые числа (по одному на каждый фронт тактового сигнала). По мере поступления данных значение выхода `maxValue` обновляется и отражает максимальное из поступивших до сих пор чисел.

Когда сигнал start сбрасывается, устанавливается выходной сигнал done, говорящий, что выход maxValue содержит максимальное число последовательности; сигнал остается установленным в течение одного такта. После этого поток ждет следующей установки сигнала start. После сброса значение maxValue равно 0, а ожидает установки start. Длина последовательности необязательно равна 4: она может быть больше или меньше.

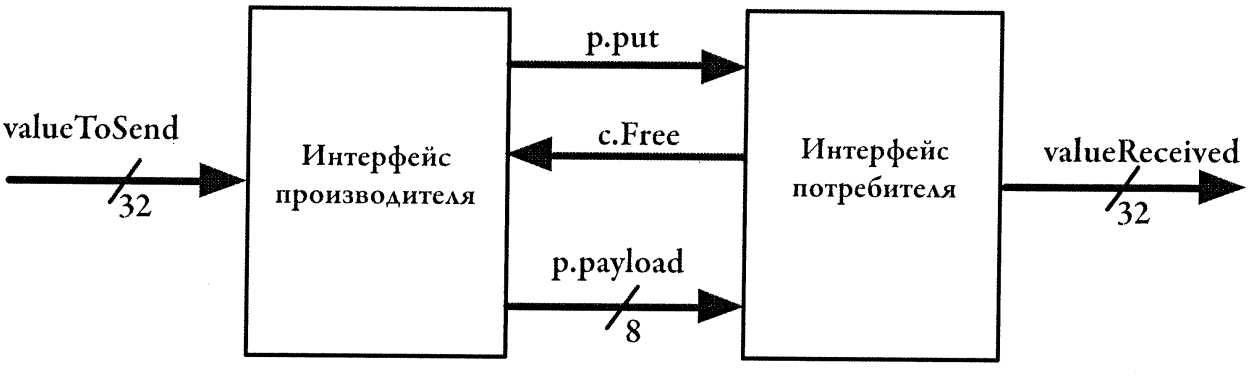
Изобразите диаграмму состояний потока и схему тракта данных. Используйте компоненты, аналогичные представленным в этой главе. Отметьте точки управления и статуса. Напишите соответствующий код на SystemVerilog. Разработайте тестовое окружение, которое посылает потоку несколько последовательностей чисел.

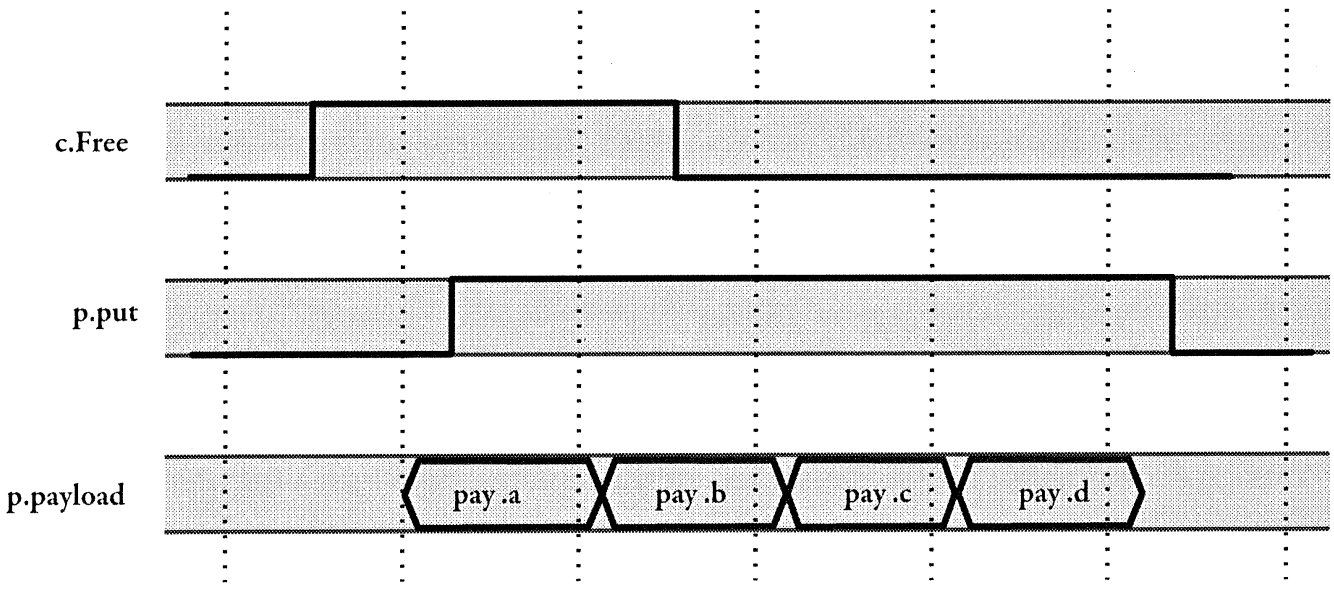
**Похожие задачи:** найти минимум последовательности, вычислить контрольную сумму чисел.

**5.3. Снова sumItUp, но не совсем!** Добавьте в поток sumItUp, представленный в этой главе, выходной сигнал error, показывающий, что в АЛУ возникло переполнение. После обнаружения переполнения error остается установленным до следующей установки go_L. Установка error не сопровождается установкой done.

**5.4. Политкорректная система.** Производитель (producer) отправляет 32 бита информации потребителю (consumer) последовательными блоками по 8 бит. Потребитель получает блоки и собирает их в одно 32-битное слово. См. диаграмму.

Передача управляется сигналами p.put и c.Free; данные передаются по линии p.payload. В названиях отражены стороны, формирующие сигналы (так, p.put означает, что производитель формирует сигнал, сообщающий потребителю о наличии данных на линии). В описании на SystemVerilog такие имена невозможны – почему?). Производитель и потребитель описываются разными конечными автоматами.






На показанной выше временно́й диаграмме видно, как данные передаются от производителя потребителю. Передача осуществляется синхронно. Когда потребитель готов к приему информации, он устанавливает сигнал `c.Free`. Производитель, увидев `c.Free`, помещает байт `pay.a` на линию `p.payload` и одновременно устанавливает сигнал `p.put`. Увидев `p.put`, потребитель делает две вещи: сбрасывает `c.Free` (теперь он не свободен, а занят приемом данных) и последовательно загружает данные с линии `p.payload` в переменные `pay.a`, `pay.b`, `pay.c` и `pay.d` по фронтам тактового сигнала (как показано выше). На следующем такте после установки `pay.d` производитель сбрасывает `p.put`. Потребитель видит, что `p.put` сброшен, и устанавливает `c.Free`.

Вход `valueToSend` и выход `valueReceived` представлены в следующем упакованном формате. Как видно на диаграмме, в каждый момент времени посылается только один элемент структуры.

```
1   struct packed {
2     bit[7:0] a;
3     bit[7:0] b;
4     bit[7:0] c;
5     bit[7:0] d;
6   } pay;
```

Спроектируйте потоки производителя и потребителя. В обоих потоках должны быть регистры для хранения сигналов и принимаемых данных. Определите входы, выходы и точки управления этих регистров. Отобразите их в диаграммах состояний конечных автоматов и в схемах трактов данных (регистров немного).

### 5.5. Последовательный сумматор

**А.** Спроектируйте последовательный сумматор; заголовок модуля и временна́я диаграмма показаны ниже. Сумматор складывает два 6-битных числа. Каждое число поступает по одному биту (сначала младший). Создайте

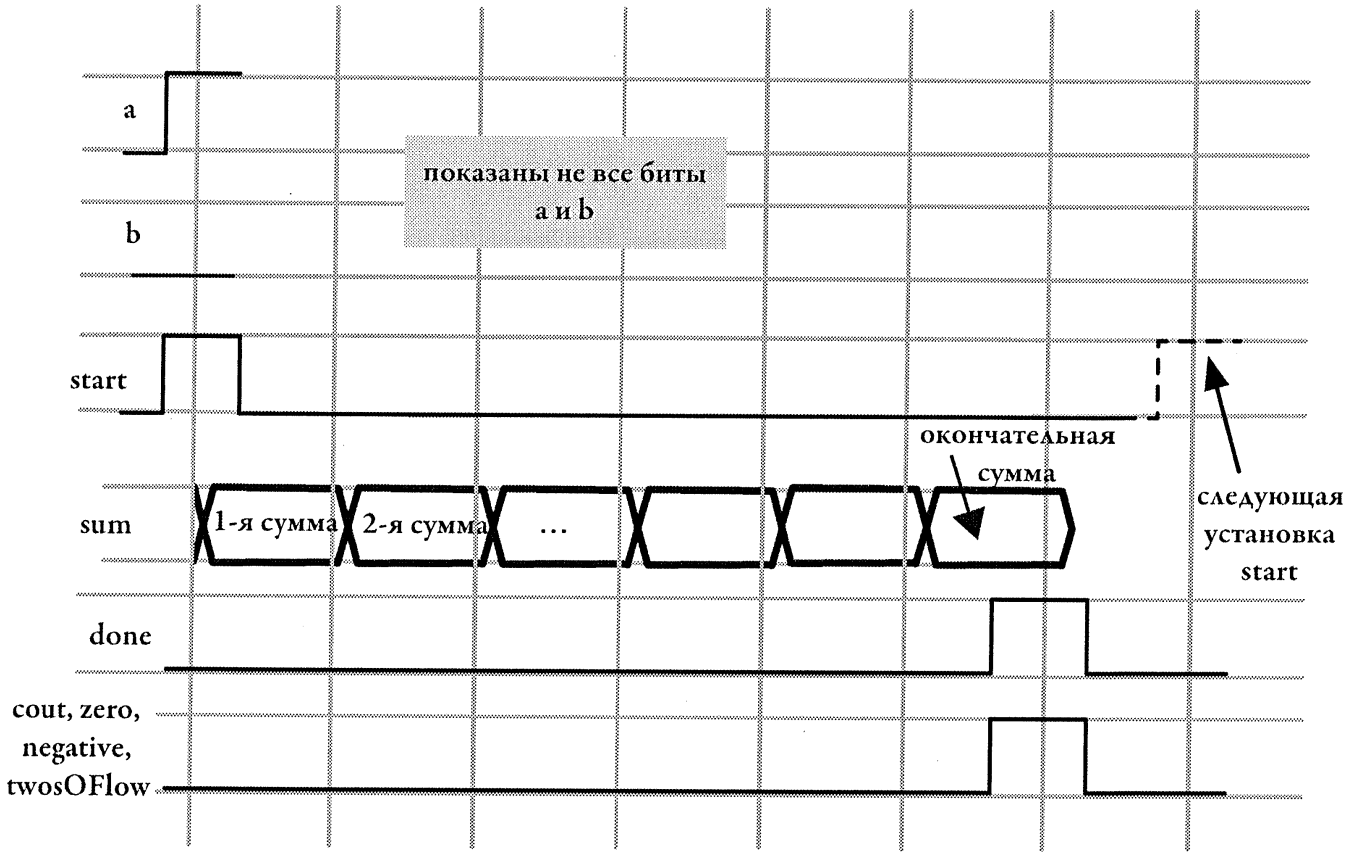
экземпляры необходимых модулей (полный сумматор, триггеры, регистры, конечный автомат и прочее), соедините их и продемонстрируйте работу в тестовом окружении.

Входы a и b – складываемые биты. Установка сигнала start обозначает момент времени, когда на входах появляются первые (младшие) биты. Сумма запоминается во внутреннем регистре sum. Установка сигнала done говорит, что на выходе sum находится окончательная 6-битная сумма и что значимы другие выходные сигналы (cout, zero, ...). Все входные и выходные сигналы устанавливаются на один такт.

Отметим, что сумма первых двух битов a и b помещается в первый бит регистра sum (1-я сумма). На следующем такте вычисляется сумма следующих битов a и b, а также бита переноса от предыдущего суммирования; результат записывается во второй бит регистра sum (2-я сумма). Так продолжается, пока во внутреннем регистре (sum) не окажется сумма всех 6 пар битов. Установленный сигнал done говорит, что суммирование завершено и на выходе sum находится результат.

Ниже приведено определение портов модуля:

```
1   module serialAdder
2     (input logic a, b, start, reset, clock,
3      output logic [5:0] sum,
4      output logic done, cout, zero, neg, twosOFlow);
```

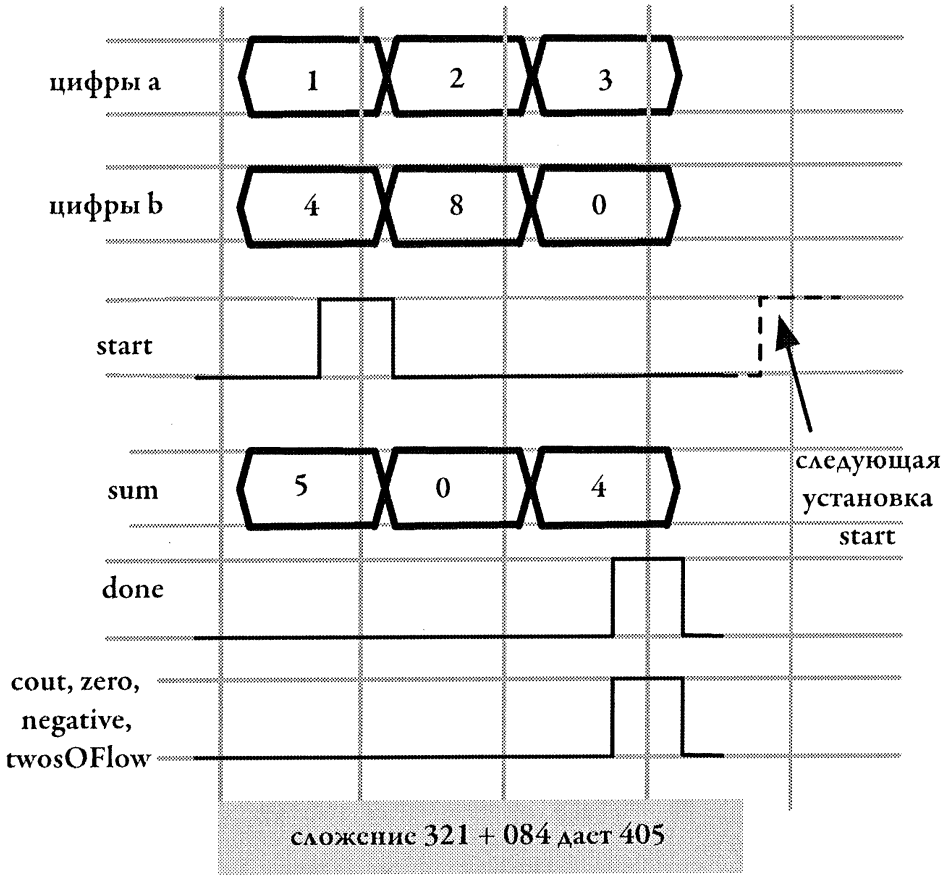


**B.** Измените протокол (и модель) так, чтобы окончательная сумма, done и прочие сигналы (cout, zero, ...) были значимы на один такт раньше, как и первая установка start.

**5.6. Последовательный двоично-десятичный сумматор.** Первые калькуляторы не преобразовывали числа, введенные в двоично-десятичном коде (Binary-Coded Decimal – BCD), в двоичную форму, а работали непосредственно с BCD-цифрами. Соответственно, результат не надо было преобразовывать в двоично-десятичный код для отображения. Глубоко в недрах калькулятора таился последовательный двоично-десятичный сумматор.

Спроектируйте последовательный двоично-десятичный сумматор с показанным ниже заголовком модуля. Первые две BCD-цифры (младшие) должны находиться на входах, когда установлен сигнал start. Установка done означает, что текущие BCD-цифры последние. По следующему после done фронту тактового сигнала может быть повторно установлен start. На выходе всегда находится сумма двух текущих BCD-цифр и величины переноса от предыдущей стадии (на первой стадии переноса нет). Помните, что на входах и выходе могут быть только допустимые BCD-цифры, то есть цифры от 0 до 9.

```
1   module serialBCDadder
2     (input logic [3:0] a, b,
3      input logic start, done, reset, clock,
4      output logic [3:0] sum);
```



**5.7. Счет от 0 до 99.** Спроектируйте двухдекадный счетчик[37] с использованием двух 4-битных синхронных каскадных декадных счетчиков[38]. Поясним, что имеется в виду:

- «синхронный» – входы синхронизации всех триггеров соединены с одним тактовым сигналом;
- «декадный счетчик» – после сброса принимает значение 4'b0000; пробегает значения 4'b0000, 4'b0001, ... 4'b1001, 4'b0000 по фронту тактового сигнала, при условии что установлен `countEnable`;
- «countEnable» – сигнал разрешения счета; счетчик не изменяет значения, если этот сигнал не установлен;
- «каскадный» – имеется выход `carryOut`, показывающий, что вход `countEnable` установлен и значение счетчика изменяется от 4'b1001 к 4'b0000; по этому сигналу инкрементируется следующая по старшинству цифра – выход `carryOut` соединен с входом `countEnable` следующего декадного счетчика.

**A.** Нарисуйте диаграмму состояний одного декадного счетчика. Состояния счетчика кодируются 4 битами, названными от A (младший бит) до D (старший бит). Это автомат Мили. Пример взаимодействия сигналов приведен на временно́й диаграмме ниже.

**B.** Напишите на SystemVerilog модуль декадного счетчика. Используйте явный стиль описания конечного автомата.

```
1   module decadeCTR
2     (input logic ck, r, countEnable,
3      output logic carryOut,
4      output logic [3:0] DCBA);
```

**C.** Напишите на SystemVerilog модуль верхнего уровня для двухдекадного счетчика. В нем должны создаваться экземпляры двух декадных счетчиков (число единиц и число десятков). Модуль верхнего уровня должен иметь такой заголовок:

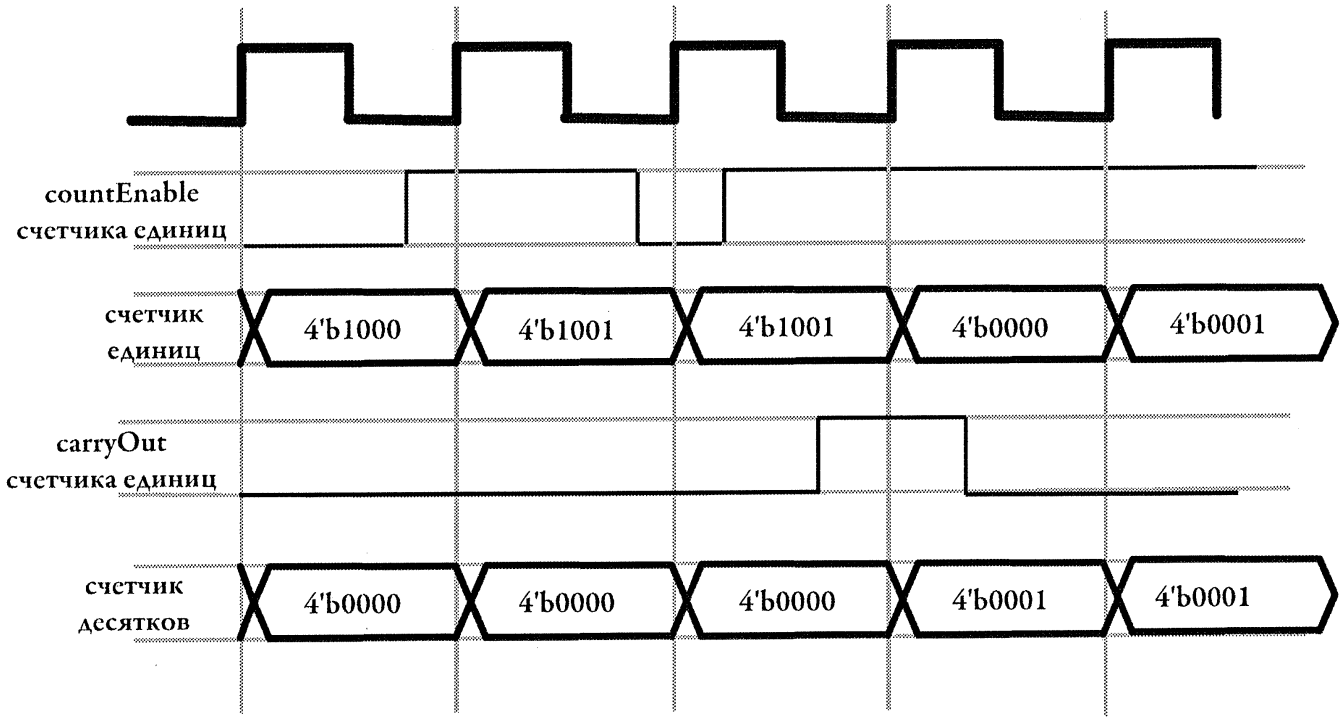
```
1   module zeroTo99
2     (input logic ck, r,
3      countEnable,    // countEnable для первой декады
4      output logic carryOut,// перенос из декады десятков
5      output logic [3:0] DCBA10, DCBA1); // выходы FF двух декад
```

Вот временна́я диаграмма работы модуля:

---

[37] Счетчик, значения которого представляются двумя десятичными цифрами (от 0 до 99).

[38] Декадный счетчик – счетчик, считающий от 0 до 9.




**5.8. Отправитель CRC.** Циклический избыточный код (Cyclic Redundancy Check – CRC) – широко распространенный метод обнаружения ошибок, возникающих при передаче данных. Отправляемая последовательность битов трактуется как двоичный многочлен; при передаче данных вычисляется остаток от деления соответствующего многочлена на некоторый фиксированный многочлен; в качестве контрольной суммы приемнику посылается дополнение вычисленного остатка[39]. Зная контрольную сумму, приемник может определить, возникла ли при передаче данных ошибка.

От вас требуется по потоку из одиннадцати последовательно передаваемых битов вычислить контрольную сумму. Представленный в примере генератор CRC-кода – один из используемых в USB-2.0. Напишите аппаратный поток, реализующий эту функциональность с использованием операторов `always_comb`, `always_ff` и `assign` (без вентилей). Разработайте тестовое окружение для проверки корректности работы.

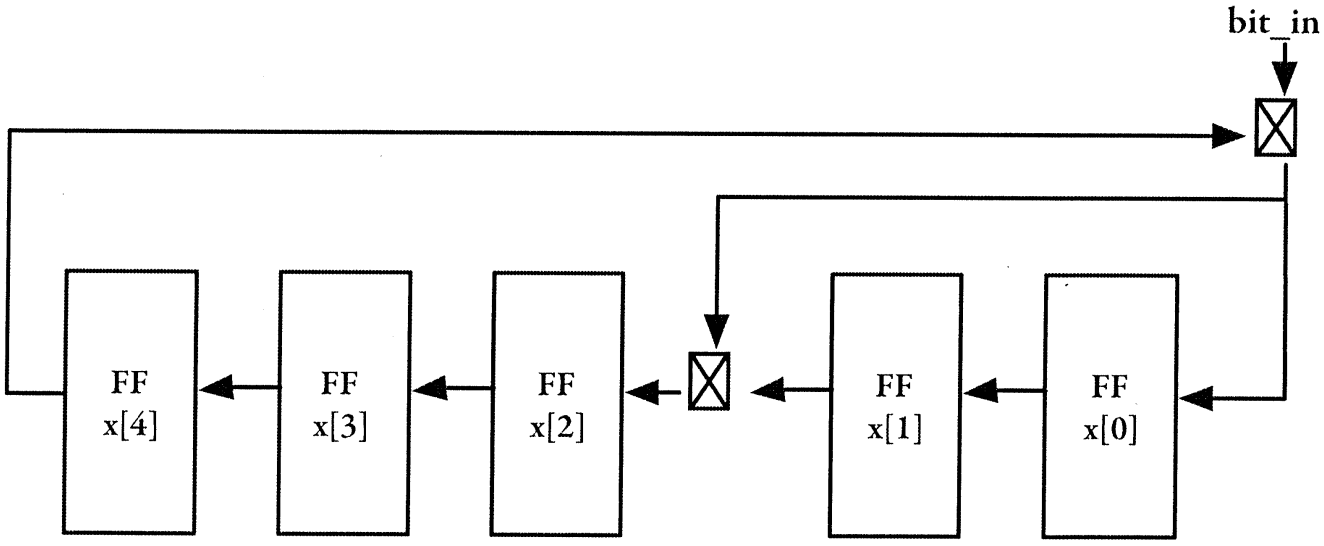
Ниже показана схема так называемого CRC-калькулятора. Символ ⊕ означает XOR, т. е. исключающее ИЛИ[40].

В начале передачи все триггеры CRC-калькулятора устанавливаются в 1. На последующих тактах биты сообщения передаются калькулятору (по одному биту за такт; старший идет первым). Биты также передаются приемнику, но здесь мы это опускаем. После того как одиннадцать бит отправлены, остаток (хранящийся в триггерах калькулятора) копируется в отдельный сдвиго-

---

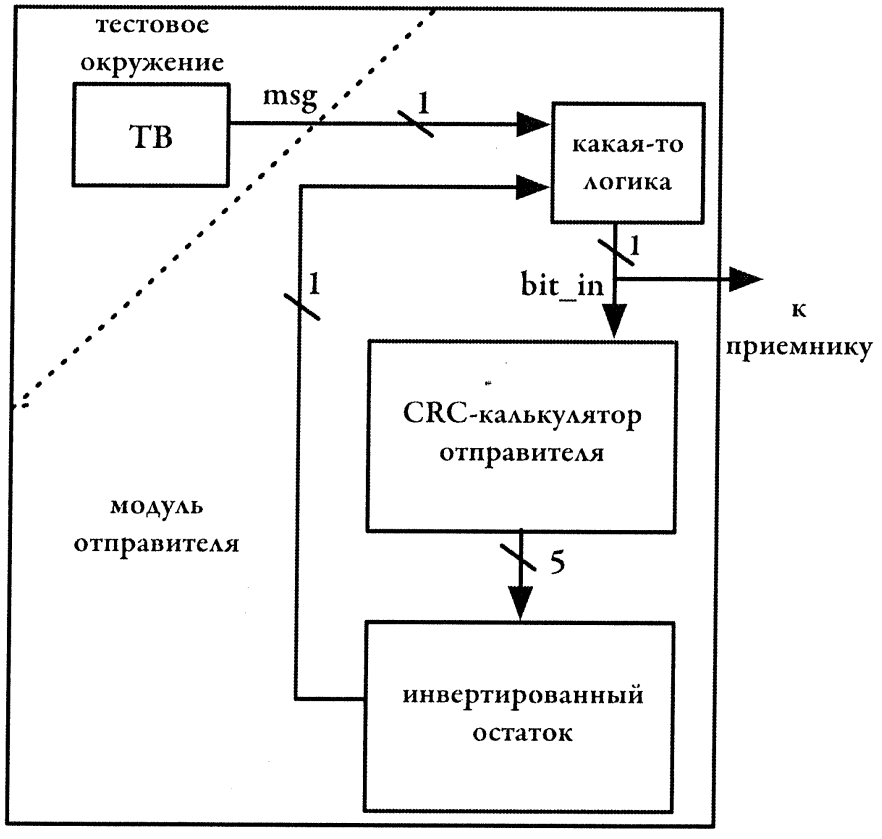
[39] Более подробно о CRC-кодировании и других арифметических «фокусах» можно прочитать в книге *Уоррен Г. С. мл.* Алгоритмические трюки для программистов. М.: Вильямс, 2014. 512 с. (в оригинале: *Warren H. S., Jr.* Hacker's Delight.)

[40] Представленная схема соответствует многочлену $x^5+x^2+1$.




вый регистр (на схеме не показан) в инвертированной форме. Биты регистра последовательно, начиная со старшего, подаются на вход калькулятору (и отправляются приемнику). После их передачи в калькуляторе должно сформироваться так называемое остаточное значение – 5'b01100.

Обратите внимание, что при инвертировании остатка состояние триггеров CRC-калькулятора не изменяется – инвертируется копия. После передачи последнего бита сообщения старший бит инвертированного остатка подается в калькулятор на следующем такте.



Завершая описание: приемник устроен аналогично, за тем исключением, что он не инвертирует остаток, а просто получает биты, переданные отправителем (сообщение и контрольную сумму). Если остаточное значение получается равным 5'b01100, значит, сообщение получено правильно.

Напишите тестовое окружение, посылающее следующее 11-битное сообщение CRC-калькулятору (старший, самый левый бит передается первым):

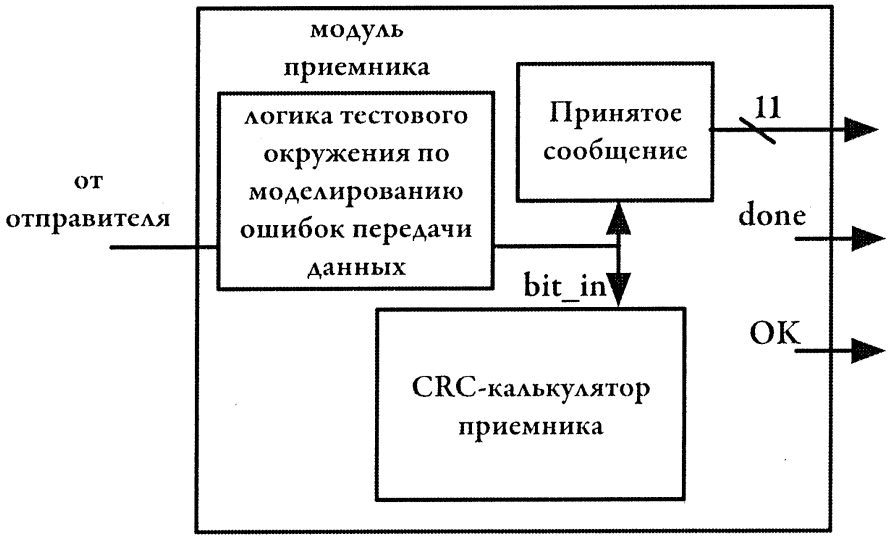
```
11'b0101_1100_101
```

Остаток, вычисленный для этого сообщения, x[4:0] = 5'b00011[41], инвертируется и отправляется, как было описано выше. Для упрощения отладки приведем первые значения x: 5'b11111 (начальное значение), 5'b11011, 5'b10110, 5'b01001[42], ...

Для вывода значений (чтобы проверить их правильность) используйте процедуру `$monitor` или `$display`.

**5.9. Приемник CRC.** Справились с предыдущей задачей? Вот вам еще одна.

Приемник инициализируется одновременно с отправителем и имеет такое же начальное состояние CRC-калькулятора – 5'b11111. Он принимает сообщение и контрольную сумму: биты сообщения (и только они) запоминаются в регистре «Принятое сообщение»; все биты (и биты сообщения, и биты контрольной суммы) подаются в калькулятор. По завершении работы в нем должно оказаться остаточное значение x=5'b01100. Установка выхода done говорит о том, что сообщение принято полностью (не важно, правильно или нет); если принятое сообщение правильно (остаточное значение равно ожидаемому), устанавливается сигнал OK.




Сделайте так, чтобы тестовое окружение инвертировало значения одного или двух передаваемых битов; проверьте, обнаружит ли приемник ошибки.

Для вывода значений используйте процедуру `$monitor` или `$display`.

---

[41] В оригинале ошибочно указан остаток 5'b01011.

[42] В оригинале приведены следующие значения: 5'b11111, 5'b11011, 5'b01001, 5'b10010,...

# Глава 6

# Интерфейсы

В синхронизированном мире, где проектируется аппаратура уровня регистровых передач, основными вычислительными объектами являются аппаратные потоки, или конечные автоматы с трактом данных, представленные в главе 5. Аппаратный поток выполняет вычисления, отвечая на запрос внешнего источника, которым может быть сенсорный интерфейс (например, кнопка) или другой аппаратный поток.

В этой главе мы познакомимся с моделированием взаимодействия между несколькими аппаратными потоками. Затем будет представлена конструкция языка SystemVerilog, предназначенная для описания интерфейсов.

## 6.1. Взаимодействующие аппаратные потоки

Организация интерфейсного взаимодействия подразумевает проектирование двух аппаратных потоков таким образом, чтобы они могли надежно обмениваться информацией. На первый взгляд может показаться, что с этой задачей справляются порты модулей, но соединение двух потоков не означает создание одних лишь линий для пересылки данных. Говоря об интерфейсах, мы имеем в виду широкий спектр функций, в том числе реализацию протокола, разбиение данных на пакеты, обнаружение и исправление ошибок, а в некоторых случаях и синхронизацию тактовых сигналов.

### 6.1.1. Организация потоков

Вычислительная система может состоять из множества совместно работающих потоков. На рис. 6.1 показано, что они могут быть организованы для направленного распространения информации, когда каждый поток передает обработанную информацию следующему потоку (или потокам) для дальнейшей обработки. Взаимодействие между любыми двумя такими потоками можно описать как взаимодействие между *производителем* и *потребителем*. Это значит, что есть один поток, который порождает информацию, и другой поток, который ее потребляет. На рисунке поток A является производителем, а поток B – потребителем. Но во взаимодействиях с потоками C и D поток B выступает в роли производителя, а потоки C и D – в роли потребителей. Таким образом, взяв любую

стрелку, мы можем считать, что на одном ее конце находится производитель, а на другом – потребитель информации, циркулирующей в системе.

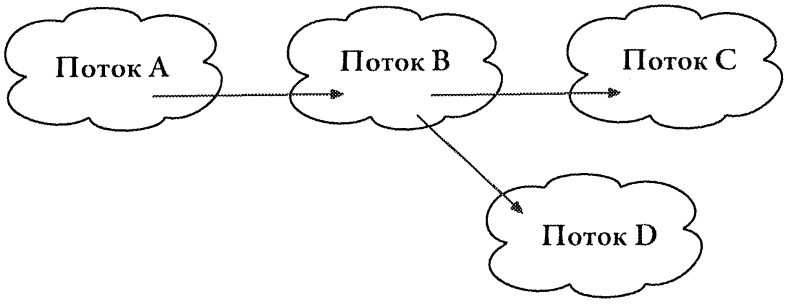



**Рис. 6.1.** Четыре взаимодействующих потока

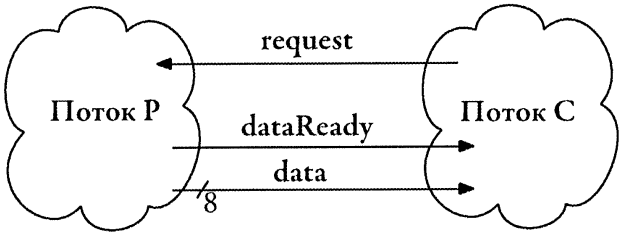



**Рис. 6.2.** Два взаимодействующих потока

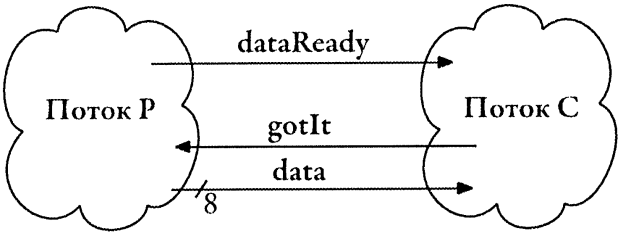



**Рис. 6.3.** Альтернативное взаимодействие потоков

Теперь рассмотрим подробно одну из стрелок на этом рисунке, соответствующую одному из возможных взаимодействий между двумя потоками (см. рис. 6.2). Система состоит из двух потоков, P и C, изображенных облаками. На рисунке показаны лишь видимые извне соединения между потоками. Поток C запрашивает у производителя данные, устанавливая сигнал `request`. Когда поток P подготовит данные, он помещает их на линии `data` и устанавливает сигнал `dataReady`, показывая, что данные на этих линиях значимы. При такой организации поток P является производителем, а поток C – потребителем, потому что данные перетекают от P к C.

На рис. 6.3 показана альтернативная организация взаимодействия потоков. Здесь поток P подготавливает данные и устанавливает сигнал `dataReady`, говорящий потоку C, что тот может их прочитать. После загрузки данных в регистр поток C устанавливает сигнал `gotIt`, чтобы поток P узнал, что можно подготавливать данные дальше. Хотя данные по-прежнему перетекают от P к C, в этом случае активную роль играет поток P: он просит поток C принять данные, а C отвечает, что исполнил просьбу.

Единственное отличие между этими двумя вариантами заключается в том, кто играет активную роль в передаче данных: либо поток P, либо поток C. Тот поток, который посылает запрос, запрашивает вычислительную услугу у другого потока. На рис. 6.2 C просит P подготовить для него данные. Например, поток P может снять показание датчика, обработать полученное значение, а затем отправить его C. На рис. 6.3 поток P просит C принять данные. Поток C может принять 8-битные данные и, например, преобразовать их для последо-

вательной передачи. В этом случае можно считать, что поток C предоставляет услугу потоку P.

При проектировании логических схем линии типа `dataReady` и `gotIt` называются *сигналами, или сигнальными переменными*. Сигналы передают информацию между потоками с единственной целью – синхронизировать их действия. В данном случае цель состоит в передаче данных от одного потока другому. Последовательность действий, определяющих взаимодействие потока с прочими частями вычислительной системы, называется *протоколом*. На рис. 6.2 протокол мог бы быть таким, что в ответ на запрос потока C поток P помещает новые подготовленные данные на выход `data` и устанавливает сигнал `dataReady` в течение одного тактового интервала. Поток C, видя сигнал `dataReady`, должен загрузить данные в свой регистр.

## 6.1.2. Синхронность

Два взаимодействующих потока могут находиться в одном домене синхронизации, т. е. синхронизироваться одним тактовым сигналом, а могут быть и в разных. В первом случае можно считать, что все сигналы между потоками синхронизированы с тактовым сигналом. Это следует из предположения о синхронности и определения домена синхронизации (см. главу 4). Если же потоки находятся в разных доменах, то отправляемый сигнал должен быть синхронизирован с тактовым сигналом принимающего потока.

Входные сигналы потока можно отнести к трем категориям.

- *Синхронный сигнал* – вход аппаратного потока является синхронным относительно этого потока, если этот сигнал был сгенерирован в том же домене синхронизации. Иначе говоря, вход удовлетворяет требованиям предположения о синхронности по отношению к текущему потоку.
- *Асинхронный сигнал* – вход аппаратного потока является асинхронным относительно этого потока, если этот сигнал был сгенерирован в другом домене синхронизации. Таким образом, для этого входа не удовлетворяется предположение о синхронности.
- *Синхронизированный сигнал* – вход из другого домена синхронизации, прошедший через триггер-синхронизатор и ставший в результате синхронным с локальным доменом. Синхронизированный сигнал удовлетворяет требованиям предположения о синхронности в локальном домене.

Единственный способ добиться, чтобы аппаратный поток надежно отвечал на асинхронный входной сигнал, – синхронизировать его. После этого синхронизированный сигнал не отличается от синхронного и становится частью локального домена синхронизации.

Выходы потока являются синхронными. В автоматах Мура выходы есть функции состояния; они обновляются по тактовому сигналу. В автоматах Мили выходы есть функции состояния и входов; они синхронны, поскольку входы автомата являются синхронными или синхронизированными.

## 6.2. Синхронные взаимодействия между потоками

Синхронность, основанная на синхронных и синхронизированных сигналах, – фундаментальное условие совместной работы двух аппаратных потоков. Предположение о синхронности гарантирует, что требования тактового сигнала и триггеров к времени выполнятся. Однако чтобы информацию можно было передать между двумя или более потоками, их состояния должны быть синхронизированы таким образом, чтобы поток, желающий прочитать данные на своем входе, знал, когда на нем появляется новая информация. То есть поток, подготавливающий данные, должен находиться в состоянии (или состояниях), когда данные присутствуют на его выходе, а принимающий поток – в состоянии, когда точно знает, что на входе присутствуют значимые данные и что их, следовательно, можно загрузить в регистр. В этой точке состояния обоих потоков становятся синхронизированными.

Поскольку состояния потоков синхронизированы, передача называется *синхронной транзакцией*.

Проблема такого рода синхронизации заключается в том, что аппаратный поток не знает состояния другого потока, если только нет специального сигнала, сообщающего эту информацию. В примере потока `sumItUp` из главы 5 такими сигналами были `go_l` и `done`. `go_l` – входной сигнал от другого потока, сообщающий, что на входной линии присутствует первое из последовательности значений; `done` – выходной сигнал потока `sumItUp`, сообщающий, что значение на выходе `sum` является суммой всей последовательности байтов. На основе текущего состояния потока сигнал `go_l` информирует поток `sumItUp` о том, что поток, подготавливающий последовательность байтов, находится в состоянии, когда на линию данных помещен первый байт, а `done` информирует зависящий от него поток, что `sumItUp` находится в состоянии, когда на выходную линию помещен окончательный результат.

Эти сигналы позволяют потокам узнать о состояниях друг друга, например узнать, что данные доступны. Взаимосвязь сигнальных линий и линий данных – это часть протокола, поддерживающая синхронизацию состояний аппаратных потоков для корректного обмена информацией между ними.

Основной элемент диаграммы состояний синхронизации двух потоков – это *состояние ожидания*. На рис. 6.4 показано состояние ожидания, переход из которого ведет в него же, если входной сигнал сброшен. (Сигнал показан толстой серой стрелкой, чтобы его можно было отличить от переходов состояний.) Когда сигнал устанавливается, поток переходит в следующее состояние. Таким образом, поток ждет, пока другой поток не установит этот сигнал. С переходами на рис. 6.4 могли бы быть ассоциированы выходы, разрешающие какие-то действия внутри потока. Часто выход (переход) из состояния ожидания сопровождается загрузкой в регистр дан-

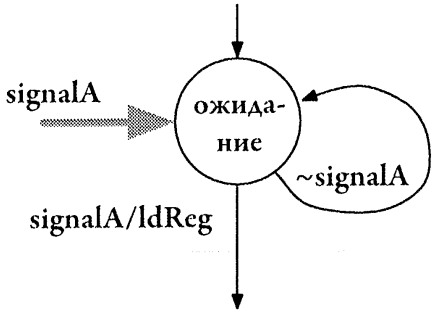



**Рис. 6.4.** Состояние ожидания

ных, находящихся на входной линии потока, что на рис. 6.4 проиллюстрировано точкой управления ldReg, активируемой при выходе из состояния ожидания.

На рис. 6.5 показан расширенный многопоточный контекст состояния ожидания. На рисунке находятся два потока, A и B, причем B – по существу, копия потока на рис. 6.4. Поток A устанавливает сигнал signalA, находясь в одноименном состоянии. Будем считать, что наличие signalA означает, что на выходе data потока A имеется новое значение, которое поток B может прочитать.

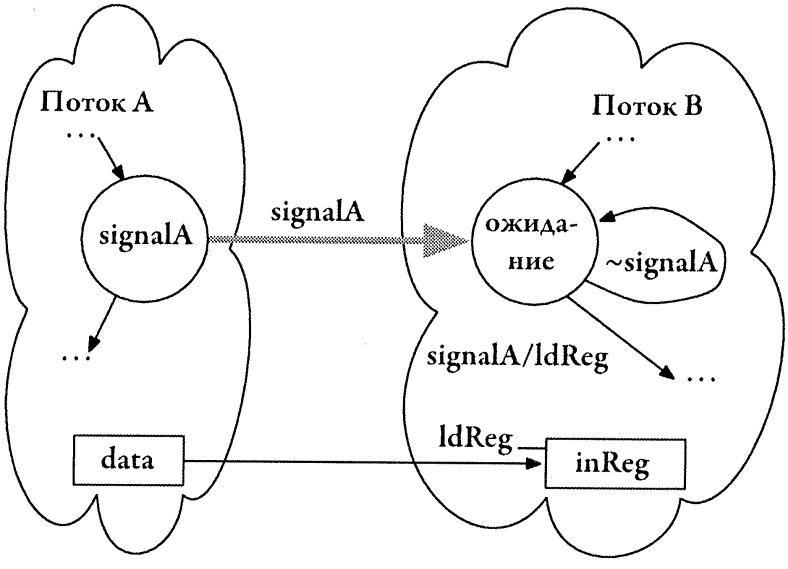



**Рис. 6.5.** Расширенный контекст состояния ожидания

Таким образом, совместная работа двух потоков заключается в том, что поток B входит в состояние ожидания и, поскольку signalA сброшен, остается в нем. Позже поток A входит в состояние signalA и устанавливает выходной сигнал signalA. По следующему фронту тактового сигнала оба потока выходят из своих состояний и переходят в следующие за ними. Одновременно с этими переходами в регистр inReg потока B загружаются данные с выхода data потока A. Это типичное взаимодействие двух потоков, передающих данные: поток-производитель (A) подготавливает новые данные и сигнализирует потоку-потребителю (B) о том, что значение можно загрузить в регистр.

Важно отметить, что основанная на состояниях работа обоих потоков стала синхронизированной, что дало возможность передавать данные между ними. Иногда мы будем называть это *рандеву* двух потоков. Они «встретились» в двух взаимно согласованных состояниях – стали синхронизированными по состоянию – и передали друг другу информацию. Интерфейсы, обеспечивающие такую передачу данных, необходимо проектировать совместно для обоих потоков.

Интересно отметить, что в этом примере поток A реализует одну половину передачи, а поток B – другую. Поскольку оба потока находятся в одном домене синхронизации и синхронизированы по состоянию (один находится в состоянии ожидания wait, другой – в состоянии signalA), то передача данных становится возможной: в регистре data потока A находятся значимые данные,

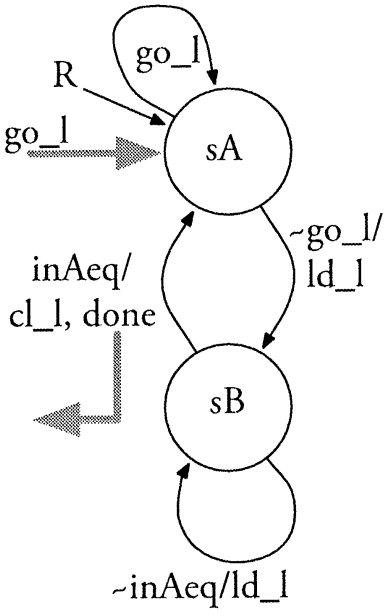



**Рис. 6.6.** Диаграмма состояний потока sumItUp

и поток B загружает эти данные в свой регистр `inReg`. В данном случае предположение о синхронности применяется к комбинации потоков A и B.

В этом примере есть скрытое предположение: поток-потребитель (ожидающий сигнала) должен находиться в состоянии ожидания, когда поток A переходит в состояние `signalA` и устанавливает соответствующий сигнал. Если он не находится в состоянии ожидания, а сигнал остается установленным только на протяжении одного тактового интервала, то этот сигнал будет пропущен. И информация, которую он нес, оказывается потеряна, потому что после выхода потока A из состояния `signalA` сигнал заново не устанавливается. Даже если сигнал остается установленным в течение нескольких тактов, он все равно будет пропущен, если поток-потребитель так и не перейдет в состояние ожидания за это время. В итоге данные просто оказались потерянными.

Вернемся к примеру потока `sumItUp`. В нем протокол, в частности, подразумевает, что сигнал `go_l` установлен в течение промежутка времени между двумя соседними тактовыми сигналами, а следующий сигнал `go_l` не может появиться раньше, чем на линию данных будет установлено нулевое значение. Диаграмма состояний, которая для удобства повторена на рис. 6.6, гарантирует, что, будучи запущенным, поток `sumItUp` не пропускает прием данных. Отметим также, что сброс переводит поток в состояние ожидания входного сигнала, так что даже в этой ситуации данные не теряются.

## 6.2.1. Двустороннее ожидание

В примере на рис. 6.5 было показано одностороннее ожидание, которое применяется в ситуации, когда разработчик может гарантировать, что один поток будет находиться в состоянии ожидания, когда другой посылает сигнал. В сложной системе не всегда легко дать такую гарантию. Решить проблему можно, воспользовавшись двусторонним ожиданием для синхронизации состояний двух потоков. Идея этого подхода в том, что для продолжения работы оба потока должны оказаться в состоянии ожидания вне зависимости от того, кто вошел в него раньше.

На рис. 6.7 показаны два потока и их состояния ожидания: $A_W$ в потоке A и $B_W$ в потоке B. Выходные значения, устанавливаемые в каждом из этих состояний (соответственно `readyA` и `readyB`), являются входными значениями для другого состояния. Протокол здесь следующий: когда поток A находится в состоянии $A_W$, он устанавливает сигнал `readyA`, а состояние $B_W$ потока B приводит к установке сигнала `readyB`. Сигналы `readyA` и `readyB` установлены только в этих состояниях. С учетом условия нахождения в состоянии ожидания из рис. 6.7, какой бы по-

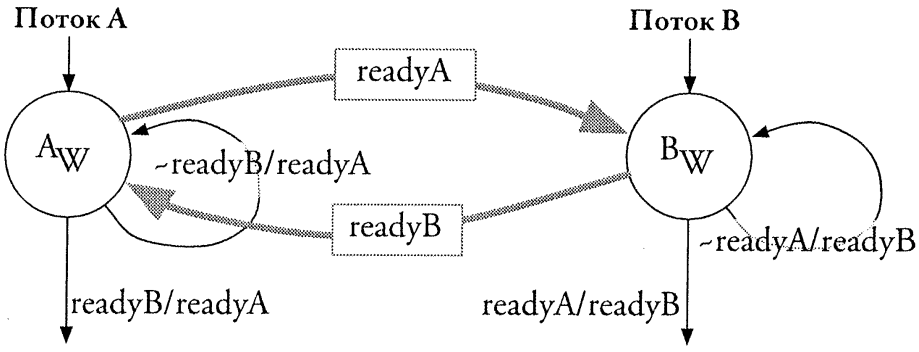



**Рис. 6.7.** Два аппаратных потока, прибывающих на рандеву, безотносительно порядка прибытия

ток ни вошел в состояние $X_W$ первым (под X понимается A и B), он будет ждать другого. Когда другой поток также войдет в состояние $X_W$, оба потока выйдут из состояний $X_W$ по одному и тому же фронту тактового сигнала. Даже если оба потока войдут в состояние $X_W$ одновременно, выйдут они по одному фронту.

Этот подход избавляет от сложностей синхронизации при использовании одностороннего ожидания.

Временна́я диаграмма двустороннего ожидания показана на рис. 6.8. Поток A прибывает на рандеву первым и устанавливает сигнал `readyA`. В следующем такте на рандеву прибывает B и устанавливает сигнал `readyB`. В этот момент поток A находится в состоянии $A_W$, а поток B – в состоянии $B_W$, и состояния обоих потоков синхронизированы. На рисунке эта совокупность состояний обозначена S2. По следующему фронту тактового сигнала оба автомата переходят в следующие состояния. Гарантированное нахождение обоих автоматов в состоянии ожидания позволяет передавать данные между потоками в любом направлении.

Как бы потоки ни прибывали на рандеву – с помощью одно- или двустороннего ожидания – важно, что они «встретились» и их состояния синхронизированы. Это значит, что каждый поток переходит из состояния, известного другому. На этом переходе и, как мы увидим ниже, возможно, также и на следующих, потоки могут обмениваться информацией с помощью регистровых передач.

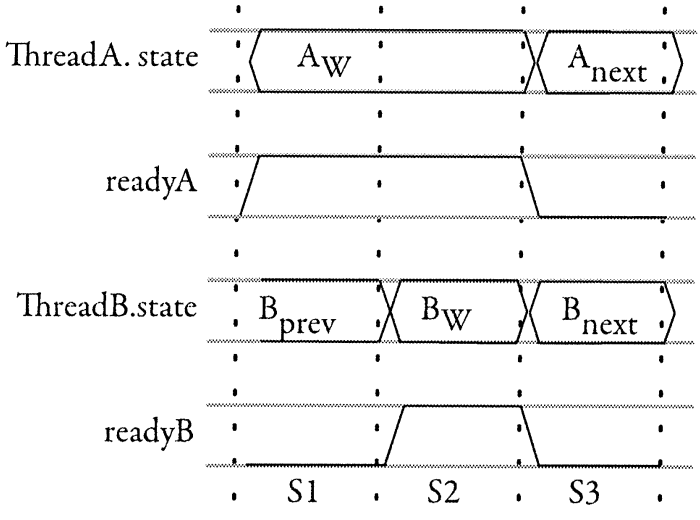



**Рис. 6.8.** Временна́я диаграмма работы рандеву

## 6.2.2. Пошаговая синхронизация

После того как два потока синхронизированы по состоянию, они могут при некоторых условиях продолжить быть синхронизированными. Это называется *пошаговой синхронизацией* (lock-step synchronization). Пошаговая синхронизация начинается в момент, когда состояния синхронизированы, и продолжается до тех пор, пока каждый поток имеет представление о состоянии другого.

Рассмотрим потоки A и B на рис. 6.9; они прибыли на рандеву, в состояния $A_W$ и $B_W$. Отметим, что на диаграмме состояний каждого потока за этими состояниями следуют еще три. Поскольку потоки находятся в одном домене синхронизации, мы можем быть уверены, что они перейдут в эти состояния одновременно: когда A находится в состоянии $A_3$, B находится в состоянии $B_7$, точно так же соответствуют друг другу $A_4$, $B_8$ и $A_5$, $B_9$. Таким образом, оба потока проходят по диаграмме синхронно.

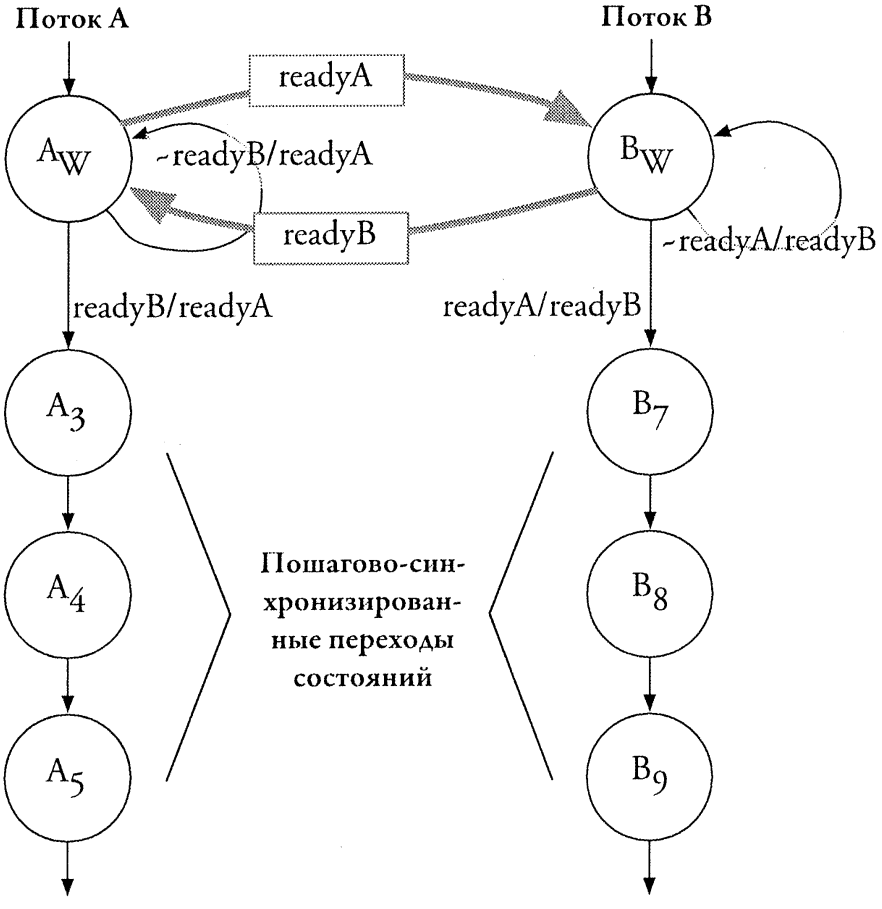



**Рис. 6.9.** Пошаговая синхронизация

Поскольку мы знаем, что потоки окажутся в состояниях $A_4$ и $B_8$ одновременно, то в состоянии $A_4$ поток A мог бы поместить данные на некоторые линии, соединяющие потоки, а поток B мог бы загрузить значения с этих линий в свой регистр без каких-либо управляющих сигналов. Благодаря пошаговой синхронизации передачу информации между потоками можно спроектировать, не вводя дополнительных сигнальных линий.

Пошаговая синхронизация может продолжаться до тех пор, пока расстояние (количество тактов) от начального состояния рандеву можно определить статически. Пошаговая синхронизация может быть нарушена при использовании циклов с условиями на данные, условных операторов с разным числом состояний в ветвях и операторов ожидания.

Считается, что потоки находятся на рандеву все время, пока они сохраняют пошаговую синхронизацию.

В качестве более сложного примера рассмотрим две диаграммы состояний для потоков P и Q, показанные на рис. 6.10. Он преследует всего одну цель – продемонстрировать хитрые ситуации, которые могут возникать при рандеву и пошаговой синхронизации. Пример, очевидно, искусственный.

Буквой R обозначены состояния сброса в обеих диаграммах. Все состояния и переходы помечены. В системе есть два состояния ожидания: $P_W$ и $Q_W$, в каждом из них поток ожидает сигнала из состояния, находящегося в начале соответствующей жирной серой стрелки. Таким образом, поток P, находясь в состоянии $P_2$, устанавливает сигнал, которого ждет поток Q, находящийся в состоянии $Q_W$. Петля H соответствует ожиданию сигнала; при установленном сигнале происходит переход A. Аналогично поток P, находясь в состоянии $P_W$ (переход e), ожидает сигнала, устанавливаемого в состоянии $Q_4$. Петля D в состоянии $Q_3$ – это цикл, управляемый данными; нельзя заранее сказать, сколько в нем будет итераций.

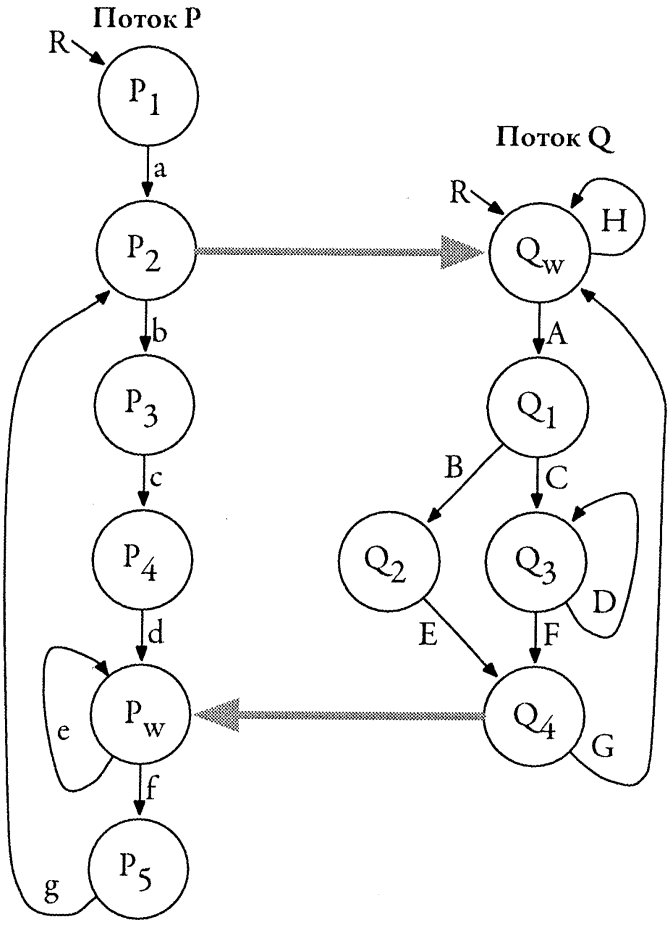



**Рис. 6.10.** Две взаимодействующие диаграммы состояний

Теперь возникает вопрос: когда эти два автомата смогут передавать информацию – на каких переходах и в каком направлении? После сброса поток Q остается в состоянии ожидания $Q_W$, пока поток P не перейдет в $P_2$. В этот момент оба потока прибыли на рандеву и по следующему фронту тактового сигнала окажутся в режиме пошаговой синхронизации. Обозначим эту ситуацию <b, A> – поток P переходит по дуге b, а поток Q – по дуге A по одному и тому же фронту. В этот момент каждый автомат может читать информацию от другого и загружать ее в свой регистр, как было показано на рис. 6.5.

Еще одно рандеву имеет место, когда поток P находится в состоянии $P_W$, а поток Q – в состоянии $Q_4$. Отметим, что P достигнет $P_W$ через три такта после

пребывания в $P_2$. Если поток Q выйдет из $Q_1$ по дуге B, то придет в $Q_4$ тоже через три такта. Но Q может выйти из $Q_1$ и по дуге C. В этом случае мы не знаем, когда он доберется до $Q_4$, потому что в D имеется цикл. Как бы то ни было, в состояниях $P_W$ и $Q_4$ имеет место рандеву и, значит, на переходах из этих состояний (<f, G>) возможен обмен данными в любом направлении.

Отметим также, что поток Q вернется в состояние $Q_W$ и будет оставаться в нем, дожидаясь возврата P в состояние $P_2$. Таким образом, эти два автомата не могут оказаться рассогласованными, т. е. они либо будут переходить в свои состояния $X_W$ одновременно, либо один автомат окажется в состоянии ожидания раньше, чем другой, и перейдет в состояние сигнализации. Следовательно, состояния двустороннего ожидания, показанные на рис. 6.9, здесь не нужны.

Имеются также состояния, где возможна передача данных в одном направлении. Рассмотрим состояния $P_3$ и $Q_1$. Поскольку из $Q_1$ возможны два перехода, это, по существу, конструкция `if-then-else`. Если условие равно TRUE, то происходит переход по дуге B, иначе по дуге C. Но мы знаем, что одновременно имеют место либо c и B, либо c и C, поскольку эти переходы удалены на фиксированное количество шагов от точки рандеву. Можно заключить, что если поток Q переходит по дуге B, то P переходит по дуге c, но не наоборот. То есть Q мог бы перейти и по дуге C. Поэтому поток Q может читать информацию от P, но не наоборот.

Более пристальное изучение показывает, что в потоке Q может существовать информация, которую поток P мог бы прочитать при переходе c. Например, данные в состоянии $Q_1$ потока Q могли бы быть только функцией текущего состояния и не зависеть от условия, которое выбирает следующее состояние. В таком случае поток P мог бы прочитать эти данные.

Потоки, взаимодействующие подобным образом, проектируются вместе, и разработчик может понять, какие значения можно передавать и когда.

## 6.3. Пример шины SimpleBus

В этом разделе рассматривается простой протокол шины, SimpleBus, на примере которого в дальнейшем будет демонстрироваться проектирование интерфейсов, основанное на состояниях ожидания, синхронизации по состоянию и пошаговой синхронизации.

### 6.3.1. Определение протокола SimpleBus

На рис. 6.11 показаны интерфейс процессора (сверху), простой интерфейс памяти (снизу) и их соединение. Информация между интерфейсами передается по линиям, составляющим шину. Поток процессора (не показан) запрашивает транзакцию чтения или записи через шину, и интерфейс процессора передает этот запрос на шину. В свою очередь, поток интерфейса памяти наблюдает за линиями шины и, когда появляется запрос, просит поток памяти (не показан) прочитать значение из памяти или записать его в память. Интерфейс процессора является ведущим (master) для шины, т. е. он инициирует ее работу.

Значения линий адреса (address), начала операции (start) и чтения (read) устанавливает только интерфейс процессора; это показано однонаправленными стрелками на рис. 6.11. Информацию с этих линий получают только потоки интерфейса памяти. data и dataValid – общие линии шины (на что указывают двунаправленные стрелки), значения на которые могут устанавливать и интерфейс процессора, и интерфейс памяти. Если ни тот, ни другой это не сделали, то на линиях будет высокий импеданс (значение z).

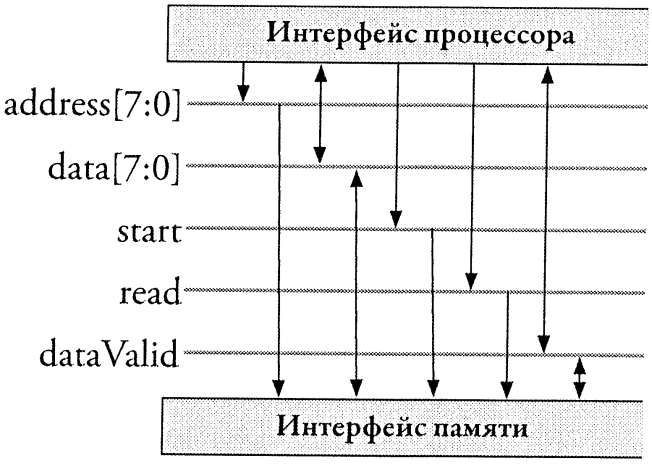



**Рис. 6.11.** Соединение процессора и памяти с помощью простой шины

Шина может выполнять транзакции двух типов: чтение и запись. Тип указывается сигналом read. Если этот сигнал установлен, то интерфейс памяти поместит на линии data прочитанное из памяти значение и установит сигнал dataValid, чтобы сообщить интерфейсу процессора о том, что на линии data есть значение. Если же тип транзакции – запись, сигнал read не устанавливается, интерфейс процессора помещает на линию data значение, которое нужно записать в память, а также устанавливает сигнал dataValid, чтобы уведомить память о наличии данных. Линия address мультиплексируется: старшая и младшая 8-битные части адреса посылаются на двух последовательных тактах.

На рис. 6.12 приведена временна́я диаграмма протокола SimpleBus для транзакции чтения. Транзакция начинается в состоянии S1 (метки состояний

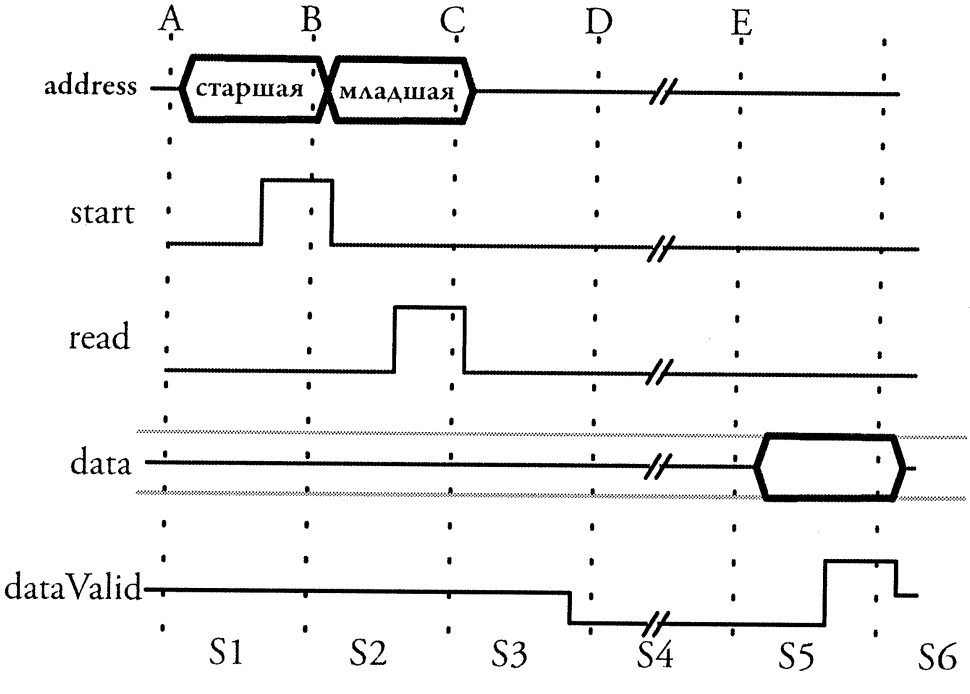



**Рис. 6.12.** Временна́я диаграмма протокола SimpleBus

находятся в нижней части диаграммы), когда процессор помещает старшие биты адреса на адресную шину и устанавливает сигнал start. В состоянии S2 процессор помещает на адресную шину младшие биты адреса и устанавливает сигнал read, если это операция чтения из памяти (в случае операции записи сигнал read не устанавливается). Память, в свою очередь, ждет появления сигнала start и, увидев его, считывает старшую половину адреса с адресной шины по фронту тактового сигнала B. В следующем состоянии (S2) она без какого-либо дополнительного сигнала считывает младшую половину адреса по фронту тактового сигнала C – в силу пошаговой синхронизации. Ориентируясь на сигнал read, память определяет следующее состояние.

Поскольку на рис. 6.12 показан цикл чтения шины, то в состоянии S3 память меняет значение dataValid с z на 0. Этот сигнал сообщает процессору, когда можно читать формируемые памятью значения, помещенные на линию data. Его не следует устанавливать, пока интерфейс памяти не будет готов поместить данные на шину. В состоянии S5 данные помещены на шину и установлен сигнал dataValid. Отметим, что S4 является промежуточным состоянием, которое может использоваться в совокупности со многими другими состояниями. В данном случае в течение S4 процессор ждет, пока память прочитает запрошенное значение и поместит его на линию данных. (В случае операции записи память может ожидать, пока процессор не поместит значение на линию data и не установит сигнал dataValid.) В состоянии S6 линия data и сигнал dataValid сбрасываются; в этом состоянии может начаться следующий цикл работы шины (для этого процессор должен сформировать значение на линии address и установить сигнал start).

При записи в память есть только одно отличие в использовании управляющих сигналов: процессор не устанавливает сигнал read в состоянии S2. Семантика остальных линий остается прежней, однако теперь *процессор* формирует значение на линии data и устанавливает сигнал dataValid, а память записывает это значение.

На рис. 6.13 протокол SimpleBus показан более детально. Здесь мы видим потоки процессора и интерфейса процессора, а также потоки памяти и интерфейса памяти. Потоки соединены сигналами и линиями данных. Обратите внимание, что соединение между потоками интерфейсов содержит линии, показанные на рис. 6.12. Соединения между остальными потоками описываются в следующих разделах.

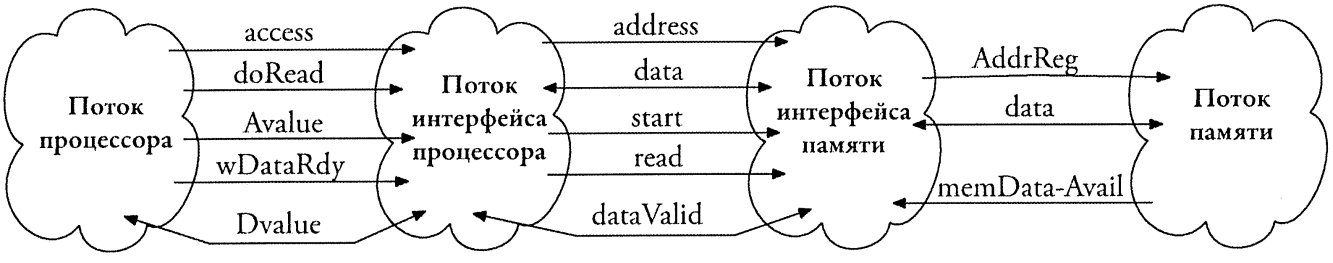



**Рис. 6.13.** Соединения между потоками в примере SimpleBus

## 6.3.2. Поток интерфейса процессора (ведущий компонент)

Поток интерфейса процессора должен формировать значения на линиях address, start и read, а также формировать и считывать значения, установленные на линиях data и dataValid. Кроме того, поскольку длина адреса составляет 16 бит, а линия address 8-битная, передача мультиплексируется: старшая и младшая части адреса передаются на двух тактах (в состояниях S1 и S2 на рис. 6.12). На рис. 6.14 показан тракт данных для интерфейса процессора.

