Фрэнк Бруно

Программирование FPGA для начинающих

Создавайте цифровые устройства и электронные схемы с помощью SystemVerilog

Под науч. ред. А. Ю. Романова, Ю. В. Ревича

ДМК Пресс издательство

FPGA (Field-Programmable Gate Array, программируемые пользователем вентильные матрицы, ПЛИС) в настоящее время стали основной частью большинства современных электронных и компьютерных систем. Чтобы реализовать свои идеи на основе FPGA, вам необходимо разобраться в их архитектуре, освоить набор инструментов разработки и изучить их важнейшие конструктивные особенности.

Данная книга открывает серию переводов зарубежных изданий по проектированию цифровых систем, которую готовят к выходу в свет издательство ДМК Пресс и МИЭМ НИУ ВШЭ при поддержке группы компаний YADRO (yadro.com). Она предназначена для тех, кто хочет узнать о технологии FPGA и получить практический опыт работы с реальными проектами. Читатель получит общее представление о ПЛИС, научится программировать на языке SystemVerilog, разработает, протестирует и реализует ряд проектов разной степени сложности от простого калькулятора до комплексного решения, использующего AXI и внешнюю периферию, подключенную через PS/2 и VGA.

Рассматриваемые темы:

- архитектура FPGA;
- проектирование на SystemVerilog RTL;
- разработка проектов на FPGA с использованием SystemVerilog;
- основы компьютерной математики, параллелизм и конвейеризация;
- взаимодействие на уровне аппаратуры с клавиатурой PS/2, датчиком температуры, микрофоном, VGA дисплеем и другой периферией.

Издание будет полезно студентам, инженерам, а также широкому кругу читателей, интересующихся современной схемотехникой.
Демонстрационные примеры реализованы на доступных платах Nexys A7 или Basys 3 (с чипами Xilinx) и сопровождаются исходными кодами.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

Packt

ДМК ПРЕСС
издательство
www.дмк.рф

ISBN 978-5-97060-986-6
9 785970 609866 >

Программирование FPGA для начинающих

Воплощайте свои идеи в жизнь и создавайте цифровые устройства и электронные схемы с помощью SystemVerilog

Фрэнк Бруно

ДМК Пресс
издательство
Москва, 2022

УДК 004.41
ББК 32.372
Б89

Главный научный редактор:
Романов А. Ю. – к.т.н., доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики».

Фрэнк Бруно
Б89 Программирование FPGA для начинающих / пер. с англ. С. Л. Плехановой; под науч. ред. А. Ю. Романова, Ю. В. Ревича. – М.: ДМК Пресс, 2022. – 304 с.: ил.

ISBN 978-5-97060-986-6

Данная книга открывает серию переводов зарубежных изданий по проектированию цифровых систем, которую готовят к выходу в свет компания «ДМК Пресс» и МИЭМ НИУ ВШЭ при поддержке группы компаний YADRO (yadro.com). Она предназначена для тех, кто хочет узнать о том, как устроена технология FPGA, и получить практический опыт работы с реальными проектами. Читатель получит общее представление о программируемых логических интегральных схемах, научится программировать на языке SystemVerilog, разработает, выполнит тестирование и реализует ряд проектов разной степени сложности от простого калькулятора до комплексного проекта, использующего AXI и внешнюю периферию, подключенную через PS/2 и VGA.

Демонстрационные примеры реализованы на доступных платах Nexys A7 или Basys 3 (с чипами Xilinx) и сопровождаются исходными кодами.

Издание будет полезно студентам, инженерам, а также широкому кругу читателей, интересующихся современной схемотехникой.

УДК 004.41
ББК 32.372

Copyright © Packt Publishing 2021. First published in the English language under the title "FPGA Programming for Beginners" – (9781789805413).

