Оглавление

Предисловие: об этой книге 13

Предисловие от издательства 15

Контекст: проектирование на уровне регистровых передач 16

Благодарности 18

Глава 1. Введение 19

1.1. Приступая к работе 20
  1.1.1. Структурное описание 20
  1.1.2. Как интерпретируется описание модуля 22
1.2. Моделирование цифровых систем 24
  1.2.1. Замечания по поводу симуляции 25
  1.2.2. Что делает симулятор? 26
  1.2.3. Более подробно о симуляции 28
  1.2.4. Модели исполнения SystemVerilog 32
  1.2.5. Зачем все это? 33
1.3. Иерархия модулей 33
1.4. Тестовое окружение для модуля mux 35
  1.4.1. Простой пример 35
  1.4.2. Более интеллектуальное тестовое окружение 38
1.5. Резюме 40
1.6. Задачи и упражнения 40

Часть I. МОДЕЛИ УРОВНЯ РЕГИСТРОВЫХ ПЕРЕДАЧ 43

Глава 2. Комбинационные схемы 45

2.1. Моделирование комбинационных схем 45
  2.1.1. Операторы assign и always_comb 46
  2.1.2. Вы уверены, что это комбинационные схемы? 49
2.2. Использование операторов assign и always_comb 50
  2.2.1. Оператор always_comb 50
  2.2.2. Оператор assign 51
  2.2.3. Процедурное моделирование с помощью if и case 52
  2.2.4. Задание несущественных комбинаций с помощью unique case 55
  2.2.5. Упрощение спецификации с помощью ? и casez 57
  2.2.6. Моделирование с учетом уровней сигналов 58
  2.2.7. Оператор priority case 59
2.3. Основы разработки тестового окружения 60
  2.3.1. Отладочная печать 61
  2.3.2. Основы тестирования комбинационных схем 62
  2.3.3. Более сложная система 64
2.4. Параметризованные модули 66

2.5. Спецификация портов ... 68
2.6. Основные типы данных ... 70
2.6.1. Двух- и четырехзначные «биты» ... 70
2.6.2. Целочисленные типы данных ... 71
2.6.3. Перечисления и определения типов ... 72
2.6.4. Тестирование комбинационных схем с перечислениями ... 76
2.6.5. Структуры ... 77
2.7. Множественные драйверы ... 79
2.7.1. Цепи ... 80
2.7.2. Тристабильные приемопередатчики ... 81
2.8. Задачи и упражнения ... 82

Глава 3. Конечные автоматы ... 87

3.1. D-триггер ... 88
3.1.1. Смотри, куда ступаешь ... 89
3.1.2. Вариации на тему ... 90
3.1.3. Тестовое окружение для D-триггера ... 90
3.2. Основы проектирования конечных автоматов ... 92
3.2.1. Описание на SystemVerilog ... 93
3.2.2. Неблокирующие (параллельные) присваивания ... 94
3.2.3. Другой взгляд на = и <= ... 96
3.2.4. Наглядное изображение диаграмм состояний ... 99
3.2.5. Формальное определение ... 100
3.3. Явный стиль описания конечных автоматов ... 101
3.4. Логическая оптимизация ... 104
3.4.1. Кодирование состояний ... 104
3.4.2. Комбинационные схемы для функций перехода и выхода ... 105
3.4.3. Так ли вам нужны несущественные элементы? ... 106
3.5. Тестовые окружения для конечных автоматов ... 106
3.6. Задачи и упражнения ... 106

Глава 4. Предположение о синхронности ... 110

4.1. Основные предположения: доверяй, но проверяй ... 110
4.2. Предположения о временных характеристиках ... 111
4.2.1. Временные характеристики D-триггера ... 111
4.2.2. Расфазировка тактового сигнала ... 113
4.2.3. Нарушение ограничения на время удержания ... 114
4.3. Домены синхронизации ... 115
4.3.1. Что ограничивает размер домена синхронизации? ... 115
4.3.2. Междоменные сигналы ... 116
4.4. Логическая оптимизация: коррекция временных характеристик ... 118
4.5. Правила проектирования синхронных систем ... 119