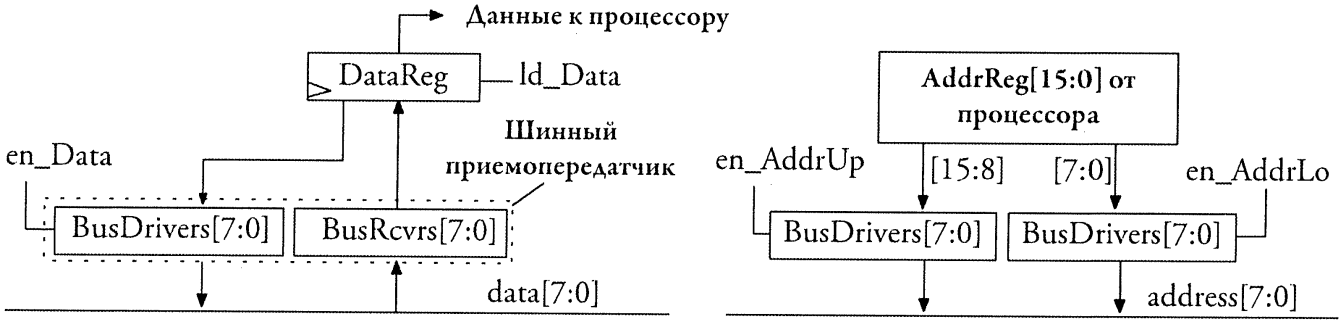



**Рис. 6.14.** Тракт данных и точки управления интерфейса процессора

Адрес хранится в регистре AddrReg. Его старшие и младшие части соединены с двумя разными тристабильными драйверами. Для активации драйверов используются сигналы en_AddrUp и en_AddrLo; когда драйвер активирован, он помещает соответствующее значение на линию address.

Работа с данными устроена немного сложнее: либо в регистре DataReg хранится значение, которое нужно поместить на шину (в случае операции записи), либо в него загружается значение, прочитанное из памяти (в случае операции чтения). Загрузка в DataReg происходит, когда установлен сигнал ld_Data. Выход DataReg соединен с тристабильным драйвером, активируемым сигналом en_Data; этот сигнал устанавливается для записи значения регистра на линию data. На рисунке также показан 8-битный приемник, который соединяет линию data с входом DataReg. Драйвер и приемник для линии data являются частью шинного приемопередатчика.

На рис. 6.15 показан конечный автомат потока интерфейса процессора, который управляет элементами тракта данных (и всем протоколом SimpleBus). Четыре его состояния обозначены MA–MD (от master A–D). Состоянием сброса является MA, а домен синхронизации тот же, что и для интерфейса процессора и интерфейса памяти, рассмотренного в следующем разделе. Входами конечного автомата, как это показано на рис. 6.13, являются:

- access – вход от потока процессора, который говорит, что поток интерфейса должен начать транзакцию на шине. Подразумевается, что регистр AddrReg уже содержит адрес памяти для чтения или записи;
- doRead – вход от процессора, сообщающий интерфейсу, что нужно выполнить либо чтение (если он установлен), либо запись (если он сброшен);

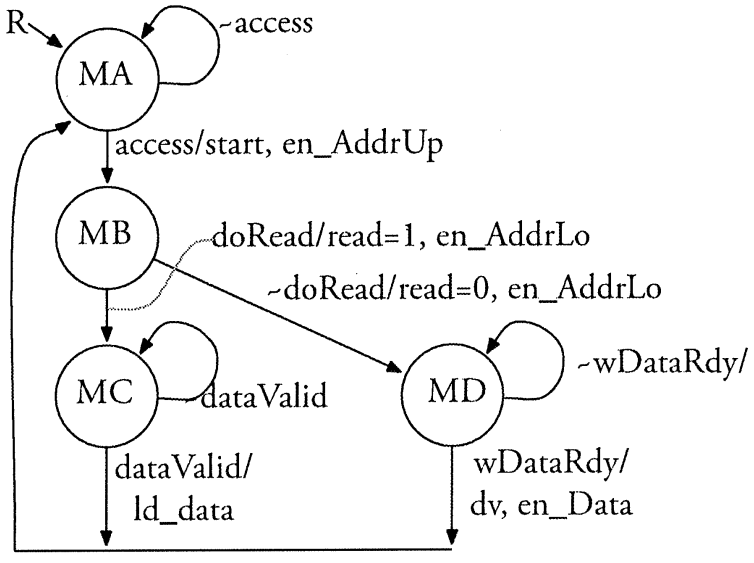



**Рис. 6.15.** Диаграмма состояний потока интерфейса процессора (драйвера шины)

- `wDataRdy` – вход от процессора, сообщающий интерфейсу, что данные для записи находятся в `DataReg`;
- `dataValid` – вход от шины, подтверждающий, что поток интерфейса памяти поместил данные на линию `data`, и, значит, интерфейс драйвера может загрузить их в `DataReg`.

Выходами того же конечного автомата являются:

- `en_AddrUp` – разрешает драйверу шины поместить старшие 8 бит `AddrReg` на линию `address`;
- `en_AddrLo` – разрешает драйверу шины поместить младшие 8 бит `AddrReg` на линию `address`;
- `en_Data` – разрешает драйверу шины поместить содержимое `DataReg` на линию `data`;
- `dv` – значение на линии `dataValid`;
- `start` – значение на линии `start`;
- `read` – значение на линии `read`.

Теперь можно объяснить, как работает поток интерфейса процессора. Прежде всего отметим, что MA, MC и MD – состояния ожидания, т. е. они имеют переход на самих себя и в них автомат ждет, когда другой поток установит внешний сигнал. В MA автомат ждет, когда процессор сообщит, что можно начинать транзакцию на шине; в MC – когда интерфейс памяти сообщит, что значение, прочитанное из памяти, находится на линии `data`; в MD – когда процессор сообщит, что значение для записи в память находится в регистре `DataReg`.

Рассмотрим состояния автомата подробнее. В MA автомат ждет сигнала для начала транзакции. Начав, он устанавливает сигнал `start` и разрешает драйверу поместить старшую половину адреса на линию `address` (`en_AddrUp`). В состоянии MB он помещает значение `doRead` на линию `read` и разрешает драйверу поместить младшую половину адреса на линию `address` (`en_AddrLo`). Кроме того, в MB принимается решение: в зависимости от сигнала `doRead` автомат переходит либо в состояние MC, либо в состояние MD.

В состоянии MC (операция чтения) автомат ждет, когда поток интерфейса памяти поместит прочитанное значение на линию data и сообщит об этом сигналом dataValid. При появлении dataValid автомат устанавливает ld_Data, и данные на линии загружаются в DataReg.

В состоянии MD (операция записи) автомат ждет, когда процессор поместит значение, подлежащее записи, в регистр DataReg и установит сигнал wDataRdy. При появлении wDataRdy автомат перемещает значение из регистра на линию (устанавливая en_Data) и устанавливает dv и en_DataValid. Из состояний MC и MD автомат возвращается в состояние MA, где ждет начала очередной транзакции, которая может начаться уже на следующем такте.

MA, MC и MD – это состояния ожидания; на их примере демонстрируется, как поток синхронизируется с другими потоками посредством внешних сигналов. Эти состояния позволяют принимать значения от других потоков. Например, в состоянии MC автомат ожидает, когда интерфейс памяти поместит значение на линию data; он ждет сигнала dataValid, после чего устанавливает сигнал ld_Data, разрешающий записать в регистр DataReg значение с линии data, и переходит в MA. Это пример регистровой передачи, охватывающей сразу несколько конечных автоматов. Как мы увидим, поток интерфейса памяти помещает прочитанное значение на линии data, а поток интерфейса процессора загружает его в DataReg. Каждый автомат исполняет свою часть регистровой передачи.

## 6.3.3. Поток интерфейса памяти (ведомый компонент)

Поток интерфейса памяти является практически зеркальным отражением потока интерфейса процессора: он реализует *другую сторону* протокола SimpleBus. На рис. 6.16 показан тракт данных для интерфейса памяти. Он должен считать адрес с линии address и загрузить значение в старшую или младшую половину регистра AddrReg. Сигналы ld_AddrUp и ld_AddrLo активируют, соответственно, загрузку старшей и младшей половин. Выход AddrReg соединен с массивом памяти. У этого простого запоминающего устройства есть порты ввода и вывода данных. Порт вывода соединен с драйвером шины, активируемым точкой управления en_Data. Если сигнал ld_Data сброшен, то значение MemData зависит от содержимого регистра адреса. Порт ввода соединен с выходом приемника шины. Загрузка в память инициируется точкой управления ld_Data. Интерфейс памяти получает также сигналы start и read, после чего (в зависимости от значения read) он либо считывает значения с линий data и dataValid, либо устанавливает их.

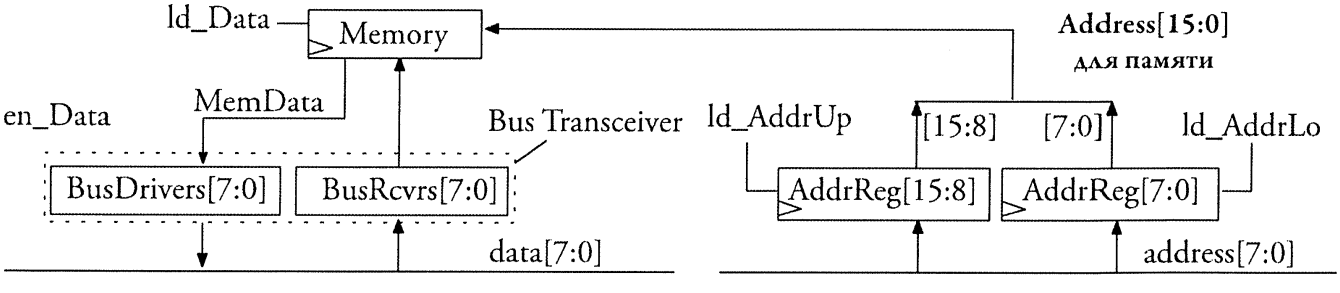



**Рис. 6.16.** Тракт данных для интерфейса памяти

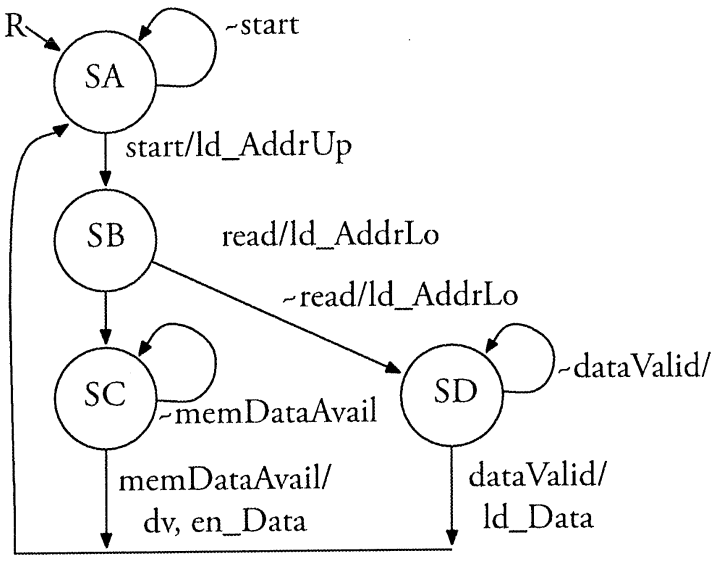



**Рис. 6.17.** Диаграмма состояний потока интерфейса памяти (приемника шины)

Входами конечного автомата для интерфейса памяти, изображенного на рис. 6.17, являются:

- `start` – сигнал шины, означающий начало транзакции;
- `read` – сигнал шины: если установлен, то это транзакция чтения, иначе транзакция записи;
- `memDataAvail` – сигнал от потока памяти, означающий, что прочитанное значение готово к помещению на линию `data`;
- `dataValid` – сигнал шины, означающий, что значение на линии `data` готово к записи в память.

Выходами конечного автомата являются:

- `ld_AddrUp` – разрешает загрузку значения с линии `address` в старшую половину регистра `AddrReg`;
- `ld_AddrLo` – разрешает загрузку значения с линии `address` в младшую половину регистра `AddrReg`;
- `en_DataValid` – разрешает драйверу шины поместить значение выхода `dv` конечного автомата на линию `dataValid`;
- `dv` – значение, помещаемое на линию `dataValid`;
- `en_Data` – разрешает драйверу шины поместить значение из порта вывода памяти на линию `data`;
- `ld_Data` – загрузить данные в память через порт ввода.

Теперь можно объяснить, как работает поток интерфейса памяти. У него есть четыре состояния, обозначенных SA–SD (от slave A-D). SA, SC и SD – состояния ожидания, в которых автомат ждет внешнего сигнала: в SA ожидается появление на шине сигнала `start`, сообщающего о начале новой транзакции; в SC (операция чтения) ожидается установка сигнала `memDataAvail`, говорящая о готовности результата чтения из памяти; в SD (операция записи) ожидается появление на шине сигнала `dataValid`, означающее, что значение с линии `data` можно записать в память.

Рассмотрим состояния автомата подробнее. В состоянии сброса SA автомат ждет начала транзакции на шине. Как только транзакция началась, устанавливается сигнал `start`, а на линию `address` помещается старшая часть адреса. При переходе из SA в SB устанавливается сигнал `ld_AddrUp`, разрешающий загрузку этого значения в старшую часть регистра `AddrReg`. В состоянии SB загружается младшая часть адреса. Далее автомат переходит либо в состояние SC, если это транзакция чтения, либо в состояние SD, если это транзакция записи. В состоянии SC автомат ждет, когда будет прочитано значение из памяти, после чего помещает это значение на линию `data` (устанавливая `en_Data`) и присваивает

выходу dataValid значение dv. Из состояния SC автомат переходит в состояние SA. В состоянии SD автомат ждет, когда на шине появится сигнал dataValid, после чего устанавливает ld_Data, чтобы записать данные в память при переходе в состояние SA.

## 6.3.4. Система в целом

Теперь рассмотрим, как работает система в целом. На рис. 6.18 показаны конечные автоматы потоков интерфейсов процессора и памяти, а также входы от потоков процессора и памяти. На рисунке изображены все точки взаимодействия между автоматами. Серыми стрелками показаны состояния ожидания и чего именно автомат в них ждет, т. е. мы видим, как оба автомата, работая совместно, исполняют транзакции на шине.

Рассмотрим работу этой системы. После сброса конечные автоматы находятся в синхронизированных состояниях <MA, SA>, и вся система ждет, когда поток процессора установит сигнал access, обозначив начало транзакции. После этого конечные автоматы переходят в состояния <MB, SB>, в процессе перехода поток интерфейса процессора помещает на линию address старшую часть адреса, а поток интерфейса памяти загружает ее в старшую часть регистра адреса. Отметим важный момент: когда два конечных автомата синхронизированы по состоянию, как в данном случае, каждый из них реализует свою часть регистровой передачи.

Совместное действие в состояниях <MB, SB> заключается в том, чтобы передать нижнюю половину адреса и затем перейти в состояние чтения или записи. Отметим, что передача нижней половины адреса возможна, потому что оба автомата пошагово синхронизированы и дополнительные сигналы не нужны.

В случае транзакции чтения конечные автоматы переходят в состояния <MC, SC> и ждут сигнала memDataAvail, показывающего, что значение прочитано из массива памяти и находится в порту вывода. Автомат может про-

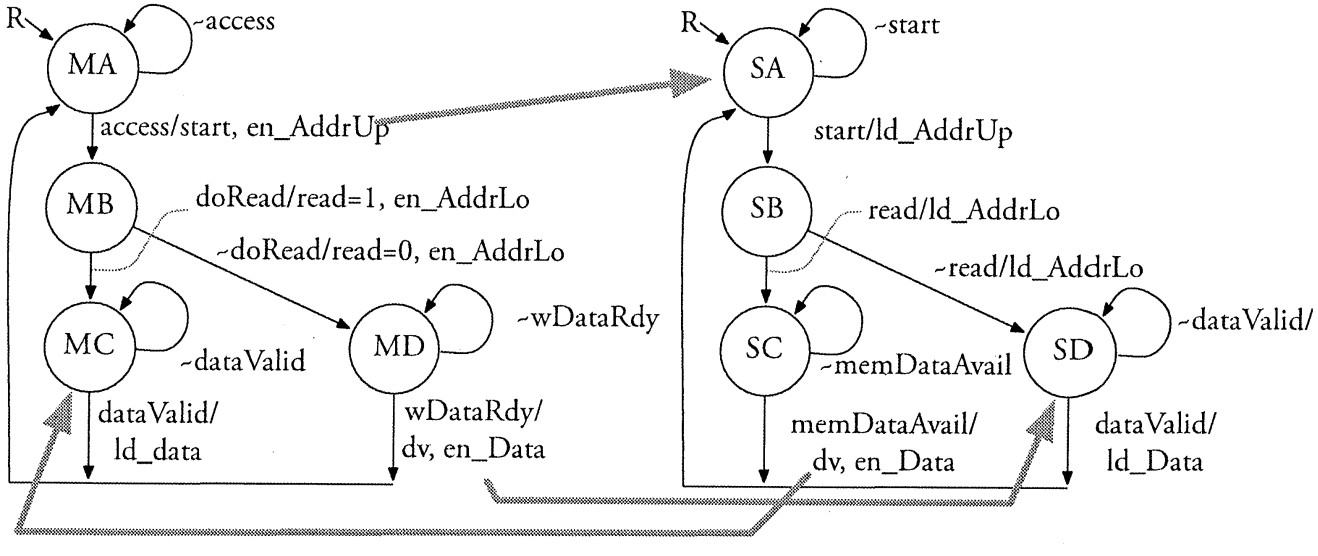



**Рис. 6.18.** Координация потоков интерфейсов процессора и памяти через состояния ожидания

вести много тактов в состоянии SC, ожидая готовности памяти, а другой автомат будет ждать в состоянии MC. Учитывая, что оба автомата являются автоматами Мили, наличие сигнала `memDataAvail` приводит к установке сигнала шины `dataValid` и далее сигнала `ld_Data` ведущим компонентом шины. Все это происходит во время одного перехода из состояний <MC, SC> в <MA, SA> вместе с передачей прочитанного из памяти значения в регистр процессора `DataReg`.

В случае транзакции записи конечные автоматы находятся в состояниях <MD, SD>. В состоянии MD автомат ожидает, когда процессор выдаст подлежащее записи значение и сообщит об этом сигналом `wDataRdy`. При появлении `wDataRdy` конечные автоматы переходят в состояния <MA, SA>, и значение из регистра `DataReg` передается в память.

Хотя это очень простая система процессор–память (кто-то скажет, что чересчур простая), она все же иллюстрирует несколько важных моментов:

- в потоках интерфейсов имеется много состояний ожидания, которые позволяют сохранять синхронизацию с другими аппаратными потоками;
- переходы состояний в таких системах координируются благодаря тесной синхронизации; в этой простой системе мы видим только пары состояний <MA, SA>, <MB, SB>, <MC, SC> и <MD, SD>;
- конечные автоматы остаются синхронизированными с помощью состояний ожидания (<MA, SA>, <MC, SC> и <MD, SD>) и пошагово синхронизированных состояний (<MB, SB> пошагово синхронизировано с <MA, SA>).

В случае такой тесной синхронизации состояний возникает вопрос: а являются ли потоки интерфейсов процессора и памяти действительно независимыми, т. е. можно ли реализовать взаимодействие между процессором и памятью в одном потоке? Если бы описанное выше составляло все содержание системы, это было бы возможно. Поток процессора мог бы напрямую управлять потоком памяти и инициировать транзакции чтения и записи (можно было бы обойтись без потоков интерфейсов).

В общем случае ответ отрицательный. Обычно к шине подключено несколько модулей памяти, и в каждый момент времени процессор может взаимодействовать только с одним из них. При такой организации системы нужно изменить переход из состояния SA в каждом модуле памяти так, чтобы он срабатывал только при соответствии старшей половины адреса базовому адресу этого модуля.

Второе соображение заключается в том, что большая часть работы интерфейса реализована в виде ИС (или IP-блока), который можно соединить с другой ИС (или IP-блоком), поддерживающей тот же протокол. При наличии стандартной спецификации шины к ней можно подключать широкий спектр устройств. Поэтому обычно в моделях потоки интерфейсов отделяются от других потоков, таких как потоки процессора и памяти в нашем примере. Это позволяет повторно использовать модели процессора и памяти с другими интерфейсами. Мы еще вернемся к этому вопросу в разделе 6.5.

### 6.3.5. Код примера SimpleBus

В этом разделе мы представим код примера SimpleBus на языке SystemVerilog. Имена и организация кода соответствуют диаграммам состояний и схемам трактов данных на рис. 6.11–6.18. Поскольку описание занимает несколько страниц, мы будет рассматривать его по частям.

Функциональность шины распределена между двумя модулями: ProcIntThread и MemIntThread. Экземпляры обоих создаются в третьем модуле (top, пример 6.1), который содержит сигналы шины, тактовый сигнал и сигнал сброса. Линии шины описаны в строках 3–4; они объявлены как тристабильные цепи, поэтому могут находиться в состоянии с высоким импедансом (иметь значение z), если драйверы шины не устанавливают никаких значений. Для задания портов интерфейсов процессора и памяти используется нотация .*, поскольку их имена и типы совпадают с именами и типами портов, которые определены в модуле top.

```
1  module top;
2    logic     clock=1, resetN=0;
3    tri       dataValid, start, read;
4    tri [7:0] data, address;
5
6    always #5 clock = ~clock;
7
8    initial #2 resetN=1;
9
10   ProcIntThread M ( .* );
11   MemIntThread S ( .* );
12 endmodule: top
```

**Пример 6.1.** Модуль top для примера SimpleBus

Объявления портов модуля для потока интерфейса процессора приведены в примере 6.2. Отметим, что dataValid и data определены как порты типа inout, потому что в зависимости от типа транзакции шины – запись или чтение – их значение устанавливается либо потоком интерфейса процессора, либо потоком интерфейса памяти. В строках 20–24 объявлены точки управления в тракте данных и сигналы от тестового окружения (например, doRead). Наконец, в перечислении (строки 26–27) объявлены имена состояний конечного автомата ведущего компонента шины.

```
13 module ProcIntThread
14   (input logic resetN, clock,
15    output logic start, read,
16    inout logic dataValid,
17    output logic [7:0] address,
18    inout logic [7:0] data);
19
20   logic en_AddrUp, en_AddrLo,
21         ld_Data, en_Data, access = 0,
22         doRead, wDataRdy, dv;
23   logic [7:0] DataReg;
24   logic [15:0] AddrReg;
25
26   enum {MA, MB, MC, MD}
27           State, NextState;
```

**Пример 6.2.** Объявление потока интерфейса процессора

В примере 6.3 показаны тракт данных и регистры для ведущего компонента шины. В строках 28–29 описаны тристабильные драйверы сигналов data и dataValid. Таким образом, если в строке 29 текущее состояние State равно MD, то dataValid формируется исходя из значения внутренней переменной dv; иначе на этой линии «формируется» высокий импеданс (значение z).

В строках 31–34 описана логика передачи адреса: младшие 8 бит, старшие 8 бит, высокий импеданс.

```
28   assign data = (en_Data) ? DataReg : 'bz;
29   assign dataValid = (State == MD) ? dv : 1'bz;
```

```
30
31  always_comb
32    if (en_AddrLo) address = AddrReg[7:0];
33    else if (en_AddrUp) address = AddrReg[15:8];
34    else address = 'bz;
35
36  always @(posedge clock)
37    if (ld_Data) DataReg <= data;
38
39  always_ff @(posedge clock, negedge resetN)
40    if (~resetN) State <= MA;
41    else State <= NextState;
```

**Пример 6.3.** Тракт данных и регистры для потока интерфейса процессора

В строках 36–37 описаны регистр DataReg ведущего компонента шины и порядок загрузки в него значения, установленного на линии data.

Наконец, в строках 39–41 описан регистр State и его поведение при сбросе системы.

В примере 6.4 показана реализация функций перехода и выхода автомата ведущего компонента шины. В этой комбинационной схеме мы сначала сбрасываем значения выходным переменным (строки 43–49). Затем в операторе case для каждого состояния определяется следующее состояние и значения выходов. Отметим важную особенность такого стиля написания оператора case: значения присваиваются не всем выходам, а только тем, которые могут поменять значение. Присваивание значений выходам, не влияющим на действия в тракте данных, только отвлекало бы внимание и затрудняло понимание кода. Кроме того, при таком подходе не создаются лишние защелки.

```
42  always_comb begin
43    start = 0;
44    en_AddrUp = 0;
45    en_AddrLo = 0;
46    read = 0;
47    ld_Data = 0;
48    en_Data = 0;
49    dv = 0;
50
51    case (State)
52      MA: begin
53          NextState = (access) ? MB : MA;
54          start = (access) ? 1 : 0;
55          en_AddrUp = (access) ? 1 : 0;
56        end
57      MB: begin
58          NextState = (doRead) ? MC : MD;
59          en_AddrLo = 1;
60          read = (doRead) ? 1 : 0;
61        end
62      MC: begin
63          NextState = (dataValid) ? MA : MC;
64          ld_Data = (dataValid) ? 1 : 0;
65        end
66      MD: begin
67          NextState = (wDataRdy) ? MA: MD;
68          en_Data = (wDataRdy) ? 1 : 0;
69          dv = (wDataRdy) ? 1 : 0;
70        end
71    endcase
72  end
```

**Пример 6.4.** Реализация функций перехода и выхода автомата ведущего компонента шины

В примере 6.5 показаны объявления портов и внутренних переменных для потока интерфейса памяти. Порты такие же, как и для потока интерфейса процессора (пример 6.2). Внутренние сигналы (строки 81–82) используются для

управления трактом данных. Массив регистров (строка 80) моделирует память. Объявлены регистры данных и адреса для интерфейса с памятью. Наконец, перечислены имена состояний конечного автомата.

```
73 module MemIntThread
74   (input logic resetN, clock,
75    input logic start, read,
76    inout logic dataValid,
77    input logic [7:0] address,
78    inout logic [7:0] data);
79
80   logic [7:0]  Mem [16'hFFFF: 0], MemData;
81   logic        ld_AddrUp, ld_AddrLo,
82                memDataAvail = 0, en_Data, ld_Data, dv;
83   logic [7:0]  DataReg;
84   logic [15:0] AddrReg;
85
86 enum {SA, SB, SC, SD} State, NextState;
```

**Пример 6.5.** Объявления интерфейса памяти

В примере 6.6 показана модель потока интерфейса памяти и тракта данных, реализующих работу памяти. В строках 87–90 память инициализируется, так что каждая ячейка содержит младшие 8 бит своего адреса (выбор произволен, можно было заполнить любым другим способом).

В строках 92–93 описаны тристабильные драйверы линий data и dataValid.

В строках 95–96 моделируется чтение памяти. Всякий раз при загрузке значения в память или изменении регистра адреса обновляется значение MemData. Память моделируется таким образом, что MemData обновляется при изменении любого элемента массива или адреса в массиве памяти. В строках 104–109 моделируется запись в массив. Копия записываемого значения сохраняется в DataReg (строка 106), чтобы тестовому окружению было проще к нему обратиться.

В строках 98–102 моделируется загрузка в регистр адреса, а в строках 111–113 – регистр State.

```
87  initial begin
88    for (int i = 0; i <= 16'hFFFF; i++)
89      Mem[i] = i[7:0];
90  end
91
92  assign data = (en_Data) ? MemData : 'bz;
93  assign dataValid = (State == SC) ? dv : 1'bz;
94
95  always @(AddrReg, ld_Data)
96    MemData = Mem[AddrReg];
97
98  always_ff @(posedge clock)
99    if (ld_AddrUp) AddrReg[15:8] <= address;
100
101 always_ff @(posedge clock)
102   if (ld_AddrLo) AddrReg[7:0] <= address;
103
104 always @(posedge clock) begin
105   if (ld_Data) begin
106     DataReg <= data;
107     Mem [AddrReg] <= data;
108   end
109 end
110
111 always_ff @(posedge clock, negedge resetN)
112   if (~resetN) State <= SA;
113   else State <= NextState;
```

**Пример 6.6.** Тракт данных и регистры для потока интерфейса памяти

В примере 6.7 показана реализация функций перехода и выхода конечного автомата. Это описание по стилю похоже на описание конечного автомата интерфейса процессора: например, сначала все выходы сбрасываются.

```
114   always_comb begin
115     ld_AddrUp = 0;
116   ld_AddrLo = 0;
117   dv = 0;
118   en_Data = 0;
119   ld_Data = 0;
120   case (State)
121     SA: begin
122           NextState = (start) ? SB : SA;
123           ld_AddrUp = (start) ? 1 : 0;
124       end
125     SB: begin
126           NextState = (read) ? SC : SD;
127           ld_AddrLo = 1;
128       end
129     SC: begin
130           NextState = (memDataAvail) ? SA : SC;
131           dv = (memDataAvail) ? 1 : 0;
132           en_Data = (memDataAvail) ? 1 : 0;
133       end
134     SD: begin
135           NextState = (dataValid) ? SA : SD;
136           ld_Data = (dataValid) ? 1 : 0;
137       end
138   endcase
139 end
```

**Пример 6.7.** Описание функций перехода и выхода

Тестовое окружение для SimpleBus состоит из блока `initial`, который моделирует поток процессора, вызывая процедуры `ReadMem` и `WriteMem`. Это заставляет поток интерфейса процессора начать на шине транзакции чтения и записи соответственно. Сначала обсудим поток тестового окружения процессора, а затем перейдем к деталям процедур `ReadMem` и `WriteMem`.

```
140 initial begin
141   repeat (2) @(posedge clock);
142
143   WriteMem (16'h0406, 8'hDC);
144   ReadMem (16'h0406);
145   WriteMem (16'h0407, 8'hAB);
146   ReadMem (16'h0406);
147   ReadMem (16'h0407);
148   $finish;
149 end
```

**Пример 6.8.** Тестовое окружение, имитирующее процессор

Тестовое окружение процессора показано в примере 6.8. В нем вызывается процедура `WriteMem` после двух тактов ожидания. В память (по адресу 16'h0406) записывается значение (8'hDC), а затем эта ячейка считывается для проверки произведенной записи. После этого по другому адресу (16'h0407) записывается другое значение. Затем мы возвращаемся к первому адресу (16'h0406) и проверяем, не было ли перезаписано хранящееся там значение. Напоследок проверяется, что по второму адресу

(16'h0407) значение записалось правильно. Это, конечно, далеко не полная проверка подсистемы шины, но она показывает, что базовая функциональность работает.

Процедура WriteMem показана в примере 6.9. Тестовое окружение процессора вызывает ее, указывая адрес и значение для записи в память. Процедура инициализирует выходы потока, за которые она отвечает (access, doRead, wDataRdy, AddrReg, DataReg). Отметим, что это не выходы самой процедуры, потому что у нее могут быть только входы. Эти переменные являются выходами потока тестового окружения, который взаимодействует с потоком интерфейса процессора. Значение access, равное 1, указывает, что поток интерфейса должен начать на шине транзакцию. Спустя один такт access сбрасывается в 0, после чего процедура ждет, когда поток интерфейса процессора перейдет в состояние MA. Затем она ждет еще два такта и завершает свою работу.

```
150 task WriteMem
151   (input [15:0] Avalue,
152    input [7:0] Dvalue);
153   begin
154     access <= 1;
155     doRead <= 0;
156     wDataRdy <= 1;
157     AddrReg <= Avalue;
158     DataReg <= Dvalue;
159     @(posedge clock) access <= 0;
160     @(posedge clock);
161     wait (State == MA);
162     repeat (2) @(posedge clock);
163   end
164 endtask
```

**Пример 6.9.** Процедура записи в память в тестовом окружении

Процедура ReadMem, показанная в примере 6.10, очень похожа на WriteMem. Отличия заключаются в том, что при вызове процедуры передается только адрес, а сигнал doRead устанавливается в 1, показывая, что следует исполнить цикл чтения шины. Как и в предыдущем случае, access устанавливается в 1 на один такт, а затем процедура ждет завершения транзакции на шине. После этого она ждет еще два такта и завершается. Прочитанное значение загружается в регистр DataReg. Это происходит, когда сигнал dataValid устанавливается памятью в состоянии SC. Поток интерфейса процессора ждет этого сигнала в состоянии MC. Когда сигнал приходит, конечный автомат потока устанавливает ld_Data, чтобы загрузить в DataReg прочитанное значение (строки 36–37 примера 6.3).

```
165 task ReadMem
166   (input [15:0] Avalue);
167   begin
168     access <= 1;
169     doRead <= 1;
170     wDataRdy <= 0;
171     AddrReg <= Avalue;
172     @(posedge clock) access <= 0;
173     @(posedge clock);
174     wait (State == MA);
175     repeat (2) @(posedge clock);
176   end
177 endtask
```

**Пример 6.10.** Процедура чтения памяти в тестовом окружении

```
178 always @(State) begin
179   bit [2:0] delay;
180   memDataAvail <= 0;
181   if (State == SC) begin
182     delay = $random;
183     repeat (2+delay)
184       @(posedge clock);
185     memDataAvail <= 1;
186   end
187 end
```

**Пример 6.11.** Тестовое окружение, имитирующее память

Все описанное выше является частью модуля ProcIntThread. Последняя часть тестового окружения (пример 6.11) моделирует задержку памяти при чтении или записи байта. Это часть модуля MemIntThread. Эта модель контролирует сигнал memDataAvail к конечному автомату про-

цесса интерфейса памяти. Перейдя в состояние SC, конечный автомат ожидает, когда память прочитает значение. Это моделируется в строках 181–186 с помощью 3-битного случайного значения, которое прибавляется к 2 для получения задержки, не меньшей 2 тактов. После задержки `memDataAvail` устанавливается в 1, что служит для конечного автомата процесса интерфейса памяти сигналом закончить цикл чтения шины.

## 6.4. Асинхронные взаимодействия между потоками

В предыдущих примерах передачи информации между аппаратными потоками использовались сигнальные переменные и состояния ожидания, чтобы организовать рандеву и продолжить передачу в режиме пошаговой синхронизации. В этих случаях – когда имеется общий тактовый сигнал и сигнальные переменные – понять, в каком состоянии находится другой поток, просто. Например, после начала транзакции шины в протоколе SimpleBus точно известно, что в следующей паре состояний (<MB, SB>) на линии `address` будет младшая половина адреса. Для определения этой ситуации не нужны никакие дополнительные сигналы; поток интерфейса памяти мог бы сразу загружать значение, установленное на линии `address`, в регистр адреса. Переходы состояний пошагово синхронизированы, а передачи данных привязаны к этим переходам.

Эти примеры иллюстрировали синхронные взаимодействия между потоками, поскольку потоки должны были быть синхронизированы по состоянию. Даже если передача данных была реализована с помощью одностороннего ожидания, все равно оба потока находились в определенном состоянии, в котором была возможна передача. Отсюда и термин «синхронный». Но для передачи информации такая тесная синхронизация не всегда необходима. Например, один поток может запросить данные, установив сигнал запроса, и заняться другими вычислениями. Когда второй поток увидит запрос, он подготовит данные и установит сигнал, означающий, что данные уже доступны, а затем займется какими-то еще вычислениями.

При таком взгляде на систему передача данных представляется асинхронной. Это значит, что аппаратный поток может подготовить некоторые данные, оставить их в буферном регистре и продолжить работу. В какой-то более поздний момент времени поток-потребитель сможет запросить и получить эти данные. Хотя данные передаются от производителя потребителю, оба потока

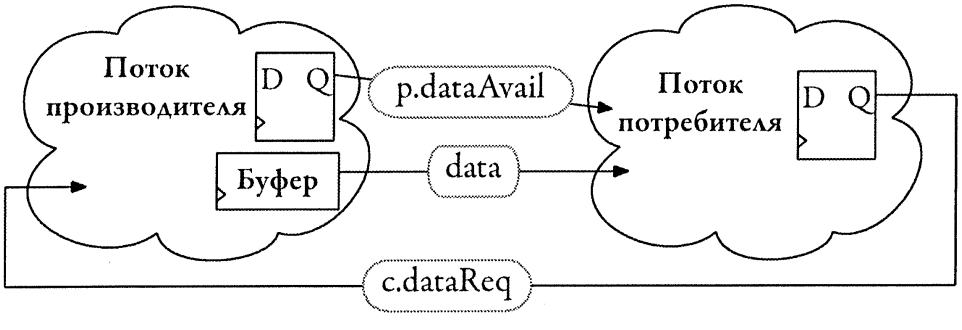



**Рис. 6.19.** Взаимодействие между потоками потребителя и производителя

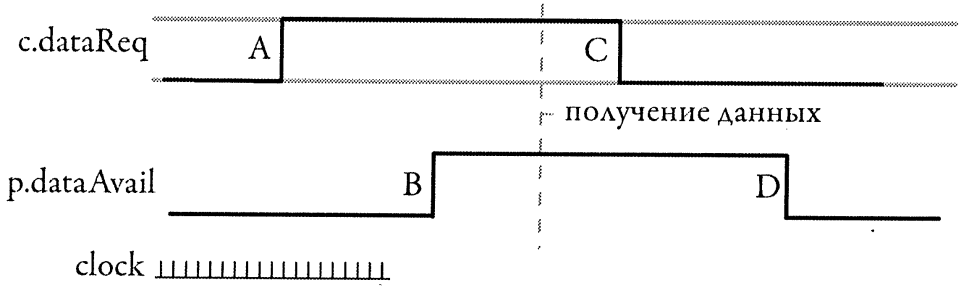



**Рис. 6.20.** Сигналы синхронизации между потоками потребителя и производителя

не обязаны для этого прибыть на рандеву, перейти в определенные состояния для осуществления передачи. Это *асинхронная передача*, и потоки не встречаются, находясь в специальных состояниях; в данном примере производитель мог бы продолжить работу – быть может, подготовить следующий элемент данных, – пока потребитель читает данные.

При синхронном взаимодействии потоков данные передаются благодаря наличию сигнала `dataValid` и тому факту, что оба потока прибыли на рандеву и синхронизированы одним тактовым сигналом. При асинхронном взаимодействии данные тоже передаются по сигналу `dataValid`, но потоки не обязаны прибыть на рандеву и вообще могут находиться в разных доменах синхронизации.

## 6.4.1. Протокол квитирования с полной взаимной синхронизацией

*Квитирование с полной взаимной синхронизацией* (*fully interlocked handshake*) используется для синхронизации потоков потребителя и производителя при асинхронном обмене данными. Рассмотрим рис. 6.19, где представлена конфигурация аппаратных элементов части тракта данных, связанной с передачей.

В конфигурации имеются потоки производителя и потребителя, а также буферный регистр, в котором хранится значение, передаваемое от одного потока другому. Два сигнала, `c.dataReq` и `p.dataAvail`, отражают текущее состояние передачи.

В определении рассматриваемого типа протокола квитирования используются две сигнальные переменные, которые на рисунке называются `p.dataAvail` и `c.dataReq`. Термин «квитирование» (*handshake*, или, буквально, рукопожатие) используется потому, что оба потока приходят к соглашению о том, когда передавать данные, и «скрепляют это рукопожатием». Префиксы имен p и c обозначают, какой поток – производитель (p) или потребитель (c) – устанавливает сигнал. Семантика сигналов следующая:

- `c.dataReq` – этот сигнал говорит, что поток-потребитель запрашивает данные, т. е. ему нужны данные для работы, а в буферном регистре ничего нет. В ответ производитель должен подготовить одно слово данных;
- `p.dataAvail` – этот сигнал говорит, что поток-производитель подготовил одно слово данных и поместил его в буферный регистр, т. е. в буферном регистре находятся непрочитанные данные.

На рис. 6.20 продемонстрировано, как совместная работа сигнальных переменных обеспечивает правильную передачу данных. Здесь показаны обе сигнальные линии, а также тактовый сигнал под ними, для того чтобы под-

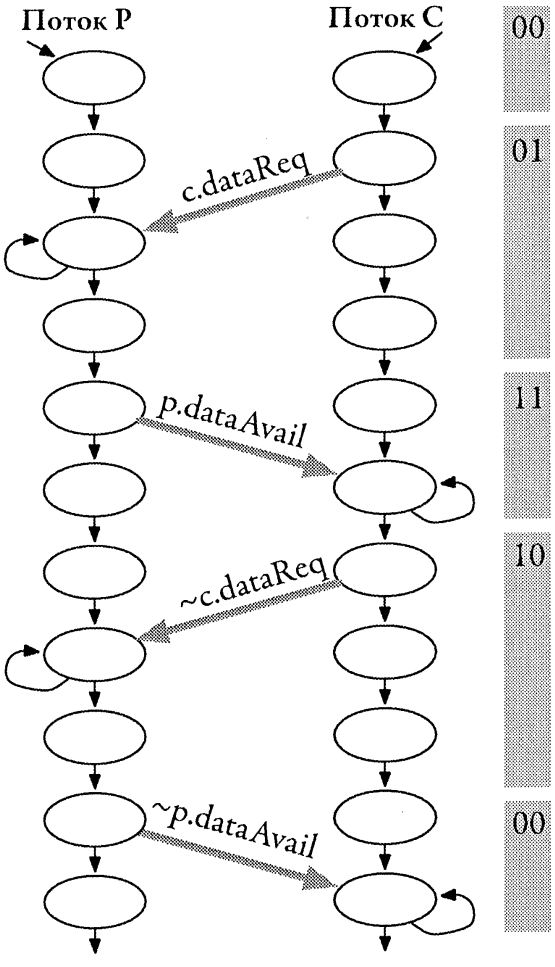



**Рис. 6.21.** Диаграммы состояний двух взаимодействующих потоков

черкнуть, что между моментами изменения сигналов может быть много состояний.

Потребитель начинает первым, устанавливая `c.dataReq` в точке A. Производитель подготавливает данные и устанавливает `p.dataAvail` в точке B. Видя `p.dataAvail`, потребитель понимает, что в буфере есть значение, которое он еще не прочитал. Поэтому он копирует его (`get_data` на рисунке) в собственный регистр и убирает `c.dataReq` (точка C): потребитель получил данные для работы и пока не должен ничего запрашивать. Производитель видит, что `c.dataReq` сброшен, т. е. значение было передано из буфера потребителю, поэтому он сбрасывает сигнал `p.dataAvail` (точка D), помечая значение как прочитанное. Когда потребителю потребуются дополнительные данные, он установит `c.dataReq`, и весь цикл начнется заново.

На рис. 6.21 показано представление этого протокола в виде двух диаграмм состояний. Производитель находится слева, а потребитель – справа. Передача значений сигнальных переменных между диаграммами иллюстрируется толстыми стрелками, каждая из них ведет в состояние ожидания соответствующего сигнала. Остальные состояния – просто заполнители, показывающие, что внутри циклов и условий может быть много других состояний.

Состояния-заполнители иллюстрируют основное свойство этого подхода – тот факт, что оба потока могут заниматься обработкой информации одновременно со взаимодействием между собой. Хотя в конечном счете первому или второму потоку придется оказаться в состоянии ожидания, большую часть времени они оба выполняют вычисления, не связанные с ожиданием. То есть их вычисления перекрываются во времени.

Рассмотрим пару сигнальных переменных `<p.dataAvail, c.dataReq>` совместно. Они показаны справа на рис. 6.21. Сначала они обе имеют значение 2'b00, потом их значение изменяется на 2'b01, затем – на 2'b11, 2'b10 и, наконец, снова на 2'b00. В этом можно убедиться, рассматривая вместе рис. 6.20 и 6.21. Важно здесь то, что состояние передачи данных (значения обеих переменных) изменяется строго в указанной последовательности. Ни один поток не сможет пропустить изменение ни одной из переменных. Каждый из них в порядке очередности ждет заданного значения сначала одной из переменных, затем – дополняющей ее. Оба потока полностью взаимно синхронизированы, отсюда и названия протокола.

Отметим несколько важных моментов. Во-первых, этот протокол *самосинхронизирующийся*, т. е. система выполняет вычисления в своем темпе, определяемом характером обработки в каждом потоке и сигнальными переменными.

Во-вторых, описанный подход к синхронизации важен для потоков, в которых подготовка данных и потребление данных происходят *асинхронно*. Каждый поток синхронизируется тактовым сигналом, и внутри каждого потока предположение о синхронности выполняется. Если потоки находятся в разных доменах синхронизации, то сигнальные переменные синхронизируются триггерами-синхронизаторами. Тем не менее этот протокол назван асинхронным, потому что временны́е диаграммы обоих сигналов определяются данными; между двумя точками сигнализации может быть сколь угодно много состояний. Кроме того, для организации передачи данных оба потока не нуждаются ни в рандеву, ни в пошаговой синхронизации. Протокол сам синхронизирует оба потока, предоставляя возможность обмена данными.

В-третьих, этот способ передачи данных работает медленнее синхронных вариантов, в которых используются рандеву и пошаговая синхронизация. Здесь для передачи одного элемента данных нужно побывать, по меньшей мере, в четырех состояниях, а в случае рандеву данные можно передавать в каждом состоянии. Но, несмотря на замедление, здесь больше возможностей для параллельной работы потоков, так как пока потребитель обрабатывает предыдущие данные, производитель может подготавливать следующие, хотя потребитель их еще и не запросил. Это показано на рис. 6.21 в парном состоянии 2'b10. Следовательно, потоки проводят меньше времени в состояниях ожидания наступления рандеву.

Наконец, рассмотрим более абстрактное представление системы, изображенное на рис. 6.22. Детальные диаграммы состояний потоков становятся набором так называемых макросостояний (закрашенные области на рисунке). Эти состояния соответствуют представлению передачи в виде пар <p.dataAvail, c.dataReq>. В рамках выбранного представления мы можем говорить, что система может находиться в одном из четырех состояний передачи. Тем самым мы отделяем интерфейсную операцию, реализующую квитирование, от производимых потоками вычислений и предлагаем более абстрактный взгляд на обмен информацией в системе.

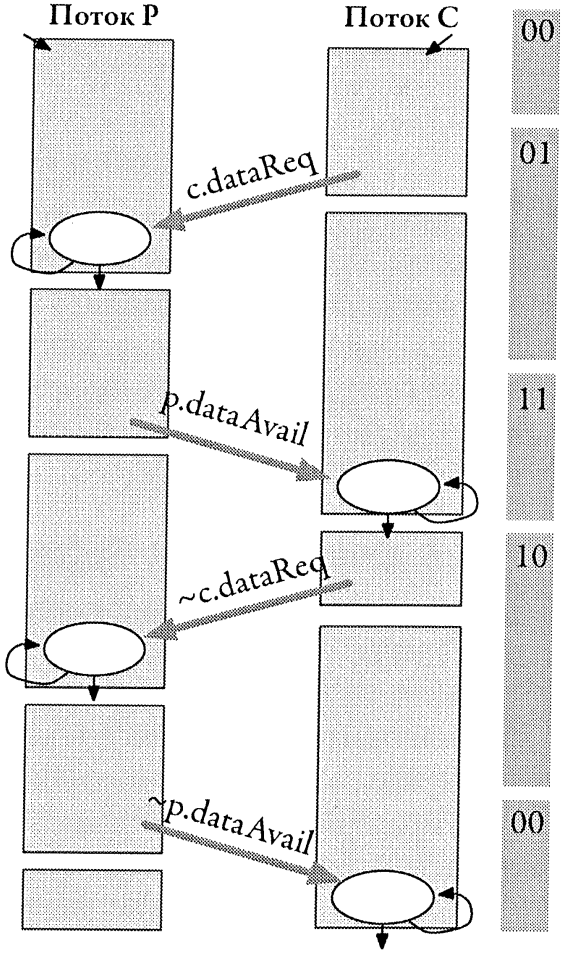



**Рис. 6.22.** Макросостояния двух взаимодействующих потоков

## 6.4.2. Вариации на тему

Мы описали квитирование с полной взаимной синхронизацией как способ передать один байт данных. Но, конечно, этот подход будет работать и для массива или структуры данных. Рассмотрим, к примеру, массив, содержащий *n* байтов. В этом случае потребитель не должен убирать сигнал `c.dataReq`, пока не скопирует в свою память все *n* байтов, а производитель не должен устанавливать `p.dataAvail`, пока все *n* байтов не будут скопированы в буферный массив.

В ситуации, когда от производителя потребителю передаются массивы данных, гораздо эффективнее оказывается схема *двойной буферизации*, показанная на рис. 6.23. Здесь мы имеем два буфера, `Abuf` и `Bbuf`, являющихся массивами, содержащими два набора данных. Пока производитель записывает в один буфер (например, `Abuf`), потребитель может читать из другого (`Bbuf`). Затем они меняются ролями – производитель пишет в `Bbuf`, а потребитель читает из `Abuf`. В отличие от рис. 6.19, синхронизироваться необходимо только в момент переключения буферов. Этот подход оставляет потокам больше возможностей для перекрытия по времени разных этапов их работы.

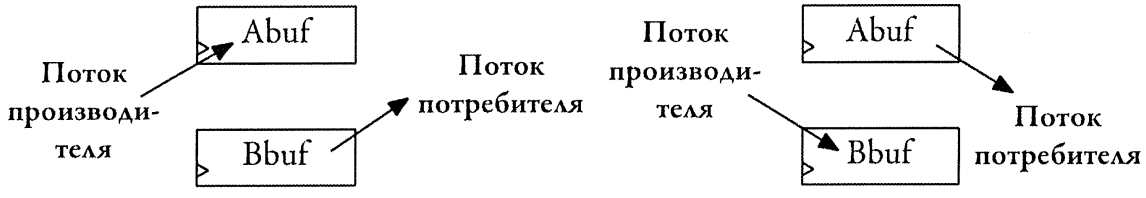



**Рис. 6.23.** Двойная буферизация данных между двумя потоками

Двойная буферизация используется очень широко. Например, графический дисплей обычно имеет два буфера: один используется для получения текущего изображения на экране, а в другом формируется следующее изображение. Визуальные элементы добавляются или удаляются из второго буфера, так что потоку формирователя дисплея ничто не мешает работать с первым буфером. Буферы меняются ролями с частотой, соответствующей частоте кадров данного дисплея.

## 6.4.3. Очереди как буферы

Еще один способ организовать буферы между потоками производителя и потребителя – обслуживать их в порядке «первым пришел – первым ушел» (First-In, First-Out – FIFO), что также иногда называется очередью. В этом случае производитель (или несколько производителей) помещает элементы данных в очередь, а потребитель (или несколько потребителей) извлекает их из очереди. Поддерживается определенный порядок работы с очередью: элемент, помещенный первым, первым и извлекается.

Очереди часто применяются в ситуациях, когда данные помещаются или извлекаются неравномерно, например сразу генерируется целая порция данных. При использовании одного буфера производитель должен был бы ждать, пока потребитель обработает каждый элемент по отдельности. Используя оче-

редь, производитель может поместить в нее все элементы, будучи уверен, что потребитель будет извлекать их по порядку и обрабатывать в своем темпе. Разумеется, очередь должна быть достаточно большой, чтобы в ней поместилась вся порция данных.

На рис. 6.24 показана организация очереди с типичными сигналами – управляющими и состояния. Здесь представлены три потока: производителя, потребителя и очереди. Перечислим входы и выходы потока очереди:

- `empty` – выход: изначально устанавливается при сбросе и указывает на то, что в очереди ничего нет. Впоследствии устанавливается, если все добавленные в очередь элементы были из нее извлечены;
- `full` – выход: указывает, что в очереди не осталось свободного места;
- `put` – вход: точка управления, сообщает очереди, что на входе имеется значение для добавления в очередь;
- `get` – вход: точка управления, сообщает очереди, что нужно извлечь значение и поместить его на выход. Обычно значение копируется в отдельный регистр.

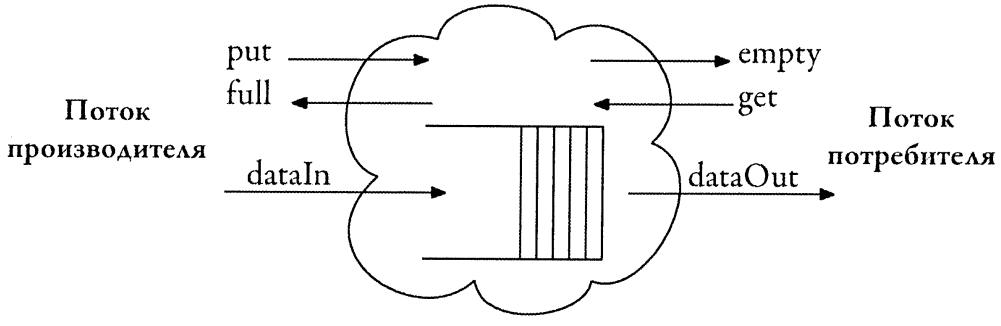



**Рис. 6.24.** Интерфейс с аппаратной очередью

Отметим, что возможны разные варианты организации очереди, отличающиеся деталями работы.

В примере 6.12 представлен модуль `queue`, в котором используются все указанные сигналы (и еще один дополнительный) для организации интерфейса между потоком производителя и потоком потребителя. Поток очереди моделируется с помощью одного блока `always_ff` (строки 14–35). Для организации работы модуля ведется учет количества элементов в очереди (`count`), хранятся указатели на позицию в массиве, в которую следует поместить следующий элемент (`putPtr`), и на позицию, из которой следует извлечь следующий элемент (`getPtr`). По сигналу сброса эти три регистра устанавливаются в 0.

В строках 11–12 определены два выхода: `empty` устанавливается, если `count` равен 0, а `full` – когда `count` равен 8. Выход `dReady` указывает на то, что значение извлечено из очереди и помещено в регистр `dOut`.

С потоком связано два действия: `put` и `get`. Действие `get` исполняется, только если очередь не пуста, а `put` – если очередь не заполнена. Сначала рассмотрим `put` (строки 24–28). Если очередь не заполнена, то входное значение `din` загружается в массив по указателю `putPtr`, затем значение этого указателя увеличи-

вается на 1, также на 1 увеличивается значение счетчика count. Поскольку все это неблокирующие присваивания, то они происходят одновременно по фронту тактового сигнала. Теперь рассмотрим операцию get (строки 29–34). Если очередь не пуста, то в регистр dOut загружается значение из массива по указателю getPtr, значение которого затем увеличивается на 1, значение счетчика count уменьшается на 1, а dReady устанавливается в 1.

Сигнал dReady показывает, что значение прочитано из очереди и находится на выходе dOut. Поток потребителя оставляет get установленным, пока не увидит dReady. Это может продолжаться на протяжении нескольких состояний, в течение которых очередь будет пуста. Иногда put и get устанавливаются одновременно, и в этом случае put имеет преимущество, а поток потребителя должен будет ждать. dReady установлен на протяжении только одного состояния.

```
1 module queue
2   (input bit ck, reset_l, get, put,
3    output bit full, dReady,
4    output bit [7:0] dOut,
5    input bit [7:0] dIn);
6
7   bit [7:0] Q [8];
8   bit [2:0] putPtr, getPtr; // указатели оборачиваются
9   bit [3:0] count;
10
11  assign empty = (count == 0),
12  full = (count == 4'd8);
13
14  always_ff @(posedge ck, negedge reset_l)
15  begin
16    if (~reset_l) begin
17      count <= 0;
18      getPtr <= 0;
19      putPtr <= 0;
20      dReady <= 0;
21    end
22    else begin
23      dReady <= 0;
24      if (put && (!full)) begin
25        Q[putPtr] <= dIn;
26        putPtr <= putPtr + 1;
27        count <= count + 1;
28      end
29      else if (get && (!empty)) begin
30        dOut <= Q[getPtr];
31        getPtr <= getPtr + 1;
32        count <= count - 1;
33        dReady <= 1;
34      end
35  end
36 endmodule: queue
```

**Пример 6.12.** Аппаратная очередь

Сделаем несколько замечаний по поводу этой реализации.

- Размер массива очереди задается степенной функцией с основанием, равным двум, поэтому указатели `putPtr` и `getPtr` могут начинать указывать на начало массива по достижении его конца. Счетчик `count` – единственный способ узнать, является ли очередь полной, пустой или частично заполненной.
- Поток производителя должен проверять, что очередь не заполнена, перед тем как устанавливать `put`.
- Поток потребителя может установить `get`, даже если очередь пуста. Однако он должен дождаться сигнала `dReady`, который показывает, что значение извлечено, и сразу после этого убрать `get`.
- Если `put` и `get` установлены одновременно, то произойдет действие `put` (предполагается, что перед установкой `put` проверено, что очередь не заполнена).

## 6.5. Интерфейсы в SystemVerilog

Все модели на уровне аппаратных потоков имеют интерфейсы к остальной части проектируемой системы. Интерфейсы включают в себя межсоединения и протоколы, описывающие временны́е соотношения между сигналами и значениями на входах и выходах аппаратных потоков.

По мере усложнения системы возникает необходимость в стандартных предопределенных интерфейсах типа USB, которые гарантируют, что аппаратный поток способен корректно взаимодействовать с другими потоками, в которых используется тот же интерфейс. Отсюда следует, что существует естественное разделение между функциональностью аппаратного потока и интерфейсом между ним и остальной частью вычислительной системы.

Конструкция interface в SystemVerilog предназначена для моделирования интерфейсов. Например, мы можем смоделировать функциональность потока, а его интерфейс построить в соответствии со стандартом USB. Впоследствии его можно будет заменить интерфейсом, отвечающим другому стандарту. Функциональность остается одной и той же, меняется только спецификация интерфейса. Такой подход дает более абстрактный взгляд на цифровую систему и способствует повторной используемости сложных моделей.

Интерфейсы SystemVerilog обеспечивают также более компактный стиль моделирования портов модуля, описывающего аппаратный поток. Рассмотрим протокол SimpleBus и порты для него, перечисленные в примерах 6.2 и 6.5, где определены заголовки модулей ведущего и ведомого компонентов шины. Эти объявления дублируют друг друга, а заодно и определения типов в модуле `top` из примера 6.1. Интерфейсы SystemVerilog позволяют поместить эти объявления в одно место – спецификацию интерфейса – и использовать во многих местах модели. В ситуации, когда интерфейс проектируется вместе с аппаратным потоком, объединение спецификации в одном месте упрощает развитие интерфейса. Кроме того, отделение спецификации интерфейса

от модели открывает возможность для получения доступа к лицензируемому стандартному интерфейсу.

## 6.5.1. Пример простого интерфейса

В примере 6.13 показан простой интерфейс ReqAck. Он определен в строках 1–4 между ключевыми словами interface и endinterface. У интерфейса есть входной порт, clk, и две переменные, определенные в строке 3. Этот интерфейс служит для соединения двух модулей, modA и modB, определенных в строках 6–18. В этих модулях показано лишь несколько строк для демонстрации того, как один модуль может взаимодействовать с другим посредством интерфейса.

```
1 interface ReqAck
2   (input clk);
3   logic req, ack;
4 endinterface: ReqAck
5
6 module modA (ReqAck c);
7   logic avail;
8
9 always_ff @(posedge c.clk)
10  c.ack <= c.req & avail;
11  ...
12 endmodule: modA
13
14 module modB (ReqAck b);
15  always_ff @(posedge b.clk)
16    b.req <= 1;
17  ...
18 endmodule: modB
19
20 module top;
21  logic clk;
22  ReqAck RA(clk);
23
24  modA t1 (RA);
25  modB t2 (RA);
26 endmodule: top
```

**Пример 6.13.** Интерфейс ReqAck

У модуля modA в спецификации входного порта c указан тип ReqAck. Это означает, что портом является интерфейс ReqAck и внутри модуля доступны его элементы (clk, req, ack). В строках 9–10 показано, как через точку обратиться к элементам интерфейса. Модуль modB имеет такой же интерфейс, как modA, хотя и называется b.

Чтобы использовать интерфейс, его необходимо определить (строки 1–4) и создать его экземпляр (строка 22). При создании задается имя экземпляра (RA) и вход интерфейса (clk). Затем при создании экземпляров модулей в строках 23–24 имя интерфейса указывается в качестве порта. Таким образом, оба модуля соединены одним и тем же экземпляром RA интерфейса ReqAck.

В интерфейсе определено несколько линий: clk, req и ack. Эти элементы определяются только в одном месте (строки 1–4), поэтому допустить опечатку при вводе их объявлений в определении каждого модуля становится сложнее. На рис. 6.25 приведена схема интерфейса примера 6.13. Мы видим, что экземпляр RA интерфейса ReqAck соединен с обоими экземплярами модулей, которые, следовательно, имеют доступ к внутренним элементам экземпляра. Поэтому интерфейс выглядит как один порт модуля, хотя в нем может быть много элементов. Для доступа к элементам интерфейса используется точка. Таким образом, когда в строке 16 мы устанавливаем b.req в 1, это значит, что переменной req в интерфейсе присваивается значение 1. Нотация c.req в строке 10 означает обращение к той же самой переменной интерфейса.

В модели можно было бы создать еще много экземпляров интерфейса, у каждого из них будут свои копии линий clk, req и ack.

Спецификацию интерфейса можно расширить, включив туда так называемые модпорты (modport). *Модпорт* определяет подмножество элементов ин-

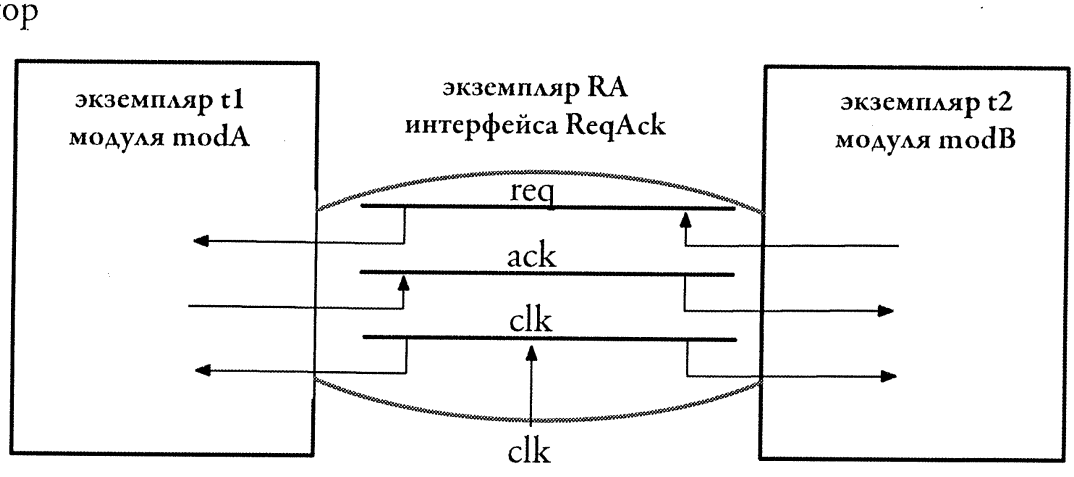



**Рис. 6.25.** Схема интерфейса для примера 6.13

терфейса и направление (вход или выход) каждого элемента. В примере 6.14 это иллюстрируется на примере интерфейса ReqAckm.

Этот пример отличается от примера 6.3 только тем, что изменилось имя интерфейса, а также строки 4–7, 19, 18, 28 и 29. Новшеством здесь является спецификация модпорта в строке 4. Здесь для модпорта G определен выход req и входы ack и clk. В строке 29 при создании экземпляра modB интерфейс указан как RA.G. В строке 18 в определении порта стоит ReqAckm.G, это означает, что будет использоваться конкретный модпорт. Вместе эти строки говорят, что доступ к переменным в модпорту G возможен только в указанных направлениях (вход или выход). Если бы в интерфейсе были другие переменные, то у modB был бы неограниченный доступ к ним. Таким образом, modB может использовать ack и clk только для ввода, а req – только для вывода. Это соответствует стрелкам, ведущим в modB и из него на рис. 6.25.

Модпорт H определяется и создается аналогично, в соответствии с направлениями стрелок, ведущих в modA и из него.

При интерпретации определений модпортов можно считать, что они находятся в том модуле, где модпорт используется. То есть, глядя на строки 4–5, где определяется модпорт G, считайте, что они принадлежат модулю modB, определяющему, как можно использовать элементы req, clk и ack.

Модпорты позволяют компилятору проверять корректность использования интерфейса.

```
1  interface ReqAckm
2    (input clk);
3    logic req, ack;
4    modport G (output req,
5               input ack, clk);
6    modport H (input req, clk
7               output ack);
8  endinterface: ReqAckm
9
10 module modA (ReqAckm.H c);
11   logic avail;
12
13   always_ff @(posedge c.clk)
14     c.ack <= c.req & avail;
15   ...
16 endmodule: modA
17
18 module modB (ReqAckm.G b);
19   always_ff @(posedge b.clk)
20     b.req <= 1;
21   ...
22 endmodule: modB
23
24 module top;
25   logic clk;
26   ReqAckm RA(clk);
27
28   modA t1 (RA.H);
29   modB t2 (RA.G);
30 endmodule: top
```

**Пример 6.14.** Интерфейс с модпортами

## 6.5.2. Характеристики интерфейса

Из примера интерфейса можно вынести два момента. Во-первых, интерфейс можно использовать для группировки сигналов (создания своего рода «пучков проводов»), которые, не будь интерфейсов, пришлось бы перечислять в заголовке модуля. Это дает более простую и допускающую повторное использование спецификацию списка портов. Во-вторых, с помощью конструкции modport можно задавать подмножества элементов интерфейса с указанием направления каждого элемента. Это позволяет компилятору обнаруживать некорректные использования портов.

Интерфейсы полезны и в более сложных ситуациях, в том числе:

- помимо объявления локальных переменных, в интерфейсах могут присутствовать блоки `initial` и `always`, включая `always_comb`, `always_ff` и `always_latch`, а также операторы `assign`;
- в интерфейсе можно определить процедуры и функции. Если модуль, соединенный с интерфейсом, их импортирует, то сможет их вызывать для создания или распознавания некоторых условий в рамках интерфейса. Это широко используется для организации взаимодействия с тестовым окружением;
- в роли порта интерфейса может выступать другой интерфейс.

Интерфейсы очень похожи на модули в том смысле, что инкапсулируют поведение. Интерфейсы создают пространство имен для объявленных переменных; они могут быть параметризованными; их можно определять на внешнем синтаксическом уровне, как модули, или внутри модуля – и в таком случае интерфейс будет известен только этому модулю. Но есть и отличия:

- создание экземпляров, в результате которого возникает структурная иерархия, внутри интерфейса не допускается. То есть нельзя создать внутренний модуль или примитивный вентиль;
- экземпляры интерфейсов могут встречаться в списке портов модуля. Экземпляры модулей – не могут.

## 6.5.3. Пример более сложного интерфейса

Чтобы полнее проиллюстрировать возможности интерфейсов в SystemVerilog, определим простой протокол «три – и готово», показанный на рис. 6.26. Согласно этому протоколу, производитель посылает 24-битный вектор по одному байту за раз. Старший байт посылается первым, за ним средний и младший. В начале передачи устанавливается сигнал `StartNow`, подтверждающий, что первый байт помещен на линию `data`. Приемник ждет сигнала `StartNow` и в режиме пошаговой синхронизации получает каждый байт в последовательных состояниях. Новый сигнал `StartNow` может появиться в состоянии, следующем за пересылкой третьего байта.

Организация системы показана на рис. 6.27. В модуле `top` создаются экземпляры трех интерфейсов и двух модулей. Ниже описаны функции этих пяти элементов.

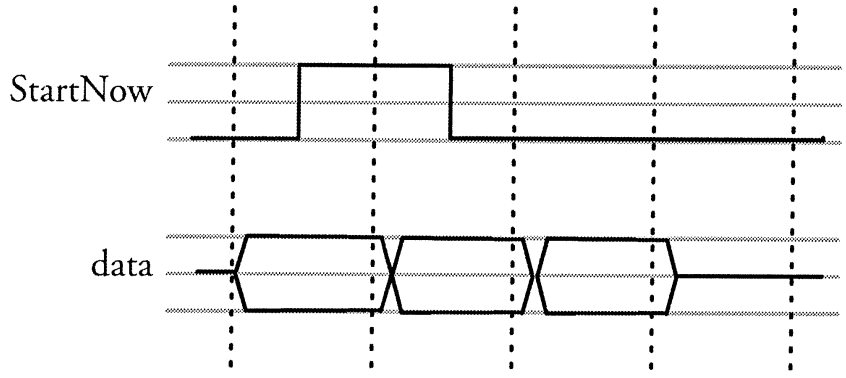



**Рис. 6.26.** Протокол интерфейса «три – и готово»

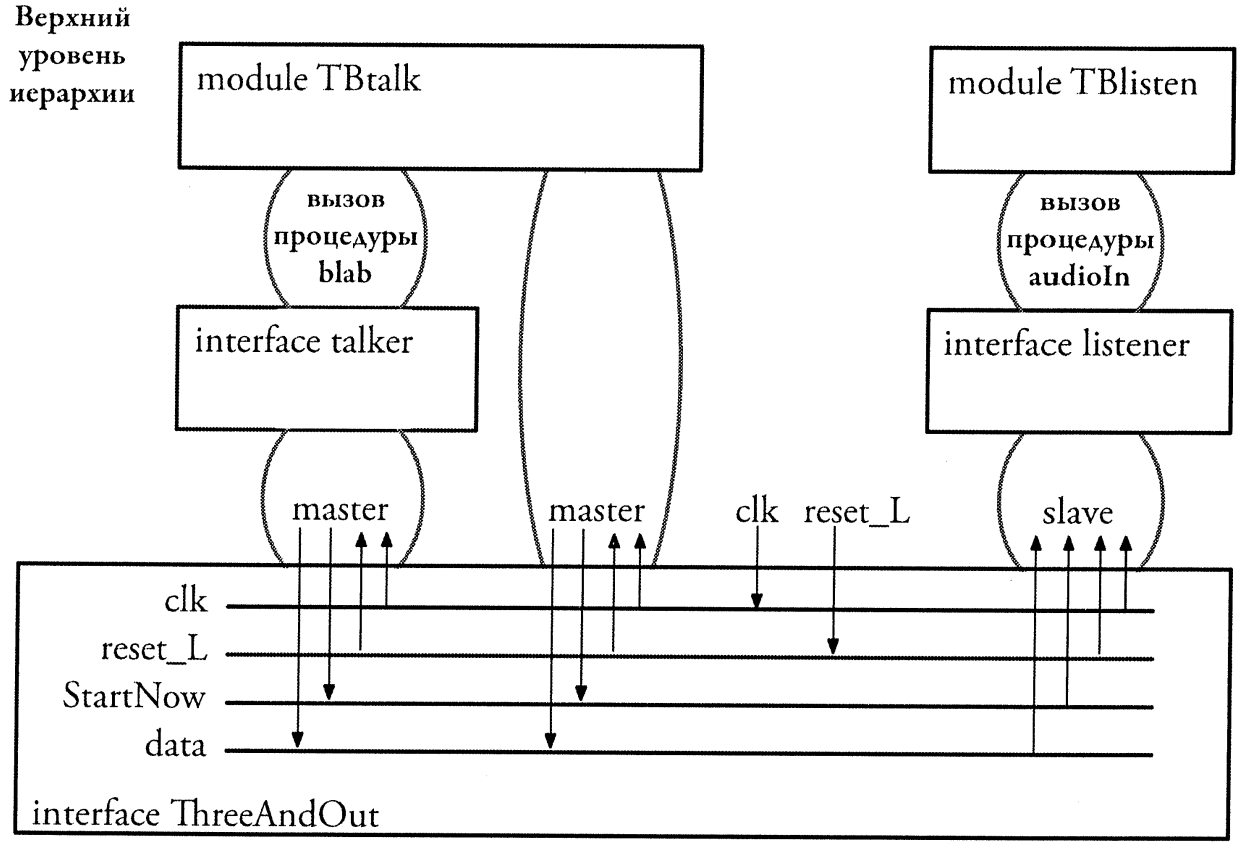



**Рис. 6.27.** Организация интерфейса «три – и готово»

- `interface ThreeAndOut` – интерфейс, содержащий линии `clk`, `reset_L`, `StartNow`, `data`.
- `interface talker` – интерфейс к аппаратному потоку, который реализует протокол `ThreeAndOut` и посылает сообщение через интерфейс `ThreeAndOut`.
- `interface listener` – интерфейс к аппаратному потоку, который реализует протокол `ThreeAndOut` и принимает сообщение, отправленное интерфейсом `talker` через интерфейс `ThreeAndOut`.
- `module TBtalk` – тестовое окружение, которое обращается к интерфейсу `talker` с запросами на передачу 24-битных сообщений. Отправка запроса происходит посредством вызова процедуры `sendit` интерфейса `talker`.

- module TBlisten – тестовое окружение, которое моделирует поток, принимающий сообщение через интерфейс listener. Модуль вызывает процедуру hearit интерфейса listener; работа этой процедуры завершается по получении полного сообщения. Затем это сообщение выводится на консоль.