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN (анг.) 978-1-78980-541-3
ISBN (рус.) 978-5-97060-986-6

Copyright © 2021 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2022
© Научное редактирование, НИУ ВШЭ, 2022

Оглавление

Об авторах **11**

Предисловие от главного редактора русского перевода **13**

Предисловие **16**

РАЗДЕЛ 1. ВВЕДЕНИЕ В FPGA **21**

Глава 1. Введение в FPGA и Xilinx Vivado **23**

Технические требования 23
Аппаратура 24
Программное обеспечение 24
Что такое ASIC? 24
Почему ASIC или FPGA? 25
Как компания создает программируемое устройство, используя ASIC 27
Базовые логические элементы 27
Более сложные операции 30
Знакомство с FPGA 31
Изучение Xilinx Artix-7 и устройств 7-й серии 33
Знакомство с набором инструментов Vivado и отладочными платами 37
Знакомство с Vivado 40
Выполнение примера 46
Программирование платы 55
Выводы 56
Вопросы 56
Задание повышенной сложности 57
Дополнительное чтение 57

РАЗДЕЛ 2. ВВЕДЕНИЕ В ПРОЕКТИРОВАНИЕ, МОДЕЛИРОВАНИЕ И СИНТЕЗ НА VERILOG RTL **59**

Глава 2. Комбинационная логика **61**

Технические требования 61
Создание модулей SystemVerilog 61
Создание многократно используемого кода с помощью параметров 62
Знакомство с типами данных 63
Представление встроенных типов данных 63
Создание массивов 64
Обращение к элементам массива 65
Присвоение значений элементам массива 66
Работа с цепями с тремя состояниями 66

Работа со знаковыми и беззнаковыми числами ... 67
Добавление битов к сигналу с помощью операции конкатенации ... 68
Преобразование знаковых и беззнаковых чисел ... 68
Создание типов, определяемых пользователем ... 69
Доступ к сигналам при использовании значений перечисляемых типов ... 69
Упаковка кода с помощью функций ... 70
Создание комбинационной логики ... 70
Операторы присваивания ... 71
Принятие решений: if-then-else ... 73
Сравнение значений ... 73
Операторы if с уникальностью или приоритетностью ... 74
Оператор выбора case ... 75
Использование пользовательских типов данных ... 76
Создание структур ... 76
Создание объединений ... 77
Проект 1. Создание комбинационной схемы ... 77
Testbench ... 78
Моделирование с помощью целевого тестирования ... 80
Моделирование с использованием рандомизированного тестирования ... 80
Моделирование с использованием ограниченной рандомизации ... 80
Реализация детектора ведущей единицы с использованием оператора case ... 80
Управление реализацией с помощью generate ... 81
Проектирование многоразового детектора ведущей единицы с помощью цикла for ... 83
Реализация сумматора/вычитателя (adder/subtractor) ... 85
Сложение ... 85
Вычитание ... 86
Умножение ... 86
Объединяем все вместе ... 87
Добавление защелки ... 89
Выводы ... 89
Вопросы ... 89
Задание повышенной сложности ... 90
Дополнительное чтение ... 90

Глава 3. Подсчет нажатий на кнопку ... 91

Технические требования ... 91
Что такое последовательностный элемент? ... 91
Синхронизация проекта ... 91
Базовый регистр ... 93
Создание триггеров ... 93
Когда использовать always@() для генерации триггера ... 95
Использование неблокирующих присваиваний ... 96
Регистры в Artix 7 ... 97
Как удерживать состояние схемы с помощью входа разрешения тактового сигнала ... 98
Сброс триггера ... 99

Проект 2. Подсчет нажатий на кнопку 100
Семисегментный индикатор 100
Обнаружение нажатия на кнопку 103
Проблемы, возникающие из-за асинхронных сигналов 104
Использование асинхронного сигнала напрямую 105
Проблема с нажатием кнопок 106
Разработка безопасной реализации 107
Переход на десятичное представление 109
Знакомство с ILA 110
Что насчет симуляции? 114
Подробное изучение синхронизации 115
Зачем использовать несколько тактовых сигналов? 115
Двухступенчатый синхронизатор 115
Синхронизация управляющих сигналов 116
Передача данных 117
Выводы 118
Вопросы 118
Задание повышенной сложности 119
Дополнительное чтение 119