Часть II. АППАРАТНЫЕ ПОТОКИ ... 123

Глава 5. Аппаратные потоки (конечные автоматы с трактом данных) ... 125

5.1. Аппаратные потоки ... 125

5.1.1. Иллюстративный пример 126
5.1.2. Временная диаграмма работы потока 127
5.1.3. Тракт данных потока 128
5.1.4. Диаграмма состояний 130
5.1.5. Совмещение конечного автомата и тракта данных 131
5.1.6. Описание на SystemVerilog 132
5.1.7. Формальное определение 133
5.2. Временные характеристики автоматов Мура и Мили 134
5.3. Компоненты тракта данных 135
5.3.1. Комбинационные элементы 136
5.3.2. Регистры 137
5.3.3. Дешифраторы 138
5.3.4. Шины 140
5.3.5. Модули памяти 141
5.4. Тестовые окружения для аппаратных потоков 143
5.5. Задачи и упражнения 143

Глава 6. Интерфейсы 153

6.1. Взаимодействующие аппаратные потоки 153
6.1.1. Организация потоков 153
6.1.2. Синхронность 155
6.2. Синхронные взаимодействия между потоками 156
6.2.1. Двустороннее ожидание 158
6.2.2. Пошаговая синхронизация 160
6.3. Пример шины SimpleBus 162
6.3.1. Определение протокола SimpleBus 162
6.3.2. Поток интерфейса процессора (ведущий компонент) 165
6.3.3. Поток интерфейса памяти (ведомый компонент) 167
6.3.4. Система в целом 169
6.3.5. Код примера SimpleBus 171
6.4. Асинхронные взаимодействия между потоками 176
6.4.1. Протокол квитирования с полной взаимной синхронизацией 177
6.4.2. Вариации на тему 180
6.4.3. Очереди как буферы 181
6.5. Интерфейсы в SystemVerilog 183
6.5.1. Пример простого интерфейса 184
6.5.2. Характеристики интерфейса 186
6.5.3. Пример более сложного интерфейса 187
6.6. Задачи и упражнения 192

Часть III. ТЕСТОВЫЕ ОКРУЖЕНИЯ 197

Глава 7. Введение в тестовые окружения 199

7.1. Организация тестового окружения 199
7.2. Программы тестового окружения 200
7.2.1. Конструкция program 201
7.2.2. Этапы работы симулятора 202
7.3. Тестовые окружения для конечных автоматов 204

7.3.1. Тактовый сигнал и сигнал сброса 204
7.3.2. Использование $monitor для отладки конечных автоматов....................206
7.3.3. Неявно заданные конечные автоматы....................208
7.4. Тестовые окружения для аппаратных потоков 213
7.4.1. Запуск аппаратного потока....................215
7.4.2. Системная инициализация 217
7.4.3. Простая программа тестового окружения 217
7.4.4. Использование процедур 219
7.4.5. Использование классов....................221
7.4.6. Обратите внимание 224
7.5. Использование случайных значений....................225
7.5.1. Определение случайной переменной 225
7.5.2. Ограничения на случайные значения....................226
7.5.3. Случайный выбор 229
7.5.4. Случайные последовательности 230
7.6. Полезные конструкции 234
7.6.1. Иерархические имена....................234
7.6.2. Операторы force/release и assign/deassign....................236
7.6.3. Завершение симуляции....................238
7.7. Отладка с использованием процедур ввода-вывода 239
7.7.1. Процедуры $display, $monitor и $strobe....................239
7.7.2. Контроль выводимых величин....................240
7.7.3. Файловый ввод/вывод....................241
7.7.4. Процедуры $readmemh и $readmemb....................241
7.8. Задачи и упражнения....................242