Интерфейс ThreeAndOut приведен в примере 6.15. Он получает сигналы clk и reset_L через свои порты и содержит сигнал StartNow и 8-битное значение данных. В интерфейсе определены два модпорта: master и slave. Модпорт master разрешает устанавливать на выходах сигналы StartNow и data. Модпорт slave разрешает принимать сигналы StartNow и data на входах. Оба модпорта разрешают принимать сигналы clk и reset_L.

```
1 interface ThreeAndOut
2   (input bit clk, reset_L);
3
4   logic StartNow;
5   logic [7:0] data;
6
7   modport master (output StartNow, data,
8                   input clk, reset_L);
9   modport slave (input StartNow, data,
10                 clk, reset_L);
11 endinterface: ThreeAndOut
```

**Пример 6.15.** Интерфейс ThreeAndOut

Интерфейс talker разбит на две части, показанные в примерах 6.16 и 6.17. Первая часть содержит объявления и регистр состояния. Интерфейс talker соединен с интерфейсом ThreeAndOut через его порт (строка 13), который определен как модпорт master и назван T. В интерфейсе также присутствуют две внутренние переменные: saywhat содержит 24-битный вектор, подлежащий отправке, а сигнал speak указывает, что интерфейс должен начать передачу.

В перечислимом типе определены имена трех состояний конечного автомата (строка 17), а в строках 29–31 определен регистр состояния и обновление его значения. Регистр управляется сигналами clk и reset_L интерфейса ThreeAndOut.

```
12 interface talker
13   (ThreeAndOut.master T);
14   logic [23:0] saywhat;
15   logic speak;
16
17   enum {waitHere, first, second} st, ns;
18
19   modport blab(import sendit);
20
21   task sendit (input logic [23:0] stuff);
22     begin
23       saywhat <= stuff;
24       speak <= 1;
25       @(posedge T.clk) speak <= 0;
26     end
27   endtask
28
29   always_ff @(posedge T.clk, negedge T.reset_L)
30     if (~T.reset_L) st <= waitHere;
31     else st <= ns;
```

**Пример 6.16.** Интерфейс talker, часть 1

По сравнению с предыдущим примером, здесь есть одно новшество: процедура и модпорт, делающий ее доступной. Процедура sendit показана в строках 21–27. Она всего лишь копирует переданное ей значение в переменную saywhat. Кроме того, она устанавливает speak в 1, сообщая интерфейсу talker, что он должен начать отправку этого сообщения. Эти переменные обновляются операторами неблокирующего присваивания, поскольку обновление должно происходить по тактовому сигналу.

Спустя один тактовый интервал speak сбрасывается в 0. Модпорт в строке 19 указывает на импортирование процедуры sendit в модуль, соединенный с модпортом blab этого интерфейса: таким образом разрешается вывоз процедуры внутри этого модуля.

На рис. 6.17 показана вторая часть интерфейса talker. Это комбинационная схема определения выхода и следующего состояния потока интерфейса talker. Поток ждет в состоянии waitHere (оно же является состоянием сброса), пока процедура sendit не установит сигнал speak. После установки speak конечный автомат формирует сигнал StartNow, устанавливает значение на линию data и переходит в состояние first.

В последующих двух состояниях на линию data помещаются остальные байты saywhat, как и определено протоколом.

Завершим рассмотрение отправки сообщений модулем TBtalk, тестовым окружением, которое вызывает процедуру sendit и тем самым инициирует отправку сообщений (см. пример 6.18). Среди портов этого модуля два являются интерфейсами. Модпорт master интерфейса ThreeAndOut предоставляет сигнал clk, а модпорт blab интерфейса talker дает доступ к процедуре sendit. Модуль ждет четыре такта (значение выбрано случайно), а затем вызывает процедуру sendit, передавая ей сообщение. Потом он ждет еще 5 тактов, пока отправляется предыдущее сообщение, и посылает следующее.

Таким образом, эти примеры иллюстрируют, как тестовое окружение использует процедуру для запуска транзакций отправки сообщения. Заметим, что рассмотренная организация его работы является типичной.

```
32  always_comb begin
33    T.StartNow = 0;
34    unique case (st)
35      waitHere: begin
36        ns = (speak) ? first : waitHere;
37        T.StartNow = speak;
38        T.data = (speak) ? saywhat[23:16] : 'bz;
39        end
40      first: begin
41        ns = second;
42        T.data = saywhat[15:8];
43        end
44      second: begin
45        ns = waitHere;
46        T.data = saywhat[7:0];
47      end
48    endcase
49  end
50 endinterface: talker
```

**Пример 6.17.** Интерфейс talker, часть 2

```
51 module TBtalk
52   (ThreeAndOut.master T,
53    talker.blab lips);
54
55   initial begin
56     repeat (4) @(posedge T.clk);
57     lips.sendit (24'h012345);
58
59     repeat (5) @(posedge T.clk);
60     lips.sendit (24'hfedcba);
61     @(posedge T.clk);
62   end
63 endmodule: TBtalk
```

**Пример 6.18.** Модуль тестового окружения для отправки сообщений

Теперь перейдем к принимающей стороне системы. Интерфейс listener показан в примерах 6.19 и 6.20. Входным портом listener является соединение с интерфейсом ThreeAndOut, а внутри listener определены две локальные переменные: IncomingMessage для хранения полученного сообщения и сигнал GotMes-

sage. В интерфейсе имеется три точки управления (ldLow, ldMid, ldHigh), контролирующие, какую часть вектора IncomingMessage загружать.

В listener есть модпорт, разрешающий импорт процедуры hearit. В модуле TBlistener эта процедура вызывается и начинает ожидать сигнала GotMessage (строки 76–77). Когда GotMessage принимает значение TRUE, входящее сообщение копируется в выход words, что и является выходным значением процедуры hearit. Таким образом, работа процедуры сводится к ожиданию сообщения и затем к его пересылке тестовому окружению.

```
64 interface listener (ThreeAndOut L);
65   logic [23:0] IncomingMessage;
66   bit GotMessage, ldLow,
67       ldMid, ldHigh;
68
69   modport audioIn (import hearit);
70
71   enum {warten, eins, zwei} st, ns;
72
73   task hearit (output logic [23:0] words);
74     begin
75       @(posedge L.clk);
76       while (~GotMessage)
77         @(posedge L.clk);
78       @(posedge L.clk);
79       words = IncomingMessage;
80     end
81   endtask
```

**Пример 6.19.** Интерфейс listener, часть 1

```
82 always_ff @(posedge L.clk, negedge L.reset_L)
83     if (~L.reset_L) st <= warten;
84     else st <= ns;
85
86   always_ff @(posedge L.clk)
87     if (ldLow) IncomingMessage[7:0] = L.data;
88     else if (ldMid) IncomingMessage[15:8] = L.data;
89     else if (ldHigh) IncomingMessage[23:16] = L.data;
90
91   always_comb begin
92     GotMessage = 0;
93     ldLow = 0;
94     ldMid = 0;
95     ldHigh = 0;
96     unique case (st)
97       warten: begin
98         ns = (L.StartNow) ? eins : warten;
99         ldHigh = (L.StartNow) ? 1 : 0;
100        end
101      eins: begin
102        ns = zwei;
```

```
103         ldMid = 1;
104         end
105       zwei: begin
106         ns = warten;
107         ldLow = 1;
108         GotMessage = 1;
109         end
110     endcase
111 end
112 endinterface: listener
```

**Пример 6.20.** Интерфейс listener, часть 2

Вторая часть интерфейса listener показана в примере 6.20. В строках 82–84 обновляется регистр состояния конечного автомата. В строках 86–89 моделируется 24-битный регистр IncomingMessage, в который можно разрешить загрузку младшего, среднего и старшего байтов, установив точки управления ldLow, ldMid и ldHigh.

В строках 91–111 описывается комбинационная схема определения выхода и следующего состояния. Отметим, что в состоянии warten конечный автомат ждет сигнала StartNow (строки 97–100). В состоянии zwei он устанавливает сигнал GotMessage в 1, извещая тем самым, что все три байта сообщения получены. Процедура hearit, ожидающая сигнала GotMessage, возвращает принятое сообщение, как описано выше. В каждом из трех состояний устанавливаются, соответственно, сигналы ldLow, ldMid и ldHigh, управляющие загрузкой в регистр IncomingMessage.

Модуль TBlistener (пример 6.21) – тестовое окружение для принимающей стороны системы. Он вызывает процедуру hearit, а когда она вернет управление, выводит на консоль полученное сообщение. Обратите внимание, что в качестве аргумента процедуре hearit передается переменная blahblah. Поскольку этот аргумент является выходным для процедуры, то перед завершением ее работы значение из переменной words (строки 73 и 79 примера 6.19) копируется обратно в переменную blahblah в строке 117 примера 6.21. Затем оно выводится на консоль.

Наконец, в примере 6.22 приведен модуль top. Он создает экземпляры всех трех

```
113 module TBlisten(listener.audioIn ear);
114  logic [23:0] blahblah;
115
116  always begin
117    ear.hearit (blahblah);
118    $display("%3d, Message is %h",
119              $stime, blahblah);
120  end
121 endmodule: TBlisten
```

**Пример 6.21.** Тестовое окружение для listener

```
122 module top;
123  bit clk, reset_L;
124
125  ThreeAndOut TAO(clk, reset_L);
126  talker tk(TAO.master);
127  listener ls(TAO.slave);
128
129  TBtalk TBT(TAO.master, tk.blab);
130  TBlisten TBL(ls.audioIn);
131
132  initial begin
133    clk = 0;
134    reset_L = 0;
135    reset_L <= #1 1;
136    repeat (30) #5 clk = ~clk;
137    #1 $finish;
138    end
139 endmodule: top
```

**Пример 6.22.** Модуль top для модели ThreeAndOut

интерфейсов и обоих модулей. Отметим, что первым нужно создать экземпляр интерфейса ThreeAndOut (строка 125), чтобы его можно было соединить с интерфейсами talker и listener в строках 126–127. Затем модуль top запускает работу системы, устанавливая сигналы clk и reset_L.

## 6.6. Задачи и упражнения

**6.1. Модификация SimpleBus.** Измените протокол SimpleBus так, чтобы сигнал read устанавливался одновременно с сигналом start. Затем измените интерфейс slave и запоминающее устройство, так чтобы к шине можно было подключить несколько запоминающих устройств. У каждого запоминающего устройства должен быть параметр, который определяет, на запрос какой страницы памяти оно будет отвечать. Страница определяется старшими 4 битами адреса. Таким образом, в начале цикла работы шины запоминающее устройство, увидевшее, что номер страницы не совпадает со старшими 4 битами на линии адреса, игнорирует транзакцию. Создайте несколько экземпляров запоминающего устройства и продемонстрируйте, что система работает правильно. Перепроектируйте конечные автоматы и тракты данных; измените описание на SystemVerilog.

**6.2. Для протокола SimpleBus нужен простой интерфейс.** Измените описание SimpleBus – первоначальное или переработанное в ходе решения предыдущей задачи, – воспользовавшись интерфейсами SystemVerilog. Процессор должен инициировать передачи по шине, вызывая процедуру, находящуюся в интерфейсе процессора, а также получать значение, прочитанное процедурой. Аналогичным образом организуйте память и интерфейс памяти.

**6.3. Разделенные транзакции.** Определите новый протокол шины, в которой присутствуют 16-битные линии адреса и данных, а также сигналы start, read и dataReady. В случае записи устанавливается сигнал start, а адрес и данные помещаются на соответствующие линии шины строго одновременно. Запоминающее устройство копирует значение в свой буфер и записывает его в память настолько быстро, насколько может. В случае чтения адрес помещается на линию шины одновременно с установкой сигналов read и start. Обе эти транзакции занимают линии шины в течение одного такта. В случае чтения память отвечает с задержкой в несколько тактов, помещая на шину прочитанное значение и устанавливая сигнал dataReady.

Отличительная особенность этого протокола состоит в том, что транзакцию можно разделить. Например, чтение начинается, как описано выше, одним тактом. Позже память отвечает, возвращая прочитанные данные. Это позволяет процессору начать чтение и запись на нескольких запоминающих устройствах, а затем дожидаться результатов. Таким образом, процессор может в одном такте обратиться для чтения к одному устройству, а в следующем такте – к другому. А спустя пару тактов появятся результаты.

Чтобы это все работало, процессор должен знать, какое именно запоминающее устройство ему отвечает. Кроме того, необходим арбитраж шины – если

два или более устройств хотят осуществить транзакцию на шине, для выбора одного из них должна существовать схема назначения приоритетов. Что касается первого вопроса, то вместе с возвратом прочитанного значения запоминающее устройство может поместить на линию адреса 4 старших бита адреса, чтобы показать, какая страница отвечает. Тогда процессор сможет разобраться, что делать с полученным значением.

Что касается схемы приоритетов: каждое устройство, способное формировать значения на шине, должно иметь линию запроса, идущую к арбитру. Арбитр может быть представлен автоматом Мура, который решает, какое устройство будет формировать значения на шине на следующем такте. Он может отправить любому устройству сигнал разрешения, сообщая, что оно может занять шину.

Эта задача не имеет однозначного решения, нужно рассмотреть разные компромиссы. Вот несколько вопросов для размышления. Может ли запоминающее устройство ставить в очередь запросы на чтение или запись, а затем обрабатывать их в порядке поступления? Если да, то какова максимальная длина очереди? Каковы приоритеты устройств? Насколько детальной вы предполагаете сделать модель памяти? Выбор конкретных решений остается за вами. Ответьте на эти вопросы в процессе проектирования – это доставит вам удовольствие.

**6.4. Контроллер памяти.** Для некоторых запоминающих устройств операция чтения-записи подразумевает чтение или запись нескольких байтов по последовательным адресам. Например, для чтения устройству посылается адрес первого байта, а в ответ передается байт, хранящийся по этому адресу, и по нескольким следующим.

Задача состоит в том, чтобы написать контроллер для показанной ниже модели запоминающего устройства.

```
module mem
 #(parameter DW = 16,
             W = 256,
             AW = $clog2(W))
 (input logic re, we, clk,
  input logic [AW-1:0] Addr,
  inout tri [DW-1:0] Data);

 logic [DW-1:0] M[W];
 logic [DW-1:0] out;

 initial
   for (int i = 0; i < W; i++)
     M[i] = 0;

 assign Data = (re) ? out: 'bz;

 always @(posedge clk)
   if (we) M[Addr] <= Data;

 always_comb
```

```
22      out = M[Addr];
23 endmodule: mem
```

Ниже приведен заголовок требуемого контроллера памяти.

```
1 module memController
2   #(parameter logic [7:0] page = 8'h02)
3   (inout tri [15:0] addrData,
4    input logic addrValid, rw, clk, reset);
```

Экземпляр модуля mem создается внутри модуля memController, а memController соединяется с процессором (который служит тестовым окружением). Для соединения memController и процессора используется 16-битная тристабильная линия (см. диаграмму ниже), сигнал addrValid, показывающий, что на линию помещен адрес, и индикатор чтения-записи rw, значение 1 которого соответствует чтению, а 0 – записи. Поскольку между контроллером и процессором есть лишь один «пучок проводов» (addrData), то адрес и данные мультиплексируются по времени.

На временно́й диаграмме ниже показано, как работает шина. Процессор устанавливает значение на линиях addrValid и rw. Линия addrData формируется как контроллером памяти, так и процессором, но в разное время. Рассмотрим пример чтения из памяти. Процессор помещает адрес на линию addrData и устанавливает сигнал addrValid; на диаграмме это происходит на такте A. Поскольку речь идет о чтении, устанавливается также сигнал rw = 1. По фронту тактового сигнала в конце такта A (в котором установлен addrValid) контроллер памяти понимает, что началась передача по шине (на это указывает addrValid), и загружает копию значения на линии addrData в свой регистр. Это адрес первого элемента данных (data1), будем называть его базовым адресом.

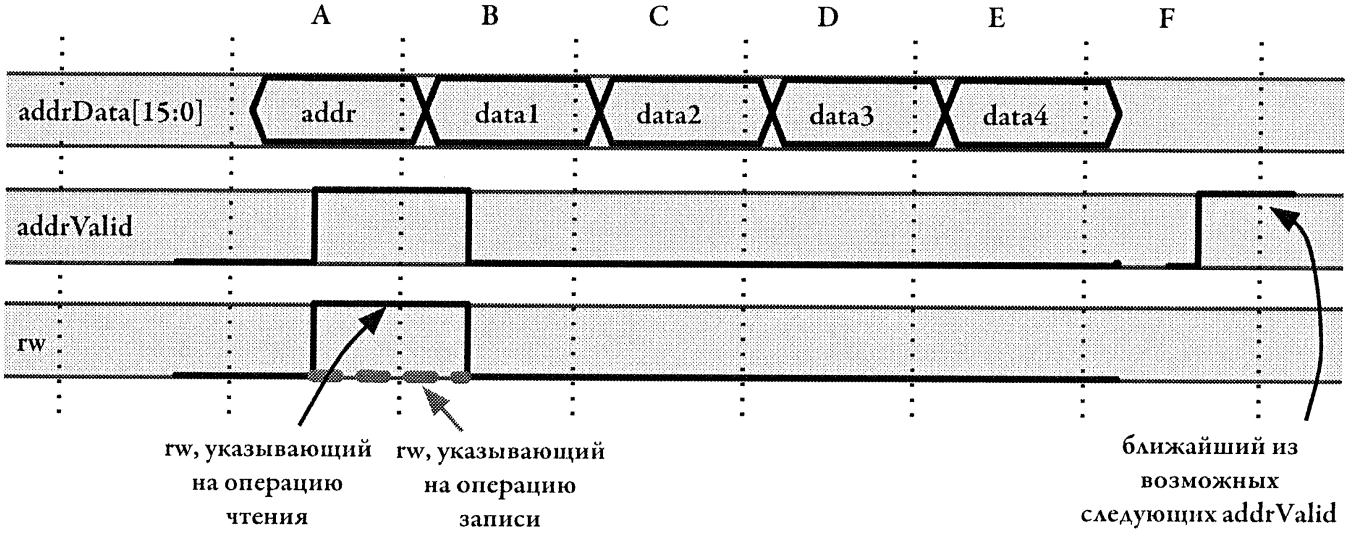
На следующем такте (B) контроллер памяти помещает на линию addrData значение (на диаграмме обозначено data1), прочитанное по базовому адресу. На такте C на линию addrData помещается значение data2 (по базовому адресу плюс 1). На такте D на линию помещается data3 из ячейки с базовым адресом плюс 2, а на такте E – data4 из ячейки с базовым адресом плюс 3. В этой точке операция чтения завершается. Следующая передача по шине может начаться уже на такте F (если так произойдет, то на линию addrData нужно будет поместить адрес чтения или записи).

При записи в память в конце такта A на линии rw должен быть 0. Тогда на протяжении всех пяти тактов (A–E) значения на линии addrData устанавливал бы процессор. Контроллер памяти увидел бы сигнал addrValid, означающий начало передачи по шине, понял бы, что это операция записи (rw равно 0), прочитал бы с шины адрес и все четыре элемента данных и записал бы их в память по адресам addr, addr+1, addr+2 и addr+3.

Отметим, что контроллер не запоминает эти значения, чтобы произвести запись в память позже. Вместо этого значение, находящееся на линии addrData, записывается в память на том же такте. Это относится и к чтению – на линию addrData помещается значение, прочитанное из памяти на текущем такте, т. е.

контроллер памяти, процессор (а значит, и линии шины) и запоминающее устройство синхронизируются одним и тем же тактовым сигналом.

Что означает параметр `page`? К шине может быть подключено несколько контроллеров памяти. Вероятно, вы обратили внимание, что на линию шины помещаются 16 бит адреса, но в модуле `mem` адрес 8-битный. Наш контроллер памяти должен отвечать, только когда старшие 8 бит адреса (они и называются страницей) равны 2. Все остальное игнорируйте. Впрочем, можно было бы создать экземпляр контроллера с другим номером страницы, и все заработало бы!




Напишите модуль `memController` и тестовое окружение, демонстрирующее, что он работает правильно.

# Часть III

# ТЕСТОВЫЕ ОКРУЖЕНИЯ

# Глава 7

# Введение в тестовые окружения

Целью тестового окружения является создание среды, в которой будет функционировать тестируемая модель. В рамках тестового окружения на тестируемую модель подаются стимулы, производится оценка корректности реакций этой модели, а также определяется, в достаточной ли степени для обнаружения всех ошибок были проверены (покрыты) отдельные ее части. Ранее в данной книге были даны примеры небольших тестовых окружений для получения читателем первого представления. В следующих нескольких главах будет представлена организация тестового окружения, использующаяся при проектировании и валидации крупных систем.

## 7.1. Организация тестового окружения

Организация тестового окружения, предназначенного для проведения динамической верификации, показана на рис. 7.1. Тестируемая модель находится в центральном прямоугольном элементе рисунка, объединяющем в себе всю функциональность модели, включая модули, находящиеся ниже по иерархии. Компоненты тестового окружения показаны рядом с моделью и полностью отделены от нее.

Серым цветом оттенены *программы* (*programs*) тестового окружения. Они создают тестовое воздействие (стимул) для тестируемой модели. Корректность ответов модели на тестовое воздействие оценивается утверждениями (assertions), сигнализирующими о неправильном поведении. Часть окружения, отвечающая за функциональное покрытие, содержит модели ошибок, возникающих при тестировании и верификации. Эта часть отслеживает, какие именно аспекты работы тестируемой модели должны быть проверены и до какой степени они должны быть проверены. Генератор тактового сигнала, необходимого для работы всей системы, находится в отдельном модуле.

Программы тестового окружения оттенены серым для подчеркивания их основной роли в управлении процессом тестирования и верификации. Они работают с утверждениями и функциональным покрытием для обеспечения корректности тестируемой модели. Элементы рисунка, обозначающие утверж-

дения и функциональное покрытие, имеют пунктирные границы с целью показать их возможное размещение как в тестовом окружении, так и в тестируемой модели. Этот вопрос будет обсуждаться в соответствующих главах, посвященных утверждениям и функциональному покрытию.

## 7.2. Программы тестового окружения

Для разработки тестового окружения требуются возможности языка, схожие с возможностями языков параллельного программирования. Иногда создание тестового окружения воспринимается как «написание программы, работающей с тактовым сигналом». Действительно, оно похоже на программный код, в котором используются специальные конструкции для управления временем: фронты сигналов (posedge, negedge), задержки (#), неблокирующие присваивания (<=), ожидание изменения значений переменных (@) и ожидание событий (wait). Это дает возможность тестовому окружению взаимодействовать с тестируемой моделью.

В языке SystemVerilog *модуль* (module) является базовой конструкцией для определения модели. Его цель – описать поведение модели либо с помощью операторов (например, блоков always и операторов непрерывного присваивания), либо с помощью соединенных друг с другом вложенных модулей, либо используя комбинацию обоих подходов. Модуль задает область (scope), внутри которой определяются и используются переменные, а также поддерживает взаимодействие с внешней средой через порты модуля. Этот подход позволяет конструировать крупные системы иерархическим образом: посредством использования в модулях экземпляров других модулей. Как уже отмечалось, на рис. 7.1 показана иерархия модулей тестируемой модели.

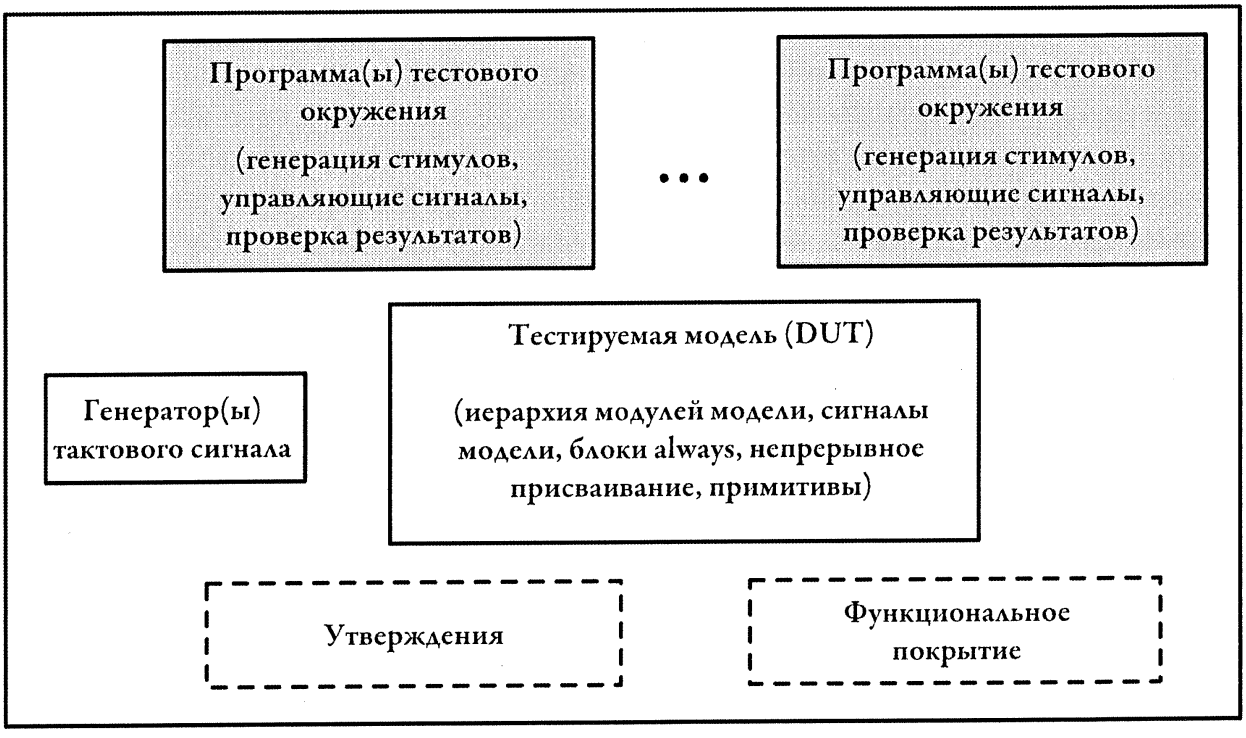



**Рис. 7.1.** Организация программ и модулей

## 7.2.1. Конструкция program

> **Обратите внимание:** использование ключевого слова `program` в SystemVerilog предназначено для помощи в организации тестирования и верификации крупных систем. Студенты при выполнении своих заданий могут не использовать программы. Все приведенные здесь примеры работают корректно, независимо от того, используется ключевое слово `program` или `module`: `program` может быть заменен на `module`, а `endprogram`, соответственно, на `endmodule`.

В тестовом окружении используется конструкция `program`, определенная в SystemVerilog, для спецификации действий программы. Определение программы похоже на определение модуля:

```
1 program tb1
2   (input   bit clk,
3    inout   logic [3:0] data,
4    output  logic [3:0] sum);
5    ...
6 endprogram: tb1
```

В тестовом окружении может быть определено несколько программ.

Программы запускаются на исполнение в блоках инициализации тестового окружения. Можно сказать, что блоки инициализации начинают работу собственно функциональной части тестового окружения. Программы могут включать в себя объявления данных, определения классов, экземпляры объектов, определения функций и процедур, операторы непрерывного присваивания, один или несколько процессов финальной обработки. Экземпляры программ могут использоваться в модулях или интерфейсах. Заметим, что хотя программы и не включают семантические конструкции, они подразумевают возможность использования структурных свойств аппаратуры. Например, программы не содержат экземпляров модулей, логических вентилей и других программ. Также они не содержат блоков `always`, используемых для моделирования аппаратуры.

Цепь или переменная, объявленная внутри программы, называется *программным сигналом*. Цепи или переменные, объявленные в модулях, интерфейсах, пакетах или за пределами любой из этих сущностей, называются *модельными сигналами*. Посредством иерархического именования программы получают доступ ко всем программным и модельным сигналам, функциям и процедурам, объявленным в программах или модулях. Однако модули не имеют доступа к программным сигналам, процедурам и функциям, объявленным внутри программ: такого рода доступ к программам запрещен даже через иерархическое именование. Это сделано для того, чтобы тестируемая модель не использовала тестовое окружение иначе, как через порты. Такой подход вполне согласуется с управлением тестовым окружением моделью и оценкой ее функционирования.

Все события, созданные программами, планируются для обработки симулятором и добавляются в список ответных событий (краткое описание этого процесса содержится в разделе 7.2.2, детальное – в разделе 14.3).

Когда все блоки инициализации, определенные внутри тела программы, завершают свое исполнение, все порожденные ими процессы также немедленно завершаются. Когда все блоки инициализации во всех программах завершены, происходит неявный вызов системной процедуры `$finish`, которая разрешает исполнение всех блоков финальной обработки. Когда и их исполнение завершено, симуляция оканчивается.

## 7.2.2. Этапы работы симулятора

Тестовое окружение создается с использованием программ SystemVerilog таким образом, чтобы отделить события тестового окружения от событий тестируемой модели. Это уменьшает вероятность гонок между ними и позволяет тестовому окружению учитывать текущие результаты тестируемой модели. События, генерируемые программами, включают в себя события, создаваемые в их блоках инициализации, в частности в вызываемых процедурах и функциях, даже если последние определены за пределами тела программы (т. е. в тестируемой модели).

Симулятор сначала обрабатывает все события, сгенерированные тестируемой моделью, а потом исполняет программы тестового окружения, генерируя новые входные значения для тестируемой модели. После этого обрабатываются эти вновь созданные события. Таким образом, события тестируемой модели отделяются от событий тестового окружения.

Более детальное представление о работе симулятора[43] дает рис. 7.2. Приведенная блок-схема показывает, как симулятор обрабатывает события для одного момента времени (так называемого *слота*). Работа симулятора разбивается на пять этапов (regions):

- *предварительная обработка* – время симулятора только что обновилось. На этом этапе значения, полученные в предыдущий момент времени, сэмплируются (сохраняются), чтобы быть использованными в тестовом окружении для проверки утверждений (это происходит на этапе наблюдения);
- *основная обработка* – на этом этапе из списка событий извлекаются и обрабатываются все обычные (regular) события, относящиеся к текущему моменту времени. Это может вызвать новые события с нулевой задержкой (0-delay events).

После обработки обычных событий (в том числе новых) из списка событий извлекаются и исполняются все неблокирующие присваивания. Это также может приводить к новым событиям с нулевой задержкой. Этап продолжается, пока не будут обработаны все обычные и неблокирующие события.

---

[43] Подробнее работа симулятора рассматривается в главе 14.

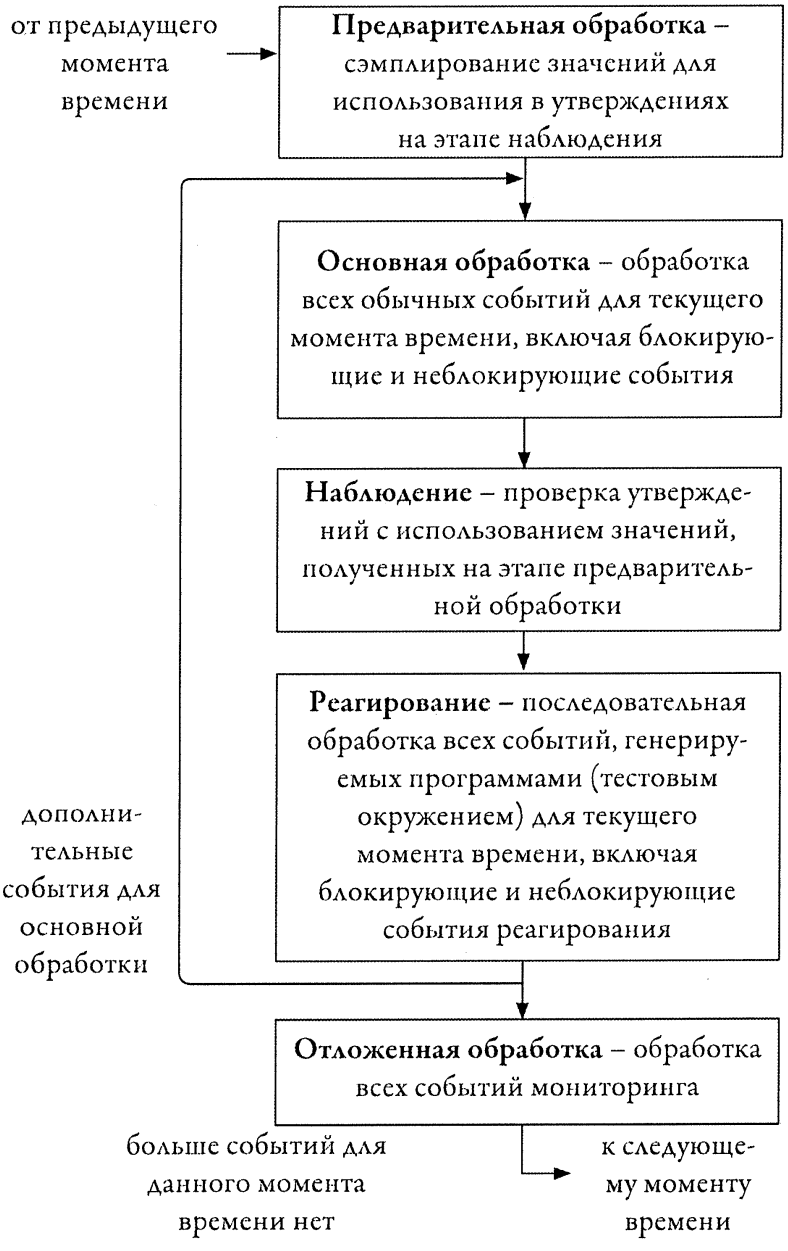



**Рис. 7.2.** Этапы работы симулятора

Прежние рассуждения о работе симулятора и обработке событий касались именно этого этапа;

- *наблюдение* – на этом этапе проверяются параллельные утверждения. Для этого используются значения, сэмплированные на этапе предварительной обработки, а не новые значения, вычисленные на этапе основной обработки;
- *реагирование* – на этом этапе обрабатываются обычные и неблокирующие события, созданные программами (тестовым окружением). События обрабатываются так же, как на этапе основной обработки: сначала – обычные, затем – неблокирующие. В результате могут появляться новые события для этапа основной обработки, вызывающие возврат к этому этапу (понятно, что изменение значений выходов тестового окружения оказывает воздействие на тестируемую модель).

Этот этап назван реагированием, поскольку элементы тестового окружения, которые работают на данном этапе, реагируют на результаты текущего времени симуляции (результатов основной обработки) и обработки утверждений;

- *отложенная обработка* – на этом этапе исполняются события мониторинга. После этого мы переходим к этапу предварительной обработки для следующего момента времени.

Эти этапы обеспечивают ясную семантику моделей уровня регистровых передач: считываются значения, полученные в предыдущий момент времени (предыдущий такт); исполняются действия тестируемой модели, запланированные на текущий такт (этап основной обработки); проверяются утверждения с использованием значений предыдущего такта (этап наблюдения); запускается тестовое окружение, устанавливающее новые значения своих выходов (входов тестируемой модели) в ответ на обновленные значения входов (этап реагирования).

Такая организация работы симулятора позволяет тестируемой модели устанавливать значения своих выходов до того, как тестовое окружение поменяет значения ее входов. Это уменьшает возможность гонок между тестируемой моделью и тестовым окружением. Поскольку этап основной обработки предшествует этапу реагирования, работа тестового окружения будет более понятной, если тактовый сигнал генерируется в модуле, а не программе.

## 7.3. Тестовые окружения для конечных автоматов

Чтобы убедиться в функциональной корректности модели конечного автомата, нужно проверить обе его функции ($\lambda$ и $\delta$) в каждом состоянии. Другими словами, нужно удостовериться в правильности переходов в каждом состоянии автомата при всех возможных комбинациях значений входов, а также в правильности выходных значений.

Для начала напишем тестовое окружение с использованием только программ тестового окружения, тактового сигнала, сигнала сброса, мониторинга и тестируемой модели. В этом разделе описывается каждый из них. Остальные элементы, представленные на рис. 7.1, будут затронуты в других главах.

**Обратите внимание еще раз:** использование ключевого слова `program` в SystemVerilog предназначено для помощи в организации тестирования и верификации крупных систем. Студенты при выполнении своих заданий могут не использовать программы. Все приведенные здесь примеры работают независимо от того, используется ключевое слово `program` или `module`: `program` может быть заменен на `module`, а `endprogram`, соответственно, на `endmodule`.

### 7.3.1. Тактовый сигнал и сигнал сброса

Тактовый сигнал управляет изменениями в тестируемой модели. Это модельный сигнал, который должен быть объявлен внутри модуля, но не в программах тестового окружения.

Простой тактовый сигнал может быть промоделирован следующим образом:

```
1 logic clock;
2 initial begin
3   clock = 0;
4   forever #5 clock = ~clock;
5 end
```

Это поведение включает инициализацию и функцию обновления тактового сигнала с периодичностью 10 единиц времени. Оператор `forever` задает непрерывно исполняющийся цикл. Так как в цикле присутствует конструкция #5, цикл приостанавливается, а возобновление его исполнения через 5 единиц времени приводит к инвертированию значения тактового сигнала `clock`. Обратите внимание на строку 3! Если тактовый сигнал `clock` имеет тип `logic` и он не инициализирован, то его значением будет `x`. Инвертирование значения `x` в бесконечном цикле `forever` будет также приводить к значению `x`. Это никуда не годится! Тактовый сигнал также может быть объявлен как переменная типа `bit`. В описанном подходе к заданию тактового сигнала требуется, чтобы где-то в тестовом окружении определялось, закончена ли работа окружения, и вызывалась процедура `$finish`.

В несколько измененном виде описанный подход выглядит следующим образом:

```
1 bit clock;
2 initial begin
3   clock = 0;
4   repeat (1000) #5 clock = ~clock;
5   $finish;
6 end
```

Оператор `repeat` задает цикл, повторяя следующий за ним оператор заданное количество раз. В данном случае значение тактового сигнала `clock` инвертируется 1000 раз, приводя к 500 изменениям состояния. Это приведет к завершению симуляции на 5000 единицах времени, независимо от того, что будет происходить в тестируемой модели в этот момент времени.

Тактовый сигнал и сигнал сброса также могут быть промоделированы следующим образом:

```
1 bit clock, reset_l;
2 initial begin
3   clock = 0;
4   reset_l = 0;
5   #1 reset_l = 1;
6   #4 forever #5 clock = ~clock;
7 end
```

В этом случае сигналы `clock` и `reset_l` инициализируются нулевым значением в начальный момент времени. Присваивание сигнала `reset_l` в 1 произойдет через 1 единицу времени. На 5 единицах времени наступает бесконечный цикл `forever`. Первый передний фронт тактового сигнала случится на 10 единицах времени.

Несколько более умная (и, возможно, запутанная) логика работы сигнала сброса использует неблокирующую задержку присваивания для достижения такого же поведения, как и было ранее.

```
1 bit clock, reset_l;
2 initial begin
3   clock = 0;
4   reset_l = 0;
5   reset_l <= #1 1;
6   forever #5 clock = ~clock;
7 end
```

Здесь сигналы `clock` и `reset_l` инициализируются, и затем в строке 5 делается неблокирующее присваивание `reset_l` значением 1. Этот же оператор является примером неблокирующей задержки *внутри присваивания*. Как вы уже знаете, неблокирующие присваивания не приводят к моментальному обновлению левой части выражения. В вышеупомянутом операторе изменение значения `reset_l` на `1` планируется (добавляется в список событий симулятора) на будущее, которое наступит через 1 единицу времени. Так как это неблокирующее присваивание, исполнение блока инициализации не приостанавливается на эту 1 единицу времени. Сразу после планирования обновления он продолжает исполняться (т. е. не блокируется), и в нашем случае в строке 6 начинается бесконечный цикл `forever`. В результате значением `reset_l` с нулевого момента времени до наступления 1 единицы времени будет `0`, а затем – `1`.

Между прочим, именно поэтому присваивание вида `<=` называется неблокирующим. Исполнение блока инициализации не приостанавливается на 1 единицу времени, т. е. блок не блокируется, а продолжает исполнение следующих операторов, не откладывая их на будущие моменты времени. В случае неблокирующих присваиваний, что мы видели в примерах обновления состояния, обновление планируется на текущее время и тестируемая модель продолжает исполняться (не блокируется). Следовательно, значением неблокирующей задержки по умолчанию внутри присваивания является 0.

Все примеры выше должны находиться внутри модуля, используемые сигналы являются модельными. Однако в некоторых тестовых окружениях сигнал сброса контролируется программами тестового окружения, позволяя программе перезапустить тестируемую модель перед дальнейшим тестированием. В этом случае сигнал сброса должен быть отделен от тактового сигнала и определен внутри программы тестового окружения.

## 7.3.2. Использование $monitor для отладки конечных автоматов

В разделе 3.2.2 уже была показана разница в использовании процедур `$display` и `$monitor` при отладке. Процедура `$monitor` более надежна при отладке моделей конечных автоматов, поскольку отображаемые функцией значения соответствуют таковым на конец текущего времени симуляции. Следовательно,

они отражают все (неблокирующие) обновления переменных состояния, а отображаемые значения являются консистентными на данный момент времени: все возможные их изменения на данный момент времени уже учтены.

До сих пор все примеры тестовых окружений работали только с портами тестируемых моделей и не учитывали их внутренние переменные. Однако понятно, что не все переменные отображаются на уровне портов модуля. *Иерархическое именование* дает возможность обращаться к переменным, не находящимся в текущем пространстве имен. Рассмотрим следующий фрагмент программы тестового окружения и экземпляра модуля с именем dut.

```
1  module top;
2    ...
3    design dut ( .*);
4    Tbench tb (.*);
5
6    initial $monitor($time, " Current State = %s", dut.state.name);
7  endmodule: top
8
9  program Tbench (объявления портов)
10   ...
11 endmodule: Tbench
12
13 module design (объявления портов);
14   enum logic [4:0] {RESET, A, B, C, ...} state;
15   ...
16 endmodule: design
```

В приведенном примере большой участок кода был заменен троеточием, будучи неважным для иллюстрации идеи. В строке 3 описания модуля top создается экземпляр модуля design с именем dut, все его порты коммутируются. В строке 4 создается экземпляр программы тестового окружения (Tbench). В строке 6 в блоке инициализации настраивается мониторинг. Переменная, на изменение которой будет реагировать монитор, объявлена в строке 14 в объявлении модуля design. Конструкция dut.state в строке 6 является *иерархическим именем*. Использование точки в иерархическом имени соответствует обращению к нижестоящему уровню иерархии. В приведенном примере dut является именем экземпляра модуля design, созданным в модуле top. Спускаясь ниже по уровням иерархии, мы находим переменную с именем state. Это та переменная, чье значение будет выводиться на консоль. Прибавление .name к концу имени переменной в определении монитора в строке 6 указывает на вывод имени элемента перечислимого типа (RESET, A, ...), а не его значения.

Другим возможным местом для определения монитора могла бы быть программа Tbench. В этом случае определение монитора выглядело бы так:

```
initial $monitor($time, " Current State = %s", top.dut.state.name);
```

При разрешении иерархических имен анализатор синтаксиса смотрит на первую часть имени (в данном случае это top). Если он не находит его в текущем мо-

дуле, происходит обращение к иерархии модулей. В данном случае модуль top находится на один уровень выше. Найдя соответствие, анализатор начинает спуск по уровням иерархии для поиска dut.state, аналогично описанному ранее.

Иерархические имена могут предоставить доступ к любой переменной модели. Рекомендуется использовать эту возможность с осторожностью и только для целей отладки. Порты и пространства имен модулей позволяют организовать модель, упростить именование, моделировать интерфейсы к другим элементам. Не стоит нарушать эти принципы.

## 7.3.3. Неявно заданные конечные автоматы

Тестирование модели конечного автомата требует проверки его функций выхода и перехода в каждом из состояний. На диаграмме состояний, показанной, например, на рис. 7.3, мы можем найти *тестовый путь*, который начинается в состоянии сброса и проходит по каждой дуге перехода всей диаграммы. На рисунке он выделен линией серого цвета. В данном простом примере действительно возможно пройти каждую дугу в рамках одной последовательности переходов между состояниями. В случае более сложных диаграмм состояний тестовое окружение может нуждаться в необходимости сбрасывать состояние автомата и исследовать другую последовательность переходов. После прохождения тестовых путей тестовое окружение может проверить факт нахождения в корректном состоянии и корректность выходных данных.

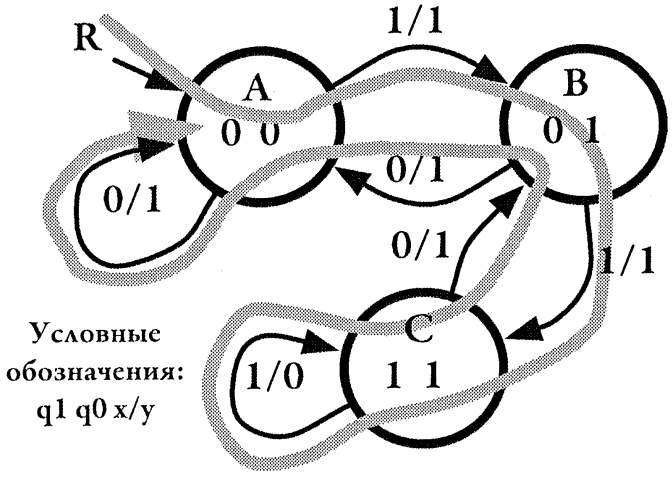



**Рис. 7.3.** Тестовый путь в модели конечного автомата

Заметим, что если конечный автомат имеет $n$ входов, то каждое состояние имеет $2^n$ возможных вариантов перехода, они все должны быть проверены, несмотря на то что некоторые из них помечены как неважные.

Возникает вопрос: как следует создавать тестовое окружение, чтобы оно делало все эти проверки, но было бы не таким сложным, как тестируемая модель?

Для этого используется *неявное задание конечных автоматов*. Явное задание, представленное ранее, называется так из-за явного перечисления состояний и явного определения функций перехода и выхода. При использовании неявного стиля последовательности состояний описываются процедурным образом. Здесь нет регистра состояния и нет его присваивания. Также здесь нет отдельной функции получения выходных данных и следующего состояния. Такие машины состояний, хотя и генерируют выходные значения, не описываются в отдельных операторах типа assign или always_comb.

Неявное описание конечного автомата основано на использовании событийных операторов:

```
@(posedge clock);
```

Этот оператор приостанавливает исполнение кода до тех пор, пока не произойдет хотя бы одно событие из списка, указанного в скобках. Этот список называется *списком чувствительности*. Под событием понимается изменение значения сигнала: передний фронт (posedge), задний фронт (negedge) или любое обновление значения.

Рассмотрим неявно заданный конечный автомат в примере 7.1. В строках 7–13 настраивается мониторинг, тактовый сигнал и сигнал сброса. Описание автомата находится в строках 19–27 в именованном блоке инициализации J внутри блока begin-end. Этот блок находится внутри программы tBench, объект которой создан в строке 5. Тестируемая модель, объект которой создан в строке 4, – это модуль FSMbehavior из примера 3.12. Объекты модуля и программы активируются по переднему фронту тактового сигнала ck. Передний фронт сигнала ck происходит в моменты времени 5, 15, 25 и т. д. Сигнал сброса (r_l) настраивается на активацию в течение 1 единицы времени в строках 10–11.

Программа tBench, расположенная в строках 16–28, имеет выходную переменную x. Тот факт, что данная переменная соответствует входной переменной тестируемой модели, не является простым совпадением: выходы теста должны быть входами модели. Тестируемая модель обновляет переменные состояния по фронтам тактового сигнала. В данном случае переменной состояния является x.

```
1  module top; // используется в главе о тестовых окружениях
2    logic x, z, ck, r_l;
3
4    FSMbehavior dut(.*);
5    tBench tb (.*);
6
7    initial begin: I
8      $monitor( $time,
9        " Current State = %s", dut.state.name);
10     ck = 0;  r_l = 0;
11     r_l <= #1 1;
12     forever #5 ck = ~ck;
13   end
14 endmodule: top
15
16 program tBench
17   (input logic  ck, output logic x);
18
19   initial begin: J // неявное описание конечного автомата
20     x <= 1;
21     @(posedge ck); // возобновление на 5 единицах времени, x не изменяется
22     @(posedge ck); // возобновление на 15 единицах времени, x не изменяется
```

```
23     @(posedge ck); // возобновление на 25 единицах времени, x изменяется
24       x <= 0;
25     @(posedge ck); // возобновление на 35 единицах времени
26     #1 $finish;
27   end
28 endprogram: tBench
29
30 module FSMbehavior
31   (input logic x, ck, r_l, output logic z);
32
33   enum {A, B, C} state;
34   always_ff @(posedge ck, negedge r_l)
35     if (~r_l)
36       state <= A;
37     else case (state)
38         A: state <= (x) ? B : A;
39         B: state <= (x) ? C : A;
40         C: state <= (x) ? C : B;
41         default: state <= A;
42       endcase
43
44   assign z = (state == C) ? ~x : 1'b1;
45 endmodule: FSMbehavior
```

**Пример 7.1.** Неявно заданный конечный автомат в программе тестового окружения

Полезно представить автомат в виде диаграммы состояний. В левой части рис. 7.4 показана диаграмма состояний для блока инициализации J. В правой части показана последовательность состояний, через которые проходит тестируемая модель в результате воздействий значений выходных переменных тестового окружения. Связь между обеими частями рисунка изображена с помощью пунктирных линий, показывающих соответствие переходов, которые происходят в каждой из машин в ответ на первые четыре фронта тактового сигнала. Здесь можно увидеть, что значение x на первом такте, равное 1, вызвало переход в тестируемом конечном автомате из состояния A в состояние B. Два перехода происходят одновременно из-за того, что они используют один и тот же тактовый сигнал. (Обратите внимание: на рисунке показаны только посещенные состояния и переходы только между этими состояниями.)

Блок инициализации J начинается с установки значения выходной переменной в 1 и ожидания в строке 21 переднего фронта сигнала ck. Рассмотрим то, что происходит в тестируемой модели в этот момент времени. Согласно рис. 7.3, состояние A – это состояние после сброса. В момент времени 5 происходит передний фронт тактового сигнала. В результате тестируемая модель переходит в состояние B, так как значение входа x равно 1, а неявно заданный конечный автомат возобновит свое исполнение со строки 21 и перейдет к исполнению событийного оператора в строке 22. Это произойдет в момент времени 5. Вся эта активность показана на рис. 7.4 выше первой пунктирной линии, соответствующей фронту тактового сигнала в момент времени 5.

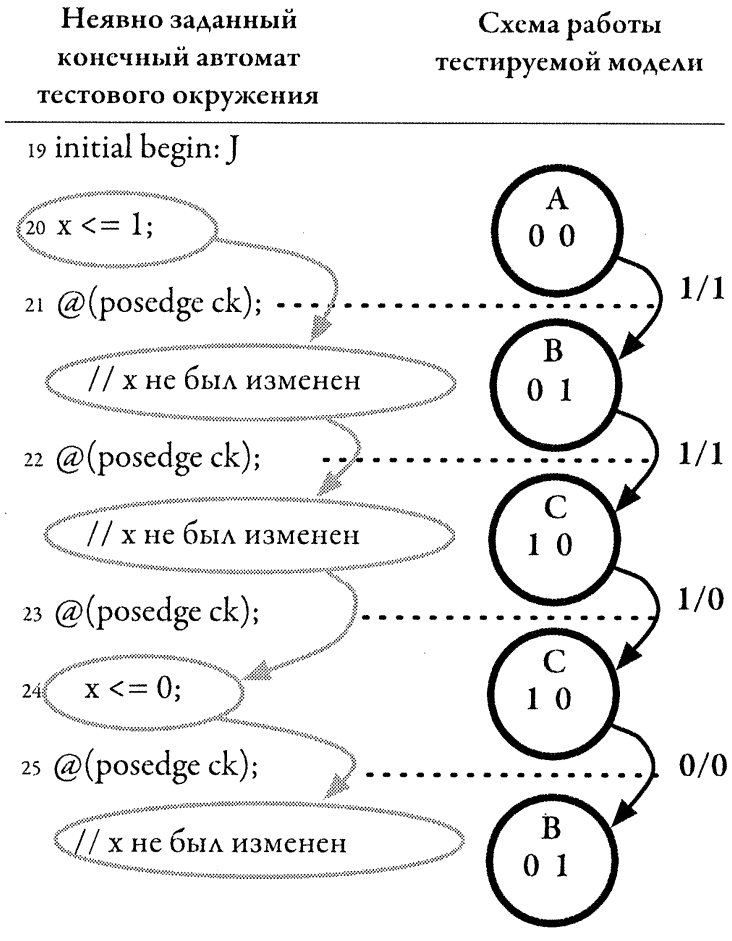



**Рис. 7.4.** Функционирование неявно заданного конечного автомата

В этот момент времени тестируемая модель находится в состоянии B, ожидая следующего фронта тактового сигнала (в момент времени 15), а неявно заданный конечный автомат в строке 22 ожидает того же самого фронта. Значение x все еще 1. Когда произойдет следующий фронт, тестируемая модель перейдет в состояние C, а неявно заданный конечный автомат возобновит свою работу, перейдя на строку 23 и ожидая следующего такта. В момент времени 25 происходит следующий фронт тактового сигнала. Тестируемая модель остается в состоянии C, так как x все еще 1; установка x в 0 произойдет в строке 24. Неявно заданный конечный автомат возобновит свое исполнение в строке 25 в момент времени 35 по следующему фронту тактового сигнала. Из-за того, что значение x равно 0, конечный автомат перейдет в состояние B. Это поведение показано на диаграмме сигналов процесса симуляции на рис. 7.5.

В результате симуляции тестируемый конечный автомат сбросился в состояние A, перешел в состояние B в момент времени 5, затем – в C в момент времени 15, опять в C в 25, и, наконец, перешел в состояние B в момент времени 35. Процедура `$monitor` в строках 8–9 выведет на консоль трассу этих событий.

Целью неявно заданных конечных автоматов в тестовых окружениях является создание последовательностей присваиваний в блоке инициализации, которые бы устанавливали значения выходов тестового окружения (входов

тестируемой модели) так, чтобы пройтись по всем переходам тестируемого конечного автомата. Неявно заданный конечный автомат может даже «перезапуститься», сбросив тестируемый автомат в начальное состояние.

Важно понимать, почему неявно заданный конечный автомат тестового окружения использует неблокирующие присваивания значений выходов. Причина заключается в том, что его выходы (x в данном примере) являются индикаторами того, в каком состоянии находится тестовое окружение: здесь нет регистра состояния, который можно было бы обновить. Это можно представить как машину, кодируемую выходными значениями. Тогда переменная x является ее состоянием, и, следовательно, ее значение должно быть обновлено с помощью неблокирующих присваиваний. Неблокирующие присваивания также отличаются тем, что они происходят одновременно: вы никогда не увидите новое значение только на одном из выходов, без новых значений на остальных.

Что касается кода в примере 7.1, неявно заданный конечный автомат (строки 19–27) находится внутри программы, что означает исполнение присваиваний в ней на этапе реагирования. Рассмотрим по шагам работу симулятора в момент времени 25, что соответствует третьему фронту тактового сигнала на рис. 7.4 и представлено также на диаграмме сигналов на рис. 7.5. Список ниже описывает то, что происходит в каждом из циклов симуляции. Напомним, что на каждом цикле симуляции симулятор удаляет все обычные события, запланированные на текущий момент времени, исполняет их и планирует обновления для текущих или будущих событий. Непосредственно перед фронтом тактового сигнала значением x является 1, и, следовательно, тестируемая модель должна оставаться в состоянии C.

- В момент времени 25 единственным событием обновления в списке событий симулятора является обработка `ck = 1`. Таким образом, когда момент времени 25 достигается, это событие исполняется и `ck` становится равным 1. В результате блок `always_ff` (строка 34) планируется на исполнение на следующий цикл симуляции. Блок инициализации `J` планируется на исполнение на этапе реагирования, который наступит после обработки всех событий этапа основной обработки (таких так блок `always_ff` в строке 34).
- Во втором цикле симуляции исполняется блок `always_ff`, что приводит к планированию неблокирующего обновления состояния на C (так как x равен 1).
- В третьем цикле симуляции неблокирующее обновление осуществляется, и состояние становится равным C. Можно представить себе этот шаг как одновременное изменение состояния триггеров тестируемой модели. Так как состояние не меняется, выходные значения также не изменятся.
- Если событий этапа основной обработки больше нет, симулятор переходит на события этапа реагирования, тех событий, что вызываются программами. В этом цикле симуляции блок инициализации `J` возобновляет исполнение, устанавливая x в 0, используя неблокирующее событие. Оно планируется в списке неблокирующих событий этапа реагирования.

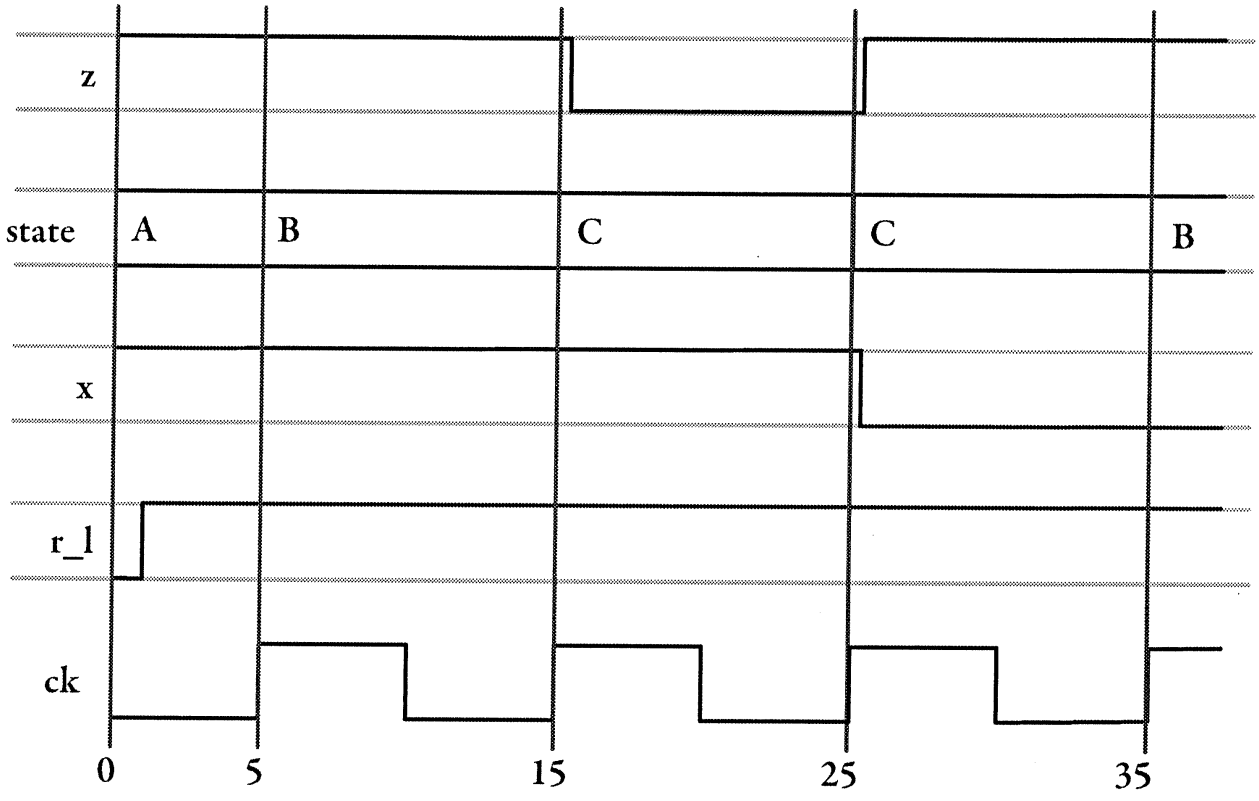



**Рис. 7.5.** Временна́я диаграмма работы неявно заданного конечного автомата

- В следующем цикле симуляции в силу того, что других событий этапа реагирования нет, исполняются неблокирующие события, устанавливая x в 0. Из-за того, что x установлен в 0 и он является входным значением в выражении присваивания в строке 44, событие обновления z планируется в списках событий этапа основной обработки.
- Так как больше событий этапа реагирования нет, симулятор возвращается к этапу основной обработки на следующем цикле симуляции, после чего значение z обновляется по всей модели. В этой точке на данный момент времени события для обработки заканчиваются.

Если бы программы не использовались для построения тестового окружения, а блок инициализации `J` был бы в модуле, последовательность циклов симуляции была бы другой. В частности, обновление состояния и значения переменной x происходило бы на третьем цикле, а значение переменной z было бы обновлено в следующем цикле.

## 7.4. Тестовые окружения для аппаратных потоков

Каждый уровень абстракции аппаратуры требует своего особого типа тестового окружения. При проектировании комбинационных схем акцент делается на применении всех возможных тестовых векторов на всех входах этих схем и проверке того, что выходная функция генерирует корректные выходные значения. В случае конечных автоматов требуется, чтобы тестовое окружение приводило к активации всех возможных переходов между состояниями.

В каждой из этих ситуаций тестовое окружение работает над исчерпывающим тестированием проверяемой модели. Когда речь заходит о тестировании системы, разработанной с поддержкой концепции аппаратных потоков и, таким образом, состоящей из множества потоков и нескольких доменов синхронизации, тестовое окружение вынуждено приспосабливаться именно к этой ситуации.

Аппаратные потоки реализуются с помощью комбинационных схем и конечных автоматов. Распространенной практикой является раздельное тестирование комбинационных и автоматных частей. После этого инженер-верификатор фокусирует проверку новой функциональности, возникающей из этого соединения.

Так как аппаратные потоки реализуют вычисления, включая протокол, по которому отправляется и принимается информация из потока, тестовое окружение поднимается до уровня представления «окружающего мира» для данного потока. Рассмотрим часть рис. 7.1, а именно тестируемое устройство, программу тестового окружения, тактовый генератор и управление сбросом, что показано на рис. 7.6. Для наглядности программа тестового окружения изображена вокруг тестируемого устройства и оттенена серым цветом. Здесь мы видим, что единственный способ управления устройством, доступный тестовому окружению, – это входные порты: `go_l` и `inA`. Наблюдение за поведением может вестись через выходные порты `outResult` и `done`. На рисунке также изображено влияние тестового окружения на аппаратные потоки и одновременно получение результатов их работы.

Отметим, что если тестируемый аппаратный поток предназначен для ввода-вывода, он также может быть использован для целей управления (как, например, включенный здесь поток `downStream`). В этом случае он не входил бы в какую-либо программу тестового окружения, так как оно не содержит функциональность, реализованную на аппаратном уровне.

Таким образом, одним из вариантов тестирования потока является проверка функциональности через его порты. Однако тестовые окружения могут также отслеживать состояния переменных внутри потока, используя их иерархические имена.

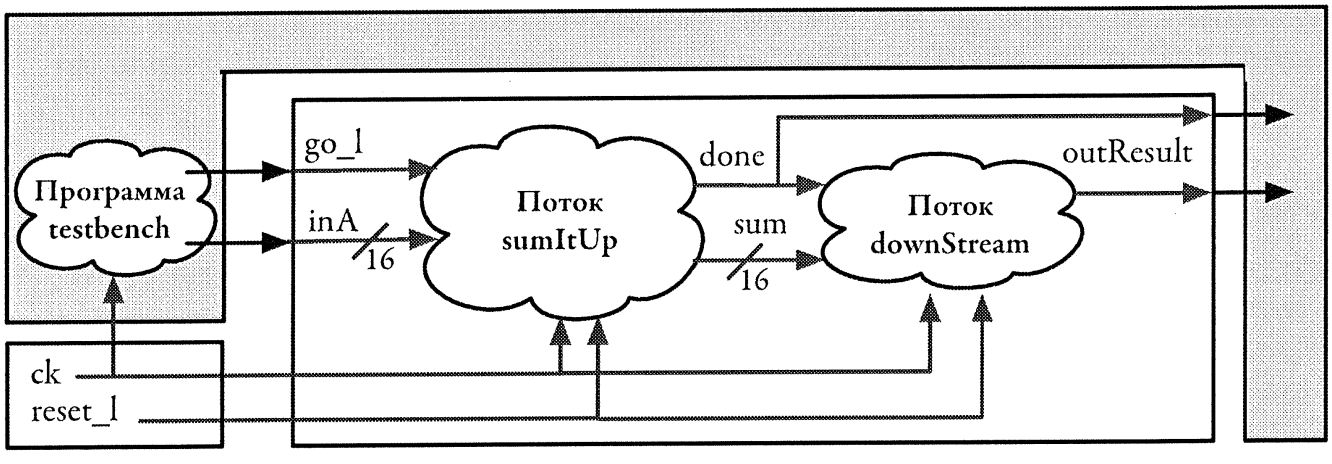



**Рис. 7.6** Два потока тестируемой модели и их программа тестового окружения

**Обратите внимание еще раз:** использование ключевого слова program в SystemVerilog предназначено для помощи в организации тестирования и верификации крупных систем. Студенты при выполнении своих заданий могут не использовать программы. Все приведенные здесь примеры работают корректно, независимо от того, используется ключевое слово program или module: program может быть заменен на module, а endprogram, соответственно, на endmodule.

## 7.4.1. Запуск аппаратного потока

Аппаратный поток sumItUp будет использован как рабочий пример, а разрабатываемые нами тестовые окружения станут через несколько примеров более сложными.

Поток sumItUp показан в примере 7.2, а связанный с ним поток downStream – в примере 7.3. Они соответствуют двум потокам в центральном белом блоке на рис. 7.6 и в основном скопированы из примеров главы 5.

Основной задачей приведенного кода является ожидание установки сигнала go_l. Когда это происходит, значение входного сигнала inA загружается в суммирующий регистр. Все значения inA, пришедшие на последующих тактах, продолжают поступать в этот регистр. Это происходит до тех пор, пока значение inA не станет равным нулю, что означает конец последовательности значений, которые необходимо просуммировать, и приводит к немедленной установке сигнала done. Этот сигнал, в свою очередь, остается установленным до очередного фронта тактового сигнала, после чего опять начинается ожидание установки сигнала go_l.

```
1  module sumItUp
2    (input logic        ck, reset_l, go_l,
3     input logic [15:0] inA,
4     output logic       done,
5     output logic [15:0]sum);
6
7    logic        ld_l, cl_l, inAeq;
8    logic [15:0] addOut;
9
10   enum bit {sA, sB} state;
11
12   always_ff @(posedge ck, negedge reset_l)
13     begin: st_machine
14     if (~reset_l)
15       state <= sA;
16     else begin
17       if (((state == sA) & go_l) | ((state == sB) & inAeq))
18         state <= sA;
19       if (((state == sA) & ~go_l) | ((state == sB) & ~inAeq))
20         state <= sB;
21     end
22   end: st_machine
23
```

```
24   always_ff @(posedge ck, negedge reset_l)
25     begin: reg_sum
26       if (~reset_l) sum <= 0;
27       else if (~ld_l) sum <= addOut;
28       else if (~cl_l) sum <= 0;
29     end: reg_sum
30
31   assign addOut = inA + sum,
32          ld_l = ~(((state == sA) & ~go_l) |
33          ((state == sB) & ~inAeq)),
34          cl_l = ~((state == sB) & inAeq),
35          done = (state == sB) & inAeq,
36          inAeq = inA == 0;
37 endmodule: sumItUp
```

**Пример 7.2.** Аппаратный поток sumItUp

Поток downStream ожидает установки сигнала done потоком sumItUp. Когда это происходит, он загружает значение сигнала sumItUp (т. е. вычисленную потоком сумму) в регистр outResult. Поскольку он ждет установки сигнала done до загрузки outResult, то следует установленному протоколу взаимодействия. Это повторяет работу реального аппаратного потока, который считывает и обрабатывает результат работы потока sumItUp.

```
38 module downStream
39   (input logic         ck, reset_l, done,
40    input  logic [15:0] sum,
41    output logic [15:0] outResult);
42
43   always_ff @(posedge ck, negedge reset_l)
44     if (~reset_l) outResult <= 0;
45     else if (done) outResult <= sum;
46 endmodule: downStream
```

**Пример 7.3.** Поток считывания downStream

```
47 module top;
48   logic        ck, reset_l, go_l;
49   logic [15:0] inA, sum, outResult;
50   logic        done;
51
52   sumItUp dut (.*);
53   downStream ds (.*);
54   TBsimple TB (.*);
55
56   initial begin
57     ck = 0;
58     reset_l = 0;
59     reset_l <= #1 1;
60     repeat (20) #5 ck = ~ck;
61   end
62 endmodule: top
```

**Пример 7.4.** Модуль верхнего уровня для рис. 7.6

Полезно рассмотреть работу блока always_ff (строки 43–45), который моделирует регистр outResult и исполняется одновременно с моделью регистра sum (строки 24–29). Они зависят от одного и того же тактового сигнала и оба используют неблокирующие присваивания. Следовательно, эти присваивания происходят одновременно. Возвращаясь к временно́й диаграмме на рис. 5.2, как sum, так и outResult обновляют свои значения на фронте D тактового сигнала. В этот момент времени значением sum становится 0, а в outResult загружается предыдущее значение sum (на рассматриваемой диаграмме это 10). Для корректного поведения все переменные состояния, даже находящиеся в тестовых окружениях, должны обновлять свои значения посредством неблокирующих присваиваний.

Рассматриваемые два модуля в виде объектов находятся в модуле верхнего уровня, как это показано в строках 52 и 53 примера 7.4. Модуль top соответствует всему рис. 7.6. Переменные в модуле top имеют одинаковые названия с переменными в тестируемых модулях. В этом случае тестируемое устройство, создаваемое в виде объекта в строке 52, может быть подключено посредством .*, который соединяет переменные, перечисленные в декларации интерфейса модуля, с переменными с совпадающими именами, находящимися в пространстве имен модуля top. Это упрощает отслеживание того, что именно проверяется и наблюдается: нет необходимости использовать альтернативные имена для любых из этих сигналов. Модуль downStream создается в виде объекта и подключается аналогичным образом.

Объект программы тестового окружения создается в строке 54. В то время как в этой и следующих главах повествование уходит в область все более сложных тестовых окружений, единственной изменяемой частью примеров будет содержание программ тестового окружения.

## 7.4.2. Системная инициализация

Строки 56–51 примера 7.4 показывают инициализацию системы модулей через активацию сигнала сброса (reset) в течение одной единицы времени. Активация reset вызывает вхождение конечного автомата в состояние sA (строка 15) и инициализацию регистров sum и outResult нулевыми значениями (строки 26 и 44 соответственно). Затем, в строке 60, задается 10 передних фронтов тактового сигнала (ck), начиная с момента времени 5. Таким образом, передние фронты тактового сигнала будут заканчиваться на пятерки единиц времени: 5, 15, 25... Этот блок инициализации – рассмотренные ранее примеры с конечными автоматами. Как было показано ранее, существуют разные способы инициализации тактового сигнала и сигнала сброса.

Блок инициализации программы тестового окружения также осуществит установку сигналов go_l и inA.

## 7.4.3. Простая программа тестового окружения

Итак, давайте рассмотрим поток программы тестового окружения, который посылает данные аппаратному потоку sumItUp, используя протокол, определенный для этой системы модулей. Реализация поведения рассматриваемой программы находится в примере 7.5. Что типично для потоков тестового окружения, в них используется неявный стиль описания конечных автоматов, представленный в разделе 7.3.3. Результаты совместной симуляции программы и модулей top, sumItUp и downStream (примеры 7.2–7.4) выводятся на консоль процедурой $monitor, как показано на рис. 7.7.

Обратите внимание, что тестовое окружение начинает свою работу при сброшенном go_l (строка 11). Поскольку go_l является сигнальной переменной для другого потока, важно как можно раньше определить ее значение, чтобы не вызвать какое-либо неожиданное поведение.