Глава 4. Разработка калькулятора 120

Технические требования 121
Реализация первого конечного автомата 121
Разработка последовательного конечного автомата 121
Разделение комбинационной и последовательностной логики в конечном автомате 122
Разработка интерфейса калькулятора 123
Проектирование конечного автомата Мура 124
Реализация конечного автомата Мили 126
Практическое проектирование конечных автоматов 126
Проект 3. Создание простого калькулятора 127
Инкапсуляция для повторного использования 127
Проектирование модуля верхнего уровня иерархии 129
Изменение тактовой частоты с помощью PLL или MMCM 130
Разработка блока деления 134
Построение конечного автомата невосстанавливающего делителя 134
Моделирование делителя 138
Определение размера промежуточного остатка 138
Проект 4. Управление перекрестком с помощью светофоров 139
Определение графа состояний 140
Отображение состояний светофоров 140
Выводы 142
Вопросы 142
Задание повышенной сложности 143
Задание еще более высокой сложности 143
Дополнительное чтение 143

Глава 5. Ресурсы FPGA, и как их использовать 144
Технические требования 144
Проект 5. Слушать и учиться 145
Что такое цифровой PDM-микрофон? 145
Моделирование работы микрофона 148
Встроенная память 150
Захват аудиоданных 154
Проект 6. Использование датчика температуры 157
Обработка данных 158
Сглаживание данных 159
Более глубокое погружение в FIFO 160
Ограничения 163
Генерация FIFO 163
Выводы 165
Вопросы 165
Дополнительное чтение 166

Глава 6. Математика, параллелизм и конвейеризация 167
Технические требования 168
Числа с фиксированной точкой 168
Проект 7. Использование чисел с фиксированной точкой для обработки данных с датчика температуры 169
Использование арифметики чисел с фиксированной точкой для очистки времени запуска 170
Преобразование температуры с помощью арифметики с фиксированной точкой 172
А как насчет чисел с плавающей точкой? 174
Сложение и вычитание с плавающей точкой 176
Умножение с плавающей точкой 176
Обратное значение для числа с плавающей точкой 176
Более практичная библиотека операций с плавающей точкой 176
Краткий обзор потокового интерфейса AXI 177
Проект 8. Обновление проекта датчика температуры до конвейерной реализации с плавающей точкой 179
Преобразование чисел из представления с фиксированной точкой в формат с плавающей точкой 179
Математические операции с плавающей точкой 181
Преобразование формата с плавающей точкой в формат с фиксированной точкой 182
Моделирование 183
Параллельные конструкции 185
ML, AI и массовый параллелизм 185
Параллельное проектирование – небольшой пример 186
Выводы 187
Вопросы 187
Задание повышенной сложности 188
Дополнительное чтение 188

РАЗДЕЛ 3. ВЗАИМОДЕЙСТВИЕ С ВНЕШНИМИ КОМПОНЕНТАМИ 189

Глава 7. Введение в AXI 191

Технические требования 191
Потоковая передача AXI 192
Проект 9. Создание IP-блоков для Vivado с использованием потоковых интерфейсов AXI 192
Потоковый интерфейс для семисегментного индикатора 193
Разработка IP ADT7420 198
Ядро t_temp 198
IP-интегратор 198
Отладка проекта с помощью IP-интегратора 206
Интерфейсы AXI4 (AXI full и AXI-Lite) 207
Разработка IP-блоков – AXI-Lite, AXI full и AXI Stream 209
Добавление неупакованного IP-блока в IP-интегратор 212
Выводы 214
Вопросы 214
Дополнительное чтение 215

Глава 8. Много данных? MIG и DDR2 216

Технические требования 216
Проект 10. Подключение внешней памяти 217
Память DDR2 218
Генерация контроллера DDR2 с помощью Xilinx MIG 219
Установка параметров интерфейса AXI 223
Настройка параметров памяти 223
Настройка параметров FPGA 224
Модификация проекта для использования на плате 232
Другие типы внешней памяти 236
Память SRAM с четырехкратной скоростью передачи данных (Quad Data Rate, QDR) 236
HyperRAM 236
SPI RAM 236
Выводы 237
Вопросы 237
Задача повышенной сложности 238
Дополнительное чтение 238