Глава 8. Параллельные тестовые окружения 244

8.1. Процессы 244
8.2. Пример и общая схема тестирования....................246
8.3. Протоколы взаимодействия....................248
8.4. Организация тестового окружения 249
8.4.1. Заголовки и создание экземпляров модулей 249
8.4.2. Тестовый передатчик 250
8.4.3. Тестовый приемник....................253
8.4.4. Основная часть тестового окружения 255
8.5. Конструкции параллельного программирования 257
8.5.1. Оператор wait....................257
8.5.2. Оператор disable 258
8.5.3. Почтовые ящики....................259
8.5.4. Семафоры....................261
8.5.5. Именованные события....................263

Глава 9. Утверждения и последовательности 265

9.1. Предварительные сведения 266
9.1.1. Пример непосредственного утверждения....................266
9.2. Введение в параллельные утверждения....................268
9.2.1. Определение и проверка свойства....................269
9.2.2. Свойства и последовательности 271

9.2.3. Как интерпретируются утверждения 272
9.2.4. Автоматный взгляд на последовательности 273
9.2.5. Еще о предыдущем примере 273
9.3. Последовательности с диапазонами и повторениями 276
9.3.1. Определение протокола SimpleBus 276
9.3.2. Спецификация свойств протокола 277
9.3.3. Операции над последовательностями 279
9.3.4. Повторение частей в последовательностях 281
9.4. Вычисления внутри последовательностей 284
9.5. Функции работы с сэмплированными значениями 287
9.5.1. Общая информация 287
9.5.2. Использование в последовательностях 289
9.6. Задачи и упражнения 292

Глава 10. Функциональное покрытие 293

10.1. План верификации 293
10.2. Группы покрытия и точки покрытия 294
10.2.1. Иллюстрация 295
10.2.2. Параметры накопителей 298
10.2.3. Активация групп покрытия в утверждениях 299
10.3. Покрытие переходов и перекрестное покрытие 301
10.3.1. Покрытие переходов 301
10.3.2. Накопители переходов 302
10.3.3. Накопители для перекрестного покрытия 304
10.4. Вычисление уровня покрытия 305
10.5. Задачи и упражнения 307

Часть IV. ДЕТАЛИ, ДЕТАЛИ, ДЕТАЛИ 309

Глава 11. Процедурные модели 311

11.1. Операторы процессов 311
11.1.1. Оператор initial 313
11.1.2. Операторы always_comb и always_latch 314
11.1.3. Оператор always_ff 316
11.1.4. Оператор непрерывного присваивания (assign) 316
11.1.5. Оператор final 317
11.2. Оператор if-else и условная операция 317
11.2.1. Логические выражения (условия) 318
11.2.2. Логические связки в выражениях 319
11.2.3. Многовариантное ветвление с помощью if-else-if 320
11.3. Оператор case и его разновидности 321
11.3.1. Операторы casex и casez 323
11.3.2. Ключевые слова unique, unique0 и priority 324
11.4. Циклы 324
11.4.1. Цикл forever 325
11.4.2. Цикл repeat 325
11.4.3. Цикл while 325

11.4.4. Цикл do-while 326
11.4.5. Цикл for 326
11.4.6. Операторы continue и break 327
11.4.7. Цикл foreach 327
11.5. Подпрограммы: функции и процедуры 328
11.5.1. Функции 329
11.5.2. Процедуры 330
11.5.3. Сходства и различия 332
11.5.4. Направление и тип аргументов 332
11.5.5. Возвращаемое значение 332
11.5.6. Автоматические и статические переменные 334
11.5.7. Значения аргументов по умолчанию 335
11.6. Таблица операций 336

Глава 12. Структурные модели 338
12.1. Вентильные примитивы 338
12.2. Цепи 341
12.2.1. Объявление цепей 343
12.2.2. Установка значений в цепях 344
12.3. Выбор части и конкатенация 346
12.3.1. Выбор бита и выбор части 346
12.3.2. Конкатенация и репликация 347
12.4. Модули, порты и экземпляры модулей 349
12.4.1. Модули и их экземпляры 349
12.4.2. Порты модулей 351
12.5. Генерация моделей 353

Глава 13. Массивы 356
13.1. Массивы 356
13.1.1. Многомерные массивы 357
13.1.2. Упакованные и неупакованные массивы 359
13.1.3. Операции над массивами 359
13.2. Динамические массивы 360
13.3. Строки 361
13.4. Очереди 362
13.5. Ассоциативные массивы 363

Глава 14. Работа симулятора 365
14.1. События, слоты времени и списки событий 365
14.2. Цикл симуляции 366
14.3. Основной и реагирующий этапы 369
14.4. Блок-схема работы симулятора 372

Предметный указатель 374

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.