В примере 7.5 обратите внимание на строки 12–14 блока инициализации, где происходит ожидание фронта тактового сигнала и затем установка inA в 55 и определение значения go_l. Фронт тактового сигнала приходит в момент времени 5 (как мы помним, блок инициализации начинает работу в момент времени 0), в результате чего в тот же момент времени 5 произойдет загрузка новых значений в inA и go_l. Хотя эта загрузка является результатом пришедшего в момент времени 5 тактового сигнала и значения inA и go_l должны будут обновиться в тот же момент времени, это не приведет ни к какому изменению состояния модуля sumItUp в тот же момент времени 5. Более того, эти новые значения не будут влиять на состояние до следующего фронта тактового сигнала в момент времени 15, когда модуль sumItUp загрузит 55 в sum. Эти изменения можно увидеть в выводе результатов симуляции на рис. 7.7. Значения на конец момента времени 5, показанные в строке 2 (результатов), соответствуют приведенным выше разъяснениям: inA равен 55, go_l установлен, но значение inA еще не было загружено в sum. Однако на конец момента времени 15 (строка 3) 55 было загружено в sum, а 22 появилось на входе. По следующему фронту тактового сигнала (строка 4) 22 будет также добавлено в sum, изменяя его значение на 77, в то время как на входе появляется следующее значение (11).

```
1  program TBsimple
2    (input  logic         ck, done,
3     output logic  [15:0] inA,
4     output logic          go_l,
5     input  logic  [15:0] sum, outResult);
6
7    initial begin
8      $monitor ($stime,
9       " inA=%3d, go_l=%b, done=%b, sum=%5d, outResult=%5d",
10        inA, go_l, done, sum, outResult);
11     go_l <= 1;
12     @(posedge ck);
13     inA <= 55;
14     go_l <= 0;
15
16     @(posedge ck);
17     inA <= 22;
18     go_l <= 1;
19
20     @(posedge ck);
21     inA <= 11;
22
23     @(posedge ck);
24     inA <= 0;
25
26     #15 $finish;
27   end
28 endprogram: TBsimple
```

**Пример 7.5.** Программа тестового окружения для потока

```
1  0 inA=  x, go_l=1, done=0, sum=  0, outResult=  0
2  5 inA= 55, go_l=0, done=0, sum=  0, outResult=  0
3 15 inA= 22, go_l=1, done=0, sum= 55, outResult=  0
4 25 inA= 11, go_l=1, done=0, sum= 77, outResult=  0
5 35 inA=  0, go_l=1, done=1, sum= 88, outResult=  0
6 45 inA=  0, go_l=1, done=0, sum=  0, outResult= 88
```

**Рисунок 7.7.** Результаты симуляции модуля sumItUp

Это поведение продолжается до фронта тактового сигнала в момент времени 35 (строка 24 примера 7.5), где inA устанавливается в 0. В результате этого sumItUp установит значение на своем выходе done (строка 5 результатов симуляции). В момент времени 45 (строка 6) sum изменится на 0, и в outResult будет загружено предыдущее значение sum (88). Симуляция заканчивается еще немного позже, в строке 26.

## 7.4.4. Использование процедур

Важной особенностью тестового окружения является то, что описываемое им поведение не предназначено для синтеза, поэтому разработчик может писать его в стиле обычной программы. В этом разделе предлагается новая программа для потока из примера 7.5, посылающая информацию в sumItUp с помощью процедуры. Новая программа, осуществляющая управление остальной системой (примеры 7.2–7.4), показана в примере 7.6. Обратите внимание, что имя программы, экземпляр которой создается в строке 54 примера 7.4, должно быть изменено на TBtask.

```
1  program TBtask
2    (input  logic         ck, done,
3     output logic  [15:0] inA,
4     output logic         go_l,
5     input  logic  [15:0] sum, outResult);
6
7    typedef struct {
8      bit [15:0]  valuesToAdd [5];
9      byte        howMany;
10   } sumItPkt_t;
11
12   sumItPkt_t pkt;
13
14   initial begin
15     $monitor ($stime,
16       " inA=%3d, go_l=%b, done=%b, sum=%5d, outResult=%5d",
17       inA, go_l, done, sum, outResult);
18     pkt.valuesToAdd[0] = 55;
19     pkt.valuesToAdd[1] = 22;
20     pkt.valuesToAdd[2] = 11;
21     pkt.howMany = 3;
22     go_l <= 1;
23
```

```
24      sendPktToAdd(pkt);
25      #100 $finish;
26    end
27
28    task sendPktToAdd (input sumItPkt_t pkt);
29
30      for (byte i = 0; i < pkt.howMany; i++) begin
31        @(posedge ck);
32        inA <= pkt.valuesToAdd[i];
33        go_l <= (i == 0) ? 0 : 1;
34      end
35      @(posedge ck);
36      inA <= 0;
37    endtask
38 endprogram: TBtask
```

**Пример 7.6.** Использование процедуры для реализации протокола посылки

Основная идея потока `sumItUp` состоит в том, что он получает несколько величин для сложения и затем сигнализирует о готовности суммы для чтения и использования где-нибудь еще в модели. В описанном далее подходе определяется структура (строки 7–10), используемая для хранения информационного пакета, который посылается в `sumItUp`. Она содержит массив из пяти 16-битных векторов, называемый `valuesToAdd`, и байт, называемый `howMany`. Использование битовых и байтовых типов данных указывает на то, что мы работаем с двоичными данными.

Вместе с идентификатором `typedef` структура создает новый тип данных, названный `sumItPkt_t`. В строке 12 мы определяем переменную `pkt` типа `sumItPkt_t`. Доступ к элементам `pkt` может быть получен, например, через `pkt.valuesToAdd[0]` и `pkt.howMany`.

Блок инициализации в строках 14–26 загружает значения в массив `pkt.valuesToAdd`. Хотя максимальное количество элементов, равное 5, определяется в строке 8 (нумеруемых в массиве от 0 до 4), байт `howMany` указывает, сколько из них действительно используются. Строки 18–20 добавляют значения 55, 22 и 11 в массив, а также 3 в `pkt.howMany`. `go_l` устанавливается в 1 для того, чтобы не допустить начала работы системы. Затем блок инициализации вызывает процедуру в строке 24 и заканчивает свою работу.

*Процедура* (`task`) в SystemVerilog схожа с процедурами языков программирования (см. раздел 11.5). Ее вызов (в SystemVerilog мы называем это *активация*) показан в строке 24. Процедура получает копии входных значений, исполняется и передает копии выходных значений по завершении своей работы. Определение процедуры показано в строках 28–37. Процедура называется `sendPktToAdd`, ей передается одно входное значение, как это определено в строке 28. Это значение будет именоваться `pkt`, его типом является `sumItPkt_t`. Процедура также использует байт, называемый `i`, определенный в строке 30 при инициализации цикла. Обратите внимание, что поскольку `sumItPkt_t` является заданным с помощью `typedef` типом, мы можем его использовать в декларации входных параметров процедуры, см. строку 28.

Основное тело процедуры находится в строках 30–36. Первым идет цикл for, который использует переменную i в качестве счетчика, последовательно повторяя исполнение тела цикла до тех пор, пока i меньше pkt.howMany. В теле цикла ожидается фронт тактового сигнала, затем устанавливаются значения выходных переменных inA и go_l при помощи неблокирующих присваиваний. При первом прохождении цикла go_l будет установлен, при всех остальных будет сброшен. Так как массив pkt.valuesToAdd индексируется счетчиком цикла, его последовательные значения будут появляться на выходе inA. После отправки pkt.howMany значений процедура ожидает еще один фронт тактового сигнала и присваивает inA значение 0, указывая на конец последовательности. Этот пример, в сущности, переносит функциональность примера 7.5 в цикл for внутри процедуры.

Положительным моментом данного подхода является то, что процедура, которая включает протокол посылки информации потоку sumItUp, пишется один раз. Этой процедуре должна лишь быть послана структура, заполненная разными значениями. Соответственно, тестовое окружение становится более высокоуровневым: создает массив чисел, загружаемых в структуру, и посылает их в тестируемую модель. И конечно, существует много вариантов того, что именно посылать и как много элементов массива должны быть посланы в каждом пакете (здесь мы произвольным образом задали 5 в качестве максимума).

Исполнение симулятором этого примера и остальной части тестового окружения приведет к результатам, аналогичным предыдущему примеру, как это показано на рис. 7.7.

## 7.4.5. Использование классов

Далее представлена еще одна версия программы тестового окружения для потока. Она использует тот же самый заголовок программы, как и предыдущий пример тестового окружения, и работает с модулями примеров 7.2–7.4. В ней определяется класс testSumThread, показанный в примере 7.7, который создает набор случайных значений для отправки в поток sumItUp и затем осуществляет их посылку. Блок инициализации примера 7.8 создает новые объекты класса testSumThread с разными длинами пакетов и значениями в них. Это тестовое окружение также проверяет корректность результатов вычисления.

Класс в SystemVerilog, грубо говоря, является точно таким же классом, как и в языках программирования C++ и Java (заметим, что «грубо говоря» было сказано только для того, чтобы вы использовали ваши знания об объектно-ориентированном программировании для понимания этого примера).

Пример 7.7 определяет класс, его атрибуты (переменные) и методы (функции и процедуры). В строках 5–8 класс определяет ту же самую структуру, которая уже была использована ранее в примере тестового окружения. Однако здесь размер массива ограничен шестью элементами с помощью ключевого слова localparam (строка 2).

```
1  class testSumThread;
2    localparam MAXSIZE = 6;
3    local bit unsigned [15:0] total;
4
5    local struct {
6      bit unsigned [15:0] valuesToAdd [MAXSIZE];
7      byte unsigned       howMany;
8    } pkt;
9
10   function new();
11     makePkt();
12   endfunction
13
14   task sendPktToAdd;
15     for (byte i = 0; i < pkt.howMany; i++) begin
16       @(posedge ck);
17       inA <= pkt.valuesToAdd[i];
18       go_l <= (i == 0) ? 0 : 1;
19     end
20     @(posedge ck);
21     inA <= 0;
22   endtask
23
24   function checkTotal (bit unsigned [15:0] value);
25     $display ("checkTotal: value=%d, total=%d",
26               value, total);
27     return value == total;
28   endfunction
29
30   local function makePkt;
31     total = 0;
32     pkt.howMany = $urandom_range(MAXSIZE,2);
33     for (byte i = 0; i < pkt.howMany; i++) begin
34       pkt.valuesToAdd [i] = $urandom_range(1000, 1);
35       total  += pkt.valuesToAdd[i];
36     end
37   endfunction
38 endclass
```

**Пример 7.7.** Использование класса для создания и отправки случайных значений в поток `sumItUp`

В строках 10–12 определяется конструктор класса. Он вызывает функцию `makePkt` (строки 30–37), которая инициализирует структуру `pkt`. Функция `makePkt`, в свою очередь, вызывает системную функцию `$urandom_range` (строка 32), которая возвращает случайное беззнаковое число – длину пакета (`pkt.howMany`). Аргументами этой функции являются верхняя и нижняя границы, в пределах которых генерируется значение. Таким образом, в строке 32 случайно выбирается число от 2 до 6 (включительно). Значение 6 взято произвольно; 2 соответствует минимальному числу складываемых чисел.

Затем цикл for (строки 33–36) создает pkt.howMany случайных беззнаковых чисел в диапазоне от 1000 до 1 и заполняет ими массив в структуре. Заметим, что 0 не является корректным значением для загрузки в массив, так как он используется в качестве признака конца пакета. Одновременно с загрузкой значений в массив вычисляется их сумма. Эта сумма будет использоваться для проверки результата вычислений потока sumItUp. (Заметим, что в SystemVerilog есть оператор сложения с присваиванием (+=), который можно использовать для короткой записи оператора total = total + pkt.valuesToAdd[i]. Оператор += является блокирующим присваиванием.) На данном этапе объект полностью сконструирован и готов к использованию.

Блок инициализации примера 7.8 пошлет пакеты в поток sumItUp и проверит корректность вычислений. В строке 40 объявляется переменная t типа testSumThread. Сигнал go_l сброшен до начала исполнения цикла (строки 45–51). На каждой итерации цикла с помощью метода new создается новый пакет. В строке 47 вызывается метод sendPktToAdd. Этот метод (строки 14–22 примера 7.7) работает, как в предыдущих примерах. На последовательных фронтах тактового сигнала для каждого элемента массива устанавливаются новые значения inA и go_l. За этим следует обнуление inA на один такт, что означает конец пакета.

Когда метод заканчивает исполнение, вход inA имеет значение 0. Блок инициализации ожидает установки done в строке 48. Когда это случается (из-за установки inA в 0), блок инициализации продолжает исполнение и вызывает метод checkTotal, передавая ему значение суммы в sum (выходное значение потока sumItUp). Метод checkTotal (строки 24–28 примера 7.7) сравнивает значение суммы в sum с вычисленным ранее значением, возвращая TRUE, если они одинаковые. Это значение используется для выбора одной из двух строк для передачи в $display (строка 50 примера 7.8).

```
39 initial begin
40   testSumThread t;
41   $monitor ($stime,
42   " inA=%3d, go_l=%b, done=%b, sum=%5d, outResult=%5d",
43     inA, go_l, done, sum, outResult);
44   go_l <= 1;
45   repeat (5) begin
46     t = new;
47     t.sendPktToAdd();
48     wait (done);
49     $display ("At time= %3d, %s", $time,
50       (t.checkTotal(sum))? "got right value": "got wrong value");
51   end
52   #15 $finish;
53 end
```

**Пример 7.8.** Использование экземпляров класса для посылки случайных значений в поток sumItUp

Когда пакетов для передачи больше нет, блок инициализации ожидает 15 единиц времени до конца работы. Часть текстового вывода данного при-

мера показана на рис. 7.8. Заметим, что в момент времени 55 значение done, равное TRUE, вызывает проверку суммы (успешную). Проверяемая сумма sum (2066) не отображается в outResult до следующего фронта тактового сигнала в момент времени 65. В этот же самый момент sum обнуляется, и поток готов к обработке следующего пакета. Отметим, что outResult сохраняет свое значение; оно будет обновлено только при следующей установке сигнала done.

```
1      0 inA=  x, go_l=1, done=0, sum=    0, outResult=    0
2      5 inA= 61, go_l=0, done=0, sum=    0, outResult=    0
3     15 inA=374, go_l=1, done=0, sum=   61, outResult=    0
4     25 inA=771, go_l=1, done=0, sum=  435, outResult=    0
5     35 inA=708, go_l=1, done=0, sum= 1206, outResult=    0
6     45 inA=152, go_l=1, done=0, sum= 1914, outResult=    0
7  checkTotal: value= 2066, total= 2066
8  At time= 55, got right value
9     55 inA=  0, go_l=1, done=1, sum= 2066, outResult=    0
10    65 inA=304, go_l=0, done=0, sum=    0, outResult= 2066
11    75 inA=223, go_l=1, done=0, sum=  304, outResult= 2066
12    85 inA=108, go_l=1, done=0, sum=  527, outResult= 2066
13 checkTotal: value= 635, total= 635
14 At time= 95, got right value
15    95 inA=  0, go_l=1, done=1, sum=  635, outResult= 2066
16   105 inA=891, go_l=0, done=0, sum=    0, outResult=  635
17    ...
```

**Рис. 7.8.** Результаты запуска примера с экземплярами класса

Важно отметить, что представленные в этом разделе тестовые окружения похожи на традиционные программы. Здесь нет ничего удивительного: функциональность тестовых окружений ближе к программам, чем к аппаратуре.

## 7.4.6. Обратите внимание

Мы рассмотрели разработку нескольких версий тестового окружения и каждый раз использовали новые возможности языка. Более подробно эти возможности будут описаны в следующих разделах.

Нужно отметить, что итоговое тестовое окружение проверяет не все. Написать тестовое окружение, которое проверяет *абсолютно все*, сложно. Сложно даже определить, что такое «абсолютно все»! Приведем примеры функциональности, которая не была охвачена.

- Протокол интерфейсного взаимодействия не был полностью проверен. Приведенные здесь примеры предполагали, что он в целом работает. Что случится, если go_l будет установлен больше, чем на один такт? До сих пор мы не рассматривали такую возможность и поэтому считали, что значение go_l имеет смысл проверять только сразу после сброса и после установки inA значения 0. Каким предполагалось поведение тестируемой модели при установке go_l в некорректные моменты времени?

- Был ли сумматор полностью проверен? Убедились ли мы, что перенос работает на всех 16 битах? А если возникает переполнение? Похоже, что нам все равно.

То есть мы только лишь набросали общую картину тестирования этого сравнительно простого аппаратного потока.

## 7.5. Использование случайных значений

Тестовое окружение создает входные значения для тестируемой модели, чтобы определить, корректно ли реализована ее функциональность и временны́е характеристики. Входные значения могут быть направленными на создание тестовых ситуаций, когда тестовый вектор или тестовая последовательность разрабатывается для проверки определенного аспекта модели. Однако часто в тестовых векторах используются и случайные значения, так как они дают возможность проверить широкий диапазон возможных ошибок.

SystemVerilog поддерживает объектно-ориентированный подход к созданию случайных значений внутри объектов, которые могут быть использованы в качестве стимулов (тестовых воздействий) для тестируемой модели. С учетом заданных ограничений на значения (constraints) генератор случайных чисел может создавать более эффективные в обнаружении ошибок тестовые векторы. Также он поддерживает средства создания случайных тестовых последовательностей.

### 7.5.1. Определение случайной переменной

Переменные класса могут быть определены в качестве случайных посредством использования модификаторов типа `rand` или `randc`, как это показано ниже.

```
1 class testVectors;
2   rand  bit   [4:0] fiveBits;
3   randc logic [8:0] nineBits;
4 endclass
```

В этом примере класс `testVectors` определен имеющим две случайные переменные: `fiveBits` и `nineBits`.

Модификатор `rand` указывает на то, что переменная `fiveBits` примет произвольное значение в диапазоне 0–31. Это произойдет при вызове метода `randomize()`, как будет вскоре показано. При последовательном вызове метода рандомизации новые значения будут произвольным образом выбраны из диапазона 0–31 с равной вероятностью. То есть рандомизация выполняется с *возвратом*, что означает выбор любого числа из заданного диапазона с вероятностью 1/32. Таким образом, в последовательности значений могут встретиться и одинаковые.

Модификатор `randc` указывает на то, что новые величины должны выбираться циклично: повторяющиеся значения появляются, когда все остальные значения диапазона уже были выбраны. Это рандомизация *без возврата*. Даже в случае переменной типа `logic` случайные величины не будут включать значения `x` и `z`.

Объект рассматриваемого класса может быть создан в процедурном блоке, а его переменные могут быть там же рандомизированы; все это показано ниже.

```
5  bit   [4:0] outputA;
6  logic [8:0] outputB;
7  bit         condition, clk;
8  testVectors tv = new;
9
10 initial begin
11   while (condition) begin
12     @(posedge clk);
13     assert (tv.randomize()) else $error ("OOPS, randomize didn't work");
14     outputA <= tv.fiveBits;
15     outputB <= tv.nineBits;
16   end
17 end
```

В примере декларируется несколько переменных. В строке 8 путем вызова метода new создается объект tv. Внутри блока инициализации пока condition имеет значение TRUE, исполняется цикл while. Каждая итерация цикла осуществляется по фронту тактового сигнала: у объекта tv вызывается метод randomize (строка 13), что приводит к присваиванию новых случайных значений переменным объекта; в строках 14–15 эти значения присваиваются выходам.

Метод randomize возвращает TRUE, если рандомизация прошла успешно, и TRUE, если нет. В последнем случае вызывается процедура $error. В следующем разделе будет рассказано, зачем проверять результат вызова randomize. (Вместо оператора assert, если вы с ним незнакомы, можно использовать оператор if.)

Рандомизация может применяться и к перечислимым типам. В примере ниже при вызове метода randomize объекта fc переменной myRainbow присвоится случайно выбранное значение перечисления roy_g_biv_t.

```
1  typedef enum
2    bit[2:0] {Red, Orange, Yellow, Green, Blue, Indigo, Violet} roy_g_biv_t;
3
4  class funnyColors;
5    rand  roy_g_biv_t myRainbow;
6  endclass
7
8  funnyColors fc = new;
```

## 7.5.2. Ограничения на случайные значения

Возможность использовать случайные переменные полезна, но возможность ввести ограничения (constraints) на их допустимые значения означает еще больший контроль над процессом тестирования. В SystemVerilog есть несколько средств описания ограничений.

Один из подходов к созданию случайных значений заключается в использовании ограничений совместно с методом рандомизации:

```
tv.randomize() with {nineBits <= 500 && nineBits >= 250;};
```

Это приводит к тому, что возможные значения nineBits ограничиваются диапазоном от 250 до 500. Никакого влияния на выбор случайного значения для fiveBits это не окажет. Чтобы использовать проверку утверждения, необходимо написать следующее:

```
1  assert (tv.randomize() with {nineBits <= 500 && nineBits >= 250;})
2    else $error ("OOPS, randomize didn't work");
```

Диапазон, используемый в ограничениях, может быть параметризован:

```
1  logic [8:0] y;
2  assert (tv.randomize() with {nineBits <= y && nineBits >= 250;})
3    else $error ("OOPS, randomize didn't work");
```

Здесь значение y может быть установлено тестовым окружением. Конечно, если y будет присвоено что-то, меньшее 250, невозможно будет найти случайное значение, удовлетворяющее ограничению, метод рандомизации возвратит FALSE, и вызовется $error.

Ограничения, заданные таким образом, могут включать несколько случайных переменных. Ограничениями могут быть любые выражения над переменными и константами типов bit, byte, reg, logic, integer, enum и packed struct.

Теперь рассмотрим создание случайных входных значений для тестирования аппаратного потока sumItUp. Этот поток, представленный в разделе 5.1, требует посылки ему информационного пакета. Пакет состоит из последовательностей чисел для сложения. Когда устанавливается go_l, первое значение подается на вход. Подача 0 на вход означает окончание последовательностей значений. То есть 0 не может быть числом, подлежащим сложению.

В примере 7.9 описываются два класса. Класс sumPkt является генератором случайных чисел, учитывающим ограничения на их значения. Класс testSumThread создает массивы со случайными числами, полученными из класса sumPkt, для подачи в аппаратный поток sumItUp. Затем он проверяет, совпадает ли значение total, вычисленное на основе подаваемых на сложение чисел, со значением sum, полученным потоком sumItUp.

```
1  class sumPkt;
2    rand bit [15:0] howMany;
3    rand bit [15:0] item[];
4
5    constraint N {howMany inside {[2:10]};}
6    constraint arraySize {item.size == howMany;}
7  endclass
8
9  class testSumThread;
10   local bit unsigned [15:0] total;
11   local sumPkt pkt = new;
12
13   task sendPktToThread;
14     total = 0;
15     assert (pkt.randomize()) else $error ("oops");
```

```
16      for (byte i = 0; i < pkt.howMany; i++) begin
17        @(posedge ck);
18        inA <= pkt.item[i];
19        total += pkt.item[i];
20        go_l <= (i == 0) ? 0 : 1;
21      end
22      @(posedge ck);
23      inA <= 0;
24    endtask
25
26    function checkTotal (bit unsigned [15:0] value);
27      return value == total;
28    endfunction
29  endclass
```

**Пример 7.9.** Классы для создания случайных входных данных для аппаратного потока sumItUp

Класс sumPkt определяется в строках 1–7 данного примера. Переменная howMany, принимающая случайное значение, показывает, сколько именно чисел будет в передаваемом пакете. Динамический массив (строка 3) будет содержать последовательность значений для передачи. В строке 5 определено ограничение N, в котором задано множество возможных значений переменной howMany – диапазон от 2 до 10 включительно. То есть, несмотря на то что это 16-битная переменная, она будет принимать значения именно от 2 до 10.

Описание множества может быть усложнено. Например,

```
constraint N {howMany inside { [2:10], 13, [25:73] }; }
```

включит также число 13 и числа в диапазоне между 25 и 73 включительно в возможные значения howMany.

Второе ограничение, называемое arraySize, указывает на то, что значение item.size (количество элементов в динамическом массиве) должно быть равно howMany. Когда будет вызвана рандомизация объекта типа sumPkt, будут разрешены оба ограничения и создан динамический массив howMany случайных чисел. Если механизму разрешения ограничений не удастся найти подходящие значения, удовлетворяющие ограничениям, – например, из-за того, что эти ограничения некорректно заданы, – метод рандомизации вернет FALSE.

Класс testSumThread показан в строках 9–29 примера 7.9. В нем присутствует внутренняя переменная для хранения вычисленной суммы. Эта переменная используется методом checkTotal, который сравнивает ее значение с передаваемым ему значением value и определяет, корректно ли аппаратный поток sumItUp осуществил суммирование. В строке 11 создается объект pkt типа sumPkt.

Метод sendPktToThread вызывает рандомизацию pkt, создавая динамический массив pkt.item[], где количество элементов будет равно pkt.howMany. Затем эти элементы посылаются в тестируемую модель (поток sumItUp), а после них в модель посылается 0. В процессе посылки вычисляется сумма передаваемых значений.

Метод sendPktToThread вызывается из блока инициализации, показанного в примере 7.10. В этом блоке создается объект t типа testSumThread, за чем следует цикл. Тело этого цикла for повторяется 100 раз, вызывая метод sendPktToThread. Как описано выше, при каждом вызове этот метод создает информационный пакет и посылает его в тестируемую модель. Когда моделью устанавливается сигнал done, вызывается метод checkTotal для проверки подсчитанной моделью суммы (sum). Если сумма некорректна, проверка утверждения в строке 6 завершается с ошибкой.

```
1  initial begin
2    testSumThread t = new;
3    for (int i = 0; i < 100; i++) begin
4      t.sendPktToThread();
5      wait (done);
6      ChkTotal: assert (t.checkTotal(sum))
7        else $error("OOPS");
8    end
9    $finish;
10 end
```

**Пример 7.10.** Тестовое окружение в виде потока для примера 7.9

Таким образом, циклом будет послано в поток 100 пакетов со случайными длинами и содержимым.

## 7.5.3. Случайный выбор

Для случайного выбора одного из нескольких вариантов используется конструкция randcase. Пусть мы хотим проверить некую сеть (network), посылая в нее пакеты разных типов. В примере используются три типа пакетов, обозначенных A, B и C.

```
1  randcase
2    1: sendPacketA;
3    3: sendPacketB;
4    6: sendPacketC;
5  endcase
```

У оператора randcase есть три альтернативы, каждая из которых вызывает посылку пакета в сеть. Оператор randcase неявно вызывает функцию $urandom_range() для генерации случайного числа. Диапазон генерации определяется суммой значений выражений во всех альтернативах[44]. Вероятность выбора конкретной альтернативы получается делением значения соответствующего выражения на сумму. В данном случае сумма равна 10; метод sendPacketB, например, будет вызываться в среднем в 3 случаях из 10.

Вес альтернатив может задаваться с помощью переменных. При каждом исполнении randcase вычисляется сумма значений переменных, и на основе этого генерируется случайное число. Если окажется так, что одна из переменных

---

[44] Все значения (так называемые *веса*) являются неотрицательными: $w_i \geq 0$, где $i \in \{1, \ldots n\}$. Генерируется случайное число $\xi$ из полуинтервала $\left[0, \sum_{i=1}^{n} w_i\right)$; альтернатива с номером $k$ выбирается тогда и только тогда, когда $\sum_{i=0}^{k-1} w_i \leq \xi < \sum_{i=0}^{k} w_i$.

имеет значение 0, эта ветвь вызываться не будет. Порядок, в котором записаны альтернативы, не важен.

## 7.5.4. Случайные последовательности

Часто посылка информации в тестируемую модель осуществляется в виде более сложной последовательности данных, чем мы рассматривали до этого. Конструкция randsequence-endsequence позволяет задать грамматику последовательностей в нотации, близкой к БНФ[45]. Используя правила грамматики (productions), генератор строит случайные последовательности с заданной структурой. Получаемые последовательности можно сразу подать на тестируемую систему, а можно сохранить, чтобы подать позднее.

Рассмотрим очень простое применение случайной последовательности в примере 7.11. Эта последовательность должна заменить собой процедуру sendPktToThread примера 7.9. В этом примере тестовое окружение посылает 16-битное слово в тестируемую модель, устанавливая go_l. Затем оно посылает некоторое число случайных ненулевых чисел при сброшенном go_l. Последним посылается число 16'h0000, указывающее на конец передачи пакета. Мы можем представить себе эти шаги как «предложение» языка, описывающего тестовые последовательности.

```
1  task  sendPktToAddRandSequence;
2    randsequence (main)
3      main:   first middle last;
4      first:  {total = 0; sendIt(0); } ;
5      middle: repeat ($urandom_range(6, 1)) item;
6      item:   {sendIt(1); };
7      last:   {@(posedge ck); inA <= 0; };
8    endsequence
9  endtask
10
11 task sendIt (int go);
12   @(posedge ck);
13   inA <= getValue;
14   go_l <= go;
15 endtask
16
17 function int getValue;
18   getValue = $urandom_range(1000,1);
19   total += getValue;
20   return getValue;
21 endfunction;
```

**Пример 7.11.** Случайные последовательности

Правила грамматики обрамляются скобками randsequence-endsequence (строки 2–8). Первое правило, main, говорит, что последовательность состоит из трех частей: first, middle и last. Это правило является *нетерминальным*[46], поскольку в его правой части есть обращения к правилам. Имя, указанное в скобках (строка 2), определяет, какое правило является целевым, т. е. активируется первым.

Правило first находится в строке 4. Оно является *терминальным*, так как полностью специфицирует то, что должно быть сделано (другие правила в нем не вызываются). Содержимое фигурных скобок – процедурный код, исполня-

[45] БНФ (Бэкуса–Наура форма) – нотация, используемая для описания контекстно-свободных грамматик.

[46] В теории формальных грамматик *терминальными* и *нетерминальными* называют символы грамматики, а не правила. Соответственно, main, first, middle и last – имена нетерминальных символов, а не правил.

емый при применении правила (так называемый *блок кода*). В данном случае переменной total (декларация не показана, но это переменная типа int) присваивается 0, после чего вызывается sendIt(0). Процедура sendIt (строки 11–15) вызывает функцию getValue, которая генерирует случайное число в диапазоне от 1 до 1000 и прибавляет его к total, после чего передает это число в тестируемую модель, устанавливая при этом go_l (значение для установки передается в качестве аргумента). После того как все это произошло, правило first завершается, а main переходит к правилу middle.

Правило middle является нетерминальным, поскольку в нем случайное число раз (от 1 до 6) применяется правило item. В item находится процедурный код, в котором вызывается sendIt(1) для посылки еще одного случайного числа в тестируемую модель, на этот раз при сброшенном go_l. Когда правило middle завершается, начинает обрабатываться правило last. Оно лишь отправляет 0 в тестируемую модель. Обработка правила main завершается: последовательность создана; процедура sendPktToAddRandSequence завершает свою работу. Можно считать, что эта процедура вызывается из тестового окружения, описанного в примере 7.10 (строка 4).

Это очень простой пример, который не раскрывает всей мощи конструкции randsequence, а лишь демонстрирует, как она работает. Единственное, что является здесь случайным, – так это вызов $urandom_range.

Пример 7.12 расширяет предыдущий рандомизацией выбора применяемого правила (строки 6–7). Теперь в item возможен выбор одного из двух правил: good и bad (они разделены вертикальной чертой |). Это означает, что часть item может быть как «хорошей» (правило good), так и «плохой» (правило bad) – оба варианты допустимы грамматикой. В процедуру sendIt был добавлен второй аргумент (bad), интерпретируемый следующим образом: когда его значение равно 1, к значению, передаваемому в тестируемую модель, прибавляется единица; это приводит к расхождению фактического и ожидаемого результатов вычислений, что можно увидеть при запуске тестов.

Итак, правило item может передавать либо «хорошие», либо «плохие» пакеты. Вероятности выбора того или иного варианта задаются в строках 6 и 7: в 99% случаев при применении item будет передан «хороший» пакет, а в 1 % случаев – «плохой». В данном примере веса заданы константами, но они могут быть произвольными выражения-

```
1  task sendPktToAddRandSequence;
2    randsequence (main)
3      main:   first middle last;
4      first:  {total = 0; sendIt(0, 0); } ;
5      middle: repeat ($urandom_range(6, 1)) item;
6      item:   good := 99
7              | bad := 1 // давайте сделаем ошибку
8              ;
9      good:   {sendIt(1, 0); };
10     bad:    {sendIt(1, 1); };
11     last:   {@(posedge ck); inA <= 0; };
12   endsequence
13 endtask
14
15 task sendIt (int go, bad);
16   @(posedge ck);
17   inA <= (bad) ? getValue + 1 : getValue;
18   go_l <= go;
19 endtask
```

**Пример 7.12.** Случайная последовательность с ошибкой

ми (тогда они записываются в скобках). Сумма их значений вычисляется при вызове правила и используется для расчета вероятностей выбора альтернатив. Если значение выражения равно 0, правило не может быть применено.

Как отмечалось ранее, внутри конструкции randsequence в фигурных скобках может находиться процедурный код. Некоторые операторы можно использовать в правилах без обрамления скобками: repeat (строке 5), if-else и case (не показаны в примере). Это дает дополнительную гибкость в определении грамматики. Например, оператор if-else можно использовать следующим образом:

```
1 randsequence
2   ...
3   Prod1:  if (errorMode) ProdE else ProdOK;
4   ProdE:  ...;
5   ProdOK: ...;
```

Пример использования оператора case приведен ниже:

```
1 randsequence
2   ...
3   Prod1:  case (typeOfMsg)
4             0:          ProdA;
5             1, 2, 3, 5: ProdB;
6             4:          ProdC;
7             default:    ProdOops;
8           endcase;
```

Операторы break и return используются для досрочного завершения правила. Оператор break позволяет завершить цикл, например repeat (если циклов несколько и они вложенные – ближайший к break), или перейти к исполнению операторов, находящихся после endsequence (при отсутствии циклов). Оператор return прерывает исполнение правила: генерация последовательности продолжается со следующего правила.

Правила могут принимать и возвращать значения. Это напоминает вызовы функций. Каждое правило создает область видимости для используемых в нем правил и блоков кода.

Передача значений иллюстрируется в примере 7.13. У правил указывается тип возвращаемого значения. Например, правило, определенное в строке 8, возвращает 8-битное число, а правила, определенные в строках 9–10 и 14–16, – строки. В блоках кода используется оператор return.

```
1 module rpCalcCheck;
2   initial begin
3     repeat (10)
4       randsequence (rp_calc)
5         rp_calc: value enter value op
6                  {$display("%d %s %d %s",
7                    value[1], enter, value[2], op); };
```

```
8          bit [7:0] value: {return $urandom; } ;
9          string enter:    {return "<enter>"; } ;
10         string op:       add := 5 {return add;}
11                          | sub := 3 {return sub;}
12                          | mul := 2 {return mul;}
13                          ;
14         string add:      {return "+"; } ;
15         string sub:      {return "-"; } ;
16         string mul:      {return "*"; } ;
17       endsequence
18   end
19 endmodule: rpCalcCheck
```

**Пример 7.13.** Генератор последовательности для калькулятора

Правило rp_calc является целевым. В нем вызываются правила value, enter, value и op (в указанном порядке). После применения всех этих правил исполняется блок кода: процедура $display выводит на консоль строку, содержащую возвращенные правилами значения. Обратите внимание, что правило value применяется дважды и оба его значения сохранены. В блоке кода правила rp_calc неявно объявлены следующие переменные:

```
bit [7:0] value [1:2];
string    enter, op;
```

То есть результаты применения правила value сохраняются в массиве.

При запуске примера на консоль будет выведено:

```
54 <enter> 60 +
226 <enter> 11 -
64 <enter> 247 +
27 <enter> 181 *
78 <enter> 21 +
114 <enter> 150 *
196 <enter> 170 *
207 <enter> 79 -
23 <enter> 136 +
44 <enter> 206 +
```

Здесь мы можем видеть случайный характер выбора альтернатив, определенных в строках 10–12.

Ниже показана возможность передачи в правила аргументов.

```
1  randomsequence
2    main:               msg("hello") msg("world") msg( ) ;
3    msg (string s="!"): {$display(s); } ;
4    ...
```

В этом фрагменте кода в main три раза вызывается правило msg, имеющее строковый аргумент (значение по умолчанию – "!"). На консоль выведется "hello\nworld\n!".

# 7.6. Полезные конструкции

## 7.6.1. Иерархические имена

*Иерархическое именование* дает возможность обращаться к переменным, процедурам и функциям, которые находятся за пределами текущей области видимости. Каждый из следующих элементов определяет свою область видимости имен: модули, программы, интерфейсы, классы, процедуры, функции, блоки begin-end, блоки fork-join, пакеты. Внутри каждой из этих областей не разрешается объявление нескольких сущностей с одинаковыми именами. Так, например, нельзя объявить битовую переменную и процедуру с одним и тем же именем. Когда компилятору встречается обращение к имени переменной, процедуры или функции, он просматривает текущую область видимости в поисках этого элемента. Однако иерархическое именование дает ему доступ ко всей тестируемой модели в целях решения задач, стоящих перед тестовым окружением.

Иерархия объектов SystemVerilog является статической древовидной структурой. Каждый из следующих элементов: экземпляр модуля, процедура, функция, именованный блок begin-end или именованный блок fork-join – добавляет в иерархию новый уровень ниже текущего. Заметим, что вложенной в эту иерархию оказывается как структурная иерархия, состоящая из модулей, программ, интерфейсов, объявленных внутри других модулей, так и процедурная иерархия, состоящая из процедур, функций и именованных блоков (begin-end и fork-join). Поскольку тестируемая модель начинается со структурной иерархии, сущностями верхнего уровня являются модули и их экземпляры. Затем внутри них начинается процедурная иерархия, так как именованные блоки становятся вложены в блоки always, блоки инициализации, функции и процедуры. Автоматические и динамические переменные не могут быть адресованы с помощью иерархических имен.

```
1 module top;
2    ...
3    design dut ( .*);
4    Tbench  tb (.*);
5    byte i;
6
7    initial $monitor($time,
8      " Current State = %b", dut.state);
9  endmodule: top
10
11 program Tbench (декларации портов)
12  int i;
13    ...
14 endprogram: Tbench
15
16 module design (декларации портов);
17  logic [4:0] state;
18  initial begin: newScope
19    byte j;
20    for (j = 0; j < 10; j++) begin: loopScope
21      bit k;
22    end: loopScope
23  end: newScope
24   ...
25 endmodule: design
```

**Пример 7.14.** Иерархические имена

Полное иерархическое имя начинается с верхнего уровня иерархии, с имени модуля верхнего уровня, и продолжается нижележащими именами вплоть до имени требуемого объекта. В примере 7.14 показано несколько простых определений объектов, а на рис. 7.9 дана их древовидная структура.

На этой диаграмме структурная иерархия начинается с модуля верхнего уровня top, в котором находятся экземпляры модуля dut и программы tb. В области видимости имен модуля top также находится байт i. Внутри экземпляра программы tb есть другая переменная i, а внутри экземпляра модуля dut – переменная state. Согласно концепции области видимости, эти две переменные i различаются. В dut также имеется процедурная иерархия, начинающаяся с именованного блока begin-end (newScope) в строке 18 (пример 7.14). Эта иерархия становится глубже за счет именованного блока begin-end в цикле for (loopScope) в строке 20. Внутри него объявляется битовая переменная k.

Разделителем в иерархических именах является точка, причем каждая точка соответствует обращению к следующему, более низкому уровню иерархии. Полными иерархическими именами в данной модели являются: top.i, top.tb.i, top.dut.state, top.dut.newScope.j и top.dut.newScope.loopScope.k. Структурная часть имен иерархии включает имена экземпляров модулей и программ. Процедурная часть имен начинается с newScope и обведена серой линией на рис. 7.9.

Если модель содержит несколько модулей top, то приведенных выше имен будет достаточно для обращения к экземплярам объектов внутри top. Если же в модели будет еще один модуль, экземпляр которого отсутствует в каком-либо другом модуле, назовем его topB, то для законченности в иерархические имена может быть добавлен $root. Тогда любое из приведенных выше имен будет записываться как $root.top..., а те имена, что обращаются к объектам из иерархии topB, должны будут записываться в виде $root.topB....

Вообще говоря, можно обойтись и без полных иерархических имен. Имя может быть записано в виде область_видимости.сущность (или несколько уровней иерархии, предшествующих адресуемой сущности). Компилятор начинает раскрывать такое имя с поиска адресуемой области_видимости в текущей области. Если она будет найдена, оставшаяся часть имени будет, в свою очередь, найдена просмотром вниз по ее уровням иерархии. Таким образом, если мы находимся в области видимости newScope примера 7.14, переменная k в нижележащем уровне иерархии может быть адресована как loopScope.k.

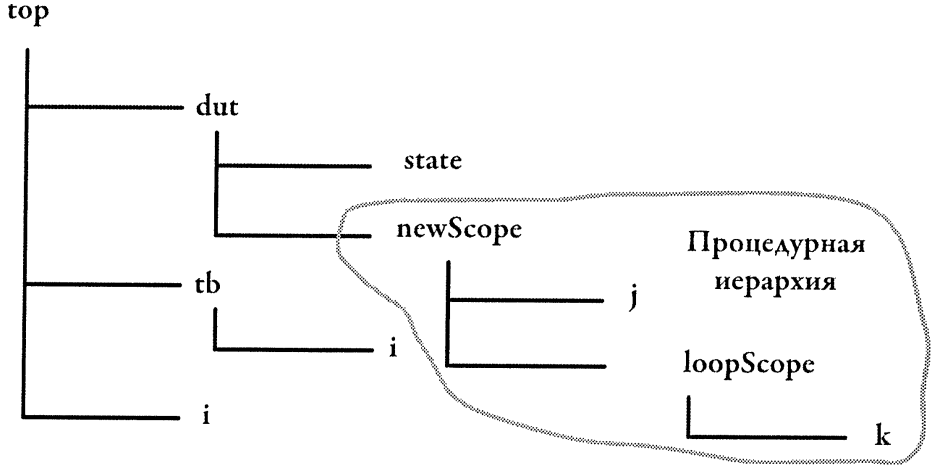



**Рис. 7.9.** Иерархические имена

Если адресуемая область_видимости не будет здесь найдена, компилятор будет искать ее в области видимости более высокого уровня; это будет означать поиск по процедурной иерархии внутри текущего элемента структурной иерархии. Если бы мы находились в области видимости программы Tbench, переменной k соответствовало бы имя dut.newScope.loopScope.k. Поскольку область видимости dut неизвестна внутри данной программы, компилятор будет искать ее по всей иерархии создания экземпляров сущностей. Когда dut найдется, оставшаяся часть имени подставляется спуском по иерархии вниз от этой точки.

В случае функций и процедур ситуация несколько иная. Если определение функции или процедуры не находится в иерархии более высокого уровня внутри структурного иерархического элемента (например, модуля), компилятор будет искать это определение на верхнем уровне иерархии имен. Если же компилятор не найдет его и здесь, то он продолжит поиски в иерархии экземпляров сущностей. Это позволяет часто используемым функциям и процедурам быть определенными на верхнем уровне иерархии, в целях их общего использования.

Если происходит прямое обращение к переменной (то есть без первой части имени, являющейся областью видимости), то компилятор будет искать это имя в текущей области видимости. Если компилятор не найдет его здесь, он будет просматривать процедурную иерархию в рамках текущего элемента структурной иерархии, но он не будет использовать иерархию создания экземпляров сущностей. Таким образом, если мы находимся в области loopScope и обращаемся к переменной state, компилятор не найдет ее в loopScope и просмотрит процедурную иерархию, найдя state в области имен модуля design. Если мы обращаемся к переменной i из loopScope, она не будет найдена компилятором, так как он не будет анализировать ничего за границами модуля design. (Поиск переменной будет начат в текущем пространстве имен и продолжится вверх по иерархии. Поиск не выйдет за пределы модуля, программы или интерфейса.)

Имена переменных считаются локальными в модулях, программах и интерфейсах; это позволяет получить читаемый стиль написания кода, который защищен от внешнего доступа к предположительно локальным переменным. Для доступа к переменным за пределами текущего элемента структурной иерархии из тестового окружения разрешено использовать только иерархические имена. А вот использование иерархических имен внутри модулей не только не нужно, но и может привести к катастрофическим последствиям.

## 7.6.2. Операторы force/release и assign/deassign

При отладке тестируемой модели иногда нужно проверить ее внутренние компоненты в связке с другими компонентами. Так, частью сценария отладки является просмотр того, как выходные сигналы влияют на работу логических вентилей, присоединенных к ним. Рассмотрим АЛУ, управляющее выходной шиной, присоединенной к нескольким регистровым входам. Может появиться

желание установить на выход АЛУ некоторую величину, проверяя, какие значения будут загружены в регистры (тем самым определяется, есть проблемы в коммутировании шины и логике разрешения загрузки).

Процедурные операторы force и release позволяют разработчику это сделать. Оператор force устанавливает значение переменной (или цепи) до вызова оператора release для той же самой переменной (или цепи). В промежутке времени между вызовами force и release переменной присваивается значение, указанное в force (все изменения этой переменной игнорируются). Вызовы release для переменных, устанавливаемых процедурными и непрерывными присваиваниями, ведут себя по-разному (см. пример ниже).

В правой части оператора force может быть выражение общего вида. В этом случае оператор работает как непрерывное присваивание: каждый раз при изменении значения правой части обновляется значение левой. С помощью оператора force нельзя присвоить значение отдельной части переменной.

Пример 7.15 показывает использование force и release для замещения выходных значений двух разных сумматоров. Первый, заданный с помощью always_comb, показан в строках 20–21, второй использует непрерывное присваивание (строка 23). Их выходными переменными являются соответственно resultA и resultB. Ниже показан результат симуляции.

```
1  module forceRelease;
2    logic [7:0] resultA, resultB, b, c;
3
4    initial begin
5      $monitor(
6      "T=%3d, resultA=%h, resultB=%h, b=%h,  c=%h",
7        $stime, resultA, resultB, b, c);
8      b = 0;
9      c = 0;
10     #5;
11     force resultA = 8'h55;
12     force resultB = 8'h55;
13     #100;
14     release resultA;
15     release resultB;
16     #5 b = 1;
17     #5 $finish;
18   end
19
20   always_comb
21     resultA = b + c;
22
23   assign resultB = b + c;
24 endmodule: forceRelease
```

**Пример 7.15.** Использование операторов force и release

При исполнении блока инициализации в строке 10 осуществляется задержка на 5 единиц времени; выходные значения сумматоров равны 0. В момент

времени 5 исполняются операторы force, которые замещают значения resultA и resultB; это показано в строке 2 результатов симуляции (см. ниже): выходные значения становятся равными 8'h55.

Симуляция возобновляется в момент времени 105. Исполняются операторы release, которые отменяют force для resultA и resultB. Обратите внимание, что значение resultB изменится, а значение resultA сохранится. Это происходит потому, что отмена force для переменной, значение которой устанавливается непрерывным присваиванием, приведет к возобновлению работы этого непрерывного присваивания. Отмена force для переменной, значение которой устанавливается с помощью процедурных присваиваний, не изменит значения переменной (значение изменится при очередном присваивании). В нашем случае это произойдет в момент времени 110, когда b будет присвоено 8'h01. В этот момент времени меняются значения как resultA, так и resultB.

```
1  Time= 0, resultA=00, resultB=00, b=00, c=00
2  Time= 5, resultA=55, resultB=55, b=00, c=00
3  Time=105, resultA=55, resultB=00, b=00, c=00
4  Time=110, resultA=01, resultB=01, b=01, c=00
```

Если переменная, используемая в операторе force, является выходной для блока always_ff, то замещающее значение этой переменной сохранится после отмены force до следующего вызова блока always_ff.

В примере 7.15 ключевое слово force могло бы быть заменено на assign, а release – на deassign. Разница между force/release и assign/deassign заключается в том, что последние не применимы к цепям.

## 7.6.3. Завершение симуляции

SystemVerilog поддерживает три разных способа нормального завершения симуляции. Нормальное завершение предполагает отсутствие найденных ошибок или просто достижение конца кода тестового окружения. Эти способы оформлены в следующие системные процедуры:

- $stop – симуляция приостанавливается, управление возвращается симулятору;
- $finish – симуляция завершается, управление передается в операционную систему;
- $exit – симулятор ожидает завершения работы всех программных блоков и делает неявный вызов $finish.

$finish и $exit используют необязательный аргумент, определяющий тип выводимого диагностического сообщения. Значение аргумента, равное 0, вообще не приведет к выводу на консоль. Значение, равное 1 (значение по умолчанию), приведет к печати текущего времени симуляции и текущей позиции. Значение, равное 2, помимо прочего, приведет к выводу статистики использования процессора при симуляции.

# 7.7. Отладка с использованием процедур ввода-вывода

## 7.7.1. Процедуры $display, $monitor и $strobe

При отладке часто используются системные процедуры `$display`, `$monitor` и `$strobe`, осуществляющие вывод на консоль.

На вход процедурам подаются аргументы (строки или выражения), разделенные запятыми. Строки могут задавать формат вывода других аргументов. Для этого используются управляющие последовательности, начинающиеся с символа %. Например, вызов

```
$display("a=%d, and b=%h", a, b);
```

приведет к выводу на консоль "a=10, and b=A", если значения аргументов a и b равны 4'd10 (%d определяет десятичный формат, а %h – шестнадцатеричный). Если у аргумента формат не указан, используется формат по умолчанию. Код ниже выводит значение c в десятичном виде, а потом – "a=10, and b=A".

```
$display(c, "a=%d, and b=%h", a, b);
```

Две запятые, идущие подряд, указывают на отсутствие аргумента:

```
$display(c,, "a=%d, and b=%h", a, b);
```

Вместо каждого пропущенного аргумента печатается единичный пробел.

Переменная c в примере выше печатается с использованием формата по умолчанию, а именно в десятичном виде. Имеются альтернативные варианты процедуры `$display` со следующими форматами по умолчанию: двоичный для `$displayb`, восьмеричный для `$displayo`, шестнадцатеричный для `$displayh`. Процедуры `$strobe` и `$monitor` также имеют альтернативные варианты.

Имеются следующие форматы вывода:

| Управляющая последовательность (допускается верхний и нижний регистры) | Формат вывода |
|---|---|
| %h или %x | Шестнадцатеричный |
| %d | Десятичный |
| %o | Восьмеричный |
| %b | Двоичный |
| %c | Символ из таблицы ASCII |
| %s | Строка |

Между символом % и буквенным спецификатором формата может быть вставлено число. Если это 0, значение печатается без дополнительных выравнивающих пробелов. Ненулевое число задает ширину области печати (даже если ширина недостаточна, значение будет выведено целиком). Если же число не задано, ширина области печати полагается максимальной для значений выводимого типа[47].

[47] Это относится к числам (и соответствующим форматам вывода), но не к строкам.

Управляющая последовательность %m указывает на печать иерархического контекста. Если мы хотим вывести %, в форматной строке необходимо указать %%.

Внутри строки могут также использоваться специальные последовательности, начинающиеся с символа \.

| Специальные последовательности | Выводимые символы |
|---|---|
| \n | Новая строка |
| \t | Табуляция |
| \\ | \ |
| \" | " |
| \ в конце текстовой строки | Не печатается ничего. Позволяет записать строку в нескольких строках кода |
| \ddd | Символ ASCII, заданный восьмеричным эквивалентом (0 ⩽ d ⩽ 7) |

## 7.7.2. Контроль выводимых величин

Вызовы системных процедур $display, $monitor и $strobe выглядят одинаково, однако работают эти процедуры по-разному: вывод ими значений осуществляется в разные моменты времени и обусловлен разными причинами.

| Системная процедура | Когда происходит печать и при каких условиях |
|---|---|
| $display | $display может находиться в любом процедурном блоке. Печать происходит при вызове процедуры, печатаются переданные ей значения |
| $monitor | $monitor вызывается из тестового окружения для настройки списка выводимых значений. Печать происходит только при изменении значений переданных переменных. Печать происходит в конце цикла симуляции (на этапе отложенной обработки), поэтому все переменные имеют финальные значения для текущего момента времени |
| $strobe | $strobe может находиться в любом процедурном блоке. Печать происходит в конце цикла симуляции (на этапе отложенной обработки), поэтому все переменные имеют финальные значения для текущего момента времени |

Рассмотрим следующий код:

```
1  always_ff @(posedge clock) begin
2      a <= a + c;
3      $display($stime,, "disp: a=%d, b=%d, c=%d", a, b, c);
4      $strobe ("At %d, strobe: a=%d, b=%d, c=%d", $stime, a, b, c);
5    end
6  end
7
8  assign b = a + 1;
```

Как только приходит передний фронт тактового сигнала, исполняется блок always_ff и вызываются $display и $strobe. $display выводит на консоль значения переменных, полученные по этому фронту (значение a не будет обновлено из-за неблокирующего присваивания). $strobe выводит на консоль значения переменных в конце цикла симуляции, связанного с пришедшим фронтом

(печатаются финальные значения переменных, включая изменения в a и b). Вывод процедуры $monitor с такой же форматной строкой, как и у $strobe, будет с ним совпадать.

Вывод $monitor может быть включен и выключен системными процедурами $monitoron и $monirоroff соответственно. В один момент времени может быть только один активный $monitor. Для изменения форматной строки вывода необходимо вызвать $monitor с новым вариантом этой строки.

### 7.7.3. Файловый ввод/вывод

Информацию можно выводить не только на консоль, но и в файл. У каждой описанной процедуры вывода есть своя f-версия, например $fdisplay. Список аргументов f-версии имеет одно отличие – наличие в нем дескриптора файла:

```
$fdisplay (fileDescriptor, "a=%d, and b=%h", a, b);
```

где fileDescriptor – значение типа int. Дескриптор файла можно получить путем вызова системной функции $fopen.

```
int fileDescriptor;
fileDescriptor = $fopen ("filename.txt", "w");
```

$fopen может открыть файл для чтения ("r"), записи ("w") или дополнения ("a") ранее созданного файла, начиная с его последней позиции. Ниже одна из f-версий функций осуществляет вывод в файл filename.txt:

```
$fmonitor (fileDescriptor, "a=%d, and b=%h", a, b);
```

Наконец, по окончании записи мы закрываем файл вызовом $fclose.

```
$fclose (fileDescriptor);
```

Значение дескриптора fileDescriptor, созданное $fopen, должно быть проверено на равенство нулю. Возвращаемое $fopen значение 0 означает, что файл не был открыт. О, нет! Файловая система исчезла со всеми вашими файлами! Шутка. Это значение может быть проверено либо с помощью оператора if, либо с помощью утверждения (assertion):

```
int fileDescriptor;

fileDescriptor = $fopen ("filename.txt", "w");
if (fileDescriptor == 0) $display ("OOPS, all your files were deleted");
```

### 7.7.4. Процедуры $readmemh и $readmemb

Системные процедуры $readmemh и $readmemb предназначены для чтения и загрузки данных из текстовых файлов в массивы памяти. Форма h предназначена для чтения шестнадцатеричных чисел, форма b касается чтения двоичных данных. Они могут быть вызваны в любом месте:

```
bit [7:0] mem[0:255];
initial $readmemh ("data.txt", mem);
```

Данный код загружает значения в память mem из файла data.txt, начиная с нулевого индекса. Если бы аргументы $readmemh были бы такие, как показано здесь:

```
initial $readmemh ("data.txt", mem, 48);
```

то память бы начала загружаться с индекса 48. Наконец, если аргументы были бы следующими:

```
initial $readmemh ("data.txt", mem, 16, 3);
```

то память бы начала загружаться с индекса 16, а закончила на индексе 3. Последние два аргумента здесь – это стартовый и конечный индексы загрузки в память.

Таким образом, память загружается, начиная с адреса, указанного в аргументе системной процедуры. Если такого аргумента нет, то загрузка начинается с самого младшего адреса памяти, а каждое следующее значение загружается с более старшим индексом. Если стартовый индекс указан, то загрузка значений начинается с него. Если указаны и стартовый, и конечный индексы, то направление загрузки определяется их относительными величинами.

Текстовые файлы для этих функций, в зависимости от их типа, содержат либо шестнадцатеричные, либо двоичные числа. Числа из файла загружаются в память по адресам последовательных слов. Пустые пространства (пробелы, новые строки, табуляции, символы окончания страницы) разрешены в текстовых файлах, как и оба типа комментариев SystemVerilog. Также могут быть использованы неизвестное значение (x), значение высокого импеданса (z) и разделитель-подчеркивание (_).

Текстовые файлы могут также содержать адреса в виде @hh...h, где h – это шестнадцатеричные цифры. Когда эта конструкция обнаруживается в файле с данными, следующая порция данных загружается уже по данному адресу (индексу), а направление загрузки сохраняется (по умолчанию в сторону более старших адресов или в соответствии с относительными значениями стартовых и конечных индексов в вызове, инициировавшем загрузку). В файле может присутствовать несколько указаний на адреса загрузки.

## 7.8. Задачи и упражнения

**7.1.** Одна из функций тестового окружения для комбинационной схемы заключается в создании всех возможных шаблонов входных значений (комбинации единиц и нулей) для тестируемой модели. Каждый шаблон должен находиться на выходе тестового окружения в течение #1 единицы времени.

**A)** Напишите тестовое окружение, которое сделает это все для тестируемой модели с 4 входами. Заголовок модуля показан ниже. Представьте, что выходы a-d соединены со входами тестируемой модели. Продемонстрируйте работоспособность окружения, используя системную процедуру $display.

```
1 module tb4bit
2   (output logic a, b, c, d);
```

**B)** Теперь добавьте параметр, как это показано ниже, задающий разрядность выхода. Вместо индивидуально именованных выходных переменных (например, `a`, `b`, ...) выход должен быть вектором. Продемонстрируйте работоспособность окружения, используя системную процедуру `$display`.

```
1  module tbParam
2    #(parameter  outSize = 4)
3    (output logic ...
```

**7.2.** Напишите тестовое окружение для задачи 3.2, которое проверит все состояния и переходы тестируемого конечного автомата, используя неявно заданный конечный автомат.

**7.3.** Напишите тестовое окружение для задачи 3.3, которое проверит все состояния и переходы тестируемого конечного автомата, используя неявно заданный конечный автомат.

**7.4.** Напишите тестовое окружение для задачи 3.2, которое проверит все состояния и переходы тестируемого конечного автомата, используя последовательность случайных входных данных.

**7.5.** Напишите тестовое окружение для задачи 3.3, которое проверит все состояния и переходы тестируемого конечного автомата, используя последовательность случайных входных данных.

**7.6.** Напишите тестовое окружение для задачи 5.2, которое проверит все состояния и переходы тестируемого конечного автомата, используя последовательности случайных входных данных разной длины. Тестовое окружение должно проверить корректность результатов.

**7.7.** Напишите тестовое окружение для задачи 5.3, которая будет проверять модель с помощью случайных входных данных и также проверит, что индикатор ошибок работает корректно.

# Глава 8

# Параллельные тестовые окружения

SystemVerilog – это язык параллельного (конкурентного) программирования. Тестовые окружения редко состоят из одного или двух блоков инициализации; часто они содержат множество процессов, подающих стимулы и наблюдающих за портами тестируемой модели. Все эти процессы работают совместно: воздействуют на модель, проверяют ее корректность, собирают информацию о функциональном покрытии. Симулятор планирует исполнение процессов в виртуальном времени, однако это выглядит так, будто все они работают параллельно.

## 8.1. Процессы

Процессы, используемые в тестовых окружениях, создаются блоками `initial` и процедурными операторами `fork-join` внутри них (см. также раздел 11.1).

Каждый процесс SystemVerilog – это поток (thread), работающий параллельно с другими потоками. При исполнении в симуляторе процессы могут находиться в одном из четырех состояний, как показано на рис. 8.1. Вначале процесс находится в состоянии *разблокирован*, что означает, что он готов к исполнению, но симулятор еще не запустил его. Разумеется, процессы не могут исполняться одновременно в одно физическое время, поэтому может быть много разблокированных процессов. Симулятор выбирает процессы в произвольном порядке. Процесс *исполняется* до тех пор, пока не достигнет блокирующего оператора, такого как @, # или `wait`. В этой точке исполнение приостанавливается: говорят, что процесс

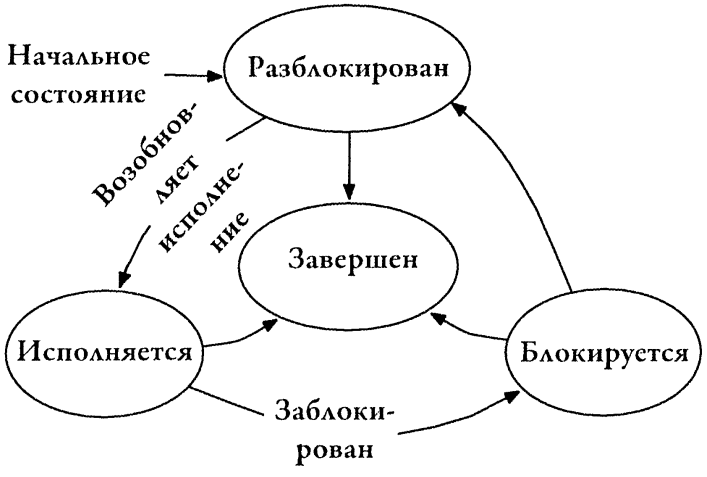



**Рис. 8.1.** Диаграмма состояний процесса

*блокируется*. Процесс остается заблокированным, пока не будет устранена причина блокировки; например, оператор #5 блокирует процесс на 5 единиц времени, `@(posedge clk)` – пока не наступит событие `posedge clk`, `wait` – пока значение выражения, указанного в скобках, равно FALSE. Когда причина блокировки снимается, процесс *разблокируется*; он *возобновляет исполнение*, когда симулятор запустит его.

Все процессы SystemVerilog, даже те, которые описывают аппаратуру с помощью конструкций `always_ff`, `always_comb` и `always_latch`, моделируются состояниями, показанными на рис. 8.1. Симулятор работает с ними точно так же, как с процессами инициализации и процессами `fork-join`, однако лучше такие процессы представлять в терминах триггеров/регистров, комбинационных схем и защелок.

Некоторые процессы, например блоки инициализации, определены статически. Они запускаются в начале симуляции и работают, пока не исполнят последнего оператора; после этого они *завершаются* (и не запускаются повторно). Другие процессы создаются динамически с помощью конструкции `fork-join`. Они исполняются, пока не достигнут конца или не будут принудительно завершены оператором `disable`.

Модель на SystemVerilog может содержать любое число блоков инициализации. Хотя они исполняются однократно, в них, например, можно использовать циклы `forever` и повторять некоторое поведение все время симуляции. Обычно блоки инициализации представляют собой неявно заданные конечные автоматы; для задания состояний в них используются конструкции `@(posedge clk)` и тактовые задержки (##n). Каждый из блоков инициализации может работать как часть тестового окружения: наблюдать за тестируемой моделью или воздействовать на нее через один из портов/интерфейсов.

Подобно обрамлению процедурного блока кода парой `begin-end`, процедурные операторы можно обрамить парой `fork-join`. В блоке `begin-end` операторы исполняются последовательно. В блоке `fork-join` операторы исполняются параллельно: для каждого оператора порождается свой процесс, исполняемый параллельно с другими операторами блока и другими процессами модели. Есть три версии блока `fork-join`, различающиеся поведением родительского процесса:

- `join` – родительский процесс блокируется; он возобновляет исполнение, только когда завершаются все процессы, порожденные в блоке;
- `join_any` – родительский процесс блокируется; он возобновляет исполнение, когда завершается любой из порожденных процессов. Другими словами, первый завершенный процесс блока возобновляет исполнение родительского процесса;
- `join_none` – родительский процесс продолжает работать. Порожденные процессы не начнут исполняться, пока родительский процесс не дойдет до блокирующего оператора.

Ниже на простом примере иллюстрируется работа блока `fork-join`.

```
1  fork
2    #5 a = 3;
3    #2 b = 4;
4  join
5 // следующий оператор
```

Здесь порождаются два процесса (строки 2 и 3). В момент времени 2 переменной b будет присвоено значение 4; в момент времени 5 переменной a будет присвоено значение 3. Время отсчитывается от начала исполнения fork. В этом примере join ожидает завершения всех порожденных процессов, поэтому оператор в строке 5 начнет исполняться в момент времени 5. Если бы в строке 4 вместо ключевого слова join было join_any, оператор в строке 5 начал бы исполняться в момент времени 2; если бы вместо join было join_none, оператор начал бы исполняться сразу, в текущий момент времени.

Следующий пример – «сторожевой» таймер. В нем порождаются два процесса: первый – временна́я задержка (строки 3–6); второй – вызов процедуры (строка 7). Таймер позволяет убедиться, что исполнение taskCall занимает не больше 10 000 единиц времени. Если это не так, будет сообщено об ошибке путем установки флага timeOut. Когда завершается один из этих процессов, исполнение продолжается со строки 9, в которой оператор disable fork останавливает другой процесс, все еще работающий. После этого можно проверить флаг timeOut и узнать, произошел ли тайм-аут.

```
1  timeOut = 0;
2  fork : watchDog
3    begin
4      #10_000 timeOut = 1;
5      $display("Вызов taskCall не должен занимать так много времени!");
6    end
7    taskCall;
8  join_any
9  disable fork;
10 // следующий оператор
```

## 8.2. Пример и общая схема тестирования

Для иллюстрации использования параллельных процессов в тестовом окружении рассмотрим простой 4-портовый маршрутизатор. Входные пакеты содержат данные и порт назначения. Получив пакет, маршрутизатор посылает его на указанный выходной порт. Общее представление маршрутизатора и его тестового окружения показано на рис. 8.2. Используемый подход к тестированию состоит в посылке тысяч рандомизированных пакетов на входные порты и проверке правильности их направления на выходные порты.

К каждому порту маршрутизатора подключаются два процесса тестового окружения: один – подает данные в маршрутизатор (TBsend – тестовый передатчик); второй – забирает данные из него (TBrecv – тестовый приемник). Когда маршрутизатор принимает пакет на одном из входных портов, он помещает

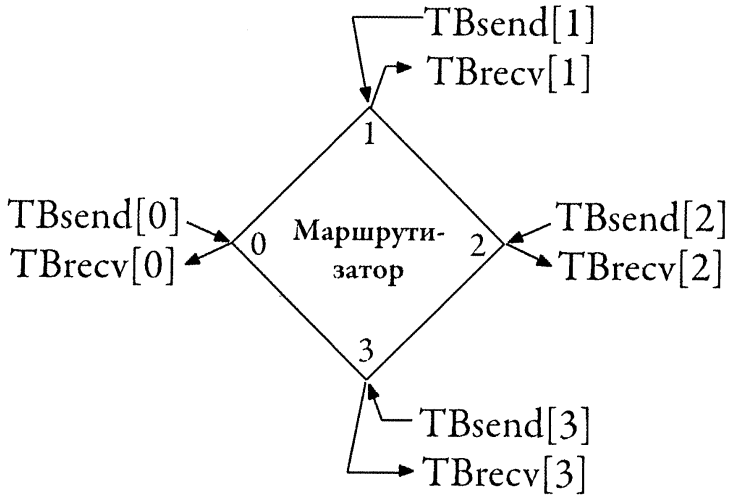



**Рис. 8.2.** Система в целом: маршрутизатор и процессы тестового окружения

содержащиеся в нем данные в указанный порт назначения. Маршрутизатор может делать до четырех передач одновременно: если пакеты на входных портах посылаются на разные выходные порты, все четыре передачи делаются разом (по одному фронту тактового сигнала). Для обеспечения корректной работы в случае, когда на один и тот же выходной порт в одно и то же время посылается несколько пакетов, между передатчиками и портами маршрутизатора определен протокол взаимодействия: посылка пакетов может быть приостановлена до тех пор, пока не появится место для их сохранения.

Тестовый передатчик должен удовлетворять протоколу взаимодействия с маршрутизатором и посылать пакеты, только когда последний готов их принять. Тестовый приемник также должен удовлетворять протоколу взаимодействия.

Для проверки того, что пакеты отправляются в правильные порты назначения, мы добавим в тестовое окружение четыре очереди. Для простоты на рис. 8.3 показана одна очередь, идущая в тестовый приемник порта 1 (очереди есть у всех тестовых приемников). Тестовые передатчики кладут в очереди копии пакетов, посылаемых в соответствующие порты. Когда выдаваемый

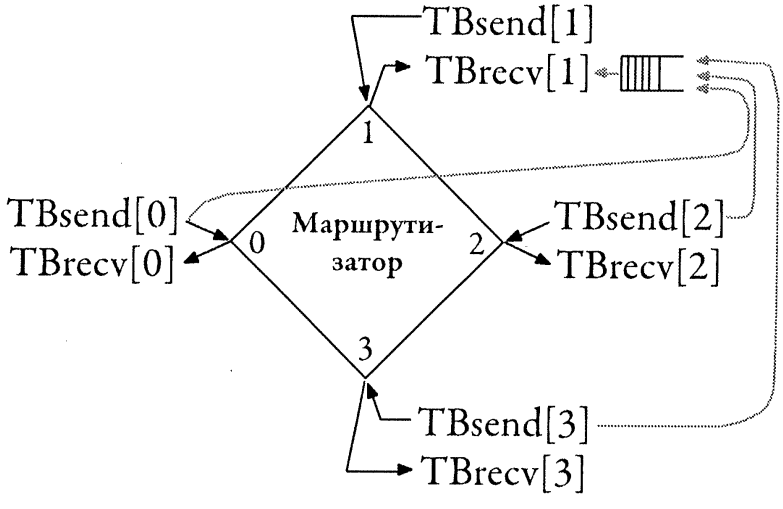



**Рис. 8.3.** Система в целом: одна из очередей тестового окружения

маршрутизатором пакет принимается тестовым приемником, проверяется, есть ли такой пакет в очереди.

В следующих разделах главы мы разработаем процессы тестового окружения. В примере будут использоваться параллельные процессы, почтовые ящики (так в SystemVerilog называются межпроцессные очереди), генераторы случайных чисел и ассоциативные массивы. Детали реализации маршрутизатора не показаны; на самом деле описание параметризовано и обобщается на большее число портов. В примере тем не менее рассматривается четырехпортовая модель и ее тестовое окружение.

## 8.3. Протоколы взаимодействия

На рис. 8.4 показаны соединения между тестовым окружением порта 0 и самим портом; на рис. 8.5 – протокол взаимодействия тестового окружения с маршрутизатором, используемый при посылке пакетов. Соединения и протоколы для других портов идентичны.

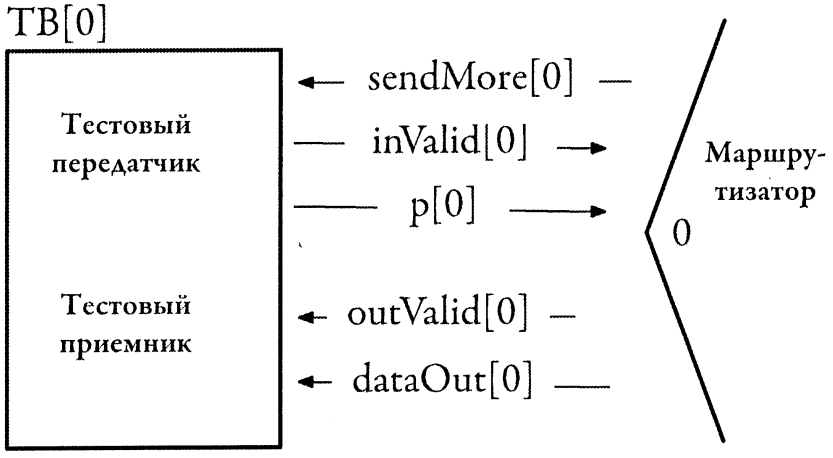



**Рис. 8.4.** Соединения между тестовым окружением и маршрутизатором

Маршрутизатор может устанавливать у входного порта сигнал запроса `sendMore`. (Надпись «router.sendMore[i]» на рисунке говорит об установке сигнала `sendMore` у порта с номером i.) Когда маршрутизатор устанавливает `sendMore`, он ждет от тестового окружения установки сигнала `inValid`, показывающего, что вход значим (не путать с invalid!). Это может занять несколько тактов или произойти немедленно (комбинационно), как показано на рис. 8.5. При установке `inValid` на вход также подается пакет p. Маршрутизатор копирует входной пакет, как только видит, что установлен `inValid`, сбрасывая `sendMore`. На рисунке это происходит на такте B. До следующей установки `sendMore` может пройти несколько тактов – выходной порт маршрутизатора, куда направляется пакет, может быть перегружен. На рисунке показано ближайшее время, когда `sendMore` может быть установлен повторно.