Глава 9. Лучший способ отображения – VGA 239

Технические требования 239
Проект 11. Основы работы с VGA 240
Определение регистров 243
Разработка простого интерфейса AXI-Lite 244
Генерация сигналов синхронизации для VGA 245
Отображение текста 251
Запрос памяти 253

Тестирование контроллера VGA 257
Проверка ограничений 257
Выводы 259
Вопросы 259
Задание повышенной сложности 260
Дополнительное чтение 260

Глава 10. Свести все воедино 261

Технические требования 261
Изучение интерфейса клавиатуры 262
Проект 12. Работа с клавиатурой 267
Моделирование работы интерфейса PS/2 270
Проект 13. Сводим все воедино 272
Отображение кодов клавиш PS/2 на экране VGA 272
Отображение показаний датчика температуры 275
Отображение аудиоданных 277
Выводы 280
Вопросы 281
Задание повышенной сложности 281
Дополнительное чтение 281

Глава 11. Темы повышенной сложности 282

Технические требования 282
Изучение более продвинутых конструкций SystemVerilog 282
Взаимодействие компонентов с использованием конструкции под названием «интерфейс» 282
Использование структур 285
Метки блоков 286
Цикл for 287
Цикл do…while 287
Выход из цикла с помощью оператора disable 288
Пропуск фрагментов кода с помощью оператора continue 288
Использование констант 289
Некоторые продвинутые конструкции языка SystemVerilog для верификации 289
Знакомство с очередями SystemVerilog 289
Продвинутое использование системной функции $display 291
Утверждения 292
Использование $error или $fatal при синтезе проекта 292
Другие проблемы, и как их избежать 293
Выведение однобитных проводов 293
Несоответствие ширины шин 294
Повышение или понижение приоритетности сообщений Vivado 294
Обработка timing closure 295
Конвейеризация 297
Выводы 301
Вопросы 302
Дополнительное чтение 303

Об авторах

Автор

Фрэнк Бруно – опытный инженер-разработчик высокопроизводительных систем, специализирующийся на FPGA и имеющий некоторый опыт работы с ASIC. Работал в таких компаниях, как Cruise, SpaceX, Allston Trading и Number Nine. В настоящее время работает инженером по разработке на FPGA в компании Cruise.

Рецензент

Джордж Калдис получил степень бакалавра электротехники в Северо-Восточном университете и имеет более чем 30-летний опыт работы с FPGA. Является президентом GK-Digital LLC, консалтинговой компании по проектированию FPGA. Реализовал множество проектов FPGA для различных приложений – от беспроводных и проводных сетей до высокочастотного трейдинга и тестового оборудования.

Редакторы русского перевода

Романов Александр Юрьевич – главный научный редактор русского перевода данной книги, доцент Московского института электроники и математики им. А. Н. Тихонова Национального исследовательского университета «Высшая школа экономики» (МИЭМ НИУ ВШЭ). С 2014 г. работает в МИЭМ НИУ ВШЭ, где возглавляет лабораторию САПР (https://miem.hse.ru/edu/ce/cadsystem), специализирующуюся на проектной деятельности, а также разработке цифровых систем на ПЛИС/микроконтроллерах, робототехнических комплексов, аппаратных реализаций систем искусственного интеллекта, многопроцессорных систем, систем удаленного доступа к лабораторному оборудованию и т. д. В 2015 г. защитил диссертацию в Институте проблем проектирования в микроэлектронике РАН (г. Зеленоград), является автором более 150 научных статей, патентов и книг. Более подробно об учебном процессе в лаборатории можно узнать из интервью: https://miem.hse.ru/news/364316102.html.

Юрий Всеволодович Ревич – научный редактор русского перевода – инженер-электронщик с многолетним стажем. Занимался автоматизацией производств, конструированием измерительных приборов для изучения океана и другими научными исследованиями. Работал редактором в периодических изданиях по IT-направлению. Автор многих статей и ряда популярных книг по электронике, среди которых «Занимательная электроника», «Практическое программирование микроконтроллеров Atmel AVR», сборник «Информационные технологии в СССР» и др.