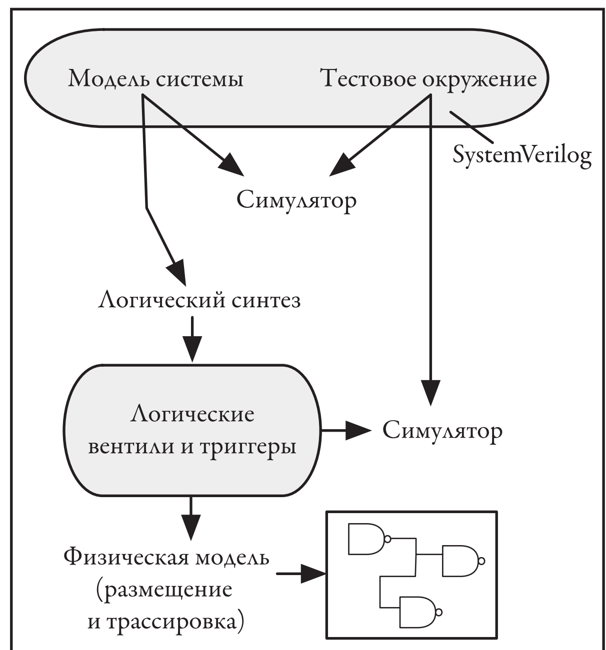
Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

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.

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.