Следует заметить, что активной стороной во входном протоколе выступает маршрутизатор: если он не готов получить очередной входной пакет, он просто

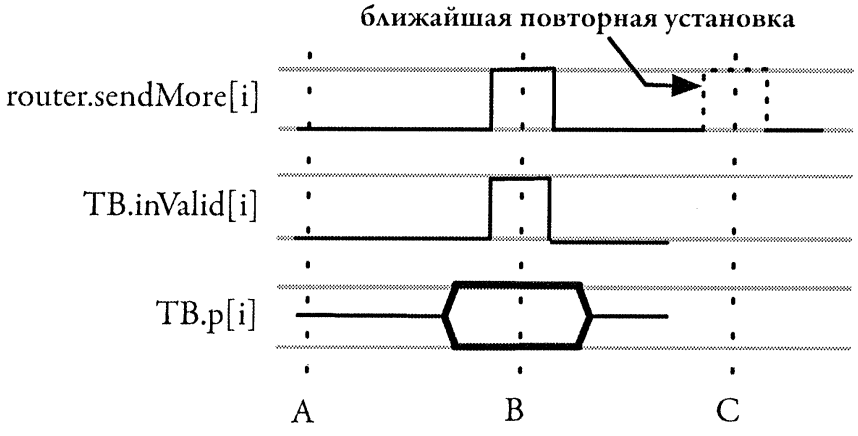



**Рис. 8.5.** Входной протокол (протокол посылки пакетов в маршрутизатор)

не запрашивает его сигналом `sendMore`. Протокол используется для приостановки данных, поступающих в маршрутизатор, пока не появится место для них.

На рис. 8.6 показан протокол взаимодействия выходного порта маршрутизатора с тестовым приемником. Здесь используется простой протокол, предполагающий, что тестовое окружение всегда готово принять данные и обработать их, – одноэтапное согласование (one-way handshake): сигнал `outValid` служит для индикации значимости `dataOut`. Передача длится один такт. Следующий выходной пакет может быть выдан уже на следующем такте, как показано на рисунке.

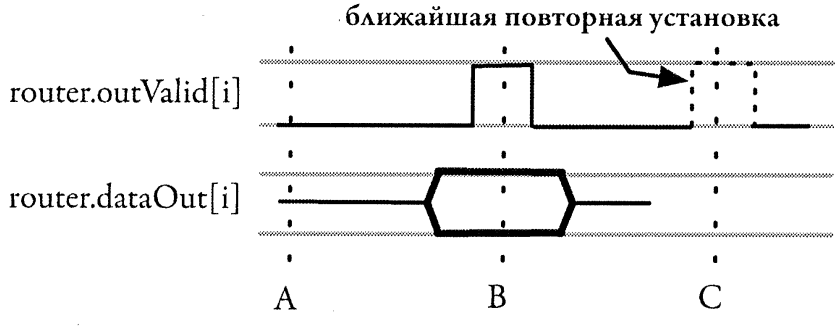



**Рис. 8.6.** Выходной протокол (протокол выдачи пакетов в тестовое окружение)

# 8.4. Организация тестового окружения

В этом разделе мы разберем разработку тестового окружения.

## 8.4.1. Заголовки и создание экземпляров модулей

В примере 8.1 определяется тип данных для представления пакета, передаваемого через маршрутизатор. Это упакованная структура – поля `src`, `dest` и `data` могут рассматриваться как одно 16-битное слово. Для типа вводится имя `pkt_t`, которое можно использовать в качестве типа портов, и не только.

Строки 7–12 определяют порты тестового окружения. Можно видеть, что тестовое окружение представлено в виде одной программы, порты которой

```
1  typedef struct packed {
2    bit [3:0] src;
3    bit [3:0] dest;
4    bit [7:0] data;
5  } pkt_t;
6
7  program automatic tb
8     (output pkt_t [3:0] p,
9      output bit [3:0] inValid,
10     input bit clk, reset,
11     input bit [3:0] sendMore, outValid,
12     input bit [3:0][7:0] dataOut );
13
14   mailbox #(pkt_t) toPort[4];
```

**Пример 8.1.** Тип данных пакета и заголовок тестового окружения

соединяются со всеми портами маршрутизатора. Обратите внимание, что порты inValid, sendMore и outValid 4-битные. Доступ к сигналам того или иного порта осуществляется путем выбора нужного бита; например, sendMore[2] – сигнал порта 2, используемый для запроса данных, inValid[1] – выход тестового окружения, соединенный с портом 1 и показывающий, что пакет p значим.

Порт p также имеет ширину 4, но в этом случае p[3] не 1 бит, а 16 – это сигналы, используемые для передачи пакетов во входной порт 3. Сигналы dataOut, выходящие из маршрутизатора, содержат только информационную часть пакета, полученного маршрутизатором. Порт dataOut имеет два измерения: dataOut[2] – 8-битный выход порта 2 маршрутизатора.

Определения, данные выше, не запрещают передавать и получать пакеты на все и со всех портов одновременно. Ниже представлен заголовок маршрутизатора; направления его портов противоположны направлениям соответствующих портов тестового окружения, не считая clk и reset.

```
module router
   (input pkt_t [3:0] p,
    input bit [3:0] inValid,
    input bit clk, reset,
    output bit [3:0] sendMore, outValid,
    output bit [3:0][7:0] dataOut,
```

Экземпляры маршрутизатора и тестового окружения создаются в модуле верхнего уровня с использованием нотации .*.

## 8.4.2. Тестовый передатчик

В программе tb определены две процедуры. Одна, названная sendFromPort, реализует протокол передачи пакетов от тестового окружения к маршрутизатору (она показана в примере 8.2 и рассматривается в этом разделе). Другая, названная receiveAtPort, реализует протокол передачи пакетов от маршрутизатора к тестовому окружению (она показана в примере 8.3 и рассматривается в разделе 8.4.3).

Назначение процедуры sendFromPort – посылать пакеты со случайными данными в случайно выбранные порты назначения. У процедуры два аргумента: номер входного порта и число пакетов для передачи. Посылка пакетов удовлетворяет входному протоколу; каждый передаваемый пакет дублируется в почтовый ящик порта назначения.

```
15 class randomNums;
16   rand bit [3:0] destNode;
17   rand bit [7:0] data;
18   constraint inRange {destNode inside {0, 1, 2, 3};}
19 endclass
20
21 task automatic sendFromPort
22   (input bit [3:0] port,
23   input int count);
24   pkt_t packet;
25   randomNums r = new;
26
27   for (int i = 1; i <= count; i++) begin
28     assert (r.randomize() with {port != destNode;})
29       else $error("Bad random numbers for port %h", port);
30     wait (sendMore[port]);
31     @(posedge clk);
32     packet.src = port;
33     packet.dest = r.destNode;
34     packet.data = r.data;
35     p[port] <= packet;
36     inValid[port] <= 1;
37     toPort[packet.dest].put(packet);
38     @(posedge clk);
39     inValid[port] <= 0;
40   end
41   $display("\nSend from port %h finishes at %d\n", port, $stime);
42 endtask
```

**Пример 8.2.** Посылка пакетов в маршрутизатор

Обратите внимание, что процедура в строке 21 объявлена как автоматическая. Идея в следующем. Процедура будет активирована (вызвана) четыре раза с использованием конструкции fork-join, при этом в каждый вызов будут переданы разные номера входных портов маршрутизатора. Автоматическое объявление используется для того, чтобы отделить внутренние переменные параллельно работающих вызовов процедуры друг от друга.

Для генерации случайных чисел процедура использует класс randomNums (строки 15–19). В классе есть два рандомизируемых поля destNode и data, а также ограничение, говорящее, что значение destNode принадлежит множеству {0, 1, 2, 3}. В строке 25 объявляется объект r типа randomNums, инициализируемый вызовом new.

В цикле for (строки 27–40) пакеты посылаются в маршрутизатор через порт, номер которого передан через аргумент port. Цикл завершается, когда число отправленных пакетов достигает значения count – другого аргумента процедуры.

Цикл разделен операторами wait и @(posedge clk) на три участка. Этим операторам соответствуют состояния конечного автомата, реализующего протокол передачи пакетов в маршрутизатор. Взгляните на рис. 8.5 и 8.7. При переходе из состояния «строка 31» в состояние «строка 38» пакет заполняется новыми

данными и портом назначения, при этом устанавливается сигнал inValid[port]. Из состояния «строка 38» процедура переходит в состояние ожидания sendMore или, если sendMore уже установлен, в состояние «строка 31»; в обоих случаях inValid[port] сбрасывается.

Рассмотрим цикл внимательнее. В строках 28–29 с помощью метода randomize() класса randomNums генерируются случайные значения полей этого класса. Вызов включает дополнительное ограничение with port != destNode, запрещающее узлу посылать пакеты самому себе (это необязательное требование; оно добавлено для иллюстративных целей). Оператор assert проверяет, что сгенерированные числа удовлетворяют заданным ограничениям (он работает как оператор if – см. раздел 9.1.1, посвященный непосредственным утверждениям). Далее цикл ожидает запроса от маршрутизатора в виде установленного сигнала sendMore[port]. На следующем фронте тактового сигнала заполняется пакет – переменная типа pkt_t. В поле src записывается номер текущего узла, а в поля dest и data – случайно сгенерированные числа. Заполненный пакет подается на выход p[port] с использованием оператора параллельного присваивания – это та лепта, которую процедура вносит в формирование выхода тестового окружения (строка 8 примера 8.1). Параллельно с подачей пакета устанавливается сигнал inValid. На рис. 8.5 эти присваивания исполняются при переходе A (присвоенные значения видны при переходе B). На следующем такте inValid сбрасывается обратно в 0, также с использованием оператора параллельного присваивания; процедура ожидает очередного запроса от маршрутизатора. Когда цикл посылает нужное число пакетов, процедура выводит сообщение и завершается.

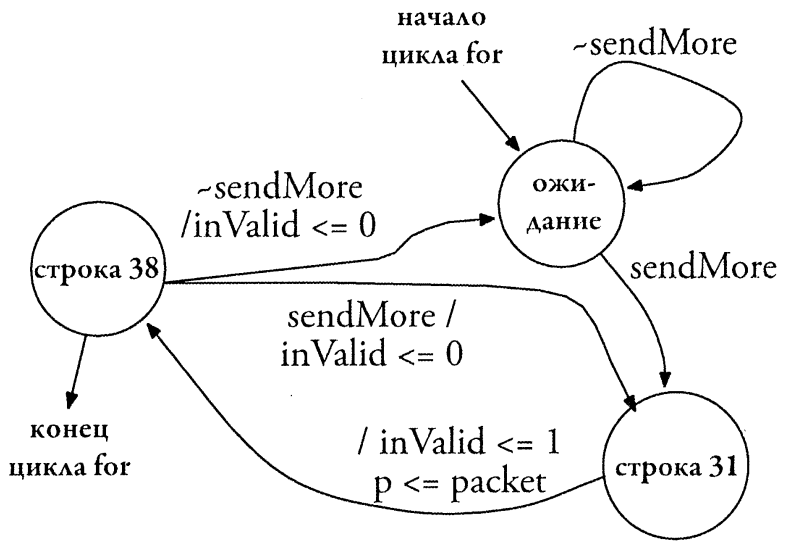



**Рис. 8.7.** Конечный автомат, неявно заданный в процедуре sendFromPort

Мы проскочили через строку 37 в описании процедуры – там используется новая конструкция, требующая разъяснения. Это то место, где сгенерированный пакет кладется в почтовый ящик, ассоциированный с портом назначения.

В строке 14 примера 8.1 создаются четыре почтовых ящика. Почтовый ящик – это межпроцессная очередь. Тестовое окружение состоит из четырех частей,

каждая из которых имеет процедуру посылки пакетов (sendFromPort) и процедуру приема пакетов (receiveAtPort); все эти процедуры, как мы увидим в дальнейшем, работают параллельно, и взаимодействие между ними осуществляется с помощью средств межпроцессного взаимодействия. Тип элементов почтового ящика задается путем переопределения параметра; можно видеть, что в нашем примере в очередь помещаются элементы типа pkt_t. В тестовом окружении объявлен массив toPort из четырех почтовых ящиков (индексируемых от 0 до3). Более детальное описание почтовых ящиков приводится в разделе 8.5.3.

Одним из методов типа mailbox («почтовый ящик») является put(). В строке 37 примера 8.2 тестовое окружение с помощью этого метода кладет пакет в почтовый ящик, связанный с узлом назначения. Такие ящики используются для проверки правильности выдачи пакетов маршрутизатором.

## 8.4.3. Тестовый приемник

Процедура, в которой определяется тестовый приемник (receiveAtPort), показана в примере 8.3. Всего активируются четыре такие процедуры (по одной на каждый порт). Их назначение – получать пакеты от маршрутизатора в соответствии с протоколом и проверять, что все пакеты, выдаваемые маршрутизатором, совпадают с пакетами, помещенными в почтовые ящики. Поскольку процедуры исполняются параллельно, они объявлены автоматическими – каждый экземпляр работает со своими собственными переменными.

У процедуры receiveAtPort один аргумент – номер порта маршрутизатора, который нужно прослушивать. Используется «бесконечный» цикл forever: процедура постоянно проверяет, выдаются ли пакеты из заданного порта маршрутизатора. По сути, это автомат с одним состоянием: ожидается, когда будет установлен сигнал outValid; когда это происходит, данные считываются с порта (без подтверждения готовности их принять); совершается одно из следующих действий:

- если полученные данные совпадают с данными в голове почтового ящика, данные из почтового ящика удаляются, а переменная receiveCount, хранящая число полученных пакетов, инкрементируется;
- если полученные данные не совпадают с данными в голове почтового ящика, они кладутся в ассоциативный массив valuesReceived (это обсуждается ниже);
- когда данные, находящиеся в голове почтового ящика, удаляются (первый пункт), проверяется, содержатся ли очередные данные в ассоциативном массиве valuesReceived. Если да, данные удаляются как из ящика, так и из массива, а переменная receiveCount инкрементируется. Данные из головы почтового ящика продолжают сопоставляться с данными ассоциативного массива. Когда результат становится отрицательным, процедура ожидает следующей установки outValid. Обратите внимание, что при таком подходе допускается изменение порядка пакетов, что возможно из-за «заторов» внутри маршрутизатора.

Для того чтобы объяснить, как функционирует код, нам потребуется знание некоторых методов типа mailbox, а также понимание принципов работы ассоциативных массивов. В примере используются следующие методы:

- `try_peek(ref item)` – см., например, строку 53. Если почтовый ящик не пуст, данные из головы копируются в элемент, переданный по ссылке, но не удаляются из ящика. Возвращается число, большее 0, если при вызове метода ящик не был пуст; 0 иначе;
- `get(ref item)` – см., например, строку 55. Если почтовый ящик не пуст, данные из головы удаляются и копируются в элемент, переданный по ссылке, после чего осуществляется возврат из метода. Если в ящике ничего нет, вызывающий процесс блокируется до тех пор, пока какой-нибудь другой процесс не положит в ящик данные. В данном примере `receiveAtPort` – единственная процедура, опустошающая почтовый ящик, – проверяет, что ящик не пуст (используя try_peek), что гарантирует отсутствие блокировки.

Дальнейшее обсуждение почтовых ящиков можно найти в разделе 8.5.3.

Ассоциативные массивы – это неупакованные массивы, индексируемые значениями некоторого типа, например строками или значениями пользовательского типа (см. раздел 13.5). В строке 47 примера 8.3 объявляется ассоциативный массив целых чисел, индексируемый 8-битными значениями. Ассоциативная память обычно используется для хранения разреженных данных, т. е. когда присутствует лишь небольшая часть элементов. В этом примере она используется для хранения данных, полученных на выходе маршрутизатора, которые в момент получения не совпадали с данными в голове почтового ящика. Использование ассоциативной памяти избавляет от необходимости писать код, осуществляющий поиск пакетов, полученных ранее. В примере используются следующие методы ассоциативного массива:

- `exists(item)` – возвращает TRUE, если переданный элемент содержится в ассоциативном массиве;
- `delete(item)` – удаляет переданный элемент.

Когда осуществляется доступ к элементу ассоциативного массива, как, например, в строке 68, если элемент по указанному индексу отсутствует, он создается (в нашем случае в роли индекса выступают данные `dataOut`). В примере инкрементируется хранящийся в массиве счетчик числа полученных пакетов (пакетов с одинаковыми данными может быть больше одного).

Посмотрим на код внимательнее. В строке 51 ожидается установка `outValid`. Если сигнал установлен (в этом случае выход `dataOut[port]` содержит выдаваемые данные), исполнение переходит на строку 53, где проверяется, есть ли что-то в почтовом ящике. Так как помещение данных в почтовый ящик не занимает времени, в нем должен находиться, по крайней мере, один пакет. В строке 54 проверяется, совпадают ли данные в голове почтового ящика с данными, полученными от маршрутизатора. Если нет, полученные данные кладутся в ассоциативный массив (строка 68). Если да, пакет удаляется из ящика (строка 55), а переменная `receiveCount` инкрементируется (строка 56).

```
43 int receiveCount[4];
44
45 task automatic receiveAtPort
46   (input bit[3:0] port);
47   int valuesReceived[bit[7:0]]; // Ассоциативный массив
48   pkt_t p;
49
50   forever begin
51     wait (outValid[port]);
52
53     if (toPort[port].try_peek(p) > 0)
54       if (p.data == dataOut[port]) begin
55         toPort[port].get(p);
56         receiveCount[port]++;
57         while (toPort[port].try_peek(p) > 0) begin
58           if (valuesReceived.exists(p.data) ) begin
59             valuesReceived[p.data]--;
60             if (valuesReceived[p.data] == 0)
61               valuesReceived.delete(p.data);
62             toPort[port].get(p);
63             receiveCount[port]++;
64           end // конец строки 58
65           else break;
66         end // конец строки 57
67       end // конец строки 54
68       else valuesReceived[dataOut[port]]++;
69     @(posedge clk);
70   end
71 endtask
```

**Пример 8.3.** Прием данных от маршрутизатора

Далее в цикле, пока почтовый ящик не пуст, процедура пытается удалить другие элементы (строки 57–66). Если данные в голове ящика присутствуют в ассоциативном массиве valuesReceived, они удаляются из них обоих. Удаление из ассоциативного массива выражается декрементом хранящегося в нем счетчика: если его значение становится равным 0, элемент удаляется с помощью метода delete (строка 61). Когда для данных в голове ящика не находится соответствия в ассоциативном массиве, осуществляется выход из цикла (строка 65).

## 8.4.4. Основная часть тестового окружения

Теперь, когда охвачены все процедуры тестового окружения, вспомним общую картинку. Четыре процедуры sendFromPort запускаются в виде отдельных процессов для посылки пакетов в маршрутизатор, а четыре процедуры receiveAtPort – для приема пакетов. Все это вместе иллюстрирует, как организовано параллельное тестовое окружение сложной системы.

```
72 initial begin
73   bit QsAreEmpty;
```

```
74   pkt_t p;
75   inValid = 0;
76
77   for (int i= 0; i <= 3; i++)
78     toPort[i] = new;
79
80   @(posedge clk);
81   @(posedge clk);
82   fork
83     receiveAtPort(0);
84     receiveAtPort(1);
85     receiveAtPort(2);
86     receiveAtPort(3);
87   join_none
88
89   fork
90     sendFromPort(0, 2000);
91     sendFromPort(1, 2000);
92     sendFromPort(2, 2000);
93     sendFromPort(3, 2000);
94   join;
95   @(posedge clk);
96
97   do begin // ожидание опустошения почтовых ящиков
98     QsAreEmpty = 1;
99     for (int i = 0; i <= 3; i++) begin
100      QsAreEmpty &= (toPort[i].num == 0);
101      if (toPort[i].num != 0) begin
102        toPort[i].try_peek(p);
103        $display("toPort %0h has %0d elements left = %h",
104            i, toPort[i].num, p.data);
105      end
106    end
107    @(posedge clk);
108   end
109  while (~QsAreEmpty);
110  $finish;
111 end
```

**Пример 8.4.** Основная часть тестового окружения

Основная часть тестового окружения, инициализирующая все и запускающая процессы sendFromPort и receiveAtPort, показана в примере 8.4. В цикле (строки 77–78) путем вызова new создаются почтовые ящики.

Далее с помощью конструкции fork-join_none запускаются процедуры receiveAtPort (строки 82–87). Конструкция порождает процессы для всех своих операторов; join_none говорит о том, что родительский процесс продолжает исполняться, при этом порожденные процессы не начнут работать, пока родительский процесс не исполнит блокирующий оператор (это join в строке 94). Запущенные процедуры никогда не завершаются, поскольку в них используется цикл forever.

Процедуры sendFromPort запускаются с помощью конструкции fork-join (строки 89–94). Конструкция запускает все свои операторы параллельно в виде процессов и блокируется на join в строке 94. В этой точке все восемь процессов начинают работать. Четыре процедуры sendFromPort работают до тех пор, пока не пошлют нужное число пакетов.

Тестовое окружение в цикле do-while ждет, когда все почтовые ящики станут пустыми (строки 97–109). Метод num типа mailbox возвращает число элементов в почтовом ящике (строка 100). При выходе из цикла все почтовые ящики пусты[48]. Вызывается системная процедура $finish, которая, помимо прочего, завершает процедуры receiveAtPort, работающие в бесконечном цикле.

Когда вызывается $finish, запускается блок final (см. пример 8.5). В нем перебираются выходные порты: проверяется, что в почтовых ящиках нет пакетов и суммируются счетчики полученных пакетов. Утверждение в конце блока проверяет, что числа полученных и посланных пакетов совпадают (см. обсуждение непосредственных утверждений в разделе 9.1.1).

```
112 final begin: finalblock
113   int total = 0;
114   pkt_t p;
115   bit [7:0] q;
116   for (int i = 0; i <= 3; i++) begin
117     QLen: assert (toPort[i].num == 0)
118       else $error("%d items still in mailbox %0h at end\n", toPort[i].num, i);
119     total += receiveCount[i];
120     $display ("valuesReceived at port %0h = %0d", i, receiveCount[i]);
121   end
122   Tot: assert (total == 8000) else $error("Total packets received is incorrect!");
123   $display ("total = %0d", total);
124 end
```

**Пример 8.5.** Блок final тестового окружения

# 8.5. Конструкции параллельного программирования

Процессы SystemVerilog взаимодействуют друг с другом, используя конструкции параллельного программирования, подобные тем, что применяются в программных системах.

## 8.5.1. Оператор wait

Оператор wait(expr) блокирует исполнение потока по заданному условию. Если выражение expr истинно, исполнение продолжается без блокировки; если же оно ложно, исполнение блокируется, пока выражение не станет истинным. Если процесс заблокирован, только другой процесс может изменить значение выражения. Оператор wait не задействуется в логическом синтезе, но активно

[48] Тестовое окружение «зависнет», если маршрутизатор из-за ошибки потеряет пакет.

применяется в тестовых окружениях для синхронизации процессов друг с другом (как это делалось в примерах предшествующего раздела).

Например, следующий оператор блокирует исполнение процесса, пока значение переменной outputDataReady отлично от TRUE. Когда значение становится TRUE, исполнение возобновляется с загрузки outputData в inputBuffer. Оператор после wait необязателен.

```
wait (outputDataReady) inputBuffer = outputData;
```

Выражение может быть более сложным:

```
wait (a + b == c) ;
```

В этом примере исполнение блокируется, пока сумма значений переменных a и b не станет равной значению переменной c. Чтобы процесс возобновил исполнение, по крайней мере, одна из этих переменных должна быть изменена в другом процессе.

Еще один вариант оператора wait – wait fork. Между словами есть пробел (в конце ставится точка с запятой). Оператор wait fork блокирует процесс до тех пор, пока не завершатся все подпроцессы, непосредственно созданные им (например, порожденные оператором fork). Если подпроцессы породили другие процессы, эти порожденные процессы не ожидаются.

Конструкция wait_order ожидает, что именованные события произойдут в определенном порядке (в порядке их записи). Если событие произошло (в свою очередь), оно может возникнуть снова, однако оставшиеся события должны быть правильно упорядочены. Первое событие может использовать свойство triggered (это обсуждается в разделе 8.5.5).

Следующий пример иллюстрирует использование wait_order.

```
1 event a, b, c;
2 ...
3 wait_order (b, c, a) $display("It worked!!");
4   else $display("Oops, something was out of order");
```

- Если последовательность событий имеет вид bca, wait_order успешно срабатывает (на консоль выводится "It worked!"). Другой допустимой последовательностью является bbccca: если событие наступило в свой черед, wait_order принимает в расчет только оставшиеся события. Последовательность abc, очевидно, не допускается.

## 8.5.2. Оператор disable

Оператор disable принудительно завершает работу процедуры или именованного блока. Исполнение процесса возобновляется с оператора, следующего за оператором вызова процедуры, или с оператора, стоящего после именного блока begin-end (или fork-join). Если процедура породила процессы, при принудительном завершении процедуры все они завершаются.

Именованный блок выглядит следующим образом:

```
1 begin: myBlock
2   while ( ) ...
3 end
4 ...
```

Если в блоке myBlock исполняется цикл while, другой процесс может завершить его с помощью оператора disable myBlock. Исполнение продолжится со строки 4.

Оператор disable можно использовать внутри цикла for для выхода из него или перехода к следующей итерации, однако лучше для этого использовать операторы break и continue (см. раздел 11.4.6) – так понятнее, что происходит в коде.

В примере 8.4 в блоке fork-join_none процедура receiveAtPort вызывается четыре раза (строки 83–86). Оператор disable receiveAtPort завершает все вызовы.

Еще одна форма disable - disable fork. Этот оператор завершает все процессы, порожденные текущим процессом (например, созданные с помощью fork-join_none или fork-join_any), включая процессы, порожденные подпроцессами (рекурсивно).

## 8.5.3. Почтовые ящики

Почтовый ящик (mailbox) – это встроенный класс, используемый для обмена сообщениями между процессами. Порядок приема сообщений совпадает с порядком их поступления в ящик.

Название «почтовый ящик» вызвано аналогией с почтой. Письма от разных людей, адресованные одному лицу, кладутся в его ящик. Если при проверке ящика оказалось, что почты нет, можно подождать у ящика, а можно вернуться к нему позднее; можно узнать, есть ли в ящике корреспонденция, не пытаясь вынуть ее.

Пример использования почтовых ящиков был показан в предыдущем разделе. Объявляются они следующим образом:

```
mailbox mbox, mymbox, m[5];
```

Здесь задаются почтовые ящики mbox и mymbox, а также массив m из 5 почтовых ящиков. Это нетипизированные почтовые ящики; если почтовый ящик используется для передачи сообщений определенного типа, этот тип можно указать. В следующем примере (немного измененном примере из раздела 8.4.1) определяется тип данных и объявляется использующий его почтовый ящик.

```
1 typedef struct packed {
2   bit [3:0] src;
3   bit [3:0] dest;
4   bit [7:0] data;
5 } pkt_t;
6 pkt_t p;
7 mailbox #(pkt_t) postal;
```

В этом примере элементы, помещаемые в почтовый ящик и извлекаемые из него, имеют тип pkt_t. Такая типизация позволяет компилятору делать статическую проверку типов при доступе к ящику.

У типа mailbox есть следующие методы:

- new() – создает почтовый ящик. Прототип следующий:

```
function new (int bound = 0);
```

  Метод возвращает дескриптор (handle) созданного почтового ящика. При нулевом значении параметра bound (значении по умолчанию) нет ограничений на число элементов в ящике (в этом случае вызов метода put() не может быть заблокирован). Если значение bound положительно, оно задает размер ящика. Отрицательным значение быть не может. Пример приведен ниже:

```
mbox = new;
```

- num() – возвращает число элементов в почтовом ящике. Прототип следующий:

```
function int num();
```

- put() – кладет сообщение в почтовый ящик (сообщения кладутся и извлекаются по принципу «первым пришел – первым ушел»). Если ящик ограниченного размера полон, процесс блокируется, пока в ящике не появится место; в противном случае процесс вставляет сообщение и возвращает управление. Прототип следующий:

```
task put(singular expression);
```

  Здесь singular expression – выражение, значение которого не является неупакованной структурой, объединением или массивом.

- try_put() – пытается положить сообщение в почтовый ящик (используется для ящиков ограниченного размера). Если ящик полон, возвращается 0; в противном случае сообщение кладется в ящик, и возвращается положительное число. Прототип следующий:

```
function try_put(singular expression);
```

- get() – извлекает сообщение из почтового ящика, удаляя его из внутренней очереди. Если ящик пуст, процесс блокируется, пока другой процесс не положит сообщение в ящик. Прототип следующий:

```
task get(ref singular expression);
```

  Сообщение, удаляемое из почтового ящика, записывается по переданной ссылке. Для ящика postal из примера выше вызов может выглядеть следующим образом:

```
postal.get(p);
```

  В этом примере сообщение, удаляемое из почтового ящика postal и имеющее тип pkt_t, копируется в переменную p.

- try_get() – метод пытается извлечь сообщение из почтового ящика. Если сообщений нет, возвращается 0; в противном случае сообщение извлекается из ящика, копируется по переданной ссылке и удаляется из внутренней очереди, возвращается положительное число. Метод возвращает от-

рицательное число, если тип сообщения, извлекаемого из ящика, отличен от типа ссылки. Прототип следующий:

```
function int try_get(ref singular expression);
```

Для ящика postal из примера выше вызов может выглядеть следующим образом:

```
if (postal.try_get(p) > 0)
 // сообщение успешно извлечено
 // иначе сообщения нет или оно другого типа
```

Если сообщение извлекается, оно копируется в переменную p.

- peek() – копирует сообщение из почтового ящика, но не удаляет его. Так можно посмотреть содержимое головного элемента очереди перед его извлечением с помощью get или try_get. Если ящик пуст, процесс блокируется, пока другой процесс не положит сообщение в ящик. Прототип следующий:

  ```
  task peek(ref singular expression);
  ```

- try_peek() – пытается скопировать сообщение из почтового ящика, не блокируя процесс. Еще раз: сообщение из почтового ящика не удаляется. Прототип следующий:

  ```
  function int try_peek(ref singular message);
  ```

Если почтовый ящик пуст, возвращается 0; в противном случае сообщение копируется по переданной ссылке, возвращается положительное число. Отрицательное число возвращается, если тип сообщения, извлекаемого из ящика, отличен от типа ссылки.

## 8.5.4. Семафоры

Семафоры используются для синхронизации доступа к общим ресурсам: общим данным или функциям и процедурам, использующим общие данные. Участки кода, где производится доступ к ресурсам, часто называют *критическими секциями* – одновременно в них может находиться не более одного процесса (или не более чем заданное число процессов).

Семафор можно представить в виде ведра, в котором лежит маркер (token). Для доступа к критической секции вам нужно взять маркер. Если маркер кто-то уже взял, следует подождать (процесс блокируется). Если маркер есть, вы берете его и входите в критическую секцию; когда заканчиваете, возвращаете маркер назад, предоставляя возможность одному из ожидающих процессов войти в критическую секцию.

Можно обобщить эту схему – задать начальное число маркеров в ведре и число ключей, необходимых для доступа. Когда есть нужное число маркеров, вы получаете доступ – это называется *захватом семафора*.

Семафор объявляется следующим образом:

```
semaphore myVars;
```

Семафор (semaphore) – это встроенный класс, имеющий следующие методы:

- new() – возвращает дескриптор семафора, который может быть присвоен переменной типа semaphore. Прототип следующий:

  ```
  function new(int tokenCount = 0);
  ```

  Значение, передаваемое new, – начальное число маркеров. В процессе работы число маркеров может превзойти это число;
- get() – берет заданное число маркеров из семафора. Если они есть в наличии, метод возвращает управление, и исполнение продолжается; в противном случае процесс блокируется, пока не появится достаточного числа маркеров. Прототип следующий:

  ```
  task get(int tokenCount = 1);
  ```

  Аргумент метода задает число маркеров, требуемое для захвата семафора (значение по умолчанию – 1). Процессы, заблокированные на семафоре, ставятся в очередь и обслуживаются по дисциплине «первым пришел – первым ушел»;
- try_get() – пытается взять заданное число маркеров из семафора, не блокируя процесс. Если они есть в наличии, метод возвращает положительное число; в противном случае – 0. Прототип следующий:

  ```
  function int try_get(int tokenCount = 1);
  ```

- put() – возвращает маркеры в семафор. Если при вызове есть процесс, заблокированный на семафоре, он начнет исполняться, если возвращается достаточное число маркеров. Прототип следующий:

  ```
  task put(int tokenCount = 1);
  ```

Семафоры можно использовать для управления доступом к почтовому ящику. Рассмотрим ситуацию, когда ящик может читаться несколькими процессами. Не зная порядка исполнения процессов, ни один процесс не может гарантировать, что знает состояние ящика (например, сколько в нем сообщений) в той или иной точке кода (другой процесс может вызвать put или get). Чтобы исправить эту ситуацию, при доступе к почтовому ящику используется семафор; каждый процесс, если хочет получить доступ к ящику, сначала должен захватить семафор.

В следующем примере процесс захватывает семафор, чтобы извлечь из почтового ящика два сообщения, идущих подряд. Если в почтовом ящике нет двух сообщений, он не извлекает ни одного.

```
1  semaphore      guard_myMessages = new(1);
2  mailbox #(int) myMessages = new;
3  int            count, m1, m2;
4
5  initial begin
6    // какие-то действия
7    guard_myMessages.get;   // при возврате из метода семафор захвачен
8    count = myMessages.num;
9    if (count >= 2) begin
10     myMessages.get(m1);
11     myMessages.get(m2);
```

```
12   end
13   guard_myMessages.put;    // семафор освобождается
14 end
```

Критическая секция в этом примере находится в строках 8–12. Так как процесс использует почтовый ящик совместно с другими процессами, нужно сделать так, чтобы другие процессы не использовали ящик одновременно с ним. Пусть, например, процесс исполнил строку 8, но еще не исполнил строку 9. Если в это время другой процесс удалит из ящика сообщение, то рассматриваемый процесс, когда доберется до строки 9, получит ошибочное значение счетчика (может так получиться, что в ящике осталось одно сообщение).

Если все процессы перед использованием почтового ящика сначала обращаются к семафору, то доступ к ящику получает процесс, захвативший семафор (другие будут заблокированы). В этом случае говорят, что система *безопасна*.

В качестве еще одного примера рассмотрим следующий фрагмент кода.

```
1  module trySem;
2    semaphore ab = new;
3
4    initial begin: p1
5      ab.get;
6      // код после вызова get
7    end
8
9    initial begin: p2
10     // код перед вызовом put
11     ab.put;
12   end
13 endmodule
```

В строке 2 объявлен семафор ab; там же для его инициализации вызывается new. Поскольку аргумент в вызове new не указан, начальное число маркеров полагается равным 0. Блоки initial могут быть запущены в произвольном порядке. Блок p1 пытается захватить семафор путем вызова get, а блок p2 вызывает put. Для этого примера можно утверждать, что код в строке 10 (расположенный перед put) будет исполнен перед кодом в строке 6 (расположенным после get). То есть семафор здесь применяется, чтобы обеспечить определенный порядок исполнения участков кода двух процессов.

## 8.5.5. Именованные события

Событие (event) – это «тип данных», «переменные» которого используются для синхронизации процессов. Объявление имеет следующую форму:

```
1  event e1, startNow;
```

Переменные e1 и startNow называются именованными событиями; они не имеют значения и работают как спусковые механизмы (triggers), которые разрешают процессам, ожидающим события, продолжить исполнение.

Событие вызывается следующим оператором:

```
2  -> startNow;
```

После исполнения этого оператора все процессы, ожидающие события `startNow`, могут продолжить исполнение.

Ожидать события можно одним из следующих способов. Во-первых, процесс может содержать следующий фрагмент кода:

```
3  // какие-нибудь процедурные операторы
4  @startNow;
5  // какие-нибудь процедурные операторы
```

Символ @ обозначает оператор контроля события. Встречая ее, процесс приостанавливается, пока не возникнет событие, указанное справа. В этом примере процесс исполняет операторы, представленные комментарием в строке 3. В строке 4 процесс блокируется и переходит в состояние ожидания события `startNow`. В другом процессе оператор вроде указанного в строке 2 разблокирует его (процесс может продолжить исполнение с кода в строке 5). Нужно отметить следующие моменты, касающиеся использования именованных событий.

- Когда возникает событие, разрешение на исполнение получает только тот процесс, который уже ожидает события. Процесс, который достигает оператора контроля события (@) после, должен ждать следующего срабатывания.
- Одно и то же именованное событие может ожидаться несколькими процессами. Когда событие возникает, все они получают разрешение на исполнение.
- Переменная-событие не имеет значения.
- Используется тот же самый механизм симулятора, что и в операторах типа `@(a)` и `@(posedge clk)`. Ожидаемые события в этих примерах – изменение значения переменной a и передний фронт сигнала clk.

Второй способ ожидания события – отслеживание установки свойства `triggered`. Такой подход называется устойчивым (persistent). Рассмотрим следующий пример кода.

```
6  // какие-нибудь процедурные операторы
7  wait(e1.triggered);
8  // какие-нибудь процедурные операторы
```

Если процесс заблокирован и находится в строке 7, то при возникновении события он получит разрешение на исполнение точно так же, как при использовании оператора @. Если же сначала произошло событие, а уже потом, но в том же слоте времени, процесс исполнил оператор в строке 7, то процесс не будет заблокирован, а продолжит исполняться со строки 8. *Слот времени* объединяет все циклы симуляции, осуществляемые в один и тот же момент времени. Когда симулятор увеличивает виртуальное время, свойство `triggered` сбрасывается.

Несколько процессов может ожидать одного и того же события, используя разные подходы.

# Глава 9

# Утверждения и последовательности

Тестирование, которое демонстрировалось до сих пор, было *направленным* (*directed*). Так называется подход, при котором тесты преследуют одну цель – проверить модель на наличие ошибок определенного класса. В случае комбинационной схемы направленное тестирование сводится к генерации всех $2^n$ комбинаций значений $n$ входов и проверке корректности выходных значений[49]. В случае конечного автомата тестовое окружение генерирует последовательность входных значений, приводящую к срабатыванию всех переходов автомата. Понятно, что каждый тест, помимо намеченной ошибки (ситуации), может покрыть и другие, но все же суть состоит в целенаправленном создании тестов, проверяющих ошибки заданного класса.

Утверждения (assertions) – это средства проверки некоторого свойства модели. Примером такого свойства может служить следующее: регистр состояния конечного автомата принимает только значения, определенные в перечислении. Если во время прогона тестов, предназначенных для выявления другого типа ошибок, это свойство окажется нарушенным, утверждение просигнализирует об ошибке. Другой пример – работа системной шины. У шины определены протоколы чтения и записи. Чтобы проверить, нарушаются ли эти протоколы, пишутся утверждения. При тестировании моделируется исполнение процессором некоторой программы: для выборки команд и работы с данными производится доступ к памяти, при этом утверждения проверяют отсутствие нарушений протоколов шины.

Отметим, что тестовые векторы и тестовые программы в этих примерах не генерируются с целью нахождения конкретных ошибок. Их можно создавать случайным образом. Система работает в обычном режиме, а утверждения следят за нарушениями.

---

[49] Направленное тестирование не обязано быть исчерпывающим (использовать все $2^n$ входных паттернов). Состав и размер тестового набора определяются *моделью ошибок* (*fault model*); например, для покрытия константных неисправностей (stuck-at faults) достаточно $N$ тестов, где $N$ – число соединений в схеме (как правило, $N$ значительно меньше $2^n$).

# 9.1. Предварительные сведения

Есть два основных типа утверждений: непосредственные (immediate) и параллельные (concurrent).

- *Непосредственное утверждение* – это процедурный оператор и потому может встречаться всюду, где допустимы процедурные операторы. Обычно непосредственное утверждение является частью тестового окружения и находится в блоке `initial` внутри program. Его функция – проверять заданное условие и сообщать о возникновении ошибки. Если ошибки не обнаружено, исполнение продолжается со следующего оператора.
- *Параллельное утверждение* – это отдельный (параллельный) процесс, цель которого – проверять заданное свойство. Будучи один раз запущен, он так и продолжает работать.

В этом разделе мы обсудим непосредственные утверждения: что именно они могут проверять и каково их место в симуляторе. Параллельные утверждения – более обширная тема, которой мы займемся в последующих разделах этой главы.

## 9.1.1. Пример непосредственного утверждения

В примере 9.1 показан 4-битный сумматор, составленный из четырех полных 1-битных сумматоров; для проверки его поведения используется непосредственное утверждение. Определение модуля полного сумматора занимает строки 26–32; далее создаются 4 экземпляра этого модуля, которые образуют структурную модель 4-битного сумматора (строки 6–9). В строках 11–23 представлен блок `initial`, где устанавливаются значения входов и проверяются значения выходов. В коде задаются значения `a=3`, `b=2`, `cIN=0`. Блок `initial` приостанавливается на одну единицу времени, давая возможность распространиться значениям, а затем вызывает `$display` для вывода результата на консоль. В строке 34 видно, что результат правильный: 3 + 2 равно 5.

В строке 16 проверяется непосредственное утверждение, описывающее правильность результата. Оно имеет следующий вид:

```
1   label: assert (expression)
2          pass_Statement else fail_Statement;
```

Метка `label` и оператор `pass_Statement` необязательны. Утверждение ведет себя как оператор `if`. Если значение выражения `expression` равно TRUE, то исполняется `pass_Statement`, иначе – `fail_Statement`. Поскольку инженеры-верификаторы просто хотят знать, была ли ошибка, `pass_Statement` часто опускают.

В строке 35 показан вывод оператора, указанного в качестве `pass_Statement`.

```
1 module adder;
2   logic [3:0] a, b, s;
3   logic [3:0] co;
4   logic cIN;
5
```

```
6    full_add b0 (s[0], co[0], a[0], b[0], cIN);
7    full_add b1 (s[1], co[1], a[1], b[1], co[0]);
8    full_add b2 (s[2], co[2], a[2], b[2], co[1]);
9    full_add b3 (s[3], co[3], a[3], b[3], co[2]);
10
11   initial begin: testbench
12     a = 3; b = 2; cIN = 0;
13     #1 $display
14       ("time=%d, a=%b, b=%b, cIN=%b, s=%b, co=%b",
15        $stime, a, b, cIN, s, co);
16     checkadd0: assert (s == a + b + cIN)
17       $display ("%m works! a=%b, b=%b, cIN=%b, s=%b,\
18                  co=%b", a, b, cIN, s, co);
19       else
20         $error ("%m says no cigar! a=%b, b=%b, cIN=%b,\
21                  s=%b, co=%b", a, b, cIN, s, co);
22 ...  // остальной код опущен
23    end
24 endmodule: adder
25
26 module full_add
27   (output sum, co,
28    input a, b, cin);
29
30   xor (sum, a, b, cin);
31   assign co = a&b | a&cin | b&cin;
32 endmodule: full_add
33
34 time=1, a=0011, b=0010, cIN=0, s=0101, co=0010
35 adder.testbench.checkadd0 works! a=0011, b=0010, cIN=0, s=0101, co=0010
```

**Пример 9.1.** Непосредственное утверждение и выведенное сообщение

Отметим ряд важных моментов.

- Непосредственное утверждение работает так же, как оператор `if`, при этом, однако, ясно, что производится проверка функциональности. Нельзя по ошибке принять утверждение за часть модели или часть тестового окружения (не связанного с проверкой поведения).
- Достаточно сложная функциональность нескольких соединенных между собой модулей (строки 6–9) проверяется простым сравнением ожидаемого результата с фактическим (строка 16). Это типично для верификации модели. Раз экземпляры модулей, соединенные друг с другом, должны вычислять сумму `a + b + cIN`, это и надо проверить; нет смысла сопоставлять одну сложную модель с другой, не менее сложной, – ведь ошибка может быть в каждой из них! А то и в обеих.

Изменим один символ в определении модуля сумматора: вместо строки 9 напишем такую:

```
full_add b3 (s[3], co[3], a[3], b[3], co[1]);
```

Обратите внимание, что входной бит переноса задан неверно: берется выходной бит переноса каскада 1 (co[1]), а не каскада 2 (что было бы правильно). Если теперь запустить симуляцию, на консоль будет выведено:

```
1 time=1, a=0011, b=0010, cIN=0, s=1101, co=0010
2 "ImmAssertBasic.sv", 14: adder.testbench.checkadd0: started at 1s failed at 1s
3     Offending "(s == ((a + b) + cIN))"
4 Error: "ImmAssertBasic.sv", 14: adder.testbench.checkadd0: at time 1
5 adder.testbench.checkadd0 says no cigar! a=0011, b=0010, cIN=0, s=1101, co=0010
```

Строка 1 выведена процедурой $display. Заметьте: s=4'b1101; очевидно, это не результат сложения 2 + 3! В строке 5 показан результат исполнения процедуры $error, находящейся в ветви else непосредственного утверждения. Эта процедура, как и $display, выводит переданное сообщение, но, в отличие от $display, она вызывает ошибку во время исполнения. Строки 2–4 выведены симулятором.

Отметим, что для форматирования вывода в строках 17 и 20 примера 9.1 использована управляющая последовательность %m. Она выводит иерархический контекст оператора, чтобы было проще определить причину ошибки. В данном случае иерархический контекст – метка checkadd0 внутри блока testbench, расположенного в модуле adder; получается adder.testbench.checkadd0.

Помимо $error, есть и другие процедуры вывода сообщений, соответствующие разным уровням критичности. Они допустимы только в утверждениях.

- $fatal – фатальная ошибка. После вывода сообщения на консоль вызывается процедура $finish, останавливающая симуляцию. Аргументы: уровень диагностической информации для передачи в $finish (по умолчанию – 1), форматная строка и выводимые значения.
- $error – ошибка (аргументы: формат и значения).
- $warning – предупреждение (аргументы: формат и значения).
- $info – информация (аргументы: формат и значения).

Форматная строка во всех этих процедурах устроена так же, как в $display.

Непосредственные утверждения можно использовать и иначе. Например, если тестовое окружение открывает файл для записи в него результатов, можно написать такой код:

```
1   open: assert ((filed = $fopen("fname.dat", "w")) != 0)
2       else $error ("oops! %m can't open file %s", "fname.dat");
```

Так проще организовать проверку ошибок!

## 9.2. Введение в параллельные утверждения

Параллельное утверждение – это независимо исполняемый процесс, проверяющий, что некоторое свойство модели выполняется в течение всего времени

симуляции. Сначала определяется свойство, затем активируется его проверка. Утверждение, следуя принципам моделирования на уровне регистровых передач, синхронизируется тактовым сигналом. Проверка осуществляется симулятором на этапе наблюдения с использованием значений, сэмплированных (считанных) на этапе предварительной обработки[50].

## 9.2.1. Определение и проверка свойства

*Свойство* описывает допустимые последовательности сигналов и может быть весьма сложным. Описания последовательностей похожи на регулярные выражения, в которых используются фронты тактовых сигналов.

Параллельное утверждение имеет вид:

```
1   label: assert property (pname) pass_Statement else fail_Statement;
```

Здесь assert, property и else – ключевые слова языка, label – метка оператора, pname – имя свойства, pass_Statement – оператор, исполняемый, когда свойство выполняется, fail_Statement – оператор, исполняемый, когда свойство нарушается; label и pass_Statement необязательны.

В строках 4–6 примера 9.2 определено очень простое свойство, названное q1r3s (строка 4). Его «действие» состоит в следующем (строка 5): по переднему фронту тактового сигнала ck проверяется, что значение переменной q равно TRUE; спустя один такт (что задается конструкцией ##1) – что значение r равно TRUE; спустя еще три такта (##3) – что значение s равно TRUE.

```
1   module simpleAssert;
2     bit q, r, s, ck;
3     ...
4     property q1r3s;
5       @(posedge ck) q ##1 r ##3 s;
6     endproperty
7
8     assert property (q1r3s) else $error("oops");
9 ...
```

**Пример 9.2.** Параллельное утверждение

В строке 8 находится параллельное утверждение, написанное в соответствии с приведенным выше синтаксисом. В отличие от непосредственного утверждения, указывается ключевое слово property и имя свойства. Здесь присутствует только оператор fail_Statement, выводящий сообщение "oops".

Как определение свойства, так и утверждение находятся внутри модуля, но вне процедурных конструкций, например блоков always и initial.

---

[50] Подробнее работа симулятора рассматривается в главе 14.

Свойство определяет ограничение на последовательность значений переменных q, r и s. Рассмотрим диаграмму сигналов на рис. 9.1. На такте A значение переменной q равно TRUE; спустя один такт (на такте B) значение r равно TRUE; спустя еще три такта (на такте E) значение s равно TRUE. Утверждение проследит за этой последовательностью и обнаружит, что свойство выполняется. Если бы мы задали оператор pass_Statement, он бы исполнился (раз мы этого не сделали, утверждение ничем не дало о себе знать).

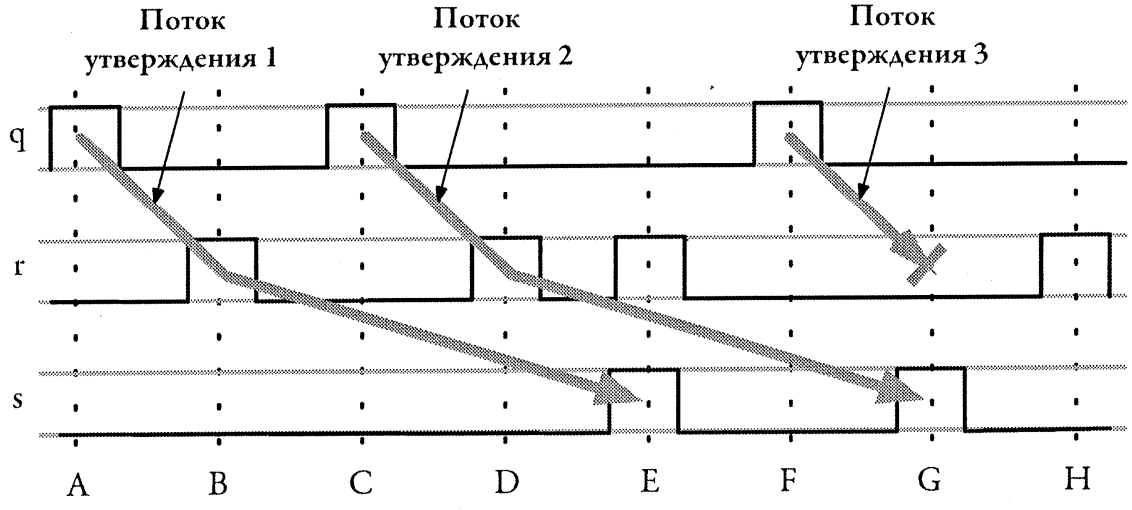



**Рис. 9.1.** Примеры последовательностей сигналов для свойства q1r3s

Сказанное выше показано на рисунке как «Поток утверждения 1». Работу утверждения можно представить следующим образом: когда первая переменная (q) имеет значение TRUE, запускается *поток утверждения*, проверяющий оставшуюся часть свойства[51]. Этот поток исполняется симулятором на этапе наблюдения и в конечном итоге завершается либо успешно, либо с ошибкой.

Когда утверждение видит начало второй последовательности (такт C), оно запускает еще один поток («Поток утверждения 2»). Оба потока перекрываются по времени (такты C и E), тем не менее это разные потоки, независимые друг от друга.

На такте F запускается третий поток («Поток утверждения 3»), завершающийся с ошибкой на такте G: значение переменной r оказывается FALSE.

Обратите внимание: каждый раз проверяется только то, что q имеет значение TRUE сейчас, r – на следующем такте, а s – спустя три такта. Свойство не говорит, что значение r равно FALSE на втором такте после установки q или что значение s равно FALSE спустя два такта после установки r. Как показано на рис. 9.1, тот факт, что значение r на тактах E и H равно TRUE, не влияет на проверку; эти значения просто игнорируются.

Мы ввели новую конструкцию – *оператор тактовой задержки* ##n. Вместо задержки на определенную величину времени (#n), ##n задает задержку

[51] Строго говоря, это не так: ложное значение переменной q – нарушение приведенного свойства, а не его игнорирование. Чтобы добиться описанного поведения, нужно добавить импликацию: `q |-> ##1 r ##3 s`.

в терминах числа тактов. В примере 9.2 тактовый сигнал и его активный фронт определяются оператором @(posedge ck) в строке 5. Таким образом, ##n – задержка на n передних фронтов сигнала ck.

Оператор тактовой задержки можно использовать и в процедурных блоках. В примере 9.3 показано, как им можно заменить @(posedge ck). Если в свойствах ясно, какой сигнал является тактовым, то в процедурных блоках это не всегда так, и тактовый сигнал нужно указывать. Для этого применяется конструкция default clocking (строки 4–6) – тактовый сигнал (часы) по умолчанию. В строках 17–20 генерируется сигнал ck: передний фронт возникает в моменты времени, заканчивающиеся на 5 (5, 15, 25, 35, ...). В строках 8–15 меняется значение переменной q: в момент 0 устанавливается значение 0; в момент 15 (по второму переднему фронту) – 1; в момент 55 – снова 0.

```
1  program cycleDelay;
2   bit ck, q;
3
4   default clocking cName
5     @(posedge ck);
6   endclocking
7
8   initial begin
9     q <= 0;
10    ##2 q <= 1;
11    ##3;
12    ##1 q <= 0;
13    ##1;
14    $finish;
15  end
16
17  initial begin
18    ck = 0;
19    forever #5 ck = ~ck;
20  end
21 endprogram: cycleDelay
```

**Пример 9.3.** Использование тактовых задержек в процедурном коде

## 9.2.2. Свойства и последовательности

В свойстве, показанном выше, использовались объявленные в модуле переменные q, r и s. Свойства могут быть параметризованы; подстановка фактических параметров вместо формальных осуществляется в утверждениях. Рассмотрим следующий код, эквивалентный примеру 9.2.

```
1   property a1b3c (a, b, c);
2     @(posedge ck) a ##1 b##3 c;
3   endproperty
4
5   assert property (a1b3c (q, r, s)) else $error("oops");
```

Единственное отличие здесь состоит в том, что свойство параметризовано. В утверждении вместо параметров a, b, c подставляются переменные q, r, s (строка 5).

Последовательности – важное средство спецификации поведения. Их можно определять отдельно от свойств. Это делается с помощью ключевого слова sequence. В нашем примере последовательность можно определить так:

```
1   sequence abxxc (a, b, c);
2     a ##1 b ##3 c;
3   endsequence
```

А затем определить свойство, использующее эту последовательность:

```
1   property checkQRS (q, r, s);
2     @(posedge ck) abxxc (q, r, s);
3   endproperty
```

Обратите внимание, что определения последовательности и свойства параметризованы – это обеспечивает гибкость.

Могло бы случиться, что последовательность abxxc – только часть проверяемого поведения. Например, если бы переменная d должна была принимать значение TRUE через два такта после abxxc, мы могли бы записать такое свойство:

```
1   property checkQRSD (q, r, s, d);
2     @(posedge ck) abxxc (q, r, s) ##2 d;
3   endproperty
```

## 9.2.3. Как интерпретируются утверждения

На рис. 9.1 мы разобрали по шагам работу параллельных утверждений. Опишем этот механизм более формально. На каждом такте утверждение пробует распознать начало своей последовательности. Возьмем, к примеру, свойство checkQRS и такое утверждение:

```
1 assert property (checkQRS (q, r, s)) $display("Yesssss!") else $error("Noooo!");
```

Оператор активирует проверку свойства с указанными фактическими параметрами. После этого на каждом фронте тактового сигнала последовательность, описанная в свойстве, сопоставляется со значениями сигналов модели. Возможны пять исходов.

- Свойство видит начало своей последовательности (в нашем примере – значение q равно TRUE)[52]. На консоль ничего не выводится; запускается поток утверждения для проверки оставшейся части последовательности.
- Свойство не видит начала своей последовательности (в нашем примере – значение q равно FALSE). Это называется *бессодержательным успехом* (*vacuous success*) – *попытка* запустить поток утверждения была, но реального запуска не было[53].
- Поток утверждения не увидел продолжения своей последовательности. Свойство нарушено: поток завершается; исполняется fail_Statement (в нашем примере выводится "Noooo!").
- Поток утверждения увидел очередной (но не последний) шаг своей последовательности. Он продолжает наблюдать за последующими шагами.
- Поток утверждения увидел последний шаг своей последовательности. Свойство выполнено: поток завершается (проверять больше нечего); исполняется pass_Statement (в нашем примере выводится "Yesssss!").

---

[52] Следует отметить, что распознавание «начала последовательности» (посылки импликации) может занять несколько тактов (например, q ##1 r |-> ##3 s).

[53] Очевидно, импликация с ложной посылкой истинна независимо от истинности/ложности ее следствия. Вопрос: исполняется ли в этом случае pass_Statement? Да (но такое поведение можно изменить с помощью системной процедуры $assertcontrol).

## 9.2.4. Автоматный взгляд на последовательности

Последовательность abxxc можно представить в виде конечного автомата, как показано на рис. 9.2. Состояния меняются по фронту тактового сигнала в зависимости от значений входов. В данном случае входами служат переменные, используемые в последовательности.

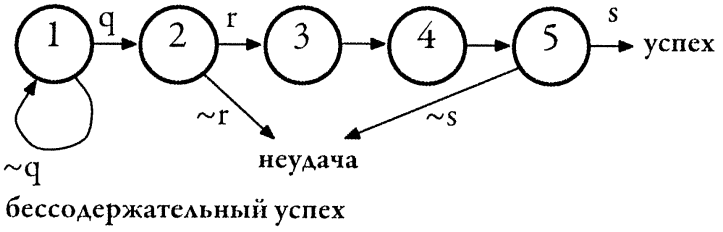



**Рис. 9.2.** Представление последовательности abxxc в виде автомата

Если в состоянии 1 (начальном состоянии) значение q равно FALSE, имеет место бессодержательный успех (автомат остается в состоянии 1); иначе – автомат переходит в состояние 2. Если в состоянии 2 значение r равно TRUE, автомат переходит в состояние 3; иначе – в состояние «неудача» (свойство нарушено). Если спустя три такта, в состоянии 5, значение s равно TRUE, автомат переходит в состояние «успех» (свойство выполнено); иначе – в состояние «неудача».

Как отмечалось выше, на каждом такте поток утверждения проверяет значения только определенных переменных (или выражений). Так, например, не проверяется, что s имеет значение FALSE в состояниях 3 и 4.

Можно провести аналогию между последовательностями в SystemVerilog и регулярными выражениями (регулярные выражения и конечные автоматы имеют одинаковую выразительную силу).

## 9.2.5. Еще о предыдущем примере

Сложим известные нам детали о работе утверждений в единую картину. Рассмотрим диаграмму сигналов из рис. 9.1, где проверяется последовательность (q ##1 r ##3 s).

Последовательность и свойство могли бы быть такими:

```
1  sequence s1 (q, r, s);
2    (q ##1 r ##3 s);
3  endsequence
4
5  property p1 (q, r, s);
6    @(posedge ck) s1(q, r, s);
7  endproperty
8
9  assert property (p1 (q, r, s)) else $error("oops");
```

Глядя на описание из раздела 9.2.3, разберем, как работает этот пример. На каждом переднем фронте тактового сигнала ck свойство p1 пытается распознать по-

следовательность s1 (строка 6). Поскольку не всегда значение q равно TRUE, неизбежны бессодержательные успехи, заставляющие ждать следующего такта.

Многие утверждения начинаются *импликацией*: поток утверждения запускается, только если определенное условие истинно. Так можно избежать бессодержательных успехов[54]. Есть две формы импликации:

```
1   i |=> j; // если i истинно сейчас, то j должно быть истинно на следующем такте
2   i |-> j; // если i истинно сейчас, то j должно быть истинно сейчас
```

```
1  module assertQRS;
2    bit q=0, r=0, s=0;
3    bit ck=0;
4
5    always #5 ck = ~ck;
6
7    initial begin
8      $monitor($stime,,,
9        "ck=%b, q=%b, r=%b, s=%b",
10         ck, q, r, s);
11     q <= #4 1;
12     q <= #6 0;
13     r <= #14 1;
14     r <= #16 0;
15     q <= #14 1;
16     q <= #16 0;
17     r <= #24 1;
18     r <= #26 0;
19     s <= #44 1;
20     s <= #46 0;
21     #56 $finish;
22   end
23
24   sequence s2(r, s);
25     (r ##3 s);
26   endsequence
27
28   property checkQRS (q, r, s);
29     @(posedge ck) q |=> s2(r, s);
30   endproperty
31
32 P1a: assert property (p2(q, r, s))
33         $display("%d Yes!", $stime);
34         else $error("%d oops", $stime);
35 ...
```

**Пример 9.4.** Свойство checkQRS и тестовое окружение для его проверки

В обоих случаях i – выражение, принимающее значение TRUE или FALSE, а j – последовательность[55]. Если значение i равно TRUE, предпринимается попытка распознать последовательность j. Подобные импликации используются только в описании свойств (property ... endproperty). Различие между этими двумя формами заключается в моменте времени, когда начинает проверяться последовательность: в первом случае (|=>) первый шаг последовательности проверяется на следующем такте; во втором (|->) – на текущем.

Перепишем свойство с использованием импликации, как показано в строках 24–30 примера 9.4.

Мы сделали два исправления: убрали переменную q из последовательности, заменив (q ##1 r ##3 s) на (r ##3 s) (строка 25), и сделали q посылкой импликации в свойстве (строка 29). Полученное свойство интерпретируется следующим образом: если по переднему фронту сигнала ck значение q равно TRUE, начиная со следующего фронта ck проверяется последовательность s2. Заметим, что задержка ##1 в исходной последовательности перешла в импликацию |=>.

В примере 9.4 показано описание, а в примере 9.5 – результат симуляции. На рис. 9.3 изображена временна́я диа-

[54] Это не так: бессодержательный успех (vacuous success) есть истинность импликации вследствие ложности посылки — применение импликации позволяет избежать нарушений свойства, заменив их на бессодержательные успехи.

[55] В общем случае i является последовательностью (выражение – это частный случай).

грамма для переменных q, r, s, соответствующая примеру 9.4. Эти детали приведены для того, чтобы перед глазами читателя были все временны́е соотношения.

Обратите внимание на сообщение "Yes!", выведенное на консоль в момент 45. Это поток утверждения 1 успешно завершился и исполнил оператор pass_Statement. Поток утверждения 2 начался в момент 15 и в момент 55 закончился неудачно, напечатав "oops", поскольку ожидал, что значение s будет равно TRUE.

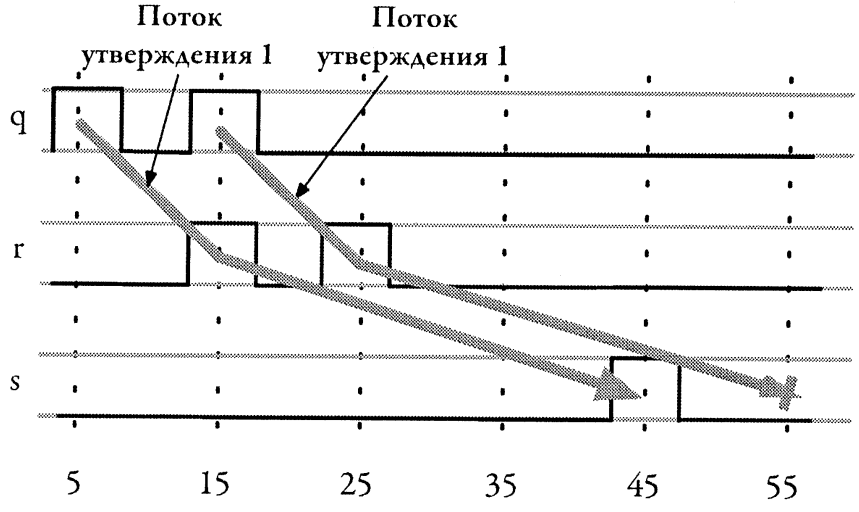



**Рис. 9.3.** Временна́я диаграмма для примера 3.4

Нельзя не отметить одну деталь. Обратите внимание, как в строках 11–20 примера 9.4 генерируются значения переменных q, r и s. Например, q устанавливается в момент 4 и сбрасывается в момент 6 (это создает первый импульс, распознанный в момент 5). Вопрос: можно ли установить q в момент 5 (на фронте тактового сигнала); увидит ли утверждение новое значение? Ответ отрицательный: в утверждениях используются значения, сэмплированные на предварительном этапе симуляции. Это показано на рис. 9.4. Из рисунка видно, что если q устанавливается в момент 4, то сэмплированное значение в момент 5 равно 1; если же q устанавливается в момент 5, сэмплированное значение – 0. Таким образом, значение q должно быть установлено *до* момента 5 – только тогда утверждение увидит это в момент 5.

```
1     0 ck=0, q=0, r=0, s=0
2     4 ck=0, q=1, r=0, s=0
3     5 ck=1, q=1, r=0, s=0
4     6 ck=1, q=0, r=0, s=0
5     10 ck=0, q=0, r=0, s=0
6     14 ck=0, q=1, r=1, s=0
7     15 ck=1, q=1, r=1, s=0
8     16 ck=1, q=0, r=0, s=0
9     20 ck=0, q=0, r=0, s=0
10    24 ck=0, q=0, r=1, s=0
11    25 ck=1, q=0, r=1, s=0
12    26 ck=1, q=0, r=0, s=0
```

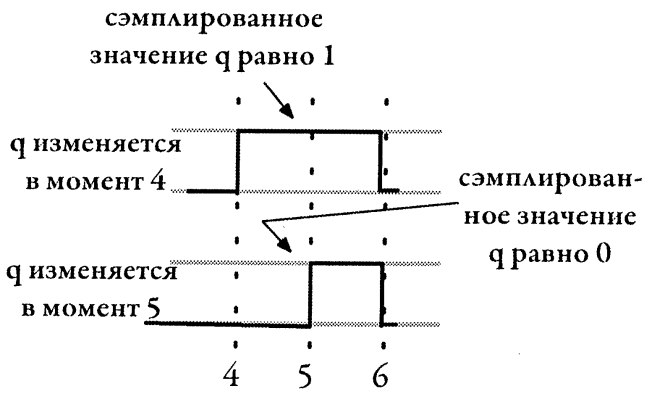



**Рис. 9.4.** Выборка значений на предварительном этапе

```
13     30 ck=0, q=0, r=0, s=0
14     35 ck=1, q=0, r=0, s=0
15     40 ck=0, q=0, r=0, s=0
16     44 ck=0, q=0, r=0, s=1
17     45 Yes!
18     45 ck=1, q=0, r=0, s=1
19     46 ck=1, q=0, r=0, s=0
20     50 ck=0, q=0, r=0, s=0
21 "ConcurrentAssertBasicBook.sv", 32:
22 assertQRS.P1a: started at 15s failed at 55s
23   Offending 's'
24 Error: "ConcurrentAssertBasicBook.sv", 32:
25 assertQRS.P1a: at time 55
26     55 oops
27     55 ck=1, q=0, r=0, s=0
28 $finish called from file
29 "ConcurrentAssertBasicBook.sv", line 20.
30 $finish at simulation time 56
```

**Пример 9.5.** Результат работы модуля assertQRS

# 9.3. Последовательности с диапазонами и повторениями

Последовательности широко используются для спецификации свойств протоколов в цифровых системах; в них предусмотрены операторы для задания разного рода деталей. В этом разделе описывается простой протокол SimpleBus и показывается, как применить последовательности для верификации его реализации.

## 9.3.1. Определение протокола SimpleBus

Протокол SimpleBus изображен на рис. 9.5. Он служит для соединения процессора и памяти через шину. Мы уже разбирали этот пример в разделе 6.3. Процессор управляет линиями `address`, `start` и `read`. Формирование значений на линиях `data` и `dataValid` может осуществляться как процессором, так и памятью. В случае чтения память помещает на линию `data` считанное значение и устанавливает сигнал `dataValid`, что говорит о наличии данных. В случае записи процессор помещает на линию `data` значение, подлежащее записи, и устанавливает сигнал `dataValid`. Шина адреса мультиплексируется – сначала передаются старшие 8 бит адреса, затем – младшие 8 бит.

Операция чтения начинается в состоянии S1: процессор помещает на линию `address` старшую половину адреса и устанавливает сигнал `start`. В состоянии S2 на линию `address` помещается младшая половина адреса и устанавливается сигнал `read`, что указывает на операцию чтения. В случае записи `read` не устанавливается. В начальный момент времени память ждет установки сигнала `start`, после чего считывает старшую половину адреса с линии `address`. В следующем состоянии (S2) она (уже безо всяких сигналов) считывает младшую половину адреса и проверяет, в каком состоянии находится `read`.

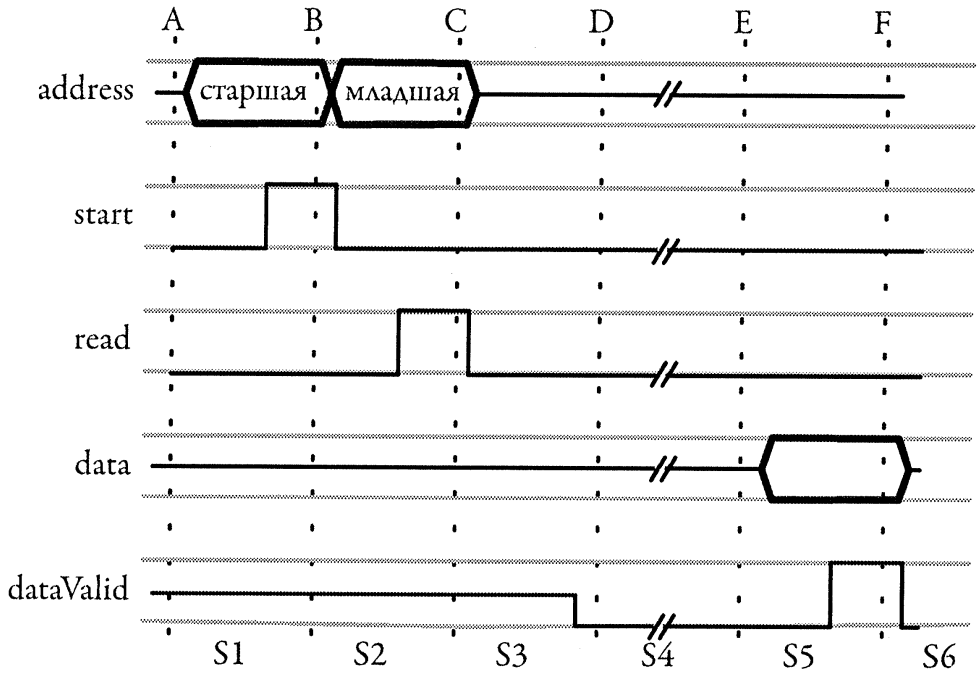



**Рис. 9.5.** Временна́я диаграмма протокола SimpleBus

На рис. 9.5 показана операция чтения: в состоянии S3 память изменяет значение `dataValid` с z на 0, чтобы в состоянии S5, установив этот сигнал, известить процессор о готовности данных. Отметим, что состояние S4 замещает много состояний, в которых процессор ждет, когда память прочитает запрошенное значение и поместит его на линию `data` (или память ждет, когда процессор поместит на линию значение для записи). В состоянии S6 сигнал `dataValid` сбрасывается: в этом состоянии может начаться следующая операция.

Операция записи отличается одним: сигнал `read` не устанавливается, остальные линии работают аналогично (разница лишь в том, что значения на линиях `data` и `dataValid` формируются процессором).

На рис. 9.5 есть ряд деталей, которые не описаны в тексте. Например, сигналы `start` и `read` сохраняют высокий уровень только в течение одного такта (остальное время они сброшены); или тот факт, что очередная установка `start` может состояться только после установки `dataValid` (в состоянии S6).

## 9.3.2. Спецификация свойств протокола

Основное свойство для протокола SimpleBus записывается в виде:

```
1   property x;
2     @(negedge ck) start |=> // какая-то последовательность
3   endproperty
```

Согласно протоколу, взаимодействие начинается по сигналу `start`; по этому же сигналу мы начинаем проверку реализации. Последовательности, используемые для проверки, можно определить отдельно, а затем вызвать в строке 2 (как мы делали ранее), либо записать прямо в строке 2. Для упрощения обсуждения мы будем записывать последовательности прямо в свойстве.