Смехов Дмитрий Геннадьевич – консультант русского перевода – инженер-разработчик на ПЛИС с многолетним стажем. Занимается разработкой и верификацией проектов на основе ПЛИС Xilinx и Altera (Intel FPGA). Является сертифицированным инструктором компании Xilinx, проводит тренинги по темам PCI Express, Vitis AI, Versal ACAP. Работал в таких компаниях, как АО «ИнСис», КТЦ «Инлайн Групп», IRQ.

Предисловие от главного редактора русского перевода

Дорогие друзья!

Книга, которую вы держите в руках, открывает серию переводов зарубежных изданий по проектированию цифровых систем, которую готовят к выходу в свет компания «ДМК Пресс» и МИЭМ НИУ ВШЭ при поддержке группы компаний YADRO (yadro.com).

Актуальность подобных книг несомненна. Очевидно, что после 24 февраля 2022 г. развитие отечественной электроники сталкивается с новыми вызовами: жесткие торговые ограничения, запрет на пользование интеллектуальными продуктами (в том числе САПРами; например, такими как Quartus), а также и множество иных санкционных действий оказывают серьезное давление на высокотехнологичные отрасли в России. Единственный путь, который позволит не оказаться в стороне от прогресса, – это предпринимать консолидированные усилия ведущими компаниями и организациями по развитию отечественной электроники и в первую очередь по подготовке высококвалифицированных кадров. Серьезный барьер на этом направлении – существенная нехватка качественных учебных материалов на русском языке.

Впрочем, благодаря усилиям многих людей в последние годы уже наметилась тенденция к ее преодолению. Прошли те времена, когда для многих практически единственными доступным источником русскоязычных материалов был известный сайт Наливатора – Егорова Антона (http://www.naliwator.narod.ru/), – где преимущественно собраны машинные переводы руководств по проектированию на FPGA Altera. Появилась и много раз переиздана знаменитая книга Харрисов «Цифровая схемотехника и архитектура компьютера», вышел к ней сборник практических работ по Verilog «Цифровой синтез: практический курс» (под. ред. А. Ю. Романова и Ю. В. Панчула), издана в русском переводе книга Д. Томаса «Логическое проектирование и верификация систем на SystemVerilog», вышло множество других изданий. (О наличии материалов по FPGA на русском языке можно узнать на «Книжном FPGA-стриме #42» с М. Коробковым и С. Иванцом https://www.youtube.com/watch?v=XaYGfNlhX0c.) Тем не менее недостаток в русскоязычных материалах по различным аспектам проектирования цифровых систем все еще ощущается.

Компания «ДМК Пресс» в сотрудничестве с МИЭМ НИУ ВШЭ выбрала стратегию, обеспечивающую (кроме подготовки собственных изданий по цифровому синтезу) создание серии переводов наиболее популярных зарубежных книг по данной проблематике. Настоящее издание – Ф. Бруно «Программирование FPGA для начинающих» – является первым в этой серии. (В обозримом будущем планируется также выход в свет перевода книги С. Сазерленда

«RTL-моделирование на SystemVerilog для моделирования и синтеза: применение SystemVerilog для проектирования ASIC и FPGA».)

Следует отметить, что процесс создания переводных изданий по проектированию цифровых систем является достаточно непростой задачей: несмотря на свою популярность, оригинальные англоязычные книги имеют множество недостатков – частые повторения, опечатки, неточности, значительные стилистические погрешности. Бытующее американское изложение от первого лица при переложении на русский язык выглядит неестественным и является чуждым русской научной речи. Оказалось, что зарубежные авторы технической литературы в погоне за прибылью часто пишут «на поток», не сильно утруждаясь стилистикой. Поэтому при редактировании переводных материалов научно-технического содержания редактору приходится решать многочисленные задачи, связанные с обеспечением выполнения основой функции научного стиля речи – точной передачи логической информации в переведенном тексте, исключающей выражение субъективных, личных эмоций автора и его отступления от стилистических норм. Таким образом, отредактированный русский перевод в его окончательном виде – это скорее переизложение начального материала, обеспечивающее его читабельность для русскоязычной аудитории; фактически это книга, написанная заново.