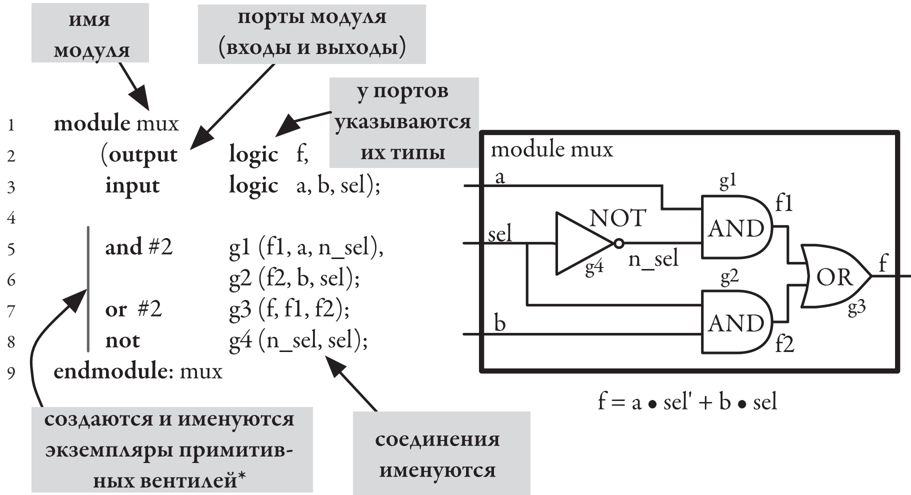
1.1.1. Структурное описание

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

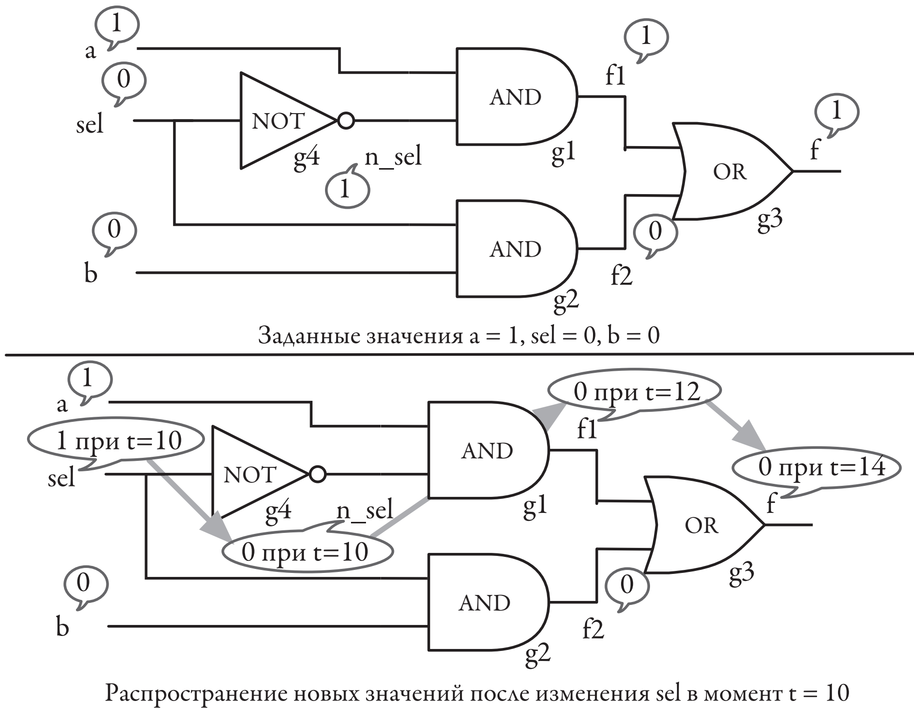


Рис. 1.1. Распространение значений во времени

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,

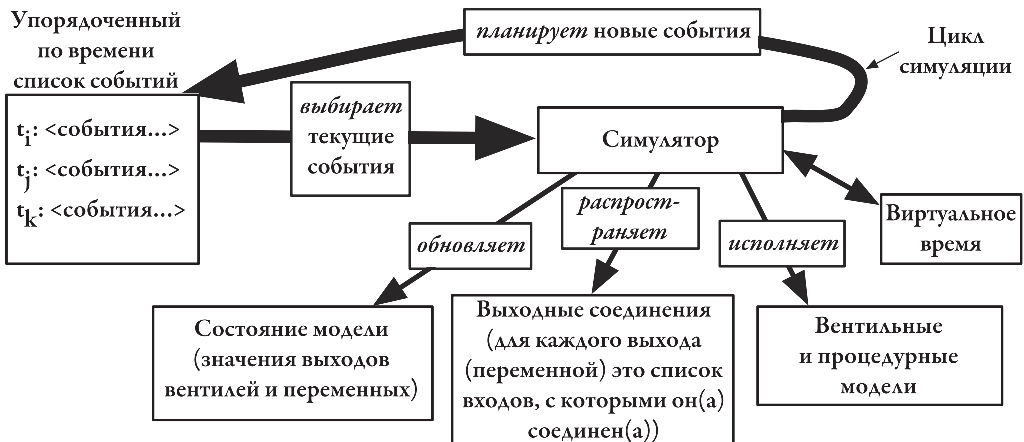


Рис. 1.3. Схема работы симулятора

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.