Сначала предположим, что состояние задержки (S4 на рис. 9.5) отсутствует. На рис. 9.6 показан новый вариант временно́й диаграммы. Теперь операции чтения и записи занимают по три такта. Ниже приведено простое свойство, говорящее, что операция чтения исполняется за три такта:

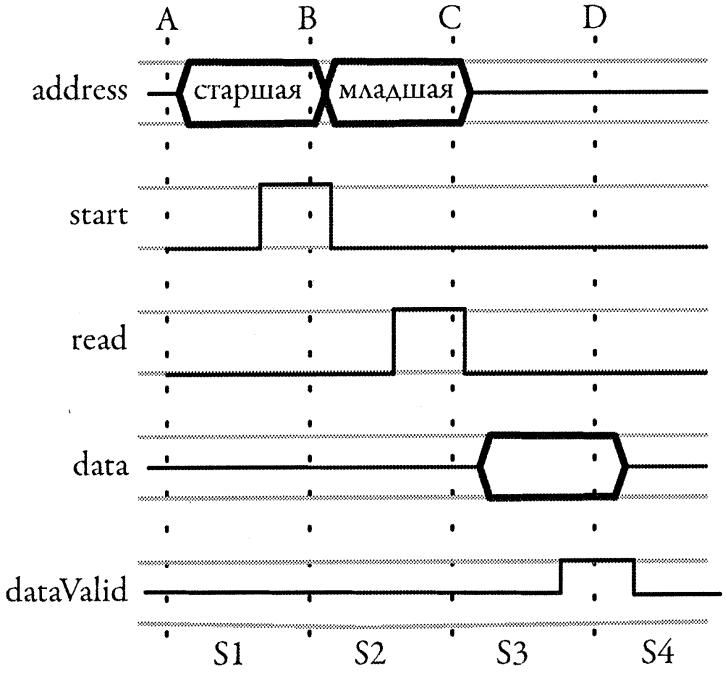



**Рис. 9.6.** Временна́я диаграмма протокола чтения SimpleBus без задержки

```
1   property readIs3ticks;
2     @(negedge ck) start |=> read ##1 dataValid;
3   endproperty
```

Здесь проверяется, что если установлен сигнал `start` (такт B), то на следующем такте (импликация `|=>`) будет установлен сигнал `read`, а спустя еще один такт (`##1`) – `dataValid`.

В SystemVerilog есть полезная системная функция `$isunknown(y)`, возвращающая TRUE, если значение выражения `y` неизвестно (равно `x` или `z`). Поскольку линия адреса обычно имеет высокий импеданс, нелишне проверить, что адрес сформирован, когда сигнал `start` установлен:

```
1   property addressBusDriven;
2     @(negedge ck) start |-> ~$isunknown(address);
3   endproperty
```

Здесь использована другая форма импликации. Ранее мы проверяли значение на следующем такте (`|=>`), а теперь нас интересует текущий момент времени (`|->`), когда значение `start` равно TRUE. Читать это свойство нужно так: в тот момент, когда сигнал `start` установлен, линия адреса не должна содержать неизвестное значение. Это классическая импликация «из `start` следует, что адрес сформирован».

Свойство `readIs3ticks`, определенное выше, проверяет операцию чтения: увидев сигнал `start`, оно смотрит, установлен ли на следующем такте сиг-

нал read. При записи это свойство нарушается, потому что read в этом случае не устанавливается. Мы не хотим, чтобы свойства нарушались, даже если это и не свидетельствует о проблеме в тестируемой модели.

Чтобы избежать этого, можно просто проверить сигнал dataValid на третьем такте:

```
1   property ckDataValidOnly;
2     @(negedge ck) start |=> ##1 dataValid;
3   endproperty
```

В этом свойстве проверка запускается на следующем такте после установки start. После этого нужно подождать еще один такт, а затем проверить значение dataValid. Это можно записать в эквивалентной форме:

```
1   property ckDataValidOnlyAlt;
2     @(negedge ck) start |-> ##2 dataValid;
3   endproperty
```

Импликация (|->) говорит о том, что проверка начинается с того же такта, на котором установлен start. Суть та же: подождать 2 такта, а затем проверить dataValid (длительность последовательности составляет 3 такта).

Немного усложнив посылку импликации, можно специфицировать операцию чтения. Рассмотрим такое свойство:

```
1   property ckRead;
2     @(negedge ck) (start ##1 read) |=> dataValid;
3   endproperty
```

Условие здесь состоит в следующем: значение start равно TRUE, а на следующем такте значение read равно TRUE. Если это так, значит, производится чтение, и на следующем такте сигнал dataValid должен быть установлен.

Свойство для операции записи имеет вид:

```
1   property ckWrite;
2     @(negedge ck) (start ##1 ~read) |=> dataValid;
3   endproperty
```

Единственное отличие состоит в том, что в условии свойства ckWrite проверяется, что сигнал read сброшен.

## 9.3.3. Операции над последовательностями

Вернемся к исходной спецификации протокола SimpleBus, в которой промежуток времени между сигналами start и dataValid не определен (рис. 9.5). Будем считать, что для записи он должен быть не меньше 2 тактов и не больше 7, а для чтения – не меньше 2 тактов и не больше 10. Таким образом, меньше всего времени операция чтения занимает, когда dataValid следует сразу после read (2 такта после start).

Для задания диапазона числа тактов в последовательности служит конструкция ##[n:m], где n – минимальное число тактов, а m – максимальное. Для операции чтения свойство имеет вид:

```
1   property ckReadRange;
2     @(negedge ck) (start ##1 read) |-> ##1 read ##[1:9] dataValid;
3   endproperty
```

Посылка импликации нам известна: на текущем такте установлен сигнал start, а на следующем – read. Если это так, то значение read должно быть равно TRUE через такт после установки start (это повторение части посылки), а dataValid – в течение промежутка от 1 до 9 тактов после read.

Границы диапазона – это неотрицательные целые числа. В качестве верхней границы можно использовать символ $, означающий *неограниченное, но конечное* число тактов. Интерпретация здесь следующая: последовательность, стоящая после диапазона с $, *в конечном итоге* произойдет, но неизвестно, когда именно (это должно случиться до конца симуляции, в противном случае свойство считается нарушенным[56]).

В данной модели имеется две операции, и допустимые времена задержек у них разные. Чтобы отразить эту ситуацию, можно воспользоваться связкой or и с ее помощью соединить условия на операции чтения и записи:

```
1   property ckReadOrWrite;
2     @(negedge ck) start |=> ((read ##[1:9] dataValid) or
3                              (~read ##[1:6] dataValid));
4   endproperty
```

Здесь заданы две последовательности, связанные оператором or. Первая успешно завершившаяся последовательность завершает и все свойство – тоже успешно. Если обе последовательности завершаются неудачно, свойство считается нарушенным.

Еще один аспект протокола, допускающий проверку, относится к сигналу start. Сигнал сообщает всем интерфейсам на шине о начале транзакции (операции чтения или записи). Протокол допускает обработку не более одной транзакции в каждый момент времени. Чтобы проверить это, мы должны убедиться, что после начала транзакции, т. е. после установки сигнала start на один такт, его значение должно быть равно FALSE вплоть до такта, следующего за установкой dataValid. Для этого нужно проверять значения start на каждом такте. Это можно записать с помощью оператора throughout.

```
1   property ckOnlyOneStart;
2     @(negedge ck) start |=> ~start throughout
3                   ((read ##[1:9] dataValid) or
4                   (~read ##[1:6] dataValid));
5   endproperty
```

В левой части оператора throughout находится выражение, которое должно сохранять значение TRUE на всем протяжении последовательности, стоящей

---

[56] Это не так: по умолчанию свойство в утверждении (в отличие от свойства в покрытии) является *слабым*; нераспознавание последовательности из-за нехватки времени не есть нарушение свойства. Свойство можно сделать сильным, указав квалификатор strong, например strong(read ##[1:$] dataValid).

в правой части. Свойство в примере утверждает: если сигнал start принял значение TRUE, то на следующем такте его значение изменяется на FALSE (~start есть TRUE) и остается таким, пока при обработке чтения или записи не появится сигнал dataValid. То есть start устанавливается только на один такт и не может быть установлен повторно до такта, следующего за установкой dataValid. Это свойство можно записать проще:

```
1   property ckOnlyOneStartAlt;
2     @(negedge ck) start |=> ~start throughout ##[1:9] (dataValid & ~start);
3   endproperty
```

При таком подходе мы смотрим только на dataValid вне зависимости от того, какая операция исполняется, чтение или запись. Соответственно, мы не сможем обнаружить, что операция записи заняла больше 7 тактов: значение сигнала start проверяется на промежутке времени, ограниченном наиболее длинной транзакцией.

Помимо упомянутых throughout и or, для последовательностей определены и другие операторы. Ниже приведен полный перечень.

- or – две последовательности начинаются на одном такте. Как только одна из них успешно завершается, утверждение завершается и считается истинным. Если обе последовательности завершаются неудачно (возможно, в разное время), утверждение считается ложным.
- and – две последовательности начинаются на одном такте. Обе должны завершиться успешно (возможно, в разное время).
- intersect – как and, но с дополнительным ограничением: последовательности должны завершиться в одно и то же время.
- throughout – выражение в левой части должно сохранять значение TRUE на всем протяжении последовательности в правой части.
- within – записывается в виде s1 within s2. Последовательность s1 должна начаться не раньше s2, а закончиться не позже s2.
- first_match – если последовательности заданы с помощью перечисленных операторов и содержат повторения (см. ниже), число которых задано в форме диапазонов, может быть несколько совпадений в разные моменты времени. Этот оператор сообщает о первом; все прочие отбрасываются.

## 9.3.4. Повторение частей в последовательностях

В некоторых протоколах сигнал или последовательность сигналов повторяется.

Рассмотрим свойство ckOnlyOneStartAlt (см. выше), утверждающее, что после начала последовательности значение start равно FALSE вплоть до такта, следующего за установкой dataValid (когда start может быть установлен повторно). Это условие было задано с помощью оператора throughout и диапазона в операторе ##delay. Есть и другой способ: задать в виде диапазона число повторений нулевого значения start:

```
1   property ckOnlyOneStartRepetitions;
2     @(negedge ck) start |=> ~start [*1:8] ##1 (dataValid & ~start);
3   endproperty
```

Свойство утверждает, что начиная с такта, следующего за установкой start, должно пройти от 1 до 8 тактов со сброшенным start, а затем – еще один такт с установленным dataValid и сброшенным start. Отметим несколько моментов.

- Оператор последовательного повторения, имеющий здесь вид [*1:8], означает, что элемент в левой части, в данном случае ~start, должен повторяться на протяжении от 1 до 8 последовательных тактов.
- На последнем такте, когда происходит выход из цикла наблюдения за ~start, выражение dataValid & ~start должно иметь значение TRUE. Тем самым гарантируется, что в момент установки dataValid сигнал start сброшен: очередная установка start может произойти не раньше, чем на такте, следующем за появлением dataValid. Опуская условие ~start в выражении, мы разрешаем установку start вместе с dataValid, а это нарушение протокола.

Рассмотрим спецификацию еще одного протокола (см. рис. 9.7). Отправитель помещает байты на линии данных на нескольких последовательных тактах. О том, что на линии есть данные, он сообщает, устанавливая сигнал готовности (dr). Пока dr установлен, получатель принимает данные. Подряд может быть передано от 2 до 73 байтов. Приведенное ниже свойство отражает это[57].

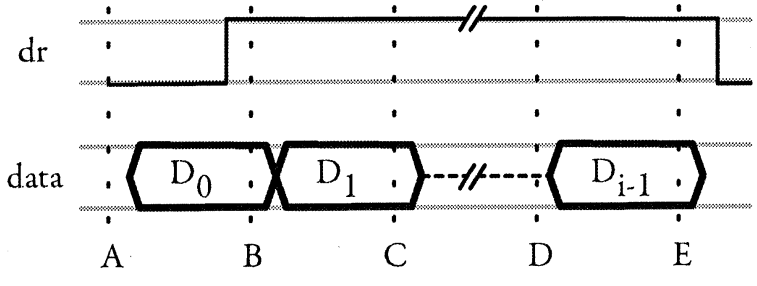



**Рис. 9.7.** Сигнал dr показывает, когда передается блок данных

```
1   property ckDataByteCountConsec;
2     @(negedge ck) dr |=> dr [*1:72] ##1 ~dr;
3   endproperty
```

Сигнал dr устанавливается на такте B. Далее идут от 1 до 72 тактов, на которых сигнал держится на высоком уровне, после чего сбрасывается. Если вы хотите, чтобы в свойстве были именно те числа, что фигурируют в спецификации (2 и 73), можете написать:

```
1   property ckDataByteCountConsecAlt;
2     @(negedge ck) dr |-> dr [*2:73] ##1 ~dr;
3   endproperty
```

---

[57] Аккуратнее было бы написать (~dr ##1 dr) |=> dr [*1:72] ##1 ~dr.

Здесь благодаря оператору импликации |-> подсчет последовательных повторений начинается на том же такте, на котором был установлен сигнал dr.

Изменим спецификацию протокола, как показано на рис. 9.8. Протокол похож на предыдущий, за тем исключением, что байты необязательно передаются подряд, на последовательных тактах. Высокий уровень dr, как и раньше, говорит о наличии данных на линии, но теперь завершение передачи сигнализируется установкой done. Как видно по рисунку, done устанавливается следом за последним dr, когда на шине нет данных. Это такт F на рис. 9.8. Между установками dr и done по-прежнему может быть передано от 2 до 73 байтов.

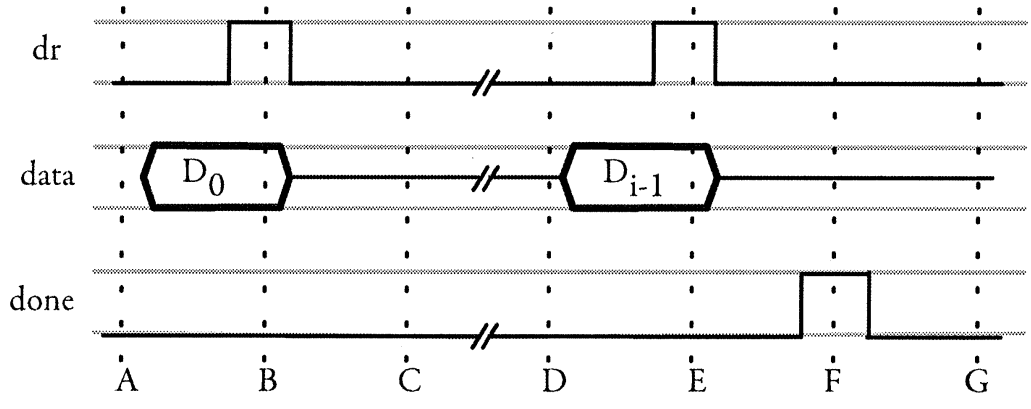



**Рис. 9.8.** Временна́я диаграмма установки dr и done

Для проверки такой передачи подойдет следующее свойство:

```
1   property ckDataByteCountNonConsec;
2     @(negedge ck) dr |-> dr [->2:73] ##1 done;
3   endproperty
```

Свойство отличается от предыдущего оператором повторения *goto* [->2:73]. Он описывает две вещи. Во-первых, сигнал dr может быть установлен от 2 до 73 раз, но необязательно подряд (между тактами, на которых сигнал установлен, могут быть такты, на которых он сброшен). Во-вторых, сигнал done в конце последовательности должен быть установлен сразу после последнего dr, как показано на рис. 9.8.

У непоследовательного повторения существует альтернативное окончание, когда допускается, что done устанавливается не сразу после последнего dr. Это описывается следующим образом:

```
1   property ckDataByteCountNonConsec;
2     @(negedge ck)
3       dr |-> dr [=2:73] ##1 done;
4   endproperty
```

На рис. 9.9 показана временна́я диаграмма: установка done отстоит на два такта от последнего dr. Такая последовательность соответствует приведенному свойству, несмотря на финальную конструкцию ##1 done. Можно считать, что повторение завершается на такте F. Нет никаких ограничений на задержку между тактом, следующим за последней установкой dr, и тактом, на котором появляется done. Кстати, последовательность на рис. 9.8 тоже удовлетворяет приведенному свойству.

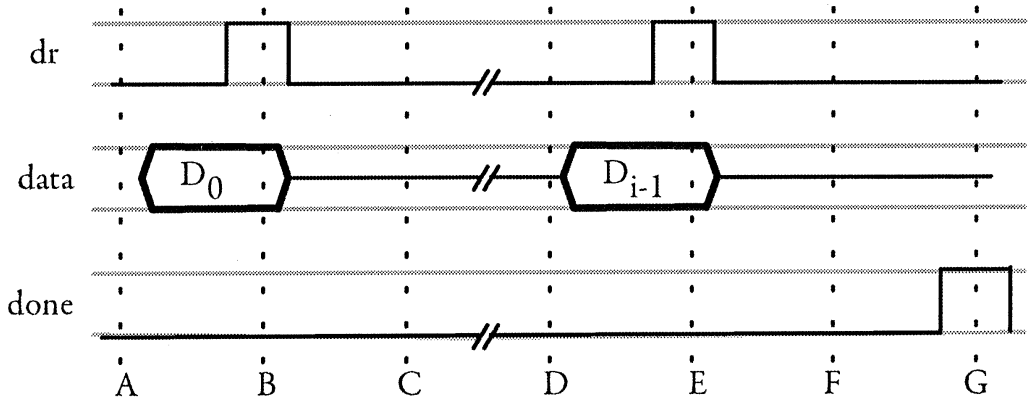



**Рис. 9.9.** Временна́я диаграмма для оператора непоследовательного повторения (=)

Итак, существуют три способа указать, что последовательность (или одиночный сигнал) должна повторяться некоторое число раз.

- *Последовательные повторения* – последовательность должна повторяться заданное число раз непрерывно, без пропусков тактов. Конструкция b[*8], например, описывает восемь подряд идущих повторений b, а конструкция b[*3:7] – от 3 до 7 таких повторений.
- *Непоследовательные повторения типа goto* – последовательность должна повторяться заданное число раз, но между повторениями возможны пропуски тактов. Например, конструкция b[->8] ##1 done означает, что сигнал b устанавливается на восьми тактах (необязательно соседних), а сигнал done появляется на такте, следующем за последней установкой b (см. рис. 9.8).
- *Непоследовательные повторения* – похожи на повторения типа goto, за тем исключением, что задержки возможны не только между повторениями, но и после последнего из них. Так, конструкция b[=8] ##1 done снимает ограничение того, что done должен идти сразу за последним b (см. рис. 9.9).

Из двух операторов непоследовательного повторения оператор -> более строгий: ему удовлетворяет только первая из последовательностей, показанных на рис. 9.8 и рис. 9.9; оператору = удовлетворяют обе.

## 9.4. Вычисления внутри последовательностей

Пусть нам нужно проверить, что по шине передается правильная последовательность байтов. На рис. 9.10 приведен протокол, в котором это может понадобиться.

Согласно этому протоколу, байты посылаются на каждом такте. Первый байт помещается на линию данных одновременно с установкой сигнала start. Последний байт, являющийся контрольной суммой, помещается на линию данных одновременно с установкой сигнала done.

Суть контрольной суммы состоит в том, что с ее помощью можно проверить правильность принятых данных. Отправитель посылает байты и попутно вычисляет их сумму по модулю 256 – это и есть контрольная сумма. Послав все байты, отправитель посылает вдогонку значение, противоположное контрольной сумме[58]. Получатель тоже вычисляет сумму всех принятых байтов по модулю 256. Поскольку отправитель послал контрольную сумму с противоположным знаком, сумма, вычисленная получателем, должна быть равна 0. Если это не так, значит, во время передачи произошла ошибка.

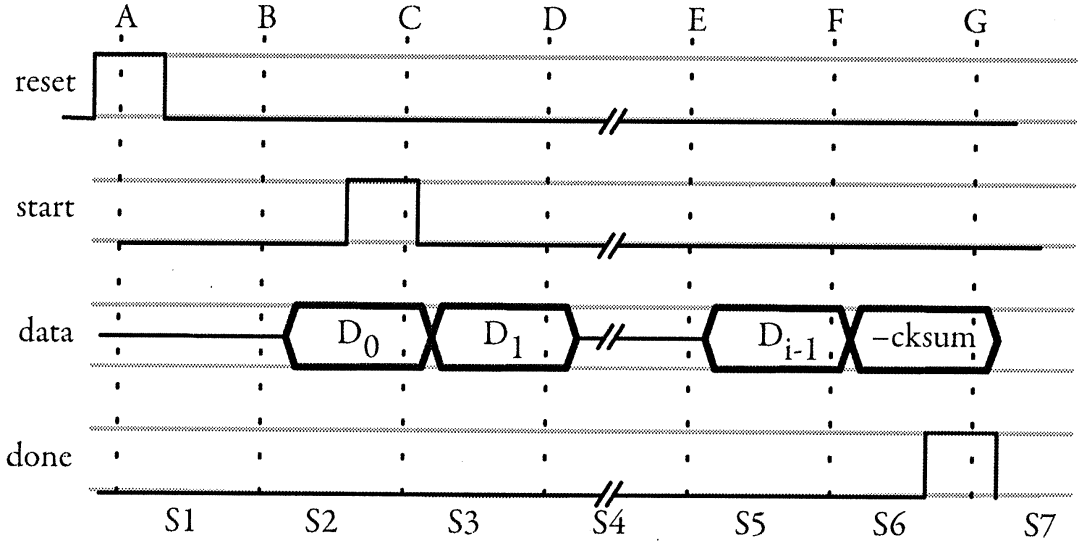



**Рис. 9.10.** Временна́я диаграмма работы калькулятора контрольной суммы

Напишем свойство, которое будет наблюдать за шиной и проверять, правильно ли отправитель посылает данные и контрольную сумму. Для этого нам понадобятся последовательности с локальными переменными. При создании нескольких потоков утверждения в каждом потоке будут использоваться свои копии переменных.

Рассмотрим следующее свойство, проверяющее контрольную сумму:

```
1  property ckCheckSumCorrect;
2    byte cSum;
3    @(negedge ck) disable iff (reset)
4      (start, cSum = data) |=> (~done, cSum += data) [*0:19]
5                          ##1 (done && ((cSum + data) == 0));
6  endproperty
```

[58] В беззнаковой интерпретации: если вычисленная контрольная сумма (*cksum*) равна 0, посылается 0; в противном случае – (256 - *cksum*).

Здесь используются новые языковые средства. Во-первых, это конструкция disable iff (reset) (строка 3). Практически во всех системах есть сигнал сброса reset, переводящий систему в начальное состояние. При сбросе не нужно запускать поток утверждения, а все запущенные потоки следует отменить. Строка 3 говорит, что если установлен reset, то все потоки, проверяющие рассматриваемое свойство, должны быть завершены.

Во-вторых, в строке 2 объявлена локальная переменная.

И наконец, последнее новшество. В строке 4 условие объединено с оператором присваивания: (start, cSum = data). Первый элемент (start) – условие начала передачи данных. Если условие истинно, текущее значение на линии данных (data) загружается в локальную переменную cSum. После этого может пройти от 0 до 19 последовательных тактов, на которых сигнал done сброшен. На каждом из этих тактов данные, прочитанные с шины, прибавляются к cSum (вычисляется контрольная сумма). Когда отправитель помещает на линию данных значение, противоположное контрольной сумме (такт G на рис. 9.10), он устанавливает сигнал done. Итак, на такте G значение на линии данных есть контрольная сумма, вычисленная отправителем (с противоположным знаком), значение переменной cSum есть контрольная сумма, вычисленная утверждением. В строке 5 проверяется, что в момент установки done их сумма равна 0.

В качестве отдельного примера рассмотрим проверку конвейерного устройства умножения. По каждому фронту тактового сигнала устройство считывает значения двух входов, а спустя три такта выдает результат, зависящий исключительно от этих значений. Спрашивается, как написать утверждение для проверки результата. Ниже приводятся описание устройства и свойства.

```
1 module pipeMult (
2   input bit [9:0] inA, inB,
3   input bit ck, reset,
4   output bit [9:0] result);
5
6   bit [9:0] stage1out, stage2out;
7
8   always_ff @(posedge ck) begin
9     stage1out <= inA * inB;
10    stage2out <= stage1out;
11    result <= stage2out;
12  end
13
14  property pipelinedFunction(inA, inB);
15    bit [9:0] A, B;
16    @(posedge ck) disable iff (reset)
17      (1, A = inA, B = inB) ##3 (result == A * B);
18  endproperty
19  ...
20 endmodule: pipeMult
```

Конвейер устройства состоит из трех стадий, показанных в строках 8–12. На первой стадии осуществляется умножение (строка 9); в это время входы все

еще значимы. В строках 10–11 вычисленное значение (stage1out) передается на следующую стадию (stage2out) и, наконец, на выход (result).

В свойстве, предназначенном для проверки устройства, используются локальные переменные (A и B, строка 15): в них запоминаются значения входов в момент запуска потока утверждения. Начальным условием последовательности является 1 (TRUE), далее идет задержка в три такта и сравнение значения выхода с вычисленным внутри последовательности произведением (строка 17). По каждому фронту тактового сигнала запускается новый поток утверждения (за исключением начальных тактов, в каждый момент времени активны три потока). Отметим, что в каждом потоке есть свои копии локальных переменных.

## 9.5. Функции работы с сэмплированными значениями

Симулятор SystemVerilog позволяет узнать, какими были значения переменных (или выражений) на этапе предварительной обработки (этапе сэмплирования)[59]. Для этого предназначены функции работы с сэмплированными значениями (sampled values).

### 9.5.1. Общая информация

Для начала рассмотрим функцию $sampled(expression), возвращающую значение выражения expression на предварительном этапе симуляции текущего слота времени. Функцию можно вызывать в любом месте (только не в моделях, предназначенных для синтеза). Следующий пример иллюстрирует ее работу.

```
1   initial begin
2     b = 0;
3     @(posedge ck);
4     b = 1;
5     $display ("%b %b", b, $sampled(b));
6   end
```

Здесь предполагается, что есть тактовый сигнал ck; в строке 3 ожидается фронт этого сигнала. При исполнении блока процедуры $display выведет "1 0": текущее значение b устанавливается в строке, предшествующей $display (оно выводится как 1); сэмплированное значение b равно 0 (это значение перед началом обработки фронта тактового сигнала).

Сама по себе функция $sampled() не очень интересна. Ниже приведен полный список функций работы с сэмплированными значениями. Имя ce – сокращение от «clock event» (событие синхронизации, что-то вроде @(posedge ck)).

- $sampled (expression) – значение выражения expression, вычисленное с использованием значений, сэмплированных на предварительном этапе симуляции для текущего слота времени.

[59] Подробнее этапы работы симулятора рассматриваются в главе 14.

- `$rose (expression, ce)` – TRUE, если младший бит выражения `expression` изменился и стал равен 1 в период между предыдущим и текущим событиями `ce` (FALSE в противном случае)[60].
- `$fell (expression, ce)` – TRUE, если младший бит выражения `expression` изменился и стал равен 0 в период между предыдущим и текущим событиями `ce` (FALSE в противном случае).
- `$stable (expression, ce)` – TRUE, если значение выражения `expression` не изменилось в период между предыдущим и текущим событиями `ce` (FALSE в противном случае).
- `$changed (expression, ce)` – TRUE, если значение выражения `expression` изменилось в период между предыдущим и текущим событиями `ce` (FALSE в противном случае).
- `$past (expression, numTicks, ce)` – значение выражения `expression`, вычисленное с использованием значений, сэмплированных на предварительном этапе симуляции `numTicks` тактов назад, не считая текущего такта.

Событие синхронизации `ce`, указанное в некоторых функциях, – необязательный аргумент. Если функции используются в свойствах, в качестве `ce` подразумевается тактовый сигнал. В других процедурных блоках события синхронизации могут быть заданы с помощью блока `clocking`, как в примере 9.3.

На рис. 9.11 показаны временны́е диаграммы для сигналов `q` и `r`, а также некоторые из перечисленных функций и их значения в разные моменты времени. В целом здесь все понятно. Отметим, что значение `$fell(r)` на такте C равно 0: сигнал `r` был сброшен на этом такте; сэмплированное значение равно 1, и функция не увидела изменения. На следующем же такте значение `$fell(r)` равно 1.

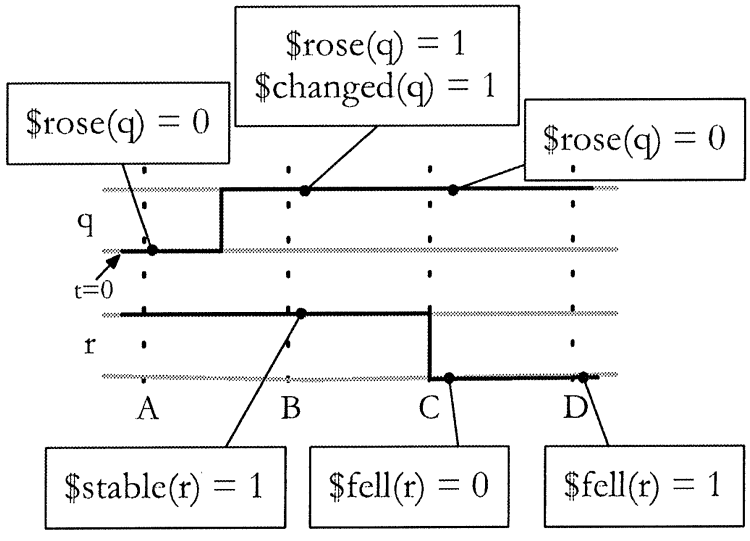



**Рис. 9.11.** Функции работы с сэмплированными значениями

[60] Значение выражения вычисляется на предварительном этапе симуляции при обработке слота времени, в котором возникает событие синхронизации.

## 9.5.2. Использование в последовательностях

Рассмотрим свойство pipelinedFunction (раздел 9.4). В нем проверяется конвейерное устройство умножения, которое на каждом такте считывает значения входов и выдает результат через три такта. Для запоминания входных значений в свойстве используются локальные переменные.

То же свойство можно записать иначе:

```
1  property pipelinedFunctionPast(inA, inB);
2    @(posedge ck) disable iff (reset)
3      result == ($past(inA, 3) * $past(inB, 3));
4  endproperty
```

Функция $past возвращает значения, которые входы inA и inB имели 3 такта назад. Произведение этих значений сравнивается с результатом, выданным устройством в текущий момент времени.

Поскольку в начале симуляции значение result равно 0, а $past возвращает 0 для моментов времени, предшествующих симуляции, то первые утверждения истинны.

В качестве другого примера рассмотрим поток sumItup из раздела 5.1. Временна́я диаграмма работы этого потока повторена на рис. 9.12. Глядя на нее, можно написать следующие свойства:

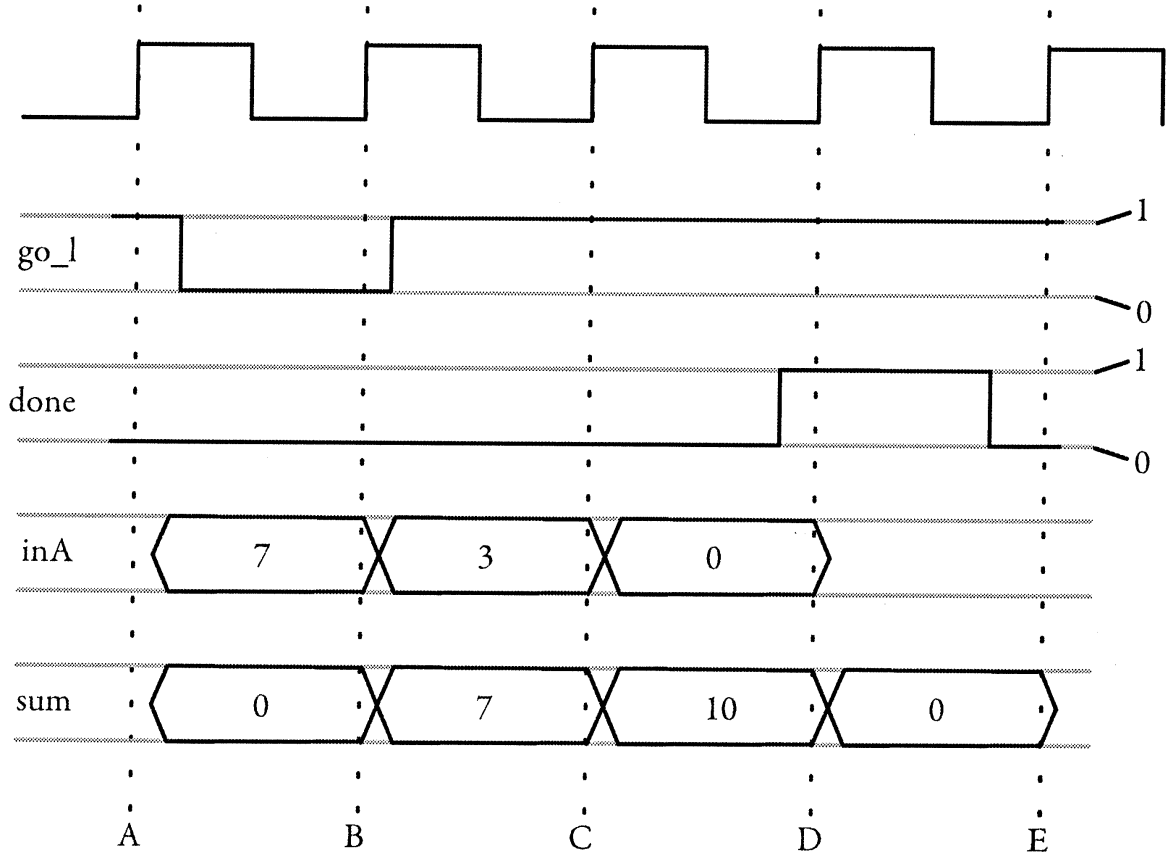



**Рис. 9.12.** Временна́я диаграмма работы потока sumItUp

```
1   property goAssertedForOneClock;
2     @(posedge ck) disable iff (~reset_l)
3       $fell(go_l) |=> $rose(go_l);
4   endproperty
```

Последовательность начинается со спада сигнала go_l. Это обнаруживается на такте B (сэмплированное значение go_l на такте A равно 1, а на такте B – 0). Затем, на такте C (спустя один такт, как того требует импликация |=>), можно видеть, что сигнал go_l снова принял значение 1.

Следующее свойство проверяет то же самое:

```
1   property goAssertedForOneClockAlt;
2     @(posedge ck) disable iff (~reset_l)
3       ~go_l |=> go_l;
4   endproperty
```

А именно: если сигнал go_l установлен сейчас (имеет значение 0), то он будет сброшен на следующем такте (иметь значение 1).

Функции $rise, $fall и $changed полезны, когда требуется отследить момент изменения уровня сигнала (чтобы проверка срабатывала по фронту). Предположим, что некоторый сигнал Q может быть установлен только в течение четырех тактов, идущих подряд. Это можно отразить в следующем свойстве:

```
1   property fourQ;
2     @(posedge ck) disable iff (~reset_l)
3       $rose(Q) |=> Q[*3] ##1 ~Q;
4   endproperty
```

В данном случае проверка начинается, только когда уровень Q меняется с низкого на высокий. Начиная со следующего такта, сигнал держится установленным еще три такта, после чего сбрасывается. Свойство перестает работать, если изменить строку 3 на следующую:

```
1       Q |=> Q[*3] ##1 ~Q;
```

Теперь проверка начинается всегда, когда значение Q равно TRUE, включая второй, третий и четвертый такты после установки Q. Если система работает правильно, эти три проверки закончатся неудачно (сигнал Q будет сброшен раньше, чем через три такта после начала проверки), что не есть хорошо.

Вернемся к нашему примеру. Нужно убедиться, что сигнал done устанавливается, только когда значение inA равно 0 (такт D). Это можно сделать с помощью такого свойства:

```
1   property doneImpliesINAis0;
2     @(posedge ck) disable iff (~reset_l)
3       done |-> (inA == 0);
4   endproperty
```

Отметим, что это утверждение нельзя написать наоборот: из того, что сигнал inA сброшен, не следует, что done установлен. Например, между операциями, когда не нужно суммировать числа, inA сброшен, однако done при этом

не устанавливается. Напомним, что done – это сигнал, который сообщает окружению, что оно может считать выходное значение. Формирование этого сигнала реализовано в конечном автомате потока, и приведенное свойство эту реализацию проверяет.

Через такт после установки done (такт D) выход sum становится равным 0 (такт E):

```
1   property doneImpliesSUMbecomes0;
2     @(posedge ck) disable iff (~reset_l)
3       done |=> (sum == 0);
4   endproperty
```

Между последними двумя свойствами есть важное отличие: первое оперирует с условиями, относящимися к одному моменту времени, а второе – с условиями, разделенными тактом. На временно́й диаграмме первое свойство касается только такта D, а второе охватывает такты D и E.

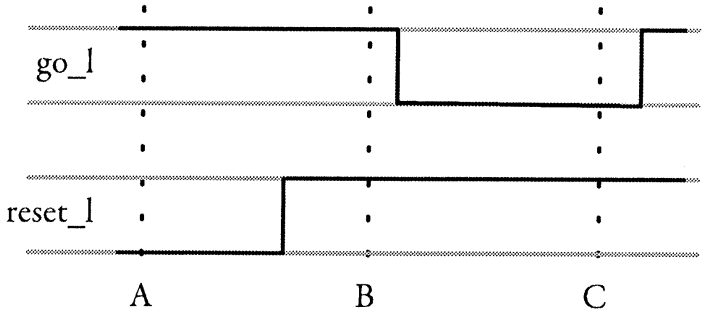



**Рис. 9.13.** Временно́е соотношение между сигналами reset_l и go_l

Действия, связанные со сбросом (reset), обычно запускаются при деактивации соответствующего сигнала (по обратному фронту)[61]. На рис. 9.13 показана временна́я диаграмма, связывающая деактивацию reset_l и установку go_l. Соотношение между этими сигналами отражено в следующем свойстве:

```
1   property goFollowsReset;
2     @(posedge ck)
3       $rose(reset_l) |-> (go_l == 1);
4   endproperty
```

Здесь говорится, что при возникновении обратного фронта reset_l (например, на такте B) go_l должен быть сброшен (иметь значение 1).

Напомним, что в реализации потока sumItUp сигнал reset_l напрямую соединен с асинхронным сбросом регистра sum. Соотношение между reset_l и sum описывается так:

```
1   property sumGetsReset;
2     @(posedge ck)
3       $rose(reset_l) |-> (sum == 0);
4   endproperty
```

Здесь также отслеживатся обратный фронт reset_l. Когда он происходит, регистр sum должен быть обнулен (результат асинхронного сброса). Отметим,

[61] То есть по сбросу (deassertion) сигнала reset: по переднему фронту, если reset имеет низкий активный уровень; по заднему фронту, если reset имеет высокий активный уровень.

что поскольку приведенные выше свойства описывают действия, связанные со сбросом, в них не используется конструкция `disable iff (~reset_l)`.

## 9.6. Задачи и упражнения

**9.1.** Для задачи 5.1 напишите утверждение, проверяющее, что в каждой последовательности из 5 бит имеется два и только два единичных бита.

**9.2.** Для задачи 5.2 напишите следующие утверждения:

**A)** В результате сброса (`rst`) значение `maxValue` становится равным 0.

**B)** После сброса установка `start` предшествует установке `done`.

**C)** После установки `done` значение `maxValue` становится равным 0.

**D)** Значение `maxValue` становится равным 0 после сброса или установки `done` и остается таким, пока на входе не появится первое ненулевое значение, знаменующее начало входной последовательности.

**E)** Сигнал `done` не устанавливается, пока установлен `start`.

**F)** Сигнал `done` устанавливается спустя такт после сброса `start`.

**G)** Значение `maxValue` вычислено правильно (если судить по входам, которые видит утверждение).

**9.3.** Для задачи 5.3 напишите следующие утверждения:

**A)** Сигналы `done` и `error` никогда не устанавливаются вместе.

**B)** Сигнал `error` устанавливается в правильный момент времени.

**C)** Сигнал `error` остается установленным до следующей установки `start`.

**9.4.** Для задачи 5.4 напишите следующие утверждения:

**A)** `c.Free` – первый сигнал, устанавливаемый после сброса (по сравнению с `p.put`).

**B)** Установка `c.Free` и `p.put` осуществляется в правильном порядке.

# Глава 10

# Функциональное покрытие

Тестовое окружение используется для генерации входных воздействий на тестируемую модель и проверки корректности ее выходных данных. Если система является сложной, оценить корректность ее поведения с помощью простого теста не получится. Необходимо проверять разные аспекты функционирования модели, да к тому же зачастую несколькими способами, потому что внутри модели могут быть взаимозависимости. Каждый отдельный аспект функционирования порождает цель в плане верификации. *Функциональное покрытие* измеряет, насколько полно цели верификации достигнуты тестовым окружением, и выражается в процентах от общего количества целей верификации. В процессе верификации его можно использовать, чтобы направить усилия на выявление еще не протестированных частей модели. SystemVerilog предоставляет средства для измерения функционального покрытия.

## 10.1. План верификации

Цель *плана верификации* – определить функциональность системы, принципиально важную для ее работы, а затем измерить, насколько полно она проверяется тестовым окружением. Для идентификации такой функциональности используется исходное техническое задание, а занимаются этим инженеры-верификаторы, так что эту деятельность можно отделить от разработки системы. На основе выявленной функциональности формулируются *цели верификации*.

Одной из целей верификации может быть измерение того, принимает ли некоторая переменная достаточно представительное подмножество возможных значений, чтобы соединенную с ней схему можно было считать хорошо протестированной. Например, принимает ли 5-битный вектор все 32 возможных значения? SystemVerilog предоставляет средства для подсчета того, сколько значений принимала переменная и сколько раз она принимала каждое значение. Если мы требуем, чтобы она принимала каждое из 32 значений хотя бы по разу, то уровень покрытия этой цели верификации можно выразить формулой:

$$\frac{\text{число разных значений, принятых переменной}}{32} * 100.$$

Если значение равно 100 %, мы говорим, что цель верификации *достигнута*.

Отсюда становится понятным, что подсчет значений – важная часть функциональной верификации. SystemVerilog предоставляет средства для задания того, что необходимо подсчитывать, с помощью чего это делать. В примере выше было бы 32 специальных накопителя, по одному для каждого возможного значения. Далее в этой главе мы расскажем о тех возможностях SystemVerilog, которые позволяют проводить функциональную верификацию.

## 10.2. Группы покрытия и точки покрытия

*Точка покрытия* (*coverpoint*) задает базовую цель верификации; это целочисленное выражение, значения которого подлежат учету. *Группа покрытия* (*covergroup*) объединяет множество точек покрытия и определяет, когда происходит сэмплирование значений. Группа покрытия задает цель верификации.

Приведем очень простой пример, пусть это будет меню вторых блюд.

```
1   enum {chicken, beef, pork, vegetarian} menuOption;
2
3   covergroup menu @(posedge clk);
4     entre: coverpoint menuOption;
5   endgroup
```

Здесь определена группа покрытия `menu`, сэмплирование значений для которой производится по переднему фронту тактового сигнала `clk`. Поскольку точка покрытия `menuOption` является переменной перечислимого типа, для нее создаются четыре *накопителя* (*bin*), по одному для каждого элемента этого типа. Накопитель содержит число, показывающее, сколько раз соответствующее значение встретилось при сэмплировании. Усложним пример, добавив в меню напитки.

```
1   enum {chicken, beef, pork, vegetarian} menuOption;
2   enum {tea, coffee, water} drinkOption;
3
4   covergroup menu @(posedge clk);
5     entre: coverpoint menuOption;
6     drink: coverpoint drinkOption
7     ExD: cross entre, drink;
8   endgroup
```

Здесь определены две категории подсчитываемых объектов (точек покрытия), а также *перекрестное покрытие* (*cross coverage*) объектов, попарно взятых из этих двух категорий (строка 7), которое служит для подсчета встретившихся комбинаций еды и напитков. В данном случае в перекрестном покрытии будет 4 × 3 накопителей, по одному для каждой пары, и мы сможем подсчитать, например, сколько раз подавали кофе с курицей. Хороший план верификации предполагает нахождение наблюдаемой системы во всех учитываемых состояниях: были заказаны все блюда, все напитки и все их комбинации. Если какая-то из этих ситуаций не произошла во время наблюдения, то не исключены скрытые дефекты.

Конечно, смысл плана верификации в SystemVerilog – не в том, чтобы следить за ресторанным меню! Идея в том, что все точки покрытия рассматриваются как состояния или значения в сложной системе.

Подводя итог, группа покрытия состоит из точек покрытия, перекрестных покрытий и общего для них механизма сэмплирования (в нашем примере – по переднему фронту clk). С точками покрытия и перекрестными покрытиями ассоциированы накопители, определяющие детализацию подсчета покрытия (в нашем примере накопители имеются для всех элементов перечислимых типов и для всех комбинаций блюд и напитков). В SystemVerilog есть и некоторые другие возможности учета покрытия, в т. ч. автоматическое создание накопителей. Также мы увидим, что у групп покрытия есть ряд настраиваемых параметров.

## 10.2.1. Иллюстрация

Вернемся к аппаратному потоку sumItUp из раздела 5.1, чтобы проиллюстрировать план верификации и его реализацию с помощью групп и точек покрытия. Вот как выглядит заголовок потока:

```
1  module sumItUp
2    (input logic ck, reset_l, go_l,
3     input logic [15:0] inA,
4     output logic done,
5     output logic [15:0] sum);
```

Поток ждет установки сигнала go_l. Как только это произойдет, на входе inA окажется первое из суммируемых значений. Сама сумма накапливается в регистре sum, который является также и выходом потока. Появление нуля на входе inA означает, что последовательность закончилась. Сигнал done показывает, что выход sum содержит сумму всех значений, переданных с момента установки go_l и до момента обнаружения нуля на входе.

Одна из целей верификации – сложить достаточно чисел, для того чтобы сумма принимала широкий диапазон значений. Ведь если мы вычислим только сумму 3 + 5 + 7 + 22, то не протестируем толком ни сумматор, ни регистр суммы, ни соединяющие их провода.

Чтобы отследить выполнение этой цели верификации, в примере 10.1 находится группа покрытия для потока sumItUp, названная sumItUp_g1. Учет значений в группе покрытия производится по переднему фронту сигнала ck. В группе определено четыре именованных накопителя для точки покрытия sum. В строках 4–9 используется ключевое слово bins для именования накопителей и указано, какие значения sum будут учитываться в каждом из них. Весь диапазон значений 16-битной суммы разбит на четыре равные части. Так, если

```
1  covergroup sumItUp @(posedge ck);
2   option.at_least = 15;
3   coverpoint sum
4     {
5     bins a0 = { [0: 16383]};
6     bins a1 = { [16384 : 32767]};
7     bins a2 = { [32768 : 49151]};
8     bins a3 = { [49152 : 65535]};
9     }
10 endgroup: sumItUp_g1
11
12 sumItUp sup = new;
```

**Пример 10.1.** Группа покрытия для потока sumItUp

бы в момент выборки значение sum было бы равно 50 000, то мы бы прибавили единицу к накопителю a3. Поскольку sum обновляется по каждому переднему фронту тактового сигнала, то в накопителях будут учитываться промежуточные суммы, вычисляемые в процессе получения всего пакета значений.

Группа покрытия определяет тип; так, в строке 12 создается новый объект sup типа sumItUp.

Для группы покрытия можно определить параметры, как показано в строке 2. Параметр at_least означает, что каждый накопитель должен инкрементировать значение не менее 15 раз, чтобы он считался покрытым.

Для проведения верификации нам необходимо тестовое окружение, которое будет посылать достаточное количество пакетов с достаточным количеством значений в каждом, чтобы покрыть диапазон значений 16-битной переменной. В примере 10.2 показана часть класса, решающего эту задачу. Код продолжается в примерах 10.3 и 10.4. В классе определена локальная неупакованная структура в строках 6–9, включающая пакет данных для отправки потоку и переменную howMany для указания количества значений в этом пакете. При вызове new для этого класса вызывается функция makePkt, которая использует системную функцию $urandom_range для генерации случайного беззнакового числа в диапазоне от MAXCOUNT до 2; это число записывается в поле pkt.howMany. Затем генерируется pkt.howMany случайных чисел в диапазоне от MAXSIZE до 1, и все они записываются в массив valuesToAdd. В процессе генерации вычисляется сумма чисел, чтобы впоследствии ее можно было сравнить с результатом, полученным моделью. Далее сгенерированный пакет необходимо отправить верифицируемому устройству.

В примере 10.3 показан метод класса task, который посылает пакет аппаратному потоку. Он содержит цикл for, в котором каждый элемент массива valuesToAdd устанавливается на вход потока. Одновременно с первым элементом (когда i=0) устанавливается сигнал go_l. После того как все значения переданы, на вход подается 0, что является признаком конца пакета (строка 30).

Последний шаг для начала процесса верификации: тестовое окружение должно создать экземпляр класса и вызывать его методы для создания пакетов и отправки их в модель. Это показано в примере 10.4. Сначала объявляется переменная t типа testSumThread. Затем в цикле while (строки 38–44) создается новый пакет (строка 39) и передается верифицируемому аппаратному потоку. Когда поток установит сигнал done, утверждение в тестовом окружении (assert в строке 42) проверит, что выход (sum) сформирован правильно, сравнив его с суммой, вычисленной в процессе генерации массива данных.

Цикл while продолжается, пока функция get_coverage возвращает значение, меньшее 100. get_coverage вычисляет число с плавающей точкой, означающее полное покрытие группы в процентном виде. В данном случае есть четыре накопителя, в каждом из которых счетчик должен быть равен как минимум 15. Поэтому если каждый из трех накопителей учел не менее 15 элементов, а последний только 14, то покрытие будет равно 75 %. Когда последний накопитель

учтет 15 элементов, покрытие станет равным 100 %. Цель верификации будет достигнута, и произойдет выход из цикла.

```
class testSumThread;
  localparam MAXSIZE = 65535,
             MAXCOUNT = 10;
  local bit unsigned [15:0] total;

  local struct {
    bit unsigned [15:0] valuesToAdd [MAXCOUNT];
    byte unsigned howMany;
  } pkt;

  function new();
    makePkt();
  endfunction

  local function makePkt;
    total = 0;
    pkt.howMany = $urandom_range(MAXCOUNT,2);
    for (byte i = 0; i < pkt.howMany; i++) begin
        pkt.valuesToAdd [i] = $urandom_range(MAXSIZE, 1);
        total += pkt.valuesToAdd[i];
    end
  endfunction
```

**Пример 10.2.** Класс для тестирования потока sumItUp

```
  task sendPktToAdd;
    for (byte i = 0; i < pkt.howMany; i++) begin
      @(posedge ck);
      inA <= pkt.valuesToAdd[i];
      go_l <= (i == 0) ? 0 : 1;
    end
    @(posedge ck);
    inA <= 0;
  endtask

  function checkTotal (bit unsigned [15:0] value);
    return value == total;
  endfunction
```

**Пример 10.3.** Процедура sendPktToAdd класса testSumThread

```
  initial begin // send packets until coverage hits 100%
    testSumThread t;
    while (sup.get_coverage() < 100.0) begin
      t = new;
      t.sendPktToThread();
      wait (done);
      ChkTotal: assert (t.checkTotal(sum))
        else $error("OOPS, At t=%4d, got wrong result!", $time);
```

```
44   end
45   #15 $finish;
46 end
```

**Пример 10.4.** Запуск тестового окружения

## 10.2.2. Параметры накопителей

Определение точек покрытия – ключевой элемент плана верификации, потому что именно они описывают, что необходимо отслеживать в процессе тестирования и как это делать. В предыдущем разделе мы разбили диапазон значений выхода sum на четыре части и складывали случайные числа до тех пор, пока каждый из четырех накопителей, сопоставленных частям диапазона, не инкрементировал свое значение хотя бы 15 раз.

```
1  covergroup sumItUp @(posedge ck);
2   option.at_least = 15;
3   coverpoint sum iff (done)
4     {
5     bins a0 = { [0: 16383]};
6     bins a1 = { [16384 : 32767]};
7     bins a2 = { [32768 : 49151]};
8     bins a3 = { [49152 : 65535]};
9     }
10 endgroup: sumItUp
```

**Пример 10.5.** Точка покрытия с охранным предикатом

Рассмотрим ситуацию, когда происходит учет только окончательной суммы, а все промежуточные в учете не участвуют. В спецификации потока указано, что при установке сигнала done на выходе находится окончательная сумма. Зададим точку покрытия с охранным предикатом в виде значения этого сигнала done (см. строку 3 примера 10.5). И учет значения sum будет происходить только тогда, когда done установлен. Еще одним подобным охранным предикатом точки покрытия часто является сигнал сброса.

Распределение учета значений sum по четырем накопителям необязательно гарантирует, что система хорошо протестирована. Быть может, лучше проверить, что в процессе работы каждый бит суммы был установлен, т. е. принимал значение 1. Альтернативная точка покрытия показана в примере 10.6. Здесь используется ключевое слово wildcard, указывающее на безразличие к значениям некоторых битов. В строке 5 описан накопитель b0, который учитывает равенство единице бита 0 суммы, а значения остальных битов sum им не отслеживаются. Если несколько битов суммы равны 1, то этот факт учитывается разными накопителями, соответствующими каждому биту в отдельности. Так, если сумма равна 3, то инкрементируются оба накопителя, b0 и b1.

Раз уж мы заговорили о проверке установленных битов, то почему не написать точку покрытия, учитывающую неустановленные биты? Или учитывающую бит, равный единице, но окруженный нулями с двух сторон. Или наоборот. Впрочем, чем более избирательны такие условия, тем ближе мы подходим к диагностике конкретной аппаратной реализации. Ведь, например, внутри системы сложение может производиться как побитовым образом, так и побайтовым. Если бы мы знали, как именно, то предложили бы другой набор точек покрытия. Напомним, однако, что план верификации разрабатывается по документации, а не по реализации. Фактическая внутренняя организация сумматора необязательно специфицирована.

```
1  covergroup sumItUp @(posedge ck);
2   option.at_least = 15;
3   bitView: coverpoint sum
4     {
5     wildcard bins b0 = { 16'b????_????_????_???1 };
6     wildcard bins b1 = { 16'b????_????_????_??1? };
7     wildcard bins b2 = { 16'b????_????_????_?1?? };
8     wildcard bins b3 = { 16'b????_????_????_1??? };
9     wildcard bins b4 = { 16'b????_????_???1_???? };
10    wildcard bins b5 = { 16'b????_????_??1?_???? };
11    wildcard bins b6 = { 16'b????_????_?1??_???? };
12    wildcard bins b7 = { 16'b????_????_1???_???? };
13    wildcard bins b8 = { 16'b????_???1_????_???? };
14    wildcard bins b9 = { 16'b????_??1?_????_???? };
15    wildcard bins b10 = { 16'b????_?1??_????_???? };
16    wildcard bins b11 = { 16'b????_1???_????_???? };
17    wildcard bins b12 = { 16'b???1_????_????_???? };
18    wildcard bins b13 = { 16'b??1?_????_????_???? };
19    wildcard bins b14 = { 16'b?1??_????_????_???? };
20    wildcard bins b15 = { 16'b1???_????_????_???? };
21    }
22 endgroup: sumItUp
```

**Пример 10.6.** Точка покрытия для учета установленных битов

Если точка покрытия не определяет число накопителей или диапазоны значений, учитываемые каждым накопителем, SystemVerilog делает это автоматически. Для переменной перечислимого типа создается по накопителю для каждого элемента перечисления. Число накопителей для M-битной переменной равно минимуму из двух чисел: $2^M$ и значения параметра `auto_bin_max` (по умолчанию 64). Этот параметр можно определить следующим образом:

```
option.auto_bin_max = 10;
```

Значение параметра распространяется на всю группу покрытия. Пример определения параметра показан в строке 2.

## 10.2.3. Активация групп покрытия в утверждениях

До сих пор мы не обращали внимания на то, правильно подсчитана сумма или нет. Несмотря на то что в строке 42 примера 10.4 это проверяется, группы покрытия в примерах 10.1, 10.5 и 10.6 срабатывают и при неправильных значениях `sum`. Мы хотели бы учитывать в покрытии только правильные значения.

Предыдущие группы покрытия сэмплировали значение `sum` по переднему фронту тактового сигнала или, как в примере 10.5, по факту установки сигнала `done`. Есть еще один способ активировать (trigger) группу покрытия – переопределить ее функцию сэмплирования. Эта функция может вызываться, например, из утверждения, которое является заключительным, или из оператора cover.

Взгляните на изменения, сделанные в примере 10.7. В строках 1–2 показано, как указать функцию, используемую для учета значений точек покрытия в группе покрытия (здесь with – ключевое слово, а sample – нет). Эти строки можно подставить вместо строки 1 примера 10.6. Теперь группа покрытия активируется только при явном вызове функции sample. В данном примере функции sample передается одно значение, но можно передать и несколько.

```
1  covergroup Cover_sumItUp with function
2   sample(logic[15:0] sum);
3  ...
4  property checkSumWithTotal;
5   bit [15:0] myTotal;
6   @(posedge ck) (~go_l, myTotal = inA) |=>
7       (~done, myTotal += inA) [*1:10] ##1
8       (done && (sum == myTotal), sup.sample(sum));
9  endproperty
10
11 SumIncorrect: assert property (checkSumWithTotal)
12    else $error("Sum value incorrect");
```

**Пример 10.7.** Переопределение функции сэмплирования

Функция сэмплирования может быть вызвана из свойства (строки 4–9), которое можно было бы включить в предыдущий пример для проверки правильности суммы. В нем ожидается установка сигнала go_l, после чего значение входа inA копируется в локальную переменную myTotal. Далее по каждому фронту тактового сигнала, при условии что сигнал done сброшен, значение inA прибавляется к myTotal. Свойство считается выполненным, когда установлен сигнал done и при этом значение выхода sum равно значению локальной переменной myTotal. В этот момент вызывается функция сэмплирования, которой передается окончательное значение sum (строка 8). Таким образом, группа покрытия учитывает только правильные значения.

Утверждения работают со значениями, полученными на этапе предварительной обработки, однако функцию сэмплирования можно вызывать с любыми значениями на любом этапе работы симулятора.

Можно вызвать функцию сэмплирования не из свойства, а из оператора cover, как показано в примере 10.8. От примера 10.7 он отличается отсутствием вызова sample в свойстве (строка 5), а также наличием оператора cover (строки 11–12). Если свойство выполняется, оператор вызывает функцию сэмплирования, передавая ей подлежащее учету значение. Отметим, что оператор cover должен передавать значения с помощью конструкции $sampled(sum). Дело в том, что в нем, в отличие от оператора assert, не используются значения, сэмплированные на этапе предварительной обработки. В нашем примере оператор cover исполняется на этапе реагирования, поэтому результаты присваиваний уже известны. Еще раз взглянув на поток sumItUp, мы увидим: если значение входа inA равно 0, то устанавливается сигнал done и по следующему фронту так-

тового сигнала в sum записывается 0. Таким образом, если не указать $sampled, в функцию сэмплирования будет передаваться значение 0.

```
1  property checkSumWithTotal;
2    bit [15:0] myTotal;
3    @(posedge ck) (~go_l, myTotal = inA) |=>
4    ~done, myTotal += inA) [*1:10] ##1
5      (done && (sum == myTotal));
6  endproperty
7
8  SumIncorrect: assert property (checkSumWithTotal)
9      else $error("Sum value incorrect");
10
11 Cover_Correct_sum_Values: cover property (checkSumWithTotal)
12     sup.sample($sampled(sum));
```

**Пример 10.8.** Вызов функции сэмплирования из оператора cover

# 10.3. Покрытие переходов и перекрестное покрытие

В предыдущих примерах было показано, как осуществляется сбор покрытия для отдельных значений. Определив специальные накопители, можно собирать покрытие и для последовательностей значений.

## 10.3.1. Покрытие переходов

Покрытие переходов мы проиллюстрируем на примере конечного автомата, показанного на рис. 10.1. Для простоты здесь отображены только состояния и переходы (без детализации значений входов и выходов). Код на SystemVerilog, определяющий следующее состояние автомата и вычисляющий значения выходов, приведен в примере 10.9 (строки 14–21 и 23–24). Этот пример абсолютно искусственный, не имеющий отношения к реальности.

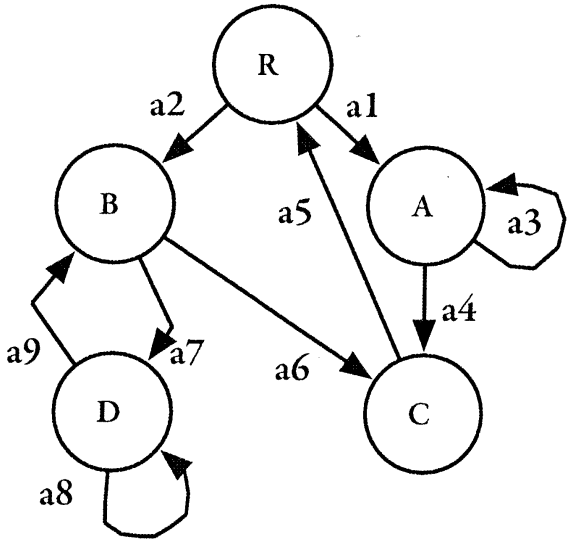



**Рис. 10.1.** Диаграмма состояний конечного автомата

```
1  typedef enum bit[3:0] {R, A, B, C, D} state_t;
2
3  module fsm
4   (output state_t st,
5    output bit out1, out2,
6    input bit in1, in2, ck, r);
7
8   state_t ns;
9
10  always_ff @(posedge ck, posedge r)
11    if (r) st <= R;
12    else s t <= ns;
13
14  always_comb
15    unique case (st)
16      R: ns = in1 ? A : B;
17      A: ns = in2 ? A : C;
18      B: ns = in1 & in2 ? C : D;
19      C: ns = R;
20      D: ns = in1 ? D : B;
21    endcase
22
23  assign out1 = in1 & (st == A || st == B),
24         out2 = st == D;
25 endmodule: fsm
26
27 module top;
28  state_t st;
29  bit out1, out2;
30  bit in1, in2, ck, r;
31
32  initial begin
33    ck = 0;
34    repeat (600) #5 ck = ~ck;
35    $finish;
36  end
37  fsm f (.*);
38  tb t (.*);
39 endmodule: top
```

**Пример 10.9.** Пример покрытия конечного автомата

В модуле верхнего уровня top (строки 27–39) создается экземпляр модуля fsm и тестового окружения (tb). Нотация .* означает, что имена всех переменных попарно совпадают. Тактовый генератор отработает 300 тактов, затем остановится.

## 10.3.2. Накопители переходов

В примере 10.10 приведена программа и определение группы покрытия для конечного автомата и его переходов. Обратите внимание на нечто новое, на накопители переходов в строках 49–59. Точка покрытия определяется для

переменной состояния st. Накопитель a1 соответствует переходу из состояния R в состояние A. Таким образом, всякий раз, как st переходит из R в A, счетчик в накопителе a1 увеличивается на 1. В строке 48 сказано, что каждый накопитель должен инкрементировать свой счетчик не менее двух раз. Отметим, что всем переходам на диаграмме сопоставлен накопитель, причем их имена совпадают с метками на рис. 10.1.

Могут быть описаны и более длинные последовательности. Так, переходы по дугам a1, a3 и a4 можно задать так:

```
bins X = (R => A => A => C);
```

В этом случае накопитель X увеличивает значение своего счетчика, только если имела место именно такая последовательность переходов. Если бы были заданы также накопители a1, a3 и a4, то значения счетчиков в них тоже были бы увеличены на единицу.

Ту же последовательность можно записать иначе:

```
bins X = (R => A [* 2] => C);
```

Здесь показано, что оператор повторения, применяемый для задания последовательностей в утверждениях, применим и в этом контексте. В данном случае специфицируется нахождение системы два раза подряд в состоянии A. Разрешено также задавать диапазон повторений – использование конструкции [* 2:4] означало бы, что последовательность переходов была бы учтена, если бы система находилась в состоянии A от 2 до 4 раз подряд. Еще разрешено указывать непоследовательные повторения: точное количество ([= 4]) и диапазоны ([= 4:6]), а также повторения типа goto ([-> 5]) и диапазоны таких повторений. Определения этих типов повторений см. в разделах 9.3.3 и 9.3.4.

В примере 10.10 показано определение класса для 2-битного случайного числа, InputTwiddle. Переменная, принимающая случайное значение, называется ins.

В примере 10.11 показана последняя часть тестового окружения. После создания нового объекта InputTwiddle (i) и завершения последовательности сброса тестовое окружение входит в цикл while, исполняемый, пока измеренное покрытие составляет меньше 100 %. То есть пока покрытие меньше 100 %, будет вызываться метод randomize объекта i, что приведет к присваиванию переменной ins случайного значения. В операторе assert про-

```
40 program tb
41   (input state_t st,
42    input bit out1, out2, ck,
43    output bit in1, in2, r);
44
45   bit [1:0] invals;
46
47   covergroup fsm @(posedge ck);
48     option.at_least = 2;
49     coverpoint st {
50       bins a1 = (R => A);
51       bins a2 = (R => B);
52       bins a3 = (A => A);
53       bins a4 = (A => C);
54       bins a5 = (C => R);
55       bins a6 = (B => C);
56       bins a7 = (B => D);
57       bins a8 = (D => D);
58       bins a9 = (D => B);
59     }
60   endgroup
61
62   fsm fcover = new;
63
64   class InputTwiddle;
65     rand bit[1:0] ins;
66   endclass
```

**Пример 10.10.** Точка покрытия для переходов

```
67  initial begin
68    InputTwiddle i = new;
69    r = 0;
70    #1 r = 1;
71    #1 r = 0;
72
73    while (fcover.get_coverage() < 100)
74      begin
75        assert (i.randomize());
76        {in1, in2} <= i.ins;
77        invals <= i.ins;
78        @(posedge ck);
79      end
80    end
81 endprogram
```

**Пример 10.11.** Тестовое окружение для покрытия накопителей переходов

веряется успешность рандомизации – если бы на случайное значение были наложены ограничения (сейчас их нет), то этот оператор сообщил бы, удовлетворяет значение им или нет. Затем значение `i.ins` записывается в два входных бита, и мы ждем следующего фронта тактового сигнала. Присваивается также значение переменной `invals`; зачем это нужно, мы объясним в следующем примере. Эти входные значения будут определять переход состояния по следующему фронту сигнала. Когда цикл завершится, покрытие составит 100 %, и достижение конца блока program приведет к окончанию симуляции.

Смысл этой программы (примеры 10.9–10.11) состоит в том, что тестовое окружение случайным образом переводит конечный автомат из одного состояния в другое, исходя из значений `in1` и `in2`. Рано или поздно будут покрыты все переходы, хотя для достижения некоторых «удаленных» состояний, например `a8`, системе может потребоваться значительное время. Если симуляция не укладывается в 300 тактов, то придется увеличить их число при настройке тактового генератора.

Отметим, что в этом примере не проверяется правильность реализации. Проверяется только, что каждый переход пройден хотя бы дважды.

## 10.3.3. Накопители для перекрестного покрытия

Существует еще один режим работы накопителей в группе покрытия, позволяющий задать *перекрестное* покрытие, как это показано в строке 4 примера 10.12. Оператор cross задает отслеживание декартова произведения значений либо двух переменных, либо, как в данном случае, переменной `invals` и точки покрытия `st`.

```
1  covergroup fsmCross @(posedge ck);
2    option.at_least = 2;
3    coverpoint st;
4    cross invals, st;
5  endgroup
6
7  fsmCross crossCover = new;
```

**Пример 10.12.** Группа покрытия с оператором cross

Идея перекрестного покрытия – подсчитать, сколько раз каждое значение одной из двух переменных комбинируется с каждым значением другой переменной. Поскольку в примере 10.12 одна из переменных – это входной вектор `invals`, а другая – переменная состояния `st`, то перекрестное покрытие подсчитывает, сколько раз каждая возможная комбинация входов встречается в каждом состоянии. Зная, что система побывала в каждом состоянии при любом допустимом входном векторе, мы можем быть уверены, что покрыта вся функциональность конечного автомата.

Чтобы использовать группу покрытия из примера 10.12 с моделью из примера 10.9, мы модифицируем программу testbench из примера 10.10, заменив группу покрытия `fsm` на `fsmCross`, а условие в цикле `while` из примера 10.11 (строка 73) – условием `(crossCover.get_coverage < 100)`.

Однако об этом примере стоит сказать еще несколько слов. В предыдущих примерах с помощью накопителей определялось осуществление подсчета отдельных значений или переходов между ними переменной точки покрытия. В данном случае никакие накопители явным образом не заданы, поэтому SystemVerilog определяет их самостоятельно. Поскольку `st` – это переменная перечислимого типа, накопитель для каждого элемента перечисления получается автоматом. Эти накопители показаны в верхней части рис. 10.2.

Перекрестное покрытие можно определить для двух переменных или для переменной и точки покрытия. В строке 4 мы видим точку покрытия и переменную. В таком случае SystemVerilog неявно определяет точку покрытия для 2-битной переменной `invals` и автоматически генерирует четыре накопителя, по одному для каждого ее значения (см. рис. 10.2). Поскольку мы задали перекрестное покрытие, то автоматически генерируется еще один набор накопителей, массив размером 5×4, для пяти значений `st` и четырех значений `invals`. В результате у нас имеется три набора накопителей: для подсчета значений `st`, `invals` и для перекрестного произведения `invals` и `st`; все они показаны на рис. 10.2. В строке 2 говорится, что покрытие считается достигнутым, если значение в каждом накопителе не меньше 2.

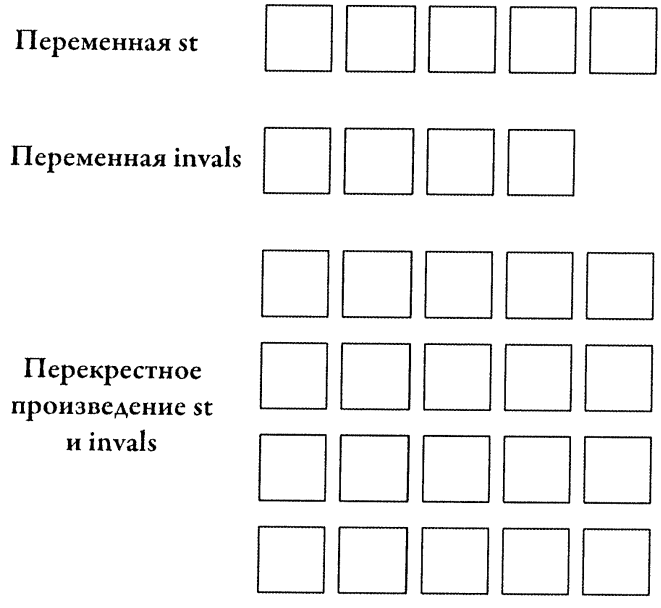



**Рис. 10.2.** Автоматически сгенерированные накопители для примера 10.12

На рис. 10.3 показано, как покрытие группы `fsmCross` возрастает по мере увеличения количества векторов, поданных на вход модели в цикле `while` из примера 10.11. В данном случае для достижения покрытия 100 % потребуется больше 200 тестовых векторов.

## 10.4. Вычисление уровня покрытия

В простой ситуации из примера 10.12, накопители для которого показаны на рис. 10.2, значение, возвращаемое функцией `crossCover.get_coverage()`, вычисляется на основе информации о покрытых накопителях (не менее двух инкрементов) трех точек покрытия.

Всего получается 29 накопителей, значение каждого из которых должно быть не менее двух. Однако не все накопители имеют одинаковый вес. Одинаково

учитываются три точки покрытия, на каждую из них приходится по 1/3 общего покрытия. Если один из пяти накопителей для st покрыт, а остальные нет, то get_coverage() вернет 1/3 * 1/5 = 0.0667 (6.67 %). Если при этом покрыт один из четырех накопителей для invals, то общее покрытие возрастет на 1/3 * 1/4 = 0.0834 (8.34 %), и get_coverage() вернет сумму обеих величин: 0.15, или 15 %. Каждый накопитель в перекрестном произведении увеличивает общее покрытие на 1/3 * 1/20, или 0.016 (1.6 %).


**Рис. 10.3.** Зависимость покрытия группы fsmCross от количества случайных входных векторов

Общая формула для покрытия группы $C_g$ имеет вид:

$$C_g = \frac{\sum_i W_i \times C_i}{\sum_i W_i}.$$

Здесь $C_i$ – покрытие каждой из точек покрытия, т. е. количество покрытых накопителей, поделенное на их общее число в точке покрытия. $W_i$ – вес точки покрытия, по умолчанию равный 1. Эта формула резюмирует приведенное выше обсуждение.