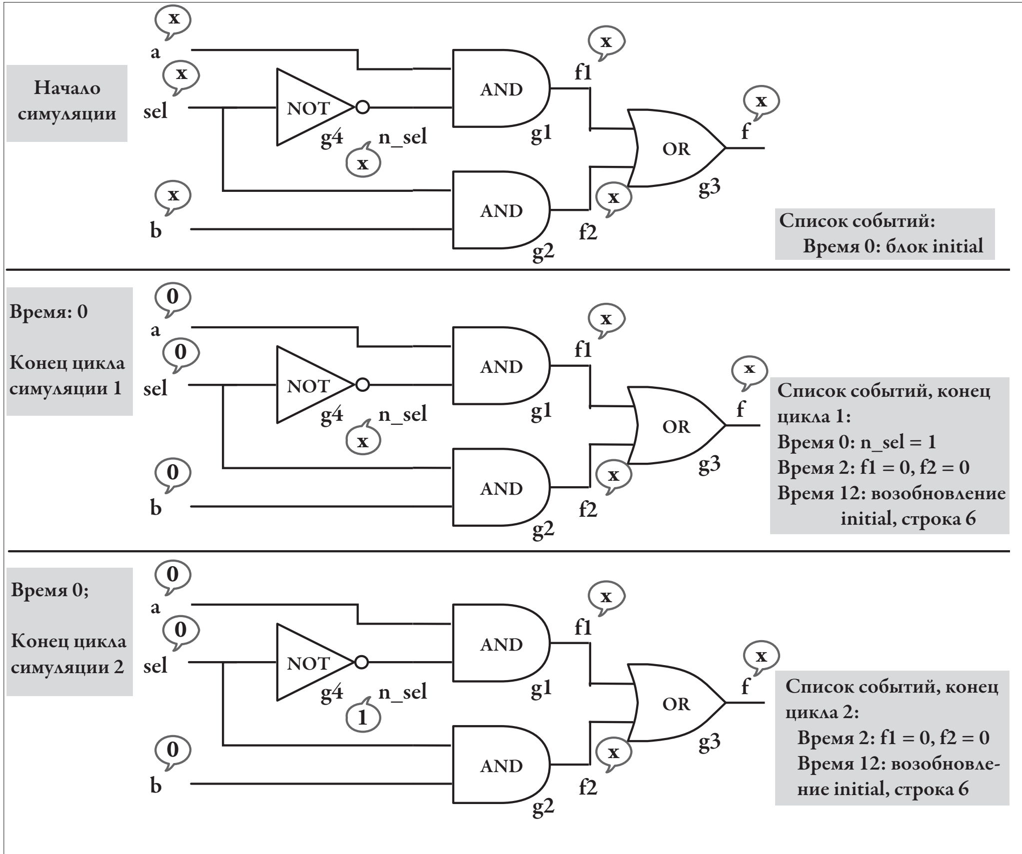


Рис. 1.4. Пошаговая схема работы симулятора (начало)

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

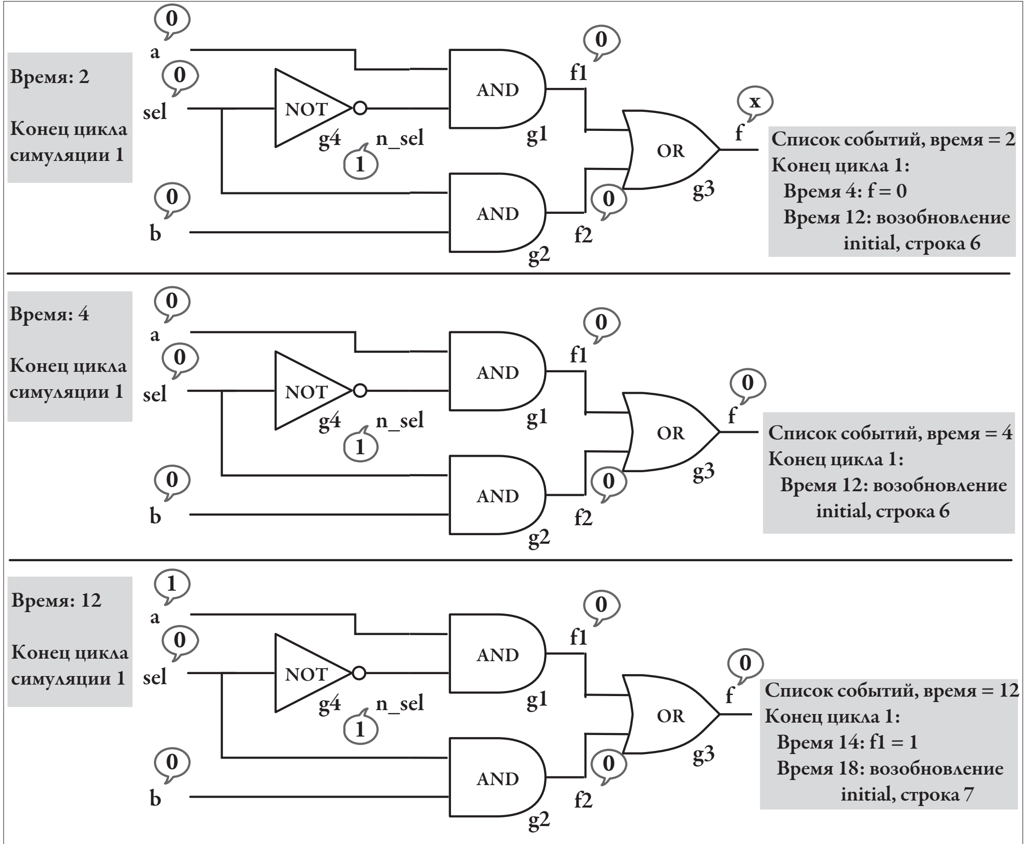


Рис. 1.5. Пошаговая схема работы симулятора (продолжение)

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.

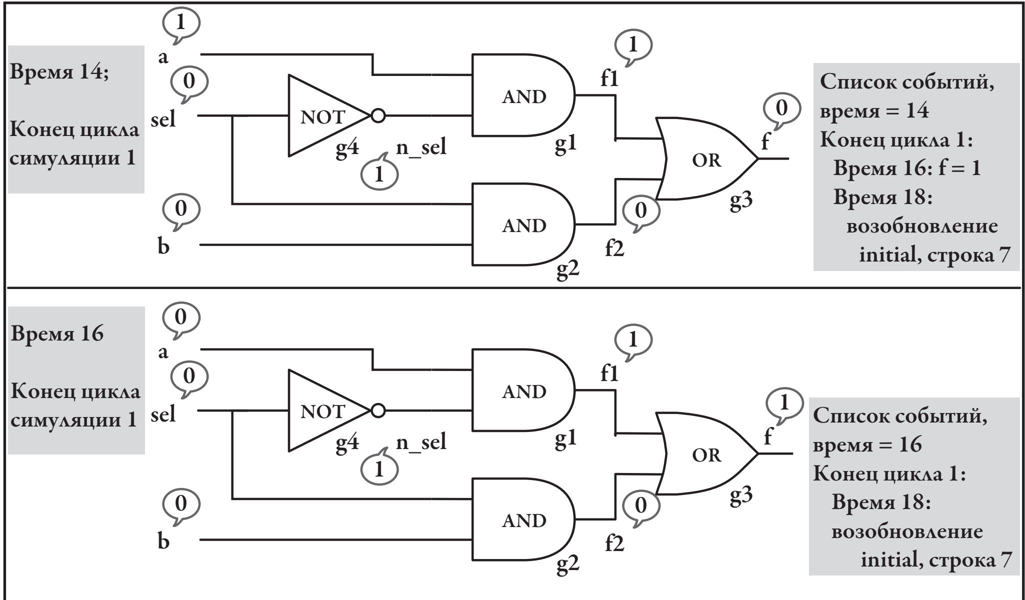


Рис. 1.6. Пошаговая схема работы симулятора (завершение)

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 др.), 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

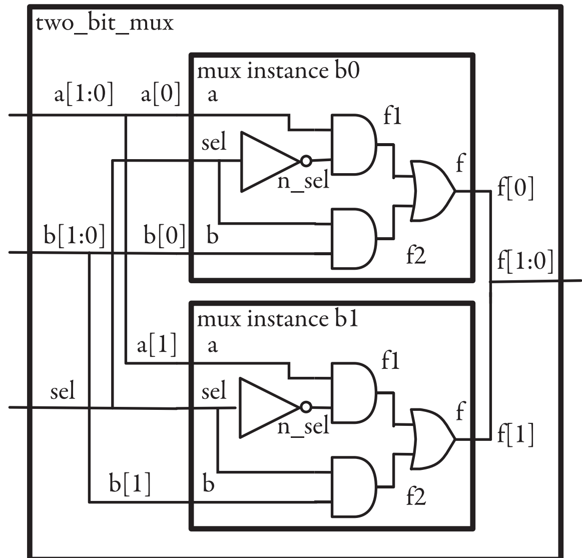


Рис. 1.7. Иерархическая логическая схема модуля two_bit_mux

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`.