Значение W можно задать в определении точки покрытия или перекрестного произведения. Оно указывается в фигурных скобках, как показано ниже:

```
coverpoint someVar
{ option.weight = 2; // должно быть неотрицательным целым числом
...
}
```

Рассмотрим диаграмму состояний на рис. 10.1 и работающий с ней код (определение группы покрытия в примере 10.10). В группе покрытия определены все переходы, допустимые в автомате `fsm`. Но что, если происходит недопустимый переход, например C => B. Его негде учесть, поскольку для него нет накопителя. С другой стороны, можно определить накопитель для недопустимых значений следующим образом:

```
illegal_bins bad_transition = {C => D};
```

Если бы этот код присутствовал внутри фигурных скобок в определении точки покрытия `st`, то появление такой последовательности привело бы к ошибке и прекращению симуляции. Конечно, можно было бы написать другой код тестового окружения и утверждения для отлавливания таких недопустимых переходов.

Накопители для недопустимых значений можно определять и в точках покрытия переменных. В следующей точке покрытия значения 22 и 25 объявлены недопустимыми:

```
coverpoint q
{
  illegal_bins bad_dog = {22, 25};
}
```

Если бы `q` приняло значение 22 или 25, то произошла бы остановка симуляции. В обоих случаях накопители для недопустимых значений или переходов не были бы учтены при вычислении покрытия.

Накопители можно также игнорировать. В этом случае нам не важно, было ли принято указанное значение или совершен указанный переход.

```
coverpoint r
{
  ignore_bins ignoredValues = {55, 57};
  ignore_bins ignoredTrans = {17 => 22};
}
```

Такие накопители не учитываются при вычислении покрытия.

## 10.5. Задачи и упражнения

**10.1.** Для задачи 7.4 напишите тестовое окружение, в котором используются накопители переходов и которое продолжает работать, пока каждый переход не будет совершен хотя бы два раза.

**10.2.** Сделайте то же самое для задачи 7.5.

**10.3.** Для задачи 7.6 используйте функциональное покрытие, чтобы убедиться, что потоку подавался на вход широкий диапазон значений.

**10.4.** В задаче 10.3 добавьте утверждения по типу тех, что встречаются в задаче 9.1. Утверждения должны фигурировать в вычислении покрытия.

# Часть IV

# ДЕТАЛИ, ДЕТАЛИ, ДЕТАЛИ

# Глава 11

# Процедурные модели

В этой главе мы обсудим использование процедурных моделей для описания синтезируемых систем и тестовых окружений для них. В *процедурных* моделях используются операторы, похожие на встречающиеся в языках программирования, в т. ч. оператор `if-then-else`, который служит для исполнения одной из двух ветвей вычислений в зависимости от значения выражения, а также оператор присваивания, который вычисляет значение выражения в правой части и присваивает его переменной в левой части. Процедурным моделям свойственна последовательность в вычислениях: операторы SystemVerilog в таких моделях исполняются друг за другом.

Процедурные модели бывают функциональными и поведенческими. Основное различие состоит в том, что для *функциональных* моделей не задаются никакие временны́е характеристики, а для *поведенческих* – задаются. Первое определение вытекает из математического определения функции, когда результат (выход функции) вычисляется на основе одних лишь входов, т. е. функция задает отображение входов в выход. Поведенческая модель допускает задание временны́х характеристик, тем самым определяя последовательность действий во времени.

Процедурные модели используются в SystemVerilog с двумя разными целями: для описания синтезируемых схем и для описания тестовых окружений. Как правило, синтезируемые описания являются чисто функциональными – время в них не задается. Временны́е характеристики определяются инструментом синтеза.

Однако в тестовом окружении действия часто описываются с помощью неявно заданного конечного автомата, как было показано в разделе 7.3.3. Поскольку они обычно включают операторы, ожидающие фронта тактового сигнала, то исполнение описания подразумевает течение времени. А это уже поведенческая модель.

## 11.1. Операторы процессов

Процедурные модели используются только в связке с так называемыми операторами *процессов*: непрерывным присваиванием, `always`, `always_comb`, `always_ff`, `always_latch`, `initial`, `final`. Все эти операторы являются параллельными, что

означает, что вычисления, описанные в их процедурных моделях, исполняются параллельно со всеми остальными операторами процессов.

При описании комбинационных схем часто используется оператор always_comb:

```
1   bit [9:0] a, b, c;
2   always_comb a = b + c;
```

Здесь a, b и c – 10-битные векторы. При любом изменении значения b или c значение переменной a вычисляется заново. Если необходимо описать более сложное поведение, можно использовать операторные скобки begin-end (как в языках программирования). Синтаксис оператора always_comb следующий:

```
always_comb <statement>
```

**Таблица 11.1.** Операторы процессов

| Название | Применение | Примечания |
|---|---|---|
| always_comb | Используется для описания комбинационных схем | Для получения комбинационной схемы требуется, чтобы выходу присваивалось значение при каждом исполнении оператора always_comb |
| always_ff | Используется для описания последовательных схем, срабатывающих по фронту сигнала | Тактовый сигнал является частью модели. Любая скалярная переменная в левой части неблокирующего присваивания (<=) будет триггером. Векторные переменные станут регистрами |
| always_latch | Используется для описания последовательных схем, срабатывающих по уровню сигнала | Управляющий сигнал является частью модели |
| initial | Операторы initial не используются для описания схем, но используются в тестовом окружении, где служат в том числе для задания начальных значений | Оператор initial исполняется один раз и заканчивается вместе с последним находящимся в нем процедурным оператором |
| always | Унаследован от языка Verilog и в целом заменен предыдущими тремя видами операторов always | Иногда используется в тестовых окружениях |
| assign | Используется для описания комбинационных схем. Описывает работу только одного оператора. Скобки begin-end недопустимы | Хотя разрешено описать работу только одного оператора, assign может вызывать функцию, включающую множество операторов |

где <statement> может быть, как в примере выше, a = b + c. Операторные скобки begin-end позволяют задать использование сразу нескольких операторов. Рассмотрим, к примеру, комбинационную схему, которая умеет выполнять сложение и вычитание:

```
1   bit [9:0] sum, dif, b, c;
2   always_comb begin
3     sum = b + c;
4     diff = b - c;
5   end
```

В этом примере, если b или c изменят свое значение, произойдет исполнение оператора always_comb и вычисление новых значений sum и diff. Поскольку для такого вычисления необходимы два оператора, для объединения их в один составной оператор используются скобки begin-end. Все как в обычных программах.

Этот базовый синтаксис применим ко всем операторам процессов, кроме непрерывного присваивания.

В табл. 11.1 сведены все операторы процессов и указания по их использованию в моделях цифровых систем.

Оператор final не является частью проектируемой системы. Он вызывается в конце симуляции для вывода отчета. Если операторов final несколько, порядок их исполнения заранее не определен.

В начале симуляции все переменные имеют значения по умолчанию (x для переменных типа logic, 0 для переменных типа bit). После этого разрешается исполнение всех операторов initial и always, причем порядок их исполнения заранее не определен. Затем однократно активируются все операторы always_comb, always_ff и always_latch, также в произвольном порядке. Таким образом, выходы операторов always_comb, always_ff и always_latch согласованы с входами, которые были инициализированы в операторах initial и always.

## 11.1.1. Оператор initial

Все операторы initial, определенные в модели, запускаются в момент времени 0, однократно исполняя содержащийся в них оператор (или несколько операторов в скобках begin-end). Затем они завершают свою работу, после чего никогда не будут запущены вновь. Они применяются для инициализации переменных перед симуляцией. Например, следующий оператор initial инициализирует все элементы массива регистров, присваивая им значение 1.

```
1   bit [11:0] mem [32];
2   initial
3     for (int addr = 0; addr < 32; addr++)
4       mem[addr] = 1;
```

В строке 1 определен 12-битный массив mem из 32 элементов. Инициализация всех элементов этого массива осуществляется оператором initial с помощью цикла for. Обратите внимание, что переменная индекса цикла (в данном случае addr) может быть объявлена прямо в заголовке цикла for (см. строку 3) и в данном случае имеет тип int. Область видимости этой переменной ограничена циклом. Сам цикл for исполняется точно так же, как в языках программирования: переменная addr принимает все значения от 0 до 31 включительно, и на каждой итерации элементу mem[addr] присваивается значение 1. Скобки begin-end не нужны, потому что цикл for – это один оператор. По завершении цикла оператор initial становится неактивным.

Хотя, как уже было сказано, оператор initial начинает свою работу в момент времени 0, он может не успеть завершить ее в течение одной единицы време-

ни. Например, следующее определение значения сигнала clock никогда не станет неактивным, потому что оператор внутри initial вообще не завершается:

```
1   bit clock;
2   initial
3     forever #5 clock = ~clock;
```

В этом примере clock сначала инициализируется значением 0, потому что любая переменная типа bit по умолчанию равна 0. Цикл forever в строке 3 откладывает исполнение оператора на 5 единиц времени, затем заменяет clock противоположным значением и повторяет эти операции бесконечно. Поскольку forever означает «вечно» (даже в виртуальном времени процесса симуляции), цикл никогда не завершится, поэтому оператор initial всегда будет активен. Ничего страшного, просто возьмите на заметку.

## 11.1.2. Операторы always_comb и always_latch

Схемы цифровой аппаратуры бывают трех основных типов:

- *комбинационные схемы* – когда значения выходов схемы задаются булевыми функциями, зависящими от значений входов;
- *последовательные схемы со срабатыванием по фронту сигнала* – когда действия производятся только в том случае, когда изменяется значение одного из входов: увеличивается (передний фронт) или уменьшается (задний фронт);
- *последовательные схемы со срабатыванием по уровню сигнала* – когда действия производится только в том случае, когда один из входов имеет определенное значение (уровень).

Эти три типа схем описываются соответственно операторами always_comb, always_ff и always_latch. В этом разделе будут рассмотрены только операторы always_comb и always_latch[62].

Примеры использования оператора always_comb были приведены в разделе 2.2, а также выше в этом разделе. Оператор always_comb – это фактически цикл, в котором неявным образом ожидается изменение значения любой встречающейся в нем переменной. Рассмотрим функциональность сумматора-вычитателя – схемы, которая производит сложение или вычитание в зависимости от значения входной переменной addSub.

```
1   bit[9:0] result, b, c;
2   bit addSub;
3   always_comb
4     if (addSub) result = b + c;
5     else result = b - c;
```

Для данного оператора always_comb (см. строку 3) неявным образом задано ожидание изменения значения любой переменной в правой части операторов присваивания (в данном случае b и c) и любой переменной в условиях (в дан-

[62] Оператор always_ff рассматривается в следующем разделе.

ном случае addSub). Таким образом, как только одна или несколько из этих переменных изменит свое значение, оператор always_comb исполнится, вычислит новое значение result и будет ждать следующего изменения. Если addSub равно TRUE, то result будет равно b+c, иначе b-c. Заметим, что поскольку if-else – это один оператор, скобки begin-end внутри always_comb не нужны.

Сделаем четыре важных замечания об операторе always_comb.

- Предполагается, что список чувствительности оператора always_comb содержит все переменные, которые находятся в правых частях операторов присваивания и в условиях. В список чувствительности входят также переменные, которые передаются вызываемым функциям в качестве аргументов, но не входят переменные, объявленные внутри оператора и внутри вызываемых функций. Не учитываются также вызовы методов объектов классов.
- Переменные, которым присваиваются значения в операторе always_comb (например, его выходы), не могут изменяться другими операторами процессов. То есть функция, определяющая значение переменной, должна быть задана в рамках одного always_comb. Обнаружив потребность написать еще один оператор, меняющий значение той же самой переменной, хлопните себя линейкой по рукам. Расширьте уже написанный оператор.
- Оператор always_comb активируется симулятором один раз в момент времени 0, после того как всем операторам initial и always дано разрешение на исполнение. Таким образом, эти операторы могут установить начальные значения переменных до исполнения оператора always_comb, так что в момент 0 на всех линиях будут согласованные значения.
- Оператор always_comb не должен содержать операторов управления временем (#, @, wait).

Операторы always_latch применяются для моделирования поведения защелок. Рассмотрим для примера следующую триггерную защелку:

```
1   bit q, d, gate;
2   always_latch
3     if (gate) q <= d;
```

Как и always_comb, оператор always_latch автоматически определяет переменные, на изменение которых ему необходимо реагировать. Если хотя бы одна из этих переменных изменяет свое значение, то цикл исполняется, в результате чего может измениться выход (q). Оператор always_latch аналогичен оператору always_comb, за исключением того, что сообщает инструменту синтеза намерение разработчика использовать защелки.

В приведенном выше примере защелка требуется, когда значение переменной gate равно FALSE. Если значение gate равно TRUE, то при изменении d оператор исполнится, и значение q будет обновлено. Если значение gate равно FALSE, то при изменении d оператор исполнится, но значение переменной q не изменится, соответственно, оно должно быть сохранено с момента предыдущего исполнения цикла. Это поведение защелки со срабатыванием по уровню сигнала. Инструмент синтеза сгенерирует защелку, реализующую такое поведение.

Все четыре замечания по поводу операторов always_comb относятся также к операторам always_latch.

### 11.1.3. Оператор always_ff

Операторы always_ff применяются для проектирования схем, работающих по фронту сигнала. Они широко используются для описания функциональности триггеров и регистров (см. раздел 3.1). Базовый триггер с асинхронным сбросом можно описать следующим образом:

```
1   bit d, q, clk, reset;
2   always_ff @(posedge clk, posedge reset)
3     if (reset) q <= 0;
4     else q <= d;
```

При установке сигнала reset, т. е. по его переднему фронту, в запоминающий элемент q будет загружено значение 0 (которое будет оставаться таким до тех пор, пока reset установлен). Если же reset сброшен, то по переднему фронту тактового сигнала в q будет загружено значение входа d.

Это можно записать и в таком виде:

```
1   bit d, q, clk, reset;
2   always_ff @(posedge clk iff ~reset, posedge reset)
3     q <= (reset) ? 0 : d;
```

Здесь выражение iff означает, что цикл always_ff будет исполняться по переднему фронту тактового сигнала тогда и только тогда, когда reset сброшен. Оба примера функционально эквивалентны.

Сделаем четыре важных замечания об операторах always_ff.

- Для обновления переменных состояния используется неблокирующее присваивание. Любая переменная в левой части неблокирующего присваивания будет храниться в триггере или в регистре, входами которых являются тактовый сигнал, сигнал сброса, а также данные (d).
- В операторе always_ff разрешается использовать блокирующее присваивание (=). Однако обычно оно применяется для вычисления промежуточных значений в процессе вычисления нового состояния (см. раздел 3.2.3). Переменные, которым значение присвоено таким образом, не должны использоваться вне always_ff.
- В always_ff можно использовать оператор @ (в самом начале). Никаких других операторов управления временем (#, wait) в нем быть не должно.
- Переменные, находящиеся в левой части операторов присваивания, не должны модифицироваться никаким другим процессом.

### 11.1.4. Оператор непрерывного присваивания (assign)

Оператор непрерывного присваивания можно интерпретировать как упрощенный always_comb. Его типичное использование рассмотрим на примере следующего 6-битного сумматора.

```
1   bit [5:0] sum, a, b;
2   assign sum = a + b;
```

При любом изменении значения a или b (или обоих значений) будет исполнен оператор assign, вычисляющий сумму.

Непрерывное присваивание отличается от always_comb только в одном отношении: операторные скобки begin-end не допускаются. Однако в правой части присваивания может находиться вызов функции, осуществляющей сложные вычисления.

Оператор непрерывного присваивания используется также для формирования соединений в цепях (см. раздел 2.7).

### 11.1.5. Оператор final

Оператор final исполняется в конце симуляции. Обычно в нем выводится на консоль статистика, собранная тестовым окружением.

```
1   final
2   $display("Simulation ended after %d instructions were executed.",
          numInstructions);
```

Операторов final может быть несколько, порядок их исполнения заранее не определен. Операторы final исполняются, когда список событий симуляции оказывается пустым, или в результате вызова системной процедуры $finish.

Все операторы final исполняются за нулевое время, поэтому не могут включать операторов управления временем (#, @ и wait), а также планировать события.

## 11.2. Оператор if-else и условная операция

Условные операторы используются в процедурном описании, чтобы влиять на поток управления. Примером является оператор if и его разновидности. Другой оператор, влияющий на поток управления, case, реализует многовариантное ветвление и представлен в разделе 11.3. Синтаксис операторов if в SystemVerilog похож на синтаксис аналогичных операторов в языках программирования. Однако есть и отличия: работа с 4-значной логикой и ключевые слова (например, unique), используемые инструментами симуляции и синтеза.

Типичное использование оператора if рассмотрим на примере простого мультиплексора:

```
1   bit mux_sel, muxOut, b, a;
2   if (mux_sel)
3     muxOut = b;
4   else muxOut = a;
```

В данном примере, если mux_sel равно TRUE, muxOut будет равно b, в противном случае (else) – a. Как и в языках программирования, и ветвь then (строка 3),

и ветвь else (строка 4) могут быть составными операторами. В таком случае для группировки нескольких операторов в один составной их следует заключать в операторные скобки begin-end.

Как и в языках программирования, часть else необязательна. Приведенный выше пример можно переписать следующим образом, полностью сохранив его функциональность:

```
1   bit mux_sel, muxOut, b, a;
2   muxOut = a;
3   if (mux_sel)
4       muxOut = b;
```

Наконец, существует условная операция, которую можно использовать вместо оператора if-else, когда в обеих ветвях значение присваивается одной и той же переменной. Приведенный выше пример можно записать и без оператора if-else:

```
1   bit mux_sel, muxOut, b, a;
2   muxOut = (mux_sel) ? b : a;
```

Если значение mux_sel равно TRUE, то переменной muxOut присваивается значение b, иначе значение a.

## 11.2.1. Логические выражения (условия)

Условие – это выражение в скобках после ключевого слова if или выражение перед знаком ? условной операции. Простейшие условия имеют вид переменной, сравнения двух переменных или сравнения переменной и константы.

Вернемся к примеру мультиплексора. На этот раз объявим переменные типа logic:

```
1   logic mux_sel, muxOut, b, a;
2   if (mux_sel)
3       muxOut = b;
4   else muxOut = a;
```

В этом примере mux_sel может принимать значения 0, 1, x или z. Для симулятора 1 это TRUE, 0 – FALSE, x и z – неизвестные значения (что в операторе if интерпретируется как FALSE). Таким образом, если значение mux_sel равно x, то переменной muxOut будет присвоено значение a.

Рассмотренный пример может быть немного изменен следующим образом (это не влияет на его функциональность):

```
1   logic mux_sel, muxOut, b, a;
2   if (mux_sel == 1)
3       muxOut = b;
4   else muxOut = a;
```

Разница между этим и предыдущим примерами – в строке 2. Здесь операция проверки равенства (==) сравнивает значение переменной mux_sel с константой 1. Если они равны, то значением выражения будет TRUE. Если же значение mux_sel равно 0, x или z, то значением выражения будет FALSE.

В качестве операции сравнения в строке 2 можно указать любую операцию из множества {==, !=, <, <=, >, >=}: «равно», «не равно», «меньше», «меньше или равно», «больше», «больше или равно». Результат работы этих операций может быть либо TRUE (1), либо FALSE (0), либо x. Результат равен x, если хотя бы один из операндов равен x или z.

Сравнить значения, которые содержат биты x и z, можно также с помощью операций проверки идентичности (case equality) (===) и проверки неидентичности (case inequality) (!==). Обе операции можно использовать только в коде тестового окружения, т. к. они не предназначены для синтеза. Эти операции возвращают только TRUE и FALSE:

```
1   logic inputWires;
2   if (inputWires === 4'b00xz)
3       $display ("The input is 4'b00xz");
4   else $display ("The input is something other than 4'b00xz");
```

Если бы в строке 2 стояла операция проверки равенства (==), а не идентичности, то строка 3 никогда бы не была исполнена, даже если значение inputWires было бы равно 4'b00xz. При проверке равенства было бы получено значение x, которое интерпретировалось бы в операторе if как FALSE. Соответственно, сообщение в строке 4 было бы выведено на консоль при любом значении inputWires.

В SystemVerilog имеются также операции неполной проверки равенства (wild-card equality) (==?) и неполной проверки неравенства (wild-card inequality) (!=?), используемые в тестовом окружении. В условии

```
1   if (a ==? b) ...
```

проверяется, совпадают ли векторы a и b. Если среди битов b есть x или z, они рассматриваются как маски (wild cards). В следующем примере проверяется, что четные биты a равны 0 (нечетные биты игнорируются).

```
1   logic [7:0] a;
2   if (a ==? 8'bx0x0x0x0)
3       $display("The even bits of a are 0");
4   else $display("At least one of the even bits of a is not 0");
```

Правая часть операции неполной проверки необязательно должна быть константой. Если a содержит значения x или z, то результат выражения равен x.

## 11.2.2. Логические связки в выражениях

Логические выражения (условия) в if не ограничиваются одной переменной или сравнением. С помощью логических связок можно строить более сложные выражения. К связкам относятся: И (&&), ИЛИ (||), импликация (->) и эквивалентность (<->). Результатом вычисления логических выражений может быть 1 (TRUE), 0 (FALSE) или x, если хотя бы один операнд имеет значение x или z. Имеется также операция отрицания (!); отрицание x или z равно x.

Приведем пример использования логических связок:

```
1   bit [5:0] a, b, c;
2   bit       enable;
3   if (((a <= b) && (c != 5)) || enable)
4   ...
```

Здесь говорится, что если a меньше или равно b и c не равно 5, или enable равно TRUE, то следует исполнить ветвь then оператора if. Поскольку приоритет операции && выше, чем у операции ||, а приоритет каждой из них ниже, чем у операций сравнения (<=, !=), строка 3 эквивалентна следующей:

```
1   if ( a <= b && c !=5 || enable )
```

Как правило, многие люди не помнят приоритетов операций. Если сомневаетесь, ставьте скобки, чтобы смысл выражения был понятен.

Операции отрицания (!) и поразрядного дополнения (~) похожи, но не идентичны. Первая предполагает, что ее операнд имеет значение TRUE или FALSE. Рассмотрим, например, следующий код:

```
1   bit [4:0] r;
2   if (!r) ...
```

В этом случае поскольку переменная r имеет тип bit, то значение 0 интерпретируется как FALSE, а значение, отличное от 0, – как TRUE. Именно на основе такой интерпретации вычисляется отрицание r в строке 2. Если значение r равно 4'b0011, то r считается равным TRUE, а дополнение к r – FALSE.

Операция поразрядного дополнения инвертирует каждый бит своего операнда. Если значение r равно 4'b0011, то значение ~r равно 4'b1100. И то, и другое интерпретируется как TRUE. Таким образом, операции работают по-разному.

Импликация

```
1   if (a -> b) ...
```

есть то же самое, что и

```
1   if (!a || b) ...
```

Эквивалентность

```
1   if (a <-> b) ...
```

есть то же самое, что и

```
1   if ((a -> b) && (b -> a)) ...
```

Импликация и эквивалентность не используются в синтезе.

## 11.2.3. Многовариантное ветвление с помощью if-else-if

Оператор if-else описывает выбор из двух вариантов: исполняется либо ветвь then, либо ветвь else. Для реализации многовариантного ветвления можно воспользоваться if-else-if.

Рассмотрим следующий фрагмент кода:

```
1 bit expression1, expression2, expression3;
2 byte a, b;
```

```
3 always_comb begin
4   a = 0;
5   b = 0;
6   if (expression1)      a = 1;
7   else if (expression2) b = 3;
8   else if (expression3) a = 43;
9   else b = 4;
10 end
```

В этом коде выбор ветви определяется тем, какое из проверяемых в порядке очереди логических выражений (условий) первым оказывается истинным. Чтобы переменной a было присвоено значение 43 в строке 8, вычисление выражений expression1 и expression2 должно вернуть FALSE, а expression3 – TRUE. Последняя ветвь else в строке 9 является выбором «по умолчанию», которая будет исполняться только в том случае, если ни одна из предыдущих ветвей не была выбрана. Она необязательна.

Если бы перед первым if в строке 6 было бы написано ключевое слово unique, это бы означало, что выражения в ветвях должны быть взаимно исключающими, и ровно одно из них должно обязательно исполняться. Точнее можно сказать, что это индикатор, показывающий, что вычисление значений выражений можно упростить и производить параллельно. Если последней ветви else нет, то unique if выведет сообщение об ошибке, если во время симуляции ни один оператор не исполнится.

Ключевое слово unique0 похоже на unique тем, что также указывает на попарную несовместность выражений. Однако если во время симуляции ни одно выражение не является истинным, то никакое сообщение об ошибке выведено не будет.

Ключевое слово priority перед первым if означает, что необходимо соблюдать указанный порядок вычислений. Обычно оно используется, когда заранее известно, что, по крайней мере, одно выражение должно быть истинным.

Результатом синтеза кода примера является комбинационная схема, потому что a и b имеют вполне определенные значения при исполнении оператора always_comb. Для возможности построения комбинационной схемы последняя ветвь else необязательна. Ключевые слова unique, unique0 и priority можно использовать для оптимизации синтеза. С точки зрения оптимизации, unique и unique0 дают одинаковый эффект.

## 11.3. Оператор case и его разновидности

Оператор case выбирает для исполнения один из нескольких операторов путем сопоставления значения выражения с заданными вариантами. В приведенном ниже фрагменте кода выражение в скобках (строка 3) определяет, какой из вариантов (строки 4–8) следует исполнить. Работа оператора состоит в том, чтобы найти первый вариант, соответствующий значению выражения, и исполнить соответствующий оператор (это может быть составной оператор,

заключенный в скобки begin-end). После этого исполнение продолжается с оператора, следующего за ключевым словом endcase.

```
1   logic [1:0] caseExp;
2   logic f;
3   case (caseExp)
4     2'b00: f = 0;
5     2'b0x: f = 1;
6     2'b1z: f = 1'bz;
7     2'b11: f = 1;
8     default: f = 1'bx;
9   endcase
```

Итак, данный оператор case вычисляет 2-битное значение caseExp и сравнивает его с каждым вариантом в указанном порядке. В первом совпавшем варианте исполняется оператор справа от двоеточия. Если ни один из вариантов не подходит, то исполняется оператор варианта default (строка 8). Этот вариант необязателен: если его нет и не было совпадения ни с одним из вариантов, то исполнение продолжается с оператора, следующего за endcase.

Отметим, что сравнение производится по правилам 4-значной логики, поэтому в данном примере могло бы быть $4^2$ вариантов. Обычно 4-значная логика используется в тестовом окружении, чтобы задать действия для случаев, когда некоторые биты принимают значения x или z. Если оператор применяется для описания синтезируемой схемы, то в вариантах нельзя использовать x и z. Однако наличие x в правой части оператора (как в строке 8) допустимо и интерпретируется инструментом синтеза как «безразлично».

Оператор case из фрагмента выше можно записать и так:

```
1   case (caseExp)
2     2'b00: f = 0;
3     2'b0x,
4     2'b11: f = 1;
5     2'b1z: f = 1'bz;
6     default: f = 1'bx;
7   endcase
```

Здесь варианты, в которых находится оператор f = 1, сгруппированы вместе и разделены запятой (строки 3 и 4).

Выражение и варианты оператора case необязательно должны быть константными выражениями. В следующем примере в вариантах указаны переменные:

```
1   bit inA, inB, inC;
2   bit [1:0] encoding;
3   case (1'b1)
4     inA:     encoding = 1;
5     inB:     encoding = 2;
6     inC:     encoding = 3;
7     default: encoding = 0;
8   endcase
```

В данном операторе case переменной encoding будет присвоено значение 1, если inA равно 1; значение 2, если inA не равно 1, и inB равно 1; значение 3, если ни inA, ни inB не равны 1, а inC равно 1; и значение 0, если ни один из первых трех вариантов не подошел. Отметим также, что если более одной переменной inA, inB и inC равны 1, то будет выбран вариант, соответствующий первой из этих переменных, значение которой равно 1, и именно в указанном порядке. Так устроен оператор case – выбирается только первый подходящий вариант.

## 11.3.1. Операторы casex и casez

Операторы casex и casez являются разновидностями case и работают так же, как описано выше, с тем лишь отличием, что биты, имеющие значение z, в случае casez и биты, имеющие значения z или x, в случае casex считаются несущественными и не учитываются при сравнении выражения с вариантами. Такие маскирующие биты могут встречаться как в выражении, так и в вариантах.

```
1   logic [3:0] caseExp, s;
2   caseExp = 4'b011x;
3   casex (caseExp)
4     4'b0100: s = 1;
5     4'bz110: s = 2;
6     4'bx01x: s = 3;
7     default: s = 0;
8   endcase
```

В данном случае x или z означают, что бит в соответствующей позиции не должен учитываться при сравнении. В результате переменной s будет присвоено значение 2, потому что при сравнении с вариантом в строке 5 рассматриваются только два средних бита, а они равны 2'b11. Таким образом, x в caseExp и z в варианте в строке 5 исключают биты в позициях 3 и 0 из сравнения. Поскольку выражение и варианты могут быть переменными, маскирование может быть динамическим.

В операторах casex и casez в целочисленных константах можно вместо z использовать вопросительный знак. Пример приведен ниже:

```
1   logic [2:0] state;
2   logic sel;
3   casez (state)
4     3'b1??: nextState = 3'b010;
5     3'b01?: nextState = 3'b001;
6     3'b001 nextState = (sel) ? 3'b100 : 3'b010;
7   endcase
```

Здесь, если бит в позиции 2 установлен, исполняется оператор в строке 4, значения остальных двух битов при этом несущественны. Это один из способов упростить синтезируемую схему.

Многие команды не используют оператор casex при разработке моделей (он объявлен вне закона!). Причина в том, что поведение синтезированной схемы может отличаться от поведения модели в симуляторе. Вместо casex для

описания несущественных комбинаций обычно применяется unique case. Ничто, однако, не мешает использовать casex в тестовом окружении.

### 11.3.2. Ключевые слова unique, unique0 и priority

Ключевые слова unique, unique0 и priority, применимые к любому из операторов case, casex и casez, позволяют уточнить их поведение. Использование этих квалификаторов открывает возможность для дополнительных проверок в процессе симуляции и для некоторых оптимизаций на этапе синтеза.

- unique – означает, что для выбора должен подходить один и только один вариант. Если это не так, симулятор сообщит об ошибке. Инструмент синтеза может воспользоваться этой информацией, чтобы упростить синтезируемую схему (обычная проверка вариантов строго по порядку необязательна).
- unique0 – то же, что unique, но симулятор не сообщает об ошибке.
- priority – означает, что хотя бы один вариант точно должен подойти. Если это не так, симулятор сообщит об ошибке. Выбирается первый подходящий вариант. Чтобы вызвать перекрытие значений вариантов, можно использовать вопросительный знак (см. выше). Это позволяет сократить размер синтезируемой схемы.

При использовании ключевых слов unique и priority нет необходимости включать вариант default для обработки отсутствия совпадений. Симулятор сделает это автоматически. Рассмотрим следующий пример использования оператора unique case:

```
1   bit [15:0] a, b, result;
2   bit [2:0] aluFn;
3   unique case (aluFn)
4     3'b110: result = a + b;
5     3'b010: result = a - b;
6     3'b011: result = b - a;
7     3'b101: result = a & b;
8     3'b000: result = -a;
9   endcase
```

Здесь описывается АЛУ с 5 операциями. Инструмент синтеза поймет, что варианты, которые здесь не указаны, невозможны. Симулятор сообщит об ошибке, если на входе появится значение, отличное от перечисленных (6, 2, 3, 5, 0).

Другие примеры unique case приведены в разделе 2.2.4 и следующих за ним.

## 11.4. Циклы

Повторяющееся поведение описывается с помощью операторов цикла: forever, repeat, while, do while, for и foreach. Циклы while, for и do while очень похожи на циклы в языках программирования. В основном они используются в тестовом окружении и моделях высокого уровня, но иногда бывают полезны при проектировании на уровне регистровых передач.

### 11.4.1. Цикл forever

Как это ни удивительно, но forever означает «всегда». Начав исполняться, этот цикл уже не останавливается. Поэтому он часто применяется для генерации тактовых сигналов:

```
1  bit clock;
2  initial
3    forever #5 clock = ~clock;
```

Этот цикл forever делает паузу на 5 единиц времени, а затем заменяет значение тактового сигнала на его дополнение; это продолжается до конца симуляции. Если вы используете этот цикл для установки значения тактового сигнала, то где-то в другом месте описания должны вызвать процедуру $finish. Поскольку переменная clock имеет тип bit, ее начальное значение равно 0.

В любом цикле forever должно быть место, где исполнение приостанавливается в ожидании внешнего события, иначе симулятор зациклится. Это значит, что в любом цикле forever должен быть хотя бы один оператор события (@) или задержки (#).

### 11.4.2. Цикл repeat

Цикл repeat исполняется фиксированное число раз, счетчик цикла в этом случае не нужен.

```
1  bit clock;
2  initial begin
3    repeat (1000) #5 clock = ~clock;
4    $finish;
5  end
```

Здесь мы 1000 раз инвертируем значение clock, что соответствует 500 изменениям состояния. После этого симуляция завершается.

Количество повторений в скобках может быть переменной. Значение x или z трактуется как 0, и цикл не исполняется ни разу.

### 11.4.3. Цикл while

Цикл while исполняет оператор, пока значение выражения не станет равным FALSE. Существует также вариант do-while, описанный в разделе 11.4.4. В примере 11.1 цикл while служит для подсчета количества единиц во входной переменной item. Результат появляется на выходе count.

Цикл while находится в строках 8–11. Выражение в скобках определяет, попадет ли управление в тело цикла и нужно

```
1  module countOnes
2   #(parameter width = 8)
3   (input bit [width-1:0] item,
4    output bit [$ceiling(width):0] count);
5
6   always_comb begin
7     count = 0;
8     while (item) begin
9       count = count + item[0];
10      item = item >> 1;
11    end
12  end
13 endmodule: countOnes
```

**Пример 11.1.** Подсчет числа единиц в цикле while

ли исполнять следующую итерацию. Как и в операторе `if-else`, нулевое значение трактуется как FALSE, а любое ненулевое – как TRUE (`x` и `z` считаются нулями).

В начале работы оператора `always_comb` переменная `count` инициализируется нулевым значением. Затем начинается цикл `while`. Пока переменная `item` отлична от нуля (в ней есть хотя бы одна единица), `count` увеличивается на значение нулевого бита `item`. После этого `item` сдвигается вправо на одну позицию, и вычисляется условие цикла `while`, определяющее необходимость следующего исполнения тела цикла. Поскольку операция сдвига вправо помещает в освободившуюся позицию 0, все единицы рано или поздно будут убраны, и цикл остановится.

Отметим, что в ситуации, когда значение `item` изначально было равно 0, вход в цикл вообще не произойдет, и его тело не будет исполнено ни разу. Значение `count` тоже будет равно 0.

Использование оператора `always_comb` предполагает построение комбинационной схемы. Благодаря тому что максимальное количество итераций цикла ограничено константой (разрядностью `item`), инструмент синтеза может развернуть цикл и соединить биты `item` с входами каскада сумматоров.

Циклы `while` также могут быть полезны для исполнения итераций в тестовом окружении. В этом случае в теле цикла обычно используются операторы управления временем (#, @).

Если говорить о данном конкретном примере, то в SystemVerilog уже имеется функция `$countones()`, подсчитывающая количество единиц в векторе. В строке

```
1   count = $countones(item);
```

делается то же, что и в приведенном выше коде. Однако ее нельзя синтезировать.

## 11.4.4. Цикл do-while

Цикл `do-while` похож на `while` (раздел 11.4.3): у обоих имеется условие и тело цикла; условие определяет, нужно ли продолжать исполнение тела цикла.

Разница состоит в том, что в цикле `while` условие проверяется до начала исполнения тела цикла (поэтому может случиться, что тело не исполнится ни разу), а в цикле `do-while` сначала исполняется тело, а уже потом проверяется условие (поэтому хотя бы один раз тело цикла исполнено будет). См. пример 11.4.

## 11.4.5. Цикл for

Цикл `for` исполняется так же, как в языках программирования. Он включает в себя оператор инициализации, условие цикла, тело цикла и оператор обновления. Сначала исполняется оператор инициализации. Затем, пока значение условия цикла равно TRUE, исполняется тело цикла, а за ним – оператор обновления. Как только значение условия становится равным FALSE, осуществляется выход из цикла. Если условие ложно сразу после инициализации, тело цикла не исполнится ни разу.

Условие цикла вычисляется так же, как в операторе if-else: нулевое значение трактуется как FALSE; любое ненулевое значение – как TRUE (x и z считаются нулями).

Цикл for в приведенном ниже фрагменте реализует ту же функциональность, что и в примере 11.1 (строки 6–12).

```
1   always_comb
2     for (count = 0; item; item = item >> 1)
3       count = count + item[0];
```

А можно это сделать и так:

```
4   always_comb
5     for (count = 0; item; item = item >> 1)
6       count += item[0];
```

Оператор += в строке 6 – сокращенный способ записи строки 3.

Допустим, что мы хотим инициализировать массив регистров единицами:

```
1   bit [15:0] mem [36];
2   initial
3     for (int i = 35; i >= 0; i--)
4       mem[i] = 16'hFFFF;
```

В этом примере цикл for используется, чтобы перебрать все элементы. Новое здесь – объявление счетчика цикла в самом цикле. Мы объявили переменную i типа int и инициализировали ее значением 35. Далее, пока значение i больше или равно 0, в элемент массива с индексом i записывается 16'hFFFF и производится декремент i.

Заметим, что переменная i не видна за пределами цикла for. Это *автоматическая* переменная, которая возникает в начале цикла и исчезает после выхода из него.

## 11.4.6. Операторы continue и break

Ключевые слова continue и break можно использовать в любом из рассмотренных выше циклов.

Оператор continue осуществляет переход в начало следующей итерации цикла. Если цикл включает в себя обновление, как в цикле for, то управление передается ему. Для остальных циклов оператор continue работает следующим образом: если цикл включает проверку условия исполнения следующей итерации, то управление передается в точку этой проверки, иначе просто начинается следующая итерация.

Оператор break прекращает текущую итерацию цикла, после чего исполнение продолжается с оператора, следующего за циклом.

## 11.4.7. Цикл foreach

Цикл foreach описывает итерирование по всем элементам массива. Можно считать, что это способ автоматически сгенерировать цикл for (или несколько вложенных циклов for в случае многомерных массивов), в котором будут перебираться все элементы.

В следующем очень простом примере сообщение “hello logic” выводится на консоль в двух строках, по одному слову в каждой:

```
1   string speak [2] = '{ “hello”, “logic”};
2   foreach (speak [j])
3     $display(speak[j]);
```

В строке 1 объявлен массив символьных строк speak. Массив содержит два элемента, инициализированных строками “hello” и “logic”. Тело цикла foreach в строке 2 исполняется для каждого элемента массива speak. В частности, будет выведена строка “hello” в первой строке и строка “logic” во второй. Это эквивалентно такому циклу for:

```
1   string speak [2] = '{ “hello”, “logic”};
2   for (int j = 0; j <= 1; j++)
3     $display(speak[j]);
```

Таким образом, цикл foreach создает неявный заголовок цикла с объявлениями переменных цикла. Переменные цикла (в данном случае только j) предназначены для обхода массива по всем его заданным размерностям (в данном случае от 0 до 1).

В примере 11.2 показано применение цикла foreach в тестовом окружении, цель которого заключается в подаче 1000 тестовых векторов на вход тестируемой модели. Предполагается, что векторы сгенерированы и загружены в массив tvector где-то в другом месте. Экземпляр устройства создается в строке 4; обратите внимание, что для краткости там показаны только входы устройства. Затем в цикле foreach каждый элемент массива подается на вход устройства и производится задержка на одну единицу времени для обработки элемента схемой устройства. Для вывода на консоль значений выходов можно было бы воспользоваться процедурой $monitor.

```
1  logic [13:0] tvectors [1000];
2  logic [13:0] dutInput;
3
4  device dut (dutInput, …);
5  initial begin
6    foreach(tvectors [i]) begin
7      dutInput = tvector[i];
8      #1;
9    end
10 end
```

**Пример 11.2.** Цикл foreach для подачи тестовых векторов на вход тестируемой модели

## 11.5. Подпрограммы: функции и процедуры

В SystemVerilog есть два вида подпрограмм: функции и процедуры. Они дают те же преимущества, что и в языках программирования.

- Повторное использование общего кода. Код пишется и отлаживается один раз, а вызываться может из многих мест.
- Абстрагирование сложного кода. Код можно разбить на блоки и оформить каждый из них в отдельную подпрограмму. Даже если эти подпрограммы вызываются только по одному разу, код становится понятнее.

В SystemVerilog функции отличаются от процедур в основном двумя аспектами.

- Функции исполняются мгновенно, поэтому в них нельзя использовать операторы управления временем (@, # и wait). Напротив, процедуры могут содержать такие операторы. Это, в частности, означает, что функции не могут вызывать процедур, но процедуры могут вызывать функции.
- Функции, вообще говоря, возвращают значение, которое можно использовать в процедурном операторе или в операторе непрерывного присваивания. Напротив, процедуры не возвращают значения и потому не могут быть использованы таким способом.

Существуют и другие различия в работе функций и процедур, которые будут освещены в этом разделе.

## 11.5.1. Функции

В примере 11.3 показаны объявление и вызов функции. Сама функция занимает строки 2–11. В строке 2 объявлено, что функция называется hamEncode и возвращает 7-битное значение. В строке 3 объявлен входной аргумент – переменная msg, а в строке 4 – внутренние переменные. Далее в строках 6–10 находится тело функции.

```
1  module ham;
2    function bit[7:1] hamEncode
3      (input bit[3:0] msg);
4      bit cbit4, cbit2, cbit1;
5
6        cbit4 = msg[3] ^ msg[2] ^ msg[1];
7        cbit2 = msg[3] ^ msg[2] ^ msg[0];
8        cbit1 = msg[3] ^ msg[1] ^ msg[0];
9        hamEncode =
10        {msg[3:1], cbit4, msg[0], cbit2, cbit1};
11   endfunction
12
13   initial begin
14    bit [7:1] result;
15    for (bit [4:0] i = 0; i <=15; i++) begin
16       result = hamEncode(i);
17       $display("msg=%4b, result=%b", i, result);
18    end
19   end
20 endmodule: ham
```

**Пример 11.3.** Определение и вызов функции

Цель этой функции – вычислить три бита кода Хэмминга для 4-битного сообщения msg. Эти три бита вычисляются в строках 6–8, а затем в строках 9–10 конкатенируются с битами сообщения. Указание возвращаемого значения происходит через присваивание значения имени функции (hamEncode). Исполнение функции прекращается при достижении endfunction или в результате исполнения оператора return, который в данном случае отсутствует. Имени функции можно присваивать значение несколько раз, возвращается последнее присвоенное. В операторе return, если он используется, тоже можно указать возвращаемое значение.

Тестовое окружение в строках 13–19 вызывает эту функцию, передавая ей все возможные комбинации четырех бит, после чего выводит на консоль каждую комбинацию и соответствующее ей закодированное сообщение. Непосредственный вызов функции происходит в строке 16 из процедурного оператора присваивания. В этой точке значение переменной i копируется во

входную переменную функции, msg, и тело функции вычисляет соответствующий ему код. После возврата из функции вычисленное значение hamEncode загружается в переменную result. В строке 17 выводятся на консоль входные и выходные значения функции.

Этим простым примером иллюстрируется еще несколько особенностей функций.

- По умолчанию входные аргументы используют «передачу по значению», т. е. во входные переменные функции копируются только подаваемые значения, но допустима и «передача по ссылке» (ref). Кроме входных переменных, у функций могут быть выходные и комбинированные (одновременно входные и выходные) аргументы.
- Оператор initial можно записать и в таком виде:

```
1   initial
2   for(bit [4:0] i = 0; i <=15; i++)
3       $display("msg=%4b, result=%b", i, hamEncode(i));
```

То есть необходимости присваивать возвращенное функцией значение промежуточной переменной перед выводом на консоль не было. Мы сделали это только для того, чтобы показать, что функцию можно вызывать из процедурного оператора (в отличие от процедуры).

- Функции можно вызывать также из оператора непрерывного присваивания. Если во фрагменте кода ниже msg изменится где-то в другой части модели, то переменная result обновится, как положено при непрерывном присваивании.

```
1   bit [7:1] result;
2   bit [3:0] msg;
3   assign result = hamEncode(msg);
```

- Функция может содержать любые процедурные операторы (в т. ч. циклы, if-else, case и т. д.), за исключением операторов контроля времени, которые потенциально могут заблокировать ее исполнение.

## 11.5.2. Процедуры

В примере 11.4 демонстрируется определение и вызов процедуры на примере простого чтения из памяти по шине. (Иногда вызов процедуры называют «активацией».) Протокол чтения памяти основан на двух сигналах: read и dataRdy. Сигнал read означает, что на линии busAddr находится значимый адрес чтения, а dataRdy – что память увидела сигнал read и поместила данные на линию busData.

Пример включает в себя процедуру readBus (строки 5–13), на вход которой подается значение адреса, а на выходе получаются прочитанные из памяти данные: в процедуре реализован протокол запроса данных. В операторе initial (строки 16–20) процедура вызывается в строке 17, и ей передается адрес (16'h100) и буфер tbData. После возврата из процедуры в tbData окажутся выходные данные. После этого оператор initial выводит на консоль прочитанные данные и завершает симуляцию.

В момент вызова процедуры значение addr копируется во входную переменную, и процедура начинает исполняться. В строках 7–8 она устанавливает значение на адресной шине busAddr и значение сигнала read, а затем входит в цикл do-while, ожидая сигнала dataRdy (строки 9–11). Цикл ждет прихода переднего фронта тактового сигнала, после чего проверяет dataRdy; таким образом, тело цикла исполняется, по крайней мере, один раз. Как только сигнал dataRdy оказывается установленным, переменная busData копируется в выходную переменную data (строка 12). После этого процедура завершается и возвращает управление, при этом выходные данные копируются в переменную tbData на вызывающей стороне.

```
1  module busReadTask;
2    bit [15:0] busAddr, busData;
3    bit clk, dataRdy, read;
4
5    task readBus(input bit [15:0] addr,
6                 output bit[15:0] data);
7      busAddr = addr;
8      read = 1;
9      do
10       @(posedge clk);
11     while (~dataRdy);
12     data = busData;
13   endtask
14
15   bit [15:0] tbData;
16   initial begin // чтение из памяти
17     readBus(16'h0100, tbData);
18     $display("data read=%h", tbData);
19     $finish;
20   end
21   initial begin // моделирование памяти
22     dataRdy = 0;
23     do
24       @(posedge clk);
25     while (~read);
26     dataRdy = 1;
27     busData = 16'h23;
28     read = 0;
29   end
30   initial forever #5 clk = ~clk;
31 endmodule: busReadTask
```

**Пример 11.4.** Простая процедура для чтения из памяти

Память моделируется оператором initial в строках 21–29. Сначала убирается (сбрасывается в 0) сигнал dataRdy (строка 22), затем по каждому переднему фронту тактового сигнала проверяется значение сигнала read (цикл do-while в строках 23–35). После установки значения сигнала read цикл завершается, и «память» отвечает, помещая значение на линию busData (16'h23) и устанавливая сигнал dataRdy. Процедура readBus видит сигнал dataRdy и считывает данные с линии busData. (Понятно, что в этом примере никакое чтение из памяти не моделируется! Мы упростили пример, чтобы проиллюстрировать работу процедуры, а не памяти.)

Тактовый сигнал для всей системы моделируется в строке 30 оператора initial.

В этом примере показаны еще некоторые особенности процедур.

- Оператор initial в строках 16–20 – часть тестового окружения для системы. Вызывая процедуру readBus, мы делаем тестовое окружение более понятным, поскольку детали протокола инкапсулированы в одном месте и скрыты от просматривающего код.
- В этом примере есть два неявно заданных конечных автомата, один в процедуре readBus, другой – в модели памяти (строки 21–29 оператора initial). Оба автомата взаимодействуют с помощью сигналов read и dataRdy. Становится очевидным, что процедуры обычно сложнее функций.

### 11.5.3. Сходства и различия

Примеры в предыдущих разделах были призваны продемонстрировать типичное использование подпрограмм: функций и процедур. Как было сказано, они различаются в двух основных аспектах: в функциях недопустимы операторы управления временем, а процедуры не могут возвращать значение.

Однако семантически у них много общего, поэтому далее мы будем рассказывать о них вместе. Там, где необходимо, мы будем отмечать различия.

### 11.5.4. Направление и тип аргументов

Аргументы функций и процедур могут быть входными (input), выходными (output), двунаправленными (inout) и передаваемыми по ссылке (ref). В случае входных и двунаправленных аргументов значение копируется в переменную в момент вызова подпрограммы. В случае выходных и двунаправленных аргументов вычисленные значения копируются в переменные вызывающей стороны при возврате из подпрограммы. Значения аргументов, передаваемых по ссылке, не копируются.

По умолчанию аргументы подпрограмм являются входными, поэтому для аргументов, указанных в начале списка, ключевое слово input можно опускать. Если для какого-то аргумента указано другое направление (output, inout или ref), то следующие за ним наследуют это направление.

Таким образом, заголовки функции hamEncode (строки 2–3 примера 11.3) и процедуры readBus (строки 5–6 примера 11.4) можно записать так:

```
function bit[7:1] hamEncode (bit[3:0] msg);
task readBus(bit [15:0] addr, output bit[15:0] data);
```

Обратите внимание, что в обоих случаях ключевое слово input опущено.

Если тип аргумента не указан, подразумевается тип logic.

### 11.5.5. Возвращаемое значение

В примерах 11.3 и 11.4 показано типичное использование функций и процедур: у них есть несколько аргументов, на основе которых вычисляется результат. Функции возвращают результат для использования в процедурных операторах или операторах непрерывного присваивания. Результат процедур может быть передан через переменную (output, inout или ref), указанную в качестве аргумента.

У функций есть специальный механизм возврата значения – для этого используется переменная, имя которой совпадает с именем функции. Тип возвращаемого значения задается при определении функции.

Функции, вызываемые из операторов непрерывного присваивания, могут иметь только входные аргументы; кроме того, такие функции должны возвращать значение (тип возвращаемого значения должен отличаться от void).

В примере 11.5 показана функция с двумя аргументами (входным и выходным), которая не возвращает значения (имеет тип void).

```
1  module ham_void;
2
3    function void hamEncode
4       (input bit[3:0] msg, output bit [7:1] result);
5       bit cbit4, cbit2, cbit1;
6
7       cbit4 = msg[3] ^ msg[2] ^ msg[1];
8       cbit2 = msg[3] ^ msg[2] ^ msg[0];
9       cbit1 = msg[3] ^ msg[1] ^ msg[0];
10
11      result = {msg[3:1], cbit4, msg[0], cbit2, cbit1};
12   endfunction
13
14   initial begin
15     bit [7:1] out;
16     for (bit [4:0] msg = 0; msg <=15; msg++) begin
17       hamEncode(msg, out);
18        $display("msg=%4b, result=%b", msg, out);
19      end
20   end
21 endmodule: ham_void
```

**Пример 11.5.** Функция типа void

Вызов функции показан в строке 17: поскольку функция не возвращает значения, ее вызов не является частью выражения; функции передаются переменные msg и out.

Значение переменной msg копируется в одноименный входной аргумент, после чего начинается исполнение функции. По достижении ключевого слова endfunction (для возврата из функции может также использоваться оператор return) значение выходного аргумента result копируется в переменную out. После этого исполнение продолжается со строки, следующей за вызовом функции: оба значения выводятся на консоль.

Функция типа void вызывается как процедура, но отличается от процедуры тем, что не может содержать операторы управления временем.

У функций и процедур могут быть аргументы, передаваемые по ссылке. В этом случае в объявлении аргумента присутствует ключевое слово ref. Часто этим механизмом пользуются, когда аргумент имеет настолько большой размер, что его копирование может замедлить симуляцию. Рассмотрим следующий вариант функции sumItUp:

```
1  bit [15:0] myArray[8000];
2  bit [15:0] sum;
3
4  function bit [15:0] sumItUp (bit [15:0] bigArray [8000]);
5    for (int i = 1; i <= 8000; i++)
6      sumItUp += bigArray[i];
7  endfunction
8  ...
9  sum = sumItUp(myArray);
```

Функция суммирует элементы входного массива, состоящего из 8000 16-битных слов, и возвращает 16-битную сумму. При вызове `sumItUp(myArray)` все 8000 слов массива `myArray` копируются в аргумент `bigArray`. Если изменить определение функции следующим образом:

```
1   function bit [15:0] sumItUp (ref bit [15:0] bigArray [8000]);
```

она будет работать напрямую с самим массивом `myArray` (без копирования).

Процедура для выполнения той же операции могла бы выглядеть так:

```
1   task sumItUp (ref bit [15:0] bigArray [8000], output bit [15:0] sum);
2   for (int i = 1; i <= 8000; i++)
3     sum += bigArray[i];
4   endtask
```

В этом случае подлежащий суммированию массив передается по ссылке, а вычисленная сумма возвращается в выходном аргументе процедуры.

## 11.5.6. Автоматические и статические переменные

Функцию и процедуру можно объявить автоматической (`automatic`) или статической (`static`), и это определение будет распространяться на все определенные внутри нее переменные. Статическая переменная создается в единственном экземпляре для всех параллельных обращений к функции или процедуре. Для автоматической переменной при каждом вызове выделяется отдельная память. По умолчанию функции и процедуры, объявленные внутри модуля, интерфейса, программы или пакета, являются статическими, и память для всех объявленных в них объектов распределяется статически. Функции и процедуры, объявленные внутри класса, по умолчанию являются автоматическими.

Автоматические переменные обычно нужны в рекурсивных функциях, т. е. таких, которые вызывают сами себя прямо или косвенно (посредством вызова промежуточной функции). В этом случае при каждом обращении должны создаваться новые копии переменных. Это относится и к процедурам, которые также могут быть рекурсивными.

Однако поскольку в процедуре могут быть операторы управления временем и время их жизни может включать много единиц времени, то вполне может случиться, что процедура будет вызвана из параллельно исполняемой сущности, например из другого оператора `always` или `initial`. Если процедура статическая, то такие обращения будут пользоваться одними и теми же переменными, а если автоматическая, то у каждого обращения будет собственный набор внутренних переменных.

В случае когда необходимо изменить объявление по умолчанию, в определении функции или процедуры (сразу после слова `function` или `task`) можно добавить ключевое слово `automatic` или `static`.

## 11.5.7. Значения аргументов по умолчанию

Аргументы функции и процедуры могут иметь значения по умолчанию, которые используются, если какой-то аргумент не передан явно. Аргумент может быть входным, выходным, двунаправленным или передаваемым по ссылке. Рассмотрим функцию getByte, которая возвращает часть вектора длиной в один байт, начинающийся с указанного бита.

```
1  bit [255:0] bigWord;
2  function byte getByte (input bit [255:0] bigVec, input byte lowBit = 0);
3    return bigVec[lowBit +: 8];
4  endfunction
```

Вектор bigWord состоит из 256 бит. Функция getByte возвращает отрезок длиной 8 бит, начинающийся с заданной позиции (см. строку 3). Например, если мы запросим байт, начинающийся с бита 2, то будет возвращен отрезок bigWord[9:2].

Поскольку аргумент lowBit инициализирован значением 0, следующие два обращения к функции эквивалентны: оба возвращают вектор bigWord[7:0]:

```
1  getByte (bigWord, 0);
2  getByte (bigWord);
```

Если у функции два входных аргумента и у каждого имеется значение по умолчанию, то любой из них или оба сразу можно опустить при вызове:

```
1  function byte fName (input byte a = 37, b = 99);
2  ...
```

В предположении, что x и y имеют тип byte, эту функцию можно вызвать любым из перечисленных ниже способов:

```
1  fName (x, y); // при вызове функции a=x, b=y
2  fName ( , y); // при вызове функции a=37 (по умолчанию), b=y
3  fName ( , );  // при вызове функции a и b принимают значения по умолчанию
4  fName (x);    // при вызове функции a=x, b=99 (по умолчанию)
```

Аргументы функции и процедуры могут связываться по имени – как при создании экземпляра модуля. В следующих двух вызовах функции передаются значения bigWord и 1. Поскольку значение связывается с аргументом по имени, аргументы можно перечислять в любом порядке. Конструкция .*, употребляемая при создании экземпляра модуля, здесь недопустима.

```
1  getByte(.bigVec(bigWord), .lowBit(1) ));
2  getByte(.lowBit(1), .bigVec(bigWord) ));
```

# 11.6. Таблица операций

| Операция | Лексема | Название и применение | Типы операндов |
|---|---|---|---|
| Операторы присваивания | = <= | Оператор присваивания | Любой |
| | += -= /= *= | Операторы присваивания, совмещенные с арифметическими операциями<br>a += b сокращение от a = a + b | Целый, вещественный |
| | %= | Оператор присваивания, совмещенный с операцией вычисления остатка от деления | Целый |
| | &= \|= ^= | Операторы присваивания, совмещенные с поразрядными операциями | Целый |
| | >>= <<= | Операторы присваивания, совмещенные с операциями логического сдвига.<br>Левый операнд сдвигается на число позиций, заданное правым операндом. Освободившиеся позиции заполняются нулями | Целый |
| | >>>= <<<= | Операторы присваивания, совмещенные с операциями арифметического сдвига.<br>Левый операнд сдвигается на число позиций, заданное правым операндом. В случае сдвига влево совпадает с оператором логического сдвига. В случае сдвига вправо освободившиеся позиции заполняются знаковым битом | Целый |
| Тернарная операция | ?: | Условная операция<br>condExpr ? trueExpr : falseExpr<br>Если condExpr истинно, то значением выражения является trueExpr, иначе – falseExpr | Любой |
| Унарные операции | + - | Унарные арифметические операции | Целый, вещественный |
| | ! | Логическое отрицание.<br>Если операнд имеет значение x или z, результатом будет x | Целый, вещественный |
| | ~ | Поразрядное дополнение.<br>Дополняется каждый разряд операнда[63] | Целый |
| | ++ -- | Операторы инкремента и декремента.<br>Это блокирующие присваивания | Любой |
| Операции редукции | & ~& \| ~\| ^ ~^ ^~ | Унарные операции логической редукции.<br>Операция применяется ко всем битам правого операнда. ~^ и ^~ обозначают исключающее ИЛИ-НЕ[3] | Целый |

[63] Дополнение разряда, имеющего значение x или z, равно x.

[64] Исключающее ИЛИ-НЕ (XNOR) – отрицание исключающего ИЛИ (XOR).

| Операция | Лексема | Название и применение | Типы операндов |
|---|---|---|---|
| Бинарные операторы | + - * / ** | Арифметические операции.<br>base**exp – возведение base в степень exp | Целый, вещественный |
| | % | Вычисление остатка от деления | Целый |
| | & \| ^ ^~ ~^ | Поразрядные операции.<br>~^ и ^~ обозначают исключающее ИЛИ-НЕ | Целый |
| | >> << | Операции логического сдвига.<br>Левый операнд сдвигается на число позиций, заданное правым операндом. Освободившиеся позиции заполняются нулями | Целый |
| | >>> <<< | Операции арифметического сдвига.<br>Левый операнд сдвигается на число позиций, заданное правым операндом. В случае сдвига влево совпадает с оператором логического сдвига. В случае сдвига вправо освободившиеся позиции заполняются значением знакового бита | Целый |
| | && \|\| -><br><-> | Логические операции.<br>Используются в качестве логических связок. Последние две операции – импликация и эквивалентность | Целый, вещественный |
| | < <= > >= | Операции сравнения на больше/меньше | Целый, вещественный |
| | === !=== | Операции проверки идентичности/неидентичности.<br>Производят сравнение значений с 4-значными разрядами | Любой, кроме вещественного |
| | == != | Операции проверки равенства/неравенства | Любой |
| | ==? !=? | Операции неполной проверки равенства/неравенства.<br>x и z в разрядах правого операнда сопоставляются с любыми значениями | Целый |
| Операция проверки вхождения во множество | inside | Операция проверки вхождения элемента во множество.<br>Является ли левый операнд элементом множества справа? | Левый операнд должен быть одиночным элементом |
| Прочие | {} {{}} | Операции конкатенации и репликации | Целый |

# Глава 12

# Структурные модели

Структурные модели описывают соединения элементов разрабатываемой системы. Для создания сложных систем структурные и процедурные модели используются совместно. В структурных моделях задействуются модули (и их экземпляры), вентильные примитивы (и их экземпляры), цепи и спецификации портов.

## 12.1. Вентильные примитивы

Вентильные примитивы моделируют простые функции, встречающиеся в логических схемах. Различные типы вентилей и их обозначения в логических схемах приведены в табл. 12.2. Учитывая, что в современной практике проектирования используются инструменты синтеза, позволяющие процедурно описывать комбинационную схему, большинство этих примитивов редко применяется при проектировании на уровне регистровых передач. Исключения – драйверы и приемники шин. Кроме того, в некоторых инструментах синтеза примитивные логические вентили используются для описания функциональности и задержки библиотечных элементов. Более подробное обсуждение и примеры вентильных примитивов см. в разделе 1.1.

Для создания экземпляров этих примитивов в модели нужно указать тип вентиля и соединения портов. В следующем описании создаются вентили AND и NAND. Первым указывается выходной порт вентиля, а затем – входные порты. Большинство компиляторов допускает использование большого числа портов, а некоторые ограничивают их количество.

```
1   logic a, b, c, d, e, f, out1, out2;
2
3   and (out1, a, b, c);
4   nand (out2, d, e, f );
```

У некоторых примитивов, например инверторов и буферов, есть только один вход. В SystemVerilog они, однако, могут иметь несколько выходов. В следующем описании первый примитив not, равно как и buffer (строки 3 и 4), имеет один выход, и он указан первым в списке портов. А вентиль not в строке 5 имеет три выхода и один вход (c).

```
1   logic a, b, c, out1, out2, out3, out4, out5;
2
3   not (out1, a);
4   buf (out2, b);
5   not (out3, out4, out5, c);
```

Выше приведены простейшие варианты создания примитивов. В общем виде создание экземпляра примитива выглядит так:

```
and (driveStrengths) #delay  instName (outPort, in1, in2, …);
```

Эта строка синтаксически некорректна, однако демонстрирует возможности языка. Возможности параметра driveStrengths в этой книге не обсуждаются, сводя общий вид объявления экземпляра к следующему:

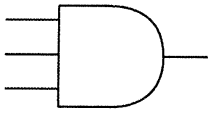
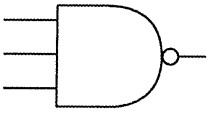
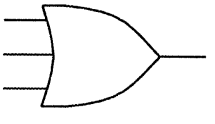
```
and #delay instName (outPort, in1, in2, …);
```

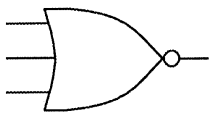
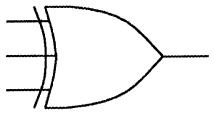
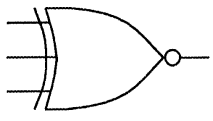
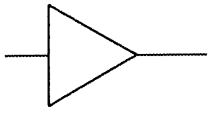
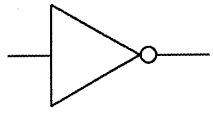
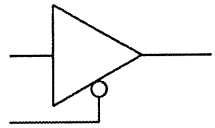
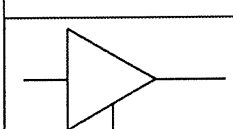
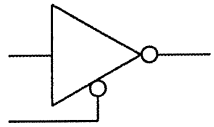
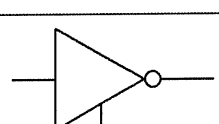
Параметр #delay определяет задержку между изменением значения любого входа и изменением значения выхода. Если параметр не указан, предполагается задержка 0. При изменении значения какого-либо входа в процессе исполнения происходит перевычисление значения выхода, и если оно изменилось, то его появление планируется через #delay единиц времени в будущем. Задержка может также иметь вид #(d1, d2), где d1 – задержка, когда новое значение равно 1, а d2 – когда оно равно 0. Для примитивов, которые могут формировать высокий импеданс, имеется также форма #(d1, d2, d3). Здесь d3 – задержка, когда новое значение равно z.

При задании задержки примитивные вентили приобретают особое свойство, которое заключается в препятствии распространения импульсных помех с входа на выход. То есть если комбинация значений входов, которая могла бы вызвать изменение значения выхода через #delay единиц времени, остается на входах в течение времени, меньшего #delay, то изменение значения выхода отменяется.

Кроме того, как показано выше, экземпляры примитивных вентилей могут быть именованными.

**Таблица 12.2.** Вентильные примитивы

| Тип вентиля | Особенности входов и выходов | Обозначение на схемах | Примеры и описание функциональности |
|---|---|---|---|
| and | Один вход, несколько выходов. Первый порт – выход |  | `and (out, in1, in2, in3);`<br>На выходе будет 0, если хотя бы на одном входе 0. На выходе будет 1, если на всех входах 1. В остальных случаях на выходе будет x |
| nand | |  | `nand (out, in1, in2, in3);`<br>На выходе будет 1, если хотя бы на одном входе 0. На выходе будет 0, если на всех входах 1. В остальных случаях на выходе будет x |
| or | |  | `or (out, in1, in2, in3);`<br>На выходе будет 1, если хотя бы на одном входе 1. На выходе будет 0, если на всех входах 0. В остальных случаях на выходе будет x |

| Тип вентиля | Особенности входов и выходов | Обозначение на схемах | Примеры и описание функциональности |
|---|---|---|---|
| nor | |  | `nor(out, in1, in2, in3);`<br>На выходе будет 0, если хотя бы на одном входе 1. На выходе будет 1, если на всех входах 0. В остальных случаях на выходе будет x |
| xor | |  | `xor(out, in1, in2, in3);`<br>На выходе будет 1, если на нечетном числе входов установлена 1. На выходе будет 0, если на четном числе входов установлена 1. В случае неизвестного значения (x или z), установленного на любом из входов, на выходе будет x |
| xnor | |  | `xnor(out, in1, in2, in3);`<br>На выходе будет 1, если на четном числе входов установлена 1. На выходе будет 0, если на нечетном числе входов установлена 1. В случае неизвестного значения (x или z), установленного на любом из входов, на выходе будет x |
| buf | Несколько выходов, один вход. Последний порт – вход, остальные – выходы (на рисунке показан только один выход) |  | `buf(out1, in);`<br>Значение на выходе совпадает со значением на входе, если на последнем 0, 1 или x. Если на входе z, то на выходе будет x |
| not | |  | `not(out1, in);`<br>Значение на выходе является инверсией значения на входе, если на последнем 0 или 1. Если на входе x или z, то на выходе будет x |
| bufif0 | Тристабильные драйверы. На единственном выходе устанавливается z, если на входе enable установлен 0. Особые случаи возникают, когда enable равен x или z; здесь они не показаны. Примеры использования тристабильных драйверов см. в разделе 2.7 |  | `bufif0(out, in, enableN);`<br>Если enableN установлен (0), то на выходе будет значение, совпадающее со значением входа, если на последнем 0 или 1. Если enableN сброшен (1), то на выходе будет z. |
| bufif1 | |  | `bufif1(out, in, enable);`<br>Если enable установлен (1), то на выходе будет значение, совпадающее со значением входа, если на последнем 0 или 1. Если enable сброшен (0), то на выходе будет z |
| notif0 | |  | `notif0(out, in, enableN);`<br>Если enableN установлен (0), то на выходе будет инверсия значения входа, если на последнем 0 или 1. Если enableN сброшен (1), то на выходе будет z |
| notif1 | |  | `notif1(out, in, enable);`<br>Если enable установлен (1), то на выходе будет инверсия значения входа, если на последнем 0 или 1. Если enable сброшен (0), то на выходе будет z |

В языке есть и другие примитивы, которые охватывают такую функциональность, как, например, проходные транзисторы (pass transistors) и другие примитивы переключательного уровня (switch-level models). В этой книге они не рассматриваются.

# 12.2. Цепи

Цепь (net) представляет собой физическое электрическое соединение между двумя структурными элементами, например вентильными примитивами. Выходы этих элементов устанавливают значения в цепи. Если к цепи не подключен ни один драйвер, то значение в ней равно z. Цепи можно классифицировать по следующим признакам:

- реализуют логическую функцию или нет (например, монтажное И);
- сохраняют ли значение в отсутствие драйвера (так делают только цепи типа `trireg`);
- ограничено ли количество подключенных к цепи драйверов (ограничение есть только в цепях типа `uwire`, `supply1` и `supply0`).

Если к цепи подключено несколько драйверов, то для определения значения в цепи симулятор использует функцию разрешения конфликтов; она вызывается симулятором при изменении выходного значения любого драйвера. На основе выходных значений всех драйверов функция вычисляет результирующее значение для установки в соединениях цепи. В табл. 12.3 приведены типы цепей, их применение в моделировании, а также их функции разрешения конфликтов.

**Таблица 12.3.** Типы цепей

<table>
<tr><th>Тип цепи</th><th>Применение в моделировании</th><th>Таблица истинности функции разрешения конфликтов для цепей с несколькими драйверами</th></tr>
<tr><td>wire<br><br>tri</td><td>Моделирует соединение без логической функции. Если драйвер только один, то используется цепь типа wire; если драйверов несколько – цепь типа tri. Хотя они функционально эквивалентны, используйте тот тип, который ясно описывает назначение цепи. Примеры использования цепей типа tri см. в разделе 2.7</td><td>
<table>
<tr><th></th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>x</td><td>x</td><td>0</td></tr>
<tr><td>1</td><td>x</td><td>1</td><td>x</td><td>1</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>0</td><td>1</td><td>x</td><td>z</td></tr>
</table>
Таблица обобщается на случай более 2 драйверов. Если хотя бы один драйвер устанавливает x, то результатом будет x. Приоритет z ниже приоритетов 0 и 1
</td></tr>
<tr><td>uwire</td><td>Моделирует соединение без логической функции. Может иметь только один драйвер; в противном случае компилятор сообщает об ошибке. Использование этого типа вместо wire или tri позволяет обнаружить ошибки, когда к одной шине случайно подключено несколько драйверов</td><td>Функции разрешения конфликтов нет, значение в цепи совпадает со значением драйвера</td></tr>
</table>

<table>
<tr><th>Тип цепи</th><th>Применение в моделировании</th><th>Таблица истинности функции разрешения конфликтов для цепей с несколькими драйверами</th></tr>
<tr><td>wand<br><br>triand</td><td>Реализует монтажное И для значений драйверов</td><td>
<table>
<tr><th></th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>x</td><td>1</td></tr>
<tr><td>x</td><td>0</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>0</td><td>1</td><td>x</td><td>z</td></tr>
</table>
0 имеет приоритет над другими значениями
</td></tr>
<tr><td>wor<br><br>trior</td><td>Реализует монтажное ИЛИ для значений драйверов</td><td>
<table>
<tr><th></th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>1</td><td>x</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>x</td><td>x</td><td>1</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>0</td><td>1</td><td>x</td><td>z</td></tr>
</table>
1 имеет приоритет над другими значениями
</td></tr>
<tr><td>tri1</td><td>Резистивная установка логической 1</td><td>
<table>
<tr><th></th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>x</td><td>x</td><td>0</td></tr>
<tr><td>1</td><td>x</td><td>1</td><td>x</td><td>1</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>0</td><td>1</td><td>x</td><td>1</td></tr>
</table>
x имеет приоритет над другими значениями. Если драйверов нет, значение в цепи равно 1
</td></tr>
<tr><td>tri0</td><td>Резистивная установка логического 0</td><td>
<table>
<tr><th></th><th>0</th><th>1</th><th>x</th><th>z</th></tr>
<tr><td>0</td><td>0</td><td>x</td><td>x</td><td>0</td></tr>
<tr><td>1</td><td>x</td><td>1</td><td>x</td><td>1</td></tr>
<tr><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td></tr>
<tr><td>z</td><td>0</td><td>1</td><td>x</td><td>0</td></tr>
</table>
x имеет приоритет над другими значениями. Если драйверов нет, значение в цепи равно 0
</td></tr>
<tr><td>supply1</td><td>Моделирует соединение с источником питания. Драйверы к цепи не подключаются (если только вы не хотите учинить пожар)</td><td>1</td></tr>
<tr><td>supply0</td><td>Моделирует соединение с землей. Драйверы к цепи не подключаются</td><td>0</td></tr>
</table>

| Тип цепи | Применение в моделировании | Таблица истинности функции разрешения конфликтов для цепей с несколькими драйверами |
|---|---|---|
| trireg | Моделирует устройство сохранения заряда. В цепи могут устанавливаться значения 0, 1 и x. Если все драйверы устанавливают z, то цепь становится драйвером, устанавливающим предыдущее сохраненное значение | Функция разрешения конфликтов такая же, как у wire и tri, за исключением случая, когда все драйверы устанавливают z. В этом случае цепь устанавливает предыдущее сохраненное значение |

## 12.2.1. Объявление цепей

Объявление цепи начинается с указания типа и, следовательно, функции разрешения конфликтов.

```
1   wire a, b;
2   wire [10:0] c;
```

В данном случае объявлены две скалярные цепи a и b и 11-битная цепь c – все типа wire. Это объявление эквивалентно такому:

```
1   wire logic a, b;
2   wire logic [10:0] c;
```

поскольку тип logic подразумевается для цепей по умолчанию. Возможны и более сложные объявления:

```
1   wire struct packed {
2     logic [15:0] address;
3     logic [7:0] data;
4   } procDataBus;
```

Важно отметить, что все элементы этой цепи имеют тип logic.

Можно определять массивы цепей.

```
1   wire [2:0] [3:0] wireBundle [7:0];
```

Здесь определено восемь элементов, каждый из которых содержит три 4-битных вектора. Слева от имени указаны упакованные измерения массива, поэтому каждая цепь трактуется как одиночный вектор; здесь каждый вектор состоит из 12 бит.

Цепи можно определять, используя разновидность непрерывного присваивания в составе объявления:

```
1   logic [7:0] value1, value2;
2   wire [7:0] data = value1 + value2;
```

В данном случае ключевое слово assign отсутствует, но подразумевается. Это объявление эквивалентно такому:

```
1   logic [7:0] value1, value2;
2   wire [7:0] data;
3   assign data = value1 + value2;
```

Цепи можно объявлять неявно. Если выход примитивного вентиля не объявлялся ранее, то он будет объявлен неявным образом с типом logic по умолчанию.

```
1   logic a, b, c;
2
3   and (cc, a, b);
```

Здесь объявлены три переменные и создан экземпляр примитивного вентиля AND. Компилятор автоматически объявил бы cc типа logic, не выдав ни сообщения об ошибке, ни предупреждения. Быть может, вы того и хотели – меньше нажимать на клавиши. Но если cc – это опечатка, было бы предпочтительно, чтобы компилятор как-то сообщил об этом. Чтобы отключить неявное объявление цепей, включите в описание такую директиву:

```
`define default_nettype none
```

В этом случае компилятор будет настаивать на явном объявлении всех цепей, в результате чего станет проще находить опечатки, например использование 0 вместо O и 1 вместо l. Включайте этот режим и продавайте ваш запас Тайленола[65].

В объявление цепи можно включить задание задержки и силы сигнала. Задержка указывается следующим образом:

```
wire [7:0] #17 busA;
```

Здесь сказано, что изменение значения любого драйвера цепи busA типа wire будет видно в других соединениях цепи через 17 единиц времени. Можно задать сразу две задержки:

```
wire [7:0] #(17, 20) busA;
```

В этом случае 17 – задержка нарастания уровня сигнала (когда значение становится равно 1), а 20 – задержка затухания уровня сигнала (когда значение становится равно 0). Какую задержку использовать в такой векторной цепи, зависит от ее младшего бита. В скобках можно указать и третью задержку – перехода к высокому импедансу (z).

Задание силы сигнала в этой книге не рассматривается.

## 12.2.2. Установка значений в цепях

Установка значений в цепях производится операторами непрерывного присваивания, примитивными вентилями и портами модулей. Значения не могут быть установлены напрямую в результате процедурного присваивания, встречающегося в операторах always и initial. То есть переменная в левой части процедурного присваивания не может быть цепью. Если значение в цепи по одну

[65] Тайленол™ (Tylenol®) – популярный в США препарат, действующим веществом которого является парацетамол. Применяется в том числе для лечения головной боли.

сторону порта модуля устанавливается с помощью значения переменной по другую его сторону (которое устанавливается процедурным присваиванием), то при создании экземпляра модуля неявно подразумевается непрерывное присваивание цепи значения этой переменной.

Отметим, что непрерывное присваивание может устанавливать значение как цепи, так и переменной. Процедурное же присваивание, используемое в разных вариантах `always` и в операторе `initial`, может загружать значение только в переменную (но не в цепь).

Рассмотрим пример цепи типа `wire` с двумя драйверами:

```
1   logic a, b, c;
2   wire d;
3
4   and (d, a, b);
5   assign d = c;
```

В этом примере значение цепи `d` устанавливается одновременно вентильным примитивом и непрерывным присваиванием. Значение `d` можно определить по таблице истинности функции разрешения конфликтов, показанной в табл. 12.4. Будем считать, что значения вентиля AND находятся слева, а значения оператора `assign` – сверху. Если выходные значения обоих драйверов равны 1 или 0, то такое же значение вернет и функция разрешения конфликтов цепи `d`. Если же значения различны или хотя бы одно значение равно 1'bx, то функция разрешения конфликтов вернет 1'bx. Если `c` принимает значение 1'bz, то результатом будет значение примитива AND. Отметим, что выход примитива AND не может быть равен 1'bz. Поэтому, согласно функции разрешения конфликтов, `d` может принимать только следующие значения: 0, 1 или x.

**Таблица 12.4.** Функция разрешения конфликтов для цепей типа wire и tri

| | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | x | x | 0 |
| 1 | x | 1 | x | 1 |
| x | x | x | x | x |
| z | 0 | 1 | x | z |

Цепи могут быть не только скалярами и векторами, но и массивами. В предыдущем разделе мы объявили цепь `wireBundle`; это объявление повторяется ниже. Эта цепь включает в себя восемь элементов, составленных из трех 4-битных векторов. В описании показаны также объявления других переменных типа `logic` с такой же структурой.

```
1   wire [2:0] [3:0] wireBundle [7:0];
2   logic [2:0] [3:0] variableArray [7:0];
3   logic [2:0] [3:0] q [7:0];
4
5   assign wireBundle = variableArray;
6   initial begin
7     ...
8     q = wireBundle;
9   end
```

В строке 5 показано, как массив variableArray может устанавливать значение всей цепи wireBundle. В строках 6–9 демонстрируется загрузка значений цепи в переменную q с такой же структурой. Пользуясь индексами, можно присвоить значения и частям массива.

# 12.3. Выбор части и конкатенация

Выбор части векторной переменной (или цепи) позволяет использовать в операции не все биты, а только указанный диапазон. Если диапазон состоит из одного элемента, говорят о выборе бита; если из нескольких – о выборе части. Если нужно выбрать биты из нескольких переменных, используется конкатенация.

## 12.3.1. Выбор бита и выбор части

Пусть вектор i определен следующим образом:

```
logic [15:0] i;
```

тогда выбор третьего бита i записывается в виде i[3]. К этому биту можно обращаться индивидуально:

```
i[3] = 1'b1;
```

или

```
j = i[3];
```

где j – скаляр.

При выборе части указывается диапазон битов переменной. Пусть имеются следующие определения:

```
1   logic [15:0] i;
2   logic [4:0] j;
```

Тогда можно написать:

```
3   j = i[11:7];
4   i[7:3] = j;
```

Номера битов, 11 и 7 в строке 3, должны быть константами. В общем случае выбор части имеет вид:

```
someVector[msb : lsb]
```

где msb – старший, а lsb – младший бит диапазона.

Индексная форма выбора части позволяет осуществлять выбор части вектора, используя переменную для задания начального индекса. Существует два варианта такой формы выбора:

```
someVector[lsb_base +: width]
someVector[msb_base -: width]
```

В обоих случаях ширина width должна быть константой, но lsb_base и msb_base могут быть переменными, т. е. выбираемая часть вектора будет фиксированной ширины, но может находиться в разных местах вектора. Отметим, что в первом варианте указывается индекс младшего бита, в состав части входит он сам и биты с большими индексами (отсюда +), а во втором – индекс старшего бита, в состав части входит он сам и биты с меньшими индексами (отсюда –).

Рассмотрим следующие определения для системы, в которой мы хотим обращаться к 4-битным отрезкам более широких векторов i и j:

```
1   logic [15:0] i, j;
2   logic [3:0] m, n;
```

В предположении, что начальные значения i и j равны 0, присваивания

```
3   i[2*4 +: 4] = 4'hF;
4   j[2*4 -: 4] = 4'hF;
```

дадут:

```
5   i = 16'b0000_1111_0000_0000
6   j = 16'b0000_0001_1110_0000
```

Объясним, как понимать результаты в строках 5 и 6. Учитывая, что номера битов векторов i и j принадлежат диапазону [15:0], в строке 3 младший бит 4'hF записывается в 8-й справа бит и в три бита с большими индексами. А в строке 4 старший бит 4'hF записывается в 8-й справа бит и в три бита с меньшими индексами.

В этих примерах номер начального бита был задан константой 2*4. Его можно также задать в виде index*4, где index – переменная, например:

```
i[index*4 +: 4] = 4'hF;
```

В этом случае если индекс принимает значения от 0 до 3, то мы будем обращаться к полубайтам векторов i и j, начинающимся в позициях 0, 4, 8 и 12.

По-другому индексацию в строке 4 выше можно записать в виде:

```
j[(index*4-1) -: 4] = 4'hF;
```

Если, например, index равен 2, то будет производиться обращение к битам [7:4].

Для векторов, имеющих нумерацию битов вида [0:15], тоже может использоваться индексная форма выбора части.

## 12.3.2. Конкатенация и репликация

Конкатенация позволяет группировать переменные в один элемент. Пусть имеются следующие определения:

```
1   logic [3:0] a, b;
2   logic [5:0] c;
3   logic [13:0] f, g;
```

Их можно конкатенировать с помощью оператора {} и создать новый элемент. Результат конкатенации первых трех переменных можно присвоить переменной f:

```
f = { a, c, b };
```

В этом случае последовательность битов f имеет вид aaaaccccccbbbb, точнее:

```
f = {a[3], a[2], a[1], a[0], c[5], c[4], c[3], c[2],c[1], c[0], b[3], b[2], b[1], b[0]};
```

Элементами конкатенации могут быть константы с указанием длины и результаты выбора части:

```
g = { a, c[3:0], b, 3'b10 };
```

Это то же самое, что:

```
g = a[3], a[2], a[1], a[0], c[3], c[2],c[1], c[0], b[3], b[2], b[1], b[0], 1'b1, 1'b0 };
```

Можно также выбрать часть конкатенации. Так,

```
{ a, c, b } [5:2]
```

не что иное, как:

```
{ c[1], c[0], b[3], b[2]}.
```

Конкатенация может находиться как в левой, так и в правой части присваивания. Вот пример использования конкатенации в левой части:

```
1   bit [7:0] a, b, sum;
2   bit carryIn, carryOut;
3
4   assign {carryOut, sum} = a + b + carryIn;
```

В этом примере младшие восемь бит результата в правой части присваиваются переменной sum, а следующий (более старший) бит – переменной carryOut.

Оператор репликации задает повторяющуюся конкатенацию элемента с самим собой. Для этого перед открывающей фигурной скобкой следует указать константу. Это может выглядеть как

```
{ 3 { a }}
```

что эквивалентно:

```
{ a, a, a} .
```

Конструкция может быть и посложнее:

```
parameter R = 2;
{ a, b, R {c, d}} .
```

Это то же самое, что { a, b, c, d, c, d }. Параметр репликации (R в этом примере и 3 в предыдущем) должен быть целой константой на этапе компиляции.

# 12.4. Модули, порты и экземпляры модулей

Модули – основная организационная единица в SystemVerilog. Они инкапсулируют части модели, давая возможность тестировать ее и создавать экземпляры этих частей в других модулях. Это позволяет создавать *иерархию модулей* – когда экземпляры одних модулей созданы в других модулях. Так создаются более крупные модули, которые, в свою очередь, становятся деталями еще более крупных.

Ключевой момент заключается в том, что с помощью модулей можно контролировать сложность модели. Крупные схемы (один большой модуль) могут состоять из большого числа мелких модулей, которые проектируются и тестируются по отдельности. Программа состоит из небольших функций; аналогичным образом логическая схема состоит из небольших модулей.

## 12.4.1. Модули и их экземпляры

С модулями можно делать две вещи: определять их и создавать экземпляр. Рассмотрим далее пример некоторого обычного модуля. Его определение ограничено ключевыми словами `module` и `endmodule`. У него есть имя (adder), порты (заданные с помощью `input` и `output`) и внутренние элементы (например, в строке 5).

```
1   module adder
2     (output logic sum,
3      input logic a, b, cIn);
4
5     assign sum = a + b + cIn;
6   endmodule: adder
```

У этого модуля есть один выходной порт `sum` и три входных – `a`, `b` и `cIn`. Будучи соединены, эти порты станут источником входных данных для оператора `assign` и позволят довести результат работы `assign` до других элементов модели, которые находятся вне модуля. Модуль определяет пространство имен, поэтому имена портов и других определенных внутри модуля переменных неизвестны за его пределами.

Модель в прямом смысле растет благодаря созданию экземпляров (копий) модулей внутри других модулей. Экземпляр модуля `adder` мог бы быть создан, например, так:

```
1   module datapath
2     logic f, g, h, i;
3
4     adder a1 (f, g, h, i);
4   endmodule: datapath
```

Здесь создается экземпляр модуля `adder` внутри модуля `datapath`, и переменные `f`–`i` соединяются с его портами. Таким образом, `sum` будет соединена с `f`, `a` с `g`, `b` с `h`, `cIn` с `i`.

Если экземпляр модуля не создан внутри какого-либо другого модуля, то автоматически создается его единственный экземпляр. Полные имена элемен-

тов, созданных внутри модуля, начинаются с имени этого модуля. Обсуждение модулей и их иерархии см. также в разделе 1.3.

Если бы мы захотели повторно использовать модуль adder для создания сумматоров разной разрядности, то могли бы параметризовать модуль. *Параметры* – это константы времени компиляции (обсуждение и примеры параметров см. в разделе 2.4). Определить параметризованный модуль adder можно, например, так:

```
1   module adderP
2     #(parameter Width = 1)
3     (output logic [Width-1:0] sum,
4      input logic [Width-1:0] a, b,
5      input logic cIn);
6
7     assign sum = a + b + cIn;
8   endmodule: adderP
```

Здесь определен модуль с параметром Width, который указан равным 1. Это так называемое *общее значение* или *значение по умолчанию*. В таком виде модуль функционально не отличается от написанного ранее модуля adder. Заметьте, однако, что в объявления портов a, b и sum входит константа Width. При создании экземпляра модуля параметр можно переопределить, указав другую разрядность сумматора. Пример создания экземпляра adderP приведен ниже:

```
1   logic [7:0] f, g, h;
2   logic i;
3   adderP #(8) biggerAdd (f, g, h, i);
```

В этом примере при создании экземпляра adderP для параметра Width указано значение 8, поэтому разрядность сумматора будет равна 8. Если бы мы опустили конструкцию #(8) в строке 3, то получился бы одноразрядный сумматор.

У модуля может быть и несколько параметров, как показано в следующем примере:

```
1   module paramDemo
2     #(parameter A = 1,
3                 B = 75,
4       parameter [3:0] C = 3'b010)
5   ...
```

Здесь определены три параметра со значениями по умолчанию. При создании экземпляра значения параметров можно переопределить:

```
paramDemo #(7, 13, 3'b110) p1 (… ports);
```

У экземпляра p1 модуля paramDemo параметр A будет равен 7, B – 13, C – 3'b110.

В рассмотренных примерах параметры задавались по порядку. Но, как и порты модулей, их можно задавать по имени (см. раздел 12.4.2). Например, экземпляр модуля paramDemo можно создать и так:

```
paramDemo #(.B(66)) p2 (... ports);
```

Здесь переопределено значение только параметра B, теперь оно равно 66;

у A и C остались значения по умолчанию. Если параметры задаются по имени, то перечислять их можно в любом порядке.

Для задания локальных констант внутри модуля используется ключевое слово localparameter. Локальные параметры указываются после объявления портов:

```
1   module Lparam
2     #(... ) // параметры
3     (... ); //порты
4     localparam Q = 17;
5   ...
```

В примере определяется константа Q, которая может быть использована внутри модуля. Q не может быть переопределена непосредственно в момент создания экземпляра, если только ее значение не зависит от одного из параметров, заданных в строке 2, например:

```
1   module anotherParam
2     #(parameter R = 3)
3     (... ); // порты
4     localparam Q = R + 17;
5   ...
```

В данном случае Q определяется через R, поэтому ее значение изменится, если переопределить R при создании экземпляра anotherParam.

При проектировании больших систем модель часто разделяется на несколько частей, компилируемых по отдельности. Для обеспечения отдельной компиляции можно объявить модуль, внешний по отношению к данной части, с помощью ключевого слова extern. В нем объявляются только порты и параметры. Предположим, что модуль adderP находится в другой части и компилируется отдельно. Тогда в компилируемой в данный момент части мы могли бы написать:

```
1   extern module adderP
2     #(parameter Width = 1)
3     (output logic [Width-1:0] sum,
4      input logic [Width-1:0] a, b,
5      input logic cIn);
```

При компиляции текущей части мы могли бы создать экземпляр этого модуля, как обычно.

## 12.4.2. Порты модулей

Порт модуля определяет соединение между двумя наборами элементов: один набор находится в создаваемом модуле, другой – в создающем. Синтаксически есть два способа описать соединения: *упорядоченный* и *именованный список портов*. В каждом конкретном случае создания экземпляра модуля используется только один способ, их нельзя использовать вместе. (Обсуждение этой темы и примеры портов см. также в разделе 2.5.)

В случае упорядоченного списка порты экземпляра должны перечисляться точно в том порядке, в каком они перечислены в определении модуля. Так,

в примере модулей adder и datapath, который для удобства повторен ниже, переменная f, определенная в строке 9, соединена (строка 11) с первым портом модуля adder (т. е. sum), переменная g – со вторым (a), h – с третьим (b) и i – с четвертым (cIn).

```
1  module adder
2    (output logic sum,
3     input logic a, b, cIn);
4
5    assign sum = a + b + cIn;
6  endmodule: adder
7
8  module datapath
9    logic f, g, h, i;
10
11   adder a1 (f, g, h, i);
12 endmodule: datapath
```

Соединение портов может осуществляться и по имени, с помощью *именованного списка портов*. В этом случае соединяемые элементы образуют пары, как показано ниже на примере модуля datapath.

```
1   module datapath
2     logic f, g, h, i;
3
4     adder a1 (.sum(f), .b(h), .a(g), .cIn(i));
5   endmodule: datapath
```

Здесь устанавливаются те же соединения, что в предыдущем примере. Но поскольку задаются сразу обе стороны соединения, перечислять пары можно в любом порядке. Например, h и g переставлены местами.

Если имена портов совпадают с именами соединяемых с ними элементов, то применима сокращенная нотация. В показанном ниже модуле newDatapath переменные названы так же, как порты модуля adder. Нотация .* в строке 4 означает, что для каждого имени порта в создаваемом экземпляре модуля компилятор будет искать в создающем модуле переменную с таким же именем, соединяя попарно эти порты и переменные.

```
1   module newDatapath
2     logic sum, cIn, a, b;
3
4     adder a1 ( .* );
5   endmodule: newDatapath
```

Такой подход, безусловно, позволяет избежать опечаток, но в этом случае код не является самодокументированным. Альтернативный подход – перечисление переменных, соответствующих портам:

```
1   adder a1 ( .sum, .cIn, .b, .a );
```

Здесь мы указали имена портов модуля, экземпляр которого создается. Воспользовавшись нотацией .name, мы документировали соединения и можем

не уточнять, что с чем соединено, поскольку предполагается, что имена одинаковы.

Наконец, если имена некоторых портов совпадают с именами переменных в создающем модуле, а остальных – нет, то можно соединить различающиеся элементы по именам, а остальные – с помощью нотации .*.

Рост и усложнение систем приводят к использованию стандартных предопределенных интерфейсов, гарантирующих, что модуль правильно взаимодействует с другими модулями, использующими тот же интерфейс. В примере 12.1 показано использование интерфейса в качестве типа порта модуля.

```
1  interface wazoo
2   (input bit clk);
3   bit dataReady;
4   bit [7:0] dataOutTheWazoo;
5  endinterface: wazoo
6
7  module listener (wazoo c);
8   bit [7:0] myCopy;
9
10  always_ff @(posedge c.clk)
11    if (c.dataReady)
12      myCopy <= c.dataOutTheWazoo;
13  ...
14 endmodule: listener
15
16 module top;
17  bit clk;
18
19  wazoo wz (clk);
20  listener l (wz);
21 endmodule: top
```

**Пример 12.1.** Порт интерфейсного типа

В строках 1–5 определен интерфейс wazoo. У него есть вход clk (строка 2) и две переменные типа bit (строки 3–4). В модуле top (строки 16–21) создается экземпляр интерфейса wazoo с именем wz (строка 19). При создании экземпляра модуля listener в строке 20 в качестве порта указывается имя экземпляра интерфейса: в определении модуля listener задан порт с типа wazoo. При такой спецификации все элементы интерфейса доступны по иерархическим именам (например, к clk можно обратиться по имени c.clk). Следовательно, модуль listener может обратиться к вектору dataOutTheWazoo.

Другие примеры интерфейсов SystemVerilog см. в разделе 6.5.

## 12.5. Генерация моделей

Конструкция generate создает внутри модуля несколько блоков, называемых *generate-блоками*. Создание производится на этапе компиляции с применением циклов for и условных операторов (if-else, case). Затем generate-блоки компилируются вместе с остальной моделью. Generate-блоки могут содержать операторы assign и создавать экземпляры модулей.

Для иллюстрации возможностей конструкции generate мы спроектируем многобитный сумматор. Один из возможных подходов – создать несколько экземпляров модуля fullAdd из примера 12.2. Для этого мы должны аккуратно выбрать имена и соединить сигналы переноса между разрядами. Но поскольку это итеративный процесс, на помощь приходит конструкция generate.

```
1  module fullAdd
2    (output logic sum, cOut,
3     input logic a, b, cIn);
4
5    always_comb begin
6      sum = a ^ b ^ cIn;
7      cOut = a & b | b & cIn | a & cIn;
8    end
9  endmodule: fullAdd
```

**Пример 12.2.** Модуль fullAdd

У нашего многобитного сумматора (пример 12.3) разрядность является параметром, а также у него есть статусные выходы, показывающие, как была исполнена операция: оказался ли результат равным 0 или отрицательным, был ли перенос из старшего разряда и имело ли место переполнение.

```
1 module adderGenMod
2   #(parameter Width = 8)
3   (output logic [Width-1:0] sum,
4    output logic cOut, neg, ovf, z,
5    input logic [Width-1:0] a, b,
6    input logic cIn);
7
8   logic [Width-1:0] c;
9
10  generate
11    for (genvar i = 0; i < Width; i = i + 1) begin: adder
12      case (i)
13        0: begin
14          fullAdd y (sum[i], c[i], a[i], b[i], cIn);
15        end
16        Width-1: begin
17          fullAdd y (sum[i], cOut, a[i], b[i], c[i-1]);
18          assign neg = sum[i];
19          assign ovf = cOut ^ c[i-1];
20          assign z = ~(|sum);
21        end
22        default: begin
23          fullAdd y (sum[i], c[i], a[i], b[i], c[i-1]);
24        end
25      endcase
26    end
27  endgenerate
28 endmodule: adderGenMod
```

**Пример 12.3.** Генерация экземпляров модуля

Ключевые слова `generate` и `endgenerate` в строках 10 и 27 определяют *область генерации*. Строго говоря, эти слова необязательны – ведь цикл `for` (строка 11) находится вне процедурных операторов, например `initial`, и этого достаточно – но мы все-таки оставили их для лучшего понимания модели.

В цикле `for` (строки 11–26) объявлена переменная генерации (`genvar`), которая служит для обхода диапазона чисел, в данном случае от 0 до значения параметра `Width`, определяющего разрядность сумматора. Можно считать, что `genvar` – целое число, задающее выбор бита и частный выбор в процессе соединения сгенерированных блоков между собой. Внутри цикла `for` находится оператор `case`, варианты которого зависят от `genvar`.

В операторе `case` три варианта: если `i = 0` (младший бит), если `i = Width-1` (старший бит) и для всех остальных `i` (ветвь `default`). В каждом варианте имеется *блок генерации*.

На этапе компиляции цикл for исполняется, и для каждого i в модели создается блок генерации, в котором вместо i подставлено текущее значение genvar. Таким образом, на первой итерации цикла, когда i = 0, создается строка:

```
1   fullAdd y (sum[0], c[0], a[0], b[0], cIn);
```

Читать ее нужно следующим образом: sum[0] – сумма на выходе нулевого разряда, а c[0] – выходной бит переноса из того же разряда. Они вычислены по значениям, поданным на входы a и b вычисления данного разряда, и входному биту переноса cIn.

На следующей итерации цикла i равно 1, и для создания блока генерации используется ветвь default (строка 23). В результате получается такой блок:

```
2   fullAdd y (sum[1], c[1], a[1], b[1], c[0]);
```

Отметим, что в этом разряде сумматора используется выходной бит переноса из предыдущего разряда. В оригинальном описании он представлен в виде c[i-1], а в процессе генерации заменен на c[0].

Генерация продолжается, пока i не станет равно Width-1. В этот момент в модель добавляется результат исполнения строк 17–20, в которые вместо i подставлено значение Width=8:

```
3   fullAdd y (sum[7], cOut, a[7], b[7], c[6]);
4
5   assign neg = sum[7];
6   assign ovf = cOut ^ c[6];
7   assign z = ~(|sum);
```

Все блоки генерации включаются в финальную модель и компилируются вместе с остальным описанием. Цикл for, в котором были созданы блоки генерации, завершен и далее в исполнении модели участия не принимает.

Вот как выглядит сгенерированный код:

```
1   fullAdd adder[0].y (sum[0], c[0], a[0], b[0], cIn);
2   fullAdd adder[1].y (sum[1], c[1], a[1], b[1], c[0]);
3   fullAdd adder[2].y (sum[2], c[2], a[2], b[2], c[1]);
4   fullAdd adder[3].y (sum[3], c[3], a[3], b[3], c[2]);
5   fullAdd adder[4].y (sum[4], c[4], a[4], b[4], c[3]);
6   fullAdd adder[5].y (sum[5], c[5], a[5], b[5], c[4]);
7   fullAdd adder[6].y (sum[6], c[6], a[6], b[6], c[5]);
8   fullAdd adder[7].y (sum[7], cOut, a[7], b[7], c[6]);
9   assign neg = sum[7];
10  assign ovf = cOut ^ c[6];
11  assign z = ~(|sum);
```

В сгенерированных операторах экземпляры модулей называются не просто y, как в примере 12.3, а содержат префикс adder[i], который был автоматически сгенерирован по имени, заданному в блоке begin-end цикла (строка 11). К переменным внутри созданных экземпляров модуля можно обратиться по иерархическим именам, начинающимся с показанных выше имен экземпляров.

# Глава 13

# Массивы

В этой главе мы рассмотрим основные типы массивов, используемые в моделях, а также более сложные типы, применяемые в тестовых окружениях.

## 13.1. Массивы

Основные типы данных – `bit` и `logic` – вводят *скалярные* (1-битные) переменные. Из них можно создавать многомерные массивы:

```
1   bit s1;
2   bit [5:0] v1;
3   bit s2[4];
4   bit [5:0] v2[8];
```

В строке 1 объявлена скалярная переменная `s1`. Далее показаны два подхода к построению массивов большей размерности. В строке 2 мы видим массив из шести скалярных элементов, называемый *вектором*, или *упакованным массивом*. Слева от имени указан диапазон индексов массива. Читать и записывать можно как массив целиком[66], так и его отдельные биты или части.

Второй подход – объявление *неупакованного массива* – показан в строке 3, где мы видим массив из четырех скаляров. Слово «неупакованный» означает, что массив представляется не одним битовым вектором (как в случае упакованного массива), а совокупностью заданного числа элементов. Доступ к элементу производится по его индексу. Справа от имени массива задается *размерность* – либо размер (это как показано в строке 3), либо диапазон индексов (например, [3:0]).

В строке 4 размерности указаны слева и справа от имени: в ней объявляется неупакованный массив из восьми 6-битных векторов. Размерность слева относится к упакованному массиву (вектору), а справа – к неупакованному. Число размерностей того и другого вида не ограничено.

Упакованные массивы могут быть составлены из одиночных битов (типов `bit` и `logic`), элементов перечислений, а также (рекурсивно) других упакован-

---

[66] Возможность работы с массивом как с единой многобитной переменной (вектором) и выражает свойство «упакованности».

ных массивов и упакованных структур. Целочисленные типы (byte, shortint, int, longint, integer и time) не могут быть типами элементов упакованных массивов[67].

На тип элементов неупакованных массивов не налагается никаких ограничений.

## 13.1.1. Многомерные массивы

Теперь посмотрим, как хранятся упакованные и неупакованные массивы и как следует обращаться к их элементам.

```
bit [3:0] [7:0] mDimArray;
```

Здесь объявлен упакованный массив из четырех 8-битных элементов. Поскольку все они упакованы, то его можно представлять 32-битным вектором, к отдельным полям которого можно обратиться, как это будет описано ниже. Этот массив можно объявить и сразу же инициализировать следующим образом:

```
bit [3:0] [7:0] mDimArray = {8'd3, 8'd22, 8'd10, 8'd5}
```

Раз это упакованный 32-битный массив, то для инициализации можно конкатенировать его подполя. Элемент 3 будет иметь значение 3, а элемент 0 – значение 5.

Рассмотрим еще один пример.

```
1  logic [3:0] a, b, c, d;
2  logic [3:0] [3:0] aa;
3
4  initial begin
5  a = 4'b01xz;
6  b = 4'b1111;
7  c = 4'xxxx;
8  d = 4'bzzzz;
9
10 aa[0] = a;
11 aa[1] = b;
12 aa[2] = c;
13 aa[3] = d;
14
15 $display("a=%b, b=%b, c=%b, d=%b, aa=%b", a, b, c, d, aa);
```

В строке 1 определены четыре 4-битных полубайта, а в строках 5–8 им присваиваются значения. В строке 2 определен упакованный массив aa из четырех полубайтов, а в строках 10–13 его элементам также присвоены значения. После этого $display выводит на консоль такую строку:

```
a=01xz, b=1111, c=xxxx, d=zzzz, aa=zzzzxxxx1111101xz
```

При выводе на консоль значения переменной aa выводится весь вектор; из вывода видно, что aa[3] содержит старшие биты, а aa[0] – младшие.

---

[67] Целочисленные типы данных представляют собой предопределенные упакованные массивы. Например, тип integer – это logic signed [31:0].

Наличие дополнительного измерения в объявлении aa открывает возможность доступа к подполям. Ниже показан вызов системной процедуры $display и результат вывода на консоль:

```
16 $display("aa[2]=%b, aa[2:1]=%b, aa[3][2:1]=%b", aa[2], aa[2:1], aa[3][2:1]);
```

```
aa[2]=xxxx, aa[2:1]=xxxx1111, aa[3][2:1]=zz
```

Первые два напечатанных элемента иллюстрируют использование индекса, а не выбора части, для вывода на консоль одного и двух последовательных элементов упакованного массива. В третьем же показано, как обратиться к части третьего элемента массива.

Продолжим знакомство и определим трехмерный упакованный массив aaa, показанный в строке 17 ниже.

```
17 logic [1:0] [3:0] [3:0] aaa;
18 aaa[1] = aa;
19 aaa[0][2] = b;
20 aaa[0][1] = d;
21
22 $display("aaa=%b, aaa[0][1][2:1]=%b", aaa, aaa[0][1][2:1]);
```

Здесь определен массив из двух 4-элементных 4-битных векторов, и некоторым элементам этого массива присвоены значения. Можно сказать, что основной единицей в этом примере является полубайт: объявлены два массива, каждый из которых содержит четыре таких полубайта. Результаты вывода на консоль показаны в следующей строке.

```
aaa=zzzzxxxx111101xzxxxx1111zzzzxxxx, aaa[0][1][2:1]=zz
```

Элемент aaa[1] представлен старшими битами выведенной строки, а элемент aaa[0] – младшими. Также показан результат выбора части массива: биты, входящие в заданный диапазон, в выводе aaa подчеркнуты.

Еще один способ организовать подобный массив – воспользоваться упакованной структурой:

```
23 struct packed {
24   logic [3:0] e;
25   logic [2:0] f;
26   logic g;
27 } st;
```

Поскольку структура упакована, все ее элементы на самом деле составляют единый 8-битный вектор. Однако для большей наглядности кода мы сопоставили частям вектора имена. Ниже разным элементам структуры присваиваются значения, а затем эти элементы выводятся на консоль.

```
28 st.e = b;
29 st.f = a[3:1];
30 st.g = 1'b0;
31
```

```
32 $display("st=%b, st.f=%b", st, st.f);

    st=111101x0, st.f=01x
```

Сначала выводится вся структура в виде упакованного вектора, содержащего сразу все элементы, а затем – только один элемент.

Наконец, приведем пример неупакованного массива cc, состоящего из двух элементов. Каждый из них представляет собой массив из четырех 4-битных векторов. В следующем фрагменте сначала определяется cc, а затем некоторым его элементам присваиваются значения.

```
33 logic [3:0] [3:0] cc[2];
34 cc[0] = aa;
35 cc[1] [2] = a;

36 cc[1] [1] [1] = 1'bz;
37 $display("cc[0]=%b, cc[1]=%b", cc[0], cc[1]);
```

В строке 34 в элемент 0 неупакованного массива записывается значение aa. В строке 35 в элемент 2 массива, находящегося в измерении 1 массива cc, записывается вектор a. Отметим, что при доступе к неупакованным массивам, у которых имеются упакованные измерения, как в случае cc, неупакованные измерения указываются сначала. В строке 36 с помощью конструкции выбора битов записывается ровно один бит. Значения, записанные в строках 35–36, подчеркнуты в следующем результате вывода на консоль:

```
cc[0]=zzzzxxxx111101xz, cc[1]=xxxx01xzxxzxxxxx
```

У переменных тоже может быть несколько неупакованных измерений[68].

## 13.1.2. Упакованные и неупакованные массивы

Смысл «упаковки» массива раскрывается в логическом синтезе. С упакованным массивом можно работать как с единым вектором. Для доступа к данным можно использовать операции выбора бита или части, а также операции доступа к полям структуры, как показано в предыдущем разделе. Что касается неупакованного массива, за одну операцию можно обратиться только к одному элементу (для этого используются индексы).

## 13.1.3. Операции над массивами

Упакованные и неупакованные массивы могут быть частью операций пересылки и сравнения. В предположении, что размеры массивов A и B и тип их элементов одинаковые, например:

```
bit [7:0] A[1024], B[1024];
```

допустимы следующие операции:

---

[68] Например, `logic a[M][N]` определяет неупакованный массив a из M элементов, каждый из которых является неупакованным массивом N элементов типа logic.

- чтение и запись элемента массива: `A[i] = B[i];`
- чтение и запись всего массива: `A = B;`
- чтение и запись среза массива: `A[i:j] = B[m:n]`. Количество элементов в диапазонах `[i:j]` и `[m:n]` должно быть одинаково;
- чтение и запись переменного среза массива: `A[y+:c] = B[z+:c];`
- операции сравнения массива или его среза на равенство или неравенство: `A != B, A[i:j] == B[m:n]`. Количество элементов в диапазонах `[i:j]` и `[m:n]` должно быть одинаково.

Такие операции полезны в тестовом окружении, где, например, один массив может быть ожидаемым ответом, а другой – фактическим. Для их сравнения достаточно одного выражения: `(A == B)`.

## 13.2. Динамические массивы

Динамическим называется массив, размер которого может изменяться во время исполнения. Тип элементов массива может быть любым, а объявление выглядит так:

```
1   bit [7:0] myDyn [];
```

Здесь объявлен динамический массив 8-битных векторов. В момент объявления размер массива равен 0, но его можно изменить, вызвав конструктор `new`, а также с помощью присваивания другого массива. При вызове `new` задается размер массива и необязательное начальное значение.

```
2   myDyn = new[10];
```

Если все элементы массива должны быть инициализированы одним и тем же значением, то объявление выглядит так:

```
3   myDyn = new[10](5);
```

В данном случае все десять элементов получают значение 5. Вызов `new` и инициализацию можно совместить в одной строке:

```
    bit [7:0] myDyn = new[10](5);
```

Динамический массив можно также инициализировать другим массивом того же размера:

```
1   bit [7:0] myA[], myB[];
2   bit [7:0] myC[4] = '{10, 25, 40, 55};
3
4   myA = new[4] (myC);
5   myB = new[4] ('{10, 25, 40, 55});
```

В строке 2 объявлен неупакованный массив `myC`, состоящий из четырех элементов с начальными значениями 10, 25, 40 и 55. В строках 4 и 5 динамические массивы `myA` и `myB` инициализированы теми же значениями.

У динамических массивов являются доступными два метода:

- size возвращает целое число, равное размеру массива в момент его создания конструктором new. Если массив не создавался, то метод возвращает 0;
- delete опустошает массив, так что получается массив нулевого размера.

Отметим, что new можно вызывать для одного и того же динамического массива несколько раз. При каждом вызове данные, хранившиеся в массиве, теряются.

## 13.3. Строки

Тип данных string используется для хранения строк символов. Переменные типа string динамические, потому что их длина может изменяться в процессе исполнения тестового окружения. А раз они динамические, то при манипуляциях с ними можно не думать о выходе за пределы объявленной длины. Каждый символ строки имеет тип byte. В конце раздела будут перечислены методы, которые можно вызывать у строк.

Символы строки длины N индексируются числами от 0 до N-1. Первый (самый левый) символ имеет индекс 0. Для обозначения конца строки не применяется специальный символ \0, как в языке C.

Ниже показано, как объявить и вывести на консоль строковую переменную:

```
1   string msg = "No VHDL here";
2   $display("%s", msg);
```

Инициализация msg в строке 1 необязательна (значение по умолчанию – строка длины 0). Для инициализации строк не нужно вызывать конструктор new.

Для строк определены следующие операторы.

- str1 == str2 – проверка строк на равенство (или на неравенство с помощью оператора !=). Возвращает TRUE, если строки равны. Неравенство – это логическое отрицание равенства.
- str1 < str2 – сравнение строк. Возвращает TRUE, если первая строка меньше второй в лексикографическом смысле. Другие операторы сравнения: <=, >, >=.
- {str1, str2, … strN} – конкатенация. Результатом является одна строка.
- {N {str}} – репликация. Результатом является строка str, повторенная N раз. Число N необязательно должно быть константой.
- str[index] – индексирование. Возвращает ASCII-код символа в указанной позиции. Если index больше длины строк, то возвращается 0.
- str.method() – вызывает один из нижеперечисленных методов.

Для типа string определен ряд методов.

- len – возвращает длину строки.
- putc(int i, byte c) – поместить символ в строку. Эквивалентно оператору str[i] = c. Длина строки не изменяется.
- getc(int i) – получить символ строки. Операторы x = str.getc(i) и x = str[i] эквивалентны. Длина строки не изменяется.
- toupper и tolower – возвращают строку, в которой все символы преобразованы в верхний или нижний регистр соответственно.

- compare(s) и icompare(s) – возвращают TRUE, если строки одинаковы. При сравнении методом icompare() регистр не учитывается. Синтаксически сравнение строк str и s записывается в виде str.compare(s).
- substr (i, j) – возвращает новую строку, содержащую символы исходной с индексами от i до j включительно.
- atoi, atohex, atooct, atobin – преобразуют строку, содержащую цифры, в эквивалентное целое число в системе счисления с основанием 10, 16, 8 и 2 соответственно. В результате исполнения кода

```
string valToConvert = "2014";
int i = valToConvert.atoi();
```

строка "2014" будет преобразована в целое десятичное число i.

- atoreal – возвращает число с плавающей точкой, являющееся результатом разбора строки.
- itoa, hextoa, octtoa, bintoa – создают строковое представление в кодировке ASCII переданного целого числа в системе счисления с основанием 10, 16, 8 и 2 соответственно.
- realtoa(real r) – возвращает строку, содержащую ASCII-представление числа с плавающей точкой r.

## 13.4. Очереди

Очередь – это упорядоченный одномерный массив элементов, который автоматически растет и уменьшается. Каждому элементу очереди присвоен индекс в диапазоне от 0 до $ (обозначает последний элемент). Элементы можно вставлять и удалять с любой стороны очереди, а также в ее середине. Таким образом очередь оказывается очень гибкой структурой, и, в частности, ее можно использовать для реализации стека. Доступ к очереди не блокирует исполнение текущего процесса.

Для определения очереди в объявлении массива используется символ $.

```
1   bit[7:0] qq [$], bQ[$:19];
```

Здесь объявлена неограниченная очередь qq и ограниченная очередь bQ; в bQ может находиться не более 20 элементов с индексами от 0 до 19. Очереди можно присвоить значение:

```
2   qq = {3, 5, 7};
```

При таком присваивании все значения, ранее хранившиеся в очереди, теряются, а в нее записываются значения новые. Объявление и инициализацию очереди можно совместить в одной строке.

Для доступа к очереди применяются нижеперечисленные методы. Отметим также, что начало очереди имеет индекс 0, и если в очереди уже находится 5 элементов с индексами 0–4, то конец очереди будет находиться в позиции с индексом 4.

- size – возвращает количество элементов в очереди. Если очередь пуста, то возвращает 0.

- `push_front (item)` – помещает элемент в начало очереди (позицию с индексом 0), а ее размер увеличивается на 1.
- `push_back (item)` – помещает элемент в конец очереди, а ее размер увеличивается на 1.
- `pop_front` – возвращает элемент, находящийся в начале очереди, а ее размер уменьшается на 1.
- `pop_back` – возвращает элемент, находящийся в конце очереди, а ее размер уменьшается на 1.
- `insert (index, item)` – вставляет элемент `item` в позицию с индексом `index`, а размер очереди увеличивается на 1.
- `delete (index)` – удаляет элемент, находящийся в позиции с индексом index. Размер очереди уменьшается на 1. Если индекс не указан, то удаляются все элементы, и остается пустая очередь размера 0.

Если очередь ограничена, а в результате операции количество элементов оказывается больше предельно допустимого, то элементы с выходящими за объявленный диапазон индексами просто теряются.

## 13.5. Ассоциативные массивы

Ассоциативные массивы применяются в случаях, когда хранимая информация разрежена или индексы не являются целочисленными, например когда в качестве индексов используются строки (см. строку 2 в примере 13.1).

```
1  module associative_array;
2    logic [7:0] colors [string];
3    string myColor = "purple";
4    string i;
5
6    initial begin
7      colors["yellow"] = 8'b110010xz;
8      colors["red"] = 0;
9      colors["green"] = 3;
10     colors["red"]++;
11     colors[myColor] = 'h17;
12
13     if (colors.first(i))
14       do
15         $display("colors[%s] = %0h",
16                  i, msg[i]);
17       while (colors.next (i));
18   end
19 endmodule: associative_array
```

Помимо ассоциативного массива, в примере 13.1 объявлены две строковые переменные; начальное значение одной из них – "purple". В строках 7–11 блока `initial` показано, как можно добавлять и изменять элементы массива. Поскольку индексы имеют тип string, для индексирования можно использовать строковые константы (например, "yellow") или переменные (например, `myColor`). Элементa-

ми являются 8-битные векторы (см. строку 7). В строке 10 элемент с индексом "red" инкрементируется.

В строках 13–17 отображаются все значения, хранящиеся в ассоциативном массиве colors вместе с их индексами. Для этого используются методы, доступные в ассоциативных массивах. Метод first() в строке 13 возвращает TRUE, если в массиве имеется хотя бы один элемент, при этом в строковую переменную i помещается индекс первого элемента. Если массив не пуст, начинается цикл do-while и на консоль выводится индекс и значение, хранящееся по этому индексу. Цикл завершается, когда метод next() возвращает FALSE. Пока этого не произошло, метод next() помещает в переменную i индекс следующего элемента и выводит на консоль этот элемент. Вот результат этого вывода:

```
colors[green] = 3
colors[purple] = 17
colors[red] = 1
colors[yellow] = cX
```

Ассоциативные массивы не упакованы и могут содержать элементы любого типа, допустимого для статических массивов. Тип индекса может быть классом и типом, определенным с помощью typedef. Если при задании типа индекса указано [*], то массив индексируется целочисленными значениями произвольного размера. Значения индексов не должны содержать x и z.

Ниже перечислены методы, доступные в ассоциативных массивах.

- num и size – возвращают число элементов массива (0, если массив пуст).
- delete(index) – удаляет элемент с индексом index. Если индекс не указан, удаляются все элементы. Метод не возвращает значения.
- exists(index) – возвращает TRUE, если в массиве существует элемент с индексом index.
- first(ref index) – возвращает TRUE, если массив не пуст. В этом случае в index записывается индекс первого элемента массива.
- last(ref index) – возвращает TRUE, если массив не пуст. В этом случае в index записывается индекс последнего элемента массива.
- next(ref index) – возвращает TRUE, если в массиве имеется элемент, следующий за элементом с индексом index. Индекс этого элемента записывается в index.
- prev(ref index) – возвращает TRUE, если в массиве имеется элемент, предшествующий элементу с индексом index. Индекс этого элемента записывается в index.

Тип аргументов, передаваемых методам обхода массива (последние четыре), должен быть совместим с типом индекса.

Подробнее использование ассоциативных массивов в тестовом окружении рассматривается в разделе 8.4.

# Глава 14

# Работа симулятора

SystemVerilog – язык моделирования параллельных систем, который позволяет описать действия, осуществляемые параллельно. Исполнение SystemVerilog-моделей осуществляется в симуляторе – среде имитационного моделирования. Симулятор отслеживает время и планирует определенные события на определенное время. Именно благодаря понятию времени мы получаем возможность моделировать физические аспекты проектируемой системы, включая события, происходящие одновременно. А попробуйте такое сделать на языке программирования!

## 14.1. События, слоты времени и списки событий

Имитационная модель есть множество процессов (потоков исполнения). Процессы являются объектами (вообще говоря, с состоянием), которые реагируют на изменение значений входов изменением значений выходов. Для исполнения одних процессов требуется несколько единиц времени, другие исполняются мгновенно (в том смысле, что время при этом не продвигается вперед). В SystemVerilog процессы описываются такими конструкциями, как `initial`, `always`, `always_comb`, `always_ff` и `always_latch`. Непрерывное присваивание – тоже процесс.

Процессы конкурируют друг с другом: действия, осуществляемые в ответ на изменение значений (общих) входов и приводящие к изменению значений (общих) выходов, могут перекрываться во времени. Симулятор оперирует с событиями, определяющими, когда изменить значение того или иного выхода и когда исполнить тот или иной процесс. Итак, есть два типа событий:

- *событие обновления* – изменение значения переменной (событие хранится в списке событий; в нужный момент значение переменной обновляется);
- *событие исполнения* – запуск или возобновление исполнения процесса (событие хранится в списке событий; в нужный момент процесс запускается).

В симуляторе имеется иерархический список событий (см. рис. 14.1). На верхнем уровне находится структура данных, содержащая упорядоченные *слоты времени* ($t_i < t_j$). С каждым слотом времени связана структура данных, содержащая списки событий, запланированных на этот слот времени. В разных списках находятся события разных категорий.

Различают две основные категории событий: *обычные* (*regular*) и *неблокирующие* (*Non-Blocking – NB*). Неблокирующие события возникают в результате неблокирующих присваиваний (<=); обычные охватывают все остальные присваивания (=). На рис. 14.1 показано, что события, относящиеся к этим двум категориям, хранятся в разных списках для каждого слота времени.

Есть и третья категория событий – *отложенные*. К ним относятся события исполнения таких операций, как мониторы. Мы обсудим их ниже.

## 14.2. Цикл симуляции

В разделе 1.2.2 была представлена схема работы симулятора; мы повторили ее на рис. 14.2. Рисунок не отражает всего происходящего в симуляторе, но все равно полезен для знакомства с основными шагами обработки событий обновления и исполнения (без неблокирующих событий).

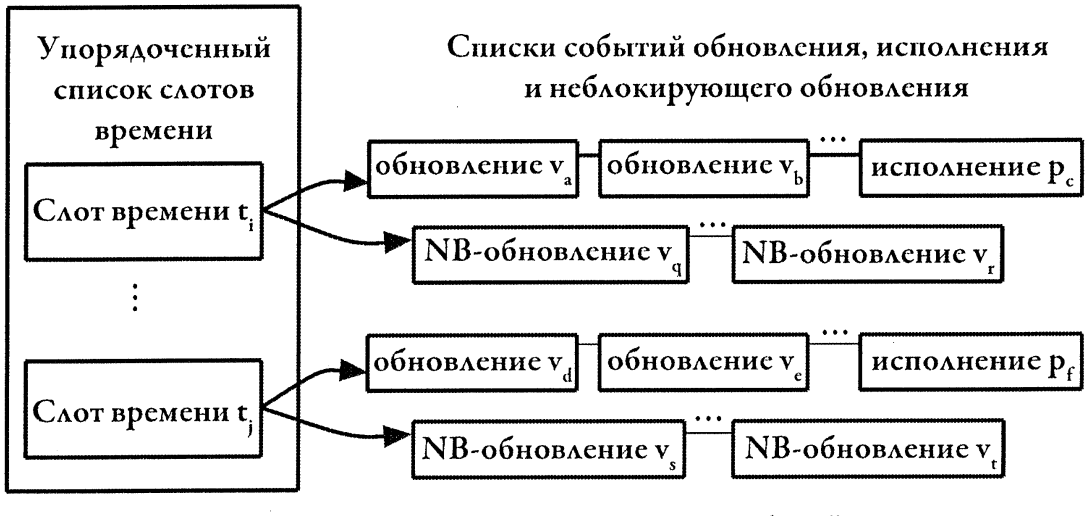



**Рис. 14.1.** Иерархический список событий

Слева в абстрактной форме показан упорядоченный по времени список событий, содержащий три слота времени. Работа симулятора в рамках одного слота времени (показана жирными стрелками) начинается с выборки обычных событий, включая события обновления и исполнения. Симулятор применяет все обновления, меняя тем самым состояние модели. После этого он следует вдоль выходных соединений обновленных переменных и ищет зависящие от них модели. Эти модели исполняются, как и процедурные модели, связанные с событиями исполнения. Это может породить новые события для текущего и будущих слотов времени (все они добавляются в список событий).

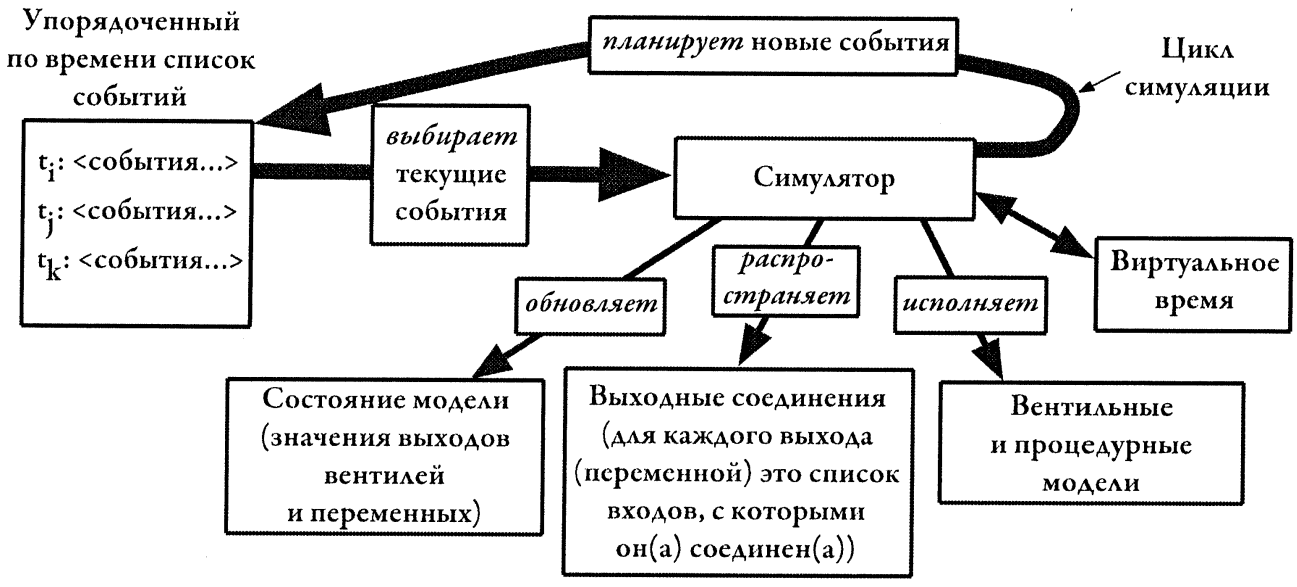



**Рис. 14.2.** Схема работы симулятора

Одна итерация этого циклического процесса (loop) называется *циклом симуляции* (*simulation cycle*). Поскольку при обработке событий могут быть порождены новые события, запланированные на тот же самый момент времени, для обработки событий, относящихся к одному времени, может потребоваться несколько циклов симуляции.

Но до сих пор мы учитывали только обычные события. Неблокирующие события хранятся в отдельном списке для каждого слота времени (см. рис. 14.1). Как показано на рис. 14.3, для каждого момента времени сначала исполняются циклы симуляции, необходимые для обработки обычных событий, в т. ч. вновь сгенерированных. Когда обычных событий больше нет, из списка извлекаются и исполняются все неблокирующие события. После этого симулятор распространяет значения по спискам соединений и исполняет вентильные и процедурные модели. В результате могут быть сгенерированы обычные и неблокирующие события для текущего и будущих слотов времени (все они добавляются в список событий).

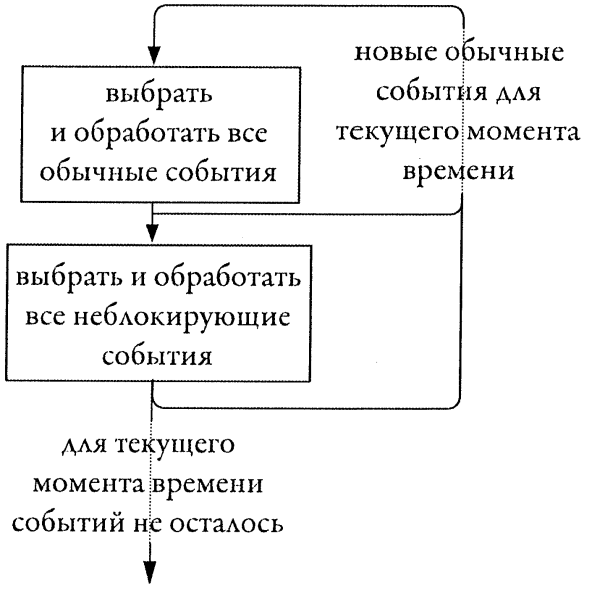



**Рис. 14.3.** Обработка обычных и неблокирующих событий

Таким образом, после того как обработаны все обычные события, относящиеся к текущему моменту времени, в отдельном цикле симуляции исполняются неблокирующие присваивания. Если на этом цикле будут запланированы новые события для того же самого момента времени, то сначала будут обработаны те из них, которые являются обычными (для этого может потребоваться несколько циклов симуляции), а затем – небло-

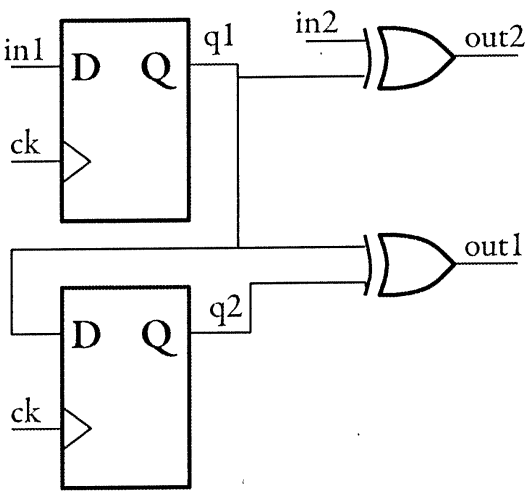



**Рис. 14.4.** Модель для демонстрации работы симулятора

кирующие. Обработка событий продолжается, пока не останется ни одного обычного или неблокирующего события.

Работа симулятора иллюстрируется на рис. 14.4 и 14.5: на первом рисунке приведена модель; на втором – действия симулятора по обработке событий модели. На логической схеме и в коде на SystemVerilog присутствуют два триггера и два вентиля XOR. Система описана как конечный автомат Мили. Рассмотрим ситуацию, изображенную на рис. 14.5, где в списке событий находятся два обычных события – события обновления входов `ck` и `in2`. Проследим, как симулятор обрабатывает эти события на протяжении нескольких циклов симуляции. Для простоты мы опустили новые значения сигналов и предполагаем, что они вызывают события обновления или исполнения.

Как было сказано выше, вначале список содержит два события обновления. На первом цикле производятся эти обновления, после чего симулятор распространяет новые значения по спискам соединений и исполняет встретившиеся модели. В данном случае по переднему фронту сигнала `ck` исполняются блоки `always_ff`; как результат обновления `q1` и `q2` добавляются в список неблокирующих событий. Кроме того, поскольку изменилось значение входа `in2`, планируется обычное событие обновления `out2`. Таким образом, после первого цикла симуляции список событий станет таким, как показано на рис. 14.5 (вторая от заголовка клетка первой строки).

На втором цикле опять выбираются обычные события; такое событие только одно – обновление `out2`. Поскольку `out2` – выход системы, его значение никуда не распространяется. На третьем цикле обычных событий нет, поэтому выбираются и обрабатываются неблокирующие события – присваивания в `q1` и `q2`. Новые значения сигналов распространяются: в список событий помещаются обычные события обновления для выходов `out1` и `out2`. Эти обновления осуществляются на следующем цикле симуляции. Больше событий нет.

Отметим, что на каждом цикле симуляции возможны разные порядки обработки событий, однако можно быть уверенными, что ни одно неблокирующее

| Список событий | reg: ck=1, in2<br>NB | reg: out2<br>NB: q1, q2 | reg:<br>NB: q1, q2 | reg: out1, out2<br>NB: | |
|---|---|---|---|---|---|
| Действия внутри цикла симуляции | | обновить ck, in2<br>запланировать q1, q2 в NB<br>запланировать out2 | обновить out2 | обновить q1, q2<br>запланировать<br>out1, out2 | обновить out1,<br>out2 |

**Рис. 14.5.** Список событий для модели на рис. 14.4

событие не будет обработано, прежде чем исчерпаются все исходные обычные события, а также обычные события, порожденные в процессе обработки исходных (и далее рекурсивно). Это дает возможность откладывать события обновления выходов триггеров до тех пор, пока не будут вычислены все выходные значения; таким образом, изменение значение выхода одного триггера не может повлиять на значение выхода другого триггера на том же фронте тактового сигнала. Все неблокирующие обновления, связанные с одним тактовым сигналом, происходят одновременно: вы никогда не увидите результат одного, не увидев результаты всех остальных.

## 14.3. Основной и реагирующий этапы

События каждого слота времени обрабатываются симулятором преимущественно на двух этапах (regions): основном (active) и реагирующем (reactive). Схема обработки событий, показанная на рис. 14.3, относилась к *основному этапу*. На этом этапе обрабатываются события модели, порождаемые в модулях: АЛУ, конечных автоматах, регистрах и т. д.

Для крупных моделей со сложными тестовыми окружениями полезно разделить события модели и события окружения. Это изолирует происходящее внутри модели от внешних взаимодействий с окружением и тем самым минимизирует гонки между внутренними и внешними процессами. Таким образом, в симулятор специально для тестовых окружений вводится еще один этап обработки – *реагирование*.

Как понять, на каком этапе обрабатываются те или иные события SystemVerilog-модели? Ответ дает конструкция `program`, представленная в разделе 7.2. Программы аналогичны модулям, но используются при создании тестовых окружений. События, генерируемые программами, обрабатываются на этапе реагирования.

Для обработки событий двух типов нужно расширить список событий. Рисунок 14.6 детализирует структуру данных, связанную с одним слотом времени. Можно видеть дополнительные списки событий для этапа реагирования. В них

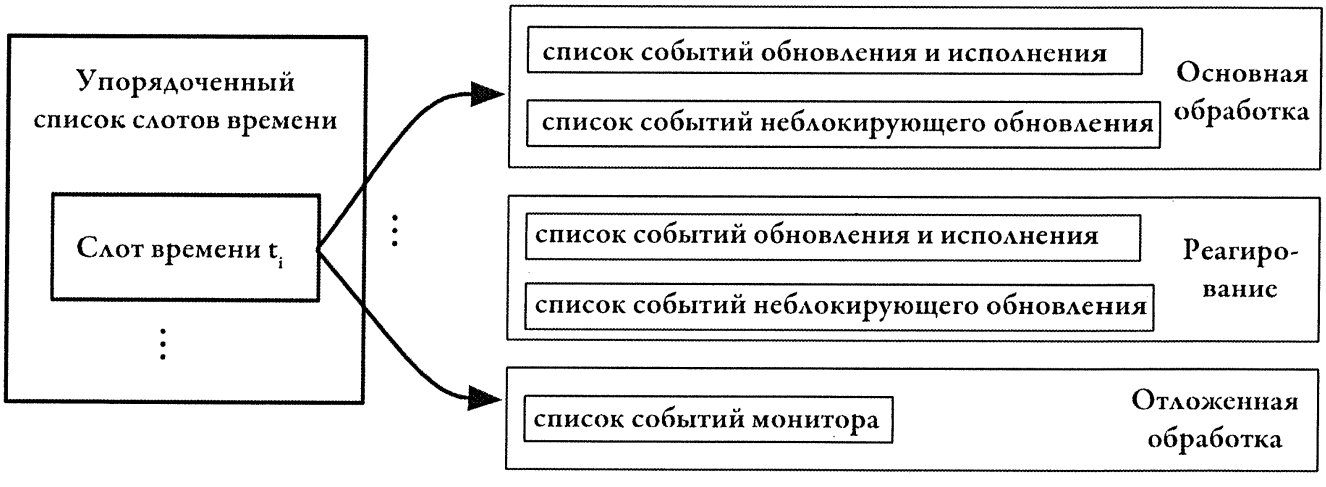



**Рис. 14.6.** Этапы обработки событий (регионы слота времени)

хранятся обычные и неблокирующие события, порожденные в программах тестового окружения.

Сначала симулятор обрабатывает все основные события (события, порожденные модулями), включая обычные и неблокирующие присваивания. Когда их не остается, обрабатываются все события реагирования (события, порожденные программами). Порядок обработки событий на этапе реагирования такой же, как на основном этапе (применяется блок-схема с рис. 14.3, но к спискам событий реагирования).

Рассмотрим пример 14.1, написанный с единственной целью – проиллюстрировать работу симулятора в случае, когда имеются события как на основном этапе, так и на этапе реагирования. В строках 29–31 показана конструкция `program` с одним блоком `initial`. Блок вызывает процедуру, определенную в строках 10–14, которая выводит значение на консоль и исполняет неблокирующее присваивание. Интересно, что этот блок `initial` исполняется только после обработки всех событий основного этапа, а все события, сгенерированные в этом блоке, планируются для исполнения на этапе реагирования.

```
1  module design;
2    byte newValue;
3    bit  lastReactive;
4
5    task calledFromDesign;
6      $display("Called By Design newValue=%3d",
7               newValue);
8    endtask
9
10   task calledFromProgram;
11     $display("Called By Program newValue=%3d",
12              newValue);
13     newValue <= 17;
14   endtask
15
16   always_comb
17     if (newValue == 17)
18       $display("Caused by reactive event");
19
20   initial begin
21     $monitor("Mon: newValue=%3d", newValue);
22     newValue = 8;
23     newValue <= 12;
24
25     calledFromDesign;
26   end
27 endmodule: design
```

```
28
29 program testbench;
30   initial design.calledFromProgram;
31 endprogram: testbench
```

**Пример 14.1.** Обработка событий основного и реагирующего этапов

Проследим за исполнением примера 14.1. Сначала исполняется блок initial модели (строки 20–26): в нем настраивается монитор, переменной newValue присваивается значение 8, планируется неблокирующее присваивание в ту же переменную значения 12, после чего вызывается процедура calledFromDesign. Эта процедура (строки 5–8) выводит на консоль значение newValue (8) и возвращает управление. Исполнение блока initial завершается.

В списке событий основного этапа остается только неблокирующее обновление переменной newValue. Это событие выбирается, и переменная получает значение 12. Симулятор переключается на события реагирования. Есть только одно событие этого типа – исполнение блока initial программы (строка 30). Вызванная из блока процедура calledFromProgram выводит на консоль текущее значение newValue (12) и планирует еще одно неблокирующее присваивание в эту переменную (на этот раз значения 17). Обычные события закончились; исполняются запланированные неблокирующие присваивания: в newValue записывается 17. Так как значение newValue изменилось, исполняется блок always_comb (строки 16–18), который печатает строку "Caused by reactive event". Все события обработаны; монитор печатает 17 – окончательное значение переменной newValue для текущего момента времени.

Таким образом, в результате исполнения этого примера на консоль будет выведено:

```
1   Called By Design newValue=  8
2   Called By Program newValue= 12
3   Caused by reactive event
4   Mon: newValue= 17
```

Подведем итоги. Первая строка выводится на консоль во время обработки обычных событий основного этапа, вторая – после исполнения неблокирующих присваиваний того же этапа, третья – на этапе реагирования после обработки события обновления newValue значением 17. Последнюю строку печатает монитор.

Смысл в том, что события модели (хранящиеся в списках событий основного этапа) обрабатываются раньше событий тестового окружения (хранящихся в списках событий этапа реагирования). При таком подходе тестовое окружение считывает выходные значения модели (используемые для направления тестирования в то или иное русло), когда они сформированы.

## 14.4. Блок-схема работы симулятора

Чтобы получить более полное представление о работе симулятора, объединим все, что мы уже знаем об этом, добавив пару деталей. На рис. 14.7 показана блок-схема работы симулятора для одного слота времени. Работа разбита на пять этапов (regions).

- *Предварительная обработка* – в этой точке значение времени уже обновлено, но обработка событий еще не началась. Здесь сэмплируются значения, используемые при проверке утверждений (см. главу 9).

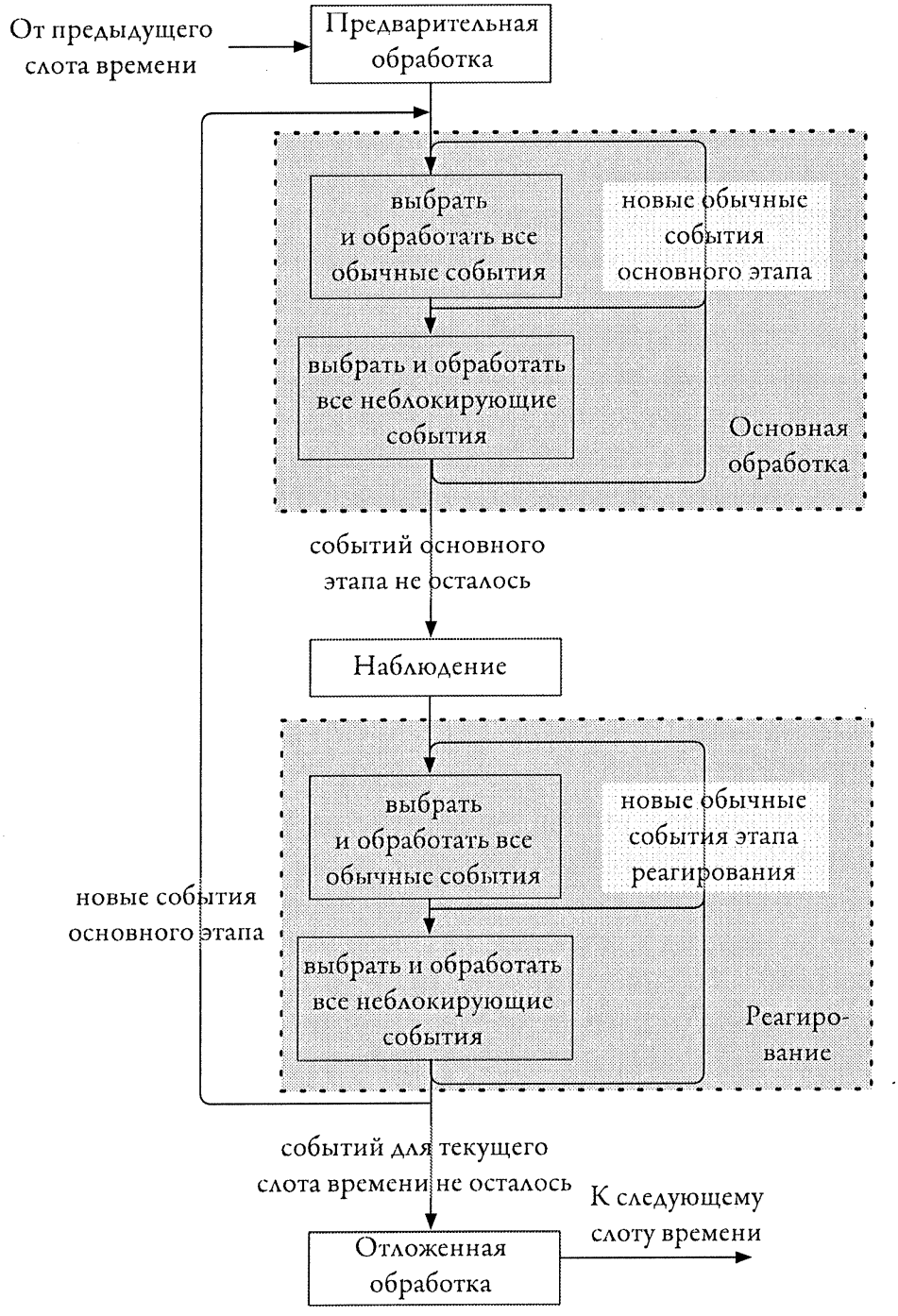



**Рис. 14.7.** Блок-схема работы симулятора и этапы обработки событий

- *Основная обработка* – этот этап мы только что обсудили (см. рис. 14.3). На нем обрабатываются все обычные и неблокирующие события из списка событий основного этапа текущего слота времени (это события, имеющие отношение к проектируемой модели).
- *Наблюдение* – на этом этапе проверяются утверждения (см. главу 9). Для этого используются значения модели, сэмплированные на этапе предварительной обработки.
- *Реагирование* – на этом этапе обрабатываются все обычные и неблокирующие события из списка событий этапа реагирования текущего слота времени (это события, имеющие отношение к блокам program).
- *Отложенная обработка* – на этом этапе обрабатываются события монитора. После того как события обработаны, симулятор обновляет время и переходит к предварительному этапу следующего слота.

Еще раз отметим, что в каждом слоте времени события модели обрабатываются раньше событий тестового окружения. Это решает две важные проблемы.

- *Реагирующее тестовое окружение.* Тестовое окружение всегда работает после обработки всех событий модели и проверки всех утверждений модели. Таким образом, у тестового окружения есть возможность оценить работу модели и, возможно, перейти к следующей фазе тестирования.
- *Гонки между тестовым окружением и моделью.* Такое упорядочение гарантирует отсутствие гонок между тестовым окружением и моделью. Рассмотрим, что могло бы случиться, если бы события тестового окружения и события модели не были разделены. Симулятор обрабатывает события в произвольном порядке, и события тестового окружения могли бы вклиниться в события модели и быть обработаны до того, как элементы модели обновили свои значения.

Следует признаться, что в работе симулятора есть и другие этапы, но они относятся к программному интерфейсу (Programming Language Interface – PLI), а эту тему мы здесь не рассматриваем.

# Предметный указатель

## P



## C

## T