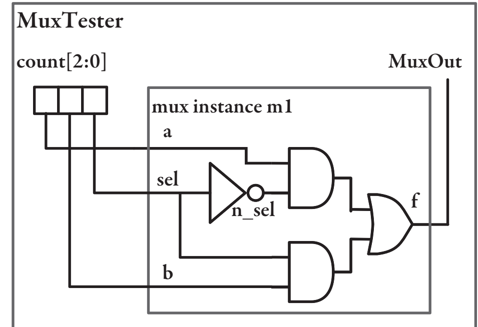


Рис. 1.8. Тестовое окружение для модуля `mux`

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.

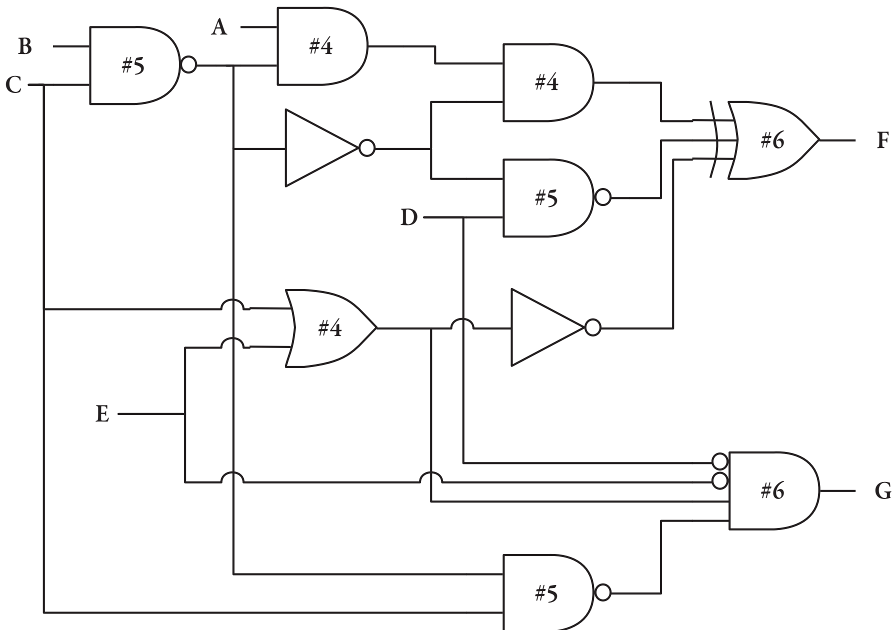
```
1 module trueBoole
2 (output logic f,
3   input logic a, b, c);
4
5   nor (f, f1, f2, f5);

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.

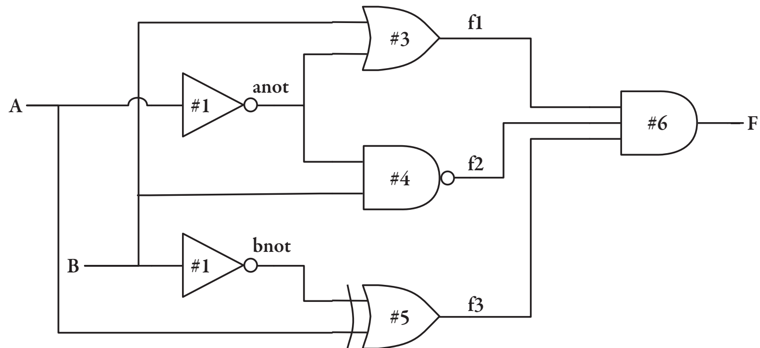


1.4. Обобщите решение задачи 1.2 для тестирования модуля из задачи 1.3.

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`.