О чем эта книга?

При выборе изданий для перевода в первую очередь преследовалась цель восполнить очевидные пробелы в массиве существующей русскоязычной литературы, при этом важным критерием отбора являлась популярность книги. Привлечение ряда экспертов (в том числе Ю. Панчула) позволило составить список наиболее интересных книг. Выбор книги «Программирование FPGA для начинающих» обусловлен тем, что она рассказывает о проектировании на FPGA компании Xilinx, демонстрационные примеры в ней реализованы на дешевых и доступных платах Nexys A7 и сопровождаются исходными кодами.

В целом абсолютному новичку эта книга поначалу может показаться несколько сложной, но если проявить немного усидчивости и вникнуть в исходные коды первых примеров, то ее восприятие заметно облегчится. В принципе, большая часть курсов по SystemVerilog, в отличие от других классических языков программирования, так и построена – обучаемого сразу погружают в несколько больших примеров, где он ведет разработку, часто копируя или модифицируя уже готовый код, постепенно изучая особенности языка (разработка программ, описывающих аппаратуру, гораздо сложнее, и постепенно изучать операторы языка нет никакого смысла).

Данная книга хороша для тех, кто раньше работал, например, в Quartus с продукцией фирмы Intel FPGA (т. е. имеет некоторый базис) и хочет расширить свой кругозор, познакомившись с Vivado и FPGA от Xilinx.

Если же у читателя имеется опыт работы в Vivado, эта книга все равно может быть полезной, потому что содержит ряд примеров работы с различной периферией (VGA, датчик температуры, микрофон, PS/2-клавиатура), а также обеспечивает знакомство с организацией проектов на основе интерконнекта AXI.

Книга, несомненно, понадобится студентам вузов (МИЭТ, ИТМО и др.), изучающим SystemVerilog, а также более продвинутым разработчикам, которые хотели бы получить практический опыт работы с реальными проектами.

Научный редактор будет очень признателен тем внимательным читателям, которые обнаружат в данном издании какие-либо ошибки или опечатки и сообщат о них на e-mail a.romanov@hse.ru. (Книги постоянно перепечатываются, и в каждом новом тираже все найденные ошибки и недочеты исправляются.)

Александр Юрьевич Романов,
главный научный редактор русского перевода,
к. т. н., доцент ДКИ МИЭМ НИУ ВШЭ,
преподаватель курсов «Проектирование систем на кристалле»
и «Системное проектирование цифровых устройств»,
г. Москва, Россия

Предисловие

Готовьтесь повеселиться. Автор этой книги разрабатывает ASIC и FPGA[1] уже 30 лет и каждый день сталкивается с новыми вызовами и волнениями, поскольку продвигает технологии к разработке новых приложений. За свою карьеру автор разработал ASIC, которые обеспечивали работу военных самолетов; графику, работающую на высокопроизводительных рабочих станциях и обычных ПК; технологию для питания следующего поколения программно-определяемых радиосистем; а также участвовал в проекте по обеспечению сети Интернет через космос на всем земном шаре. Часть этого опыта представлена в данной книге.

Для кого эта книга

Эта книга предназначена для тех, кто хочет узнать о технологии FPGA и о том, как ее можно использовать в своих проектах. Предполагается, что читатель ничего не знает о цифровой логике, поэтому книга начинается с представления базовых логических элементов и их функций, а заканчивается разработкой полноценной системы на кристалле. Некоторые знания в области программирования или аппаратного обеспечения полезны, но не обязательны. Если вы сможете установить программу, подключить USB-кабель и следовать пошаговым инструкциям, вы узнаете много нового.

Что включает в себя эта книга

Глава 1. Введение в FPGA и Xilinx Vivado. В главе объясняется, что такое ASIC и FPGA и как установить Xilinx Vivado и создать небольшой проект.

Глава 2. Комбинационная логика. В главе описано, как разработать с нуля полноценный модуль на SystemVerilog для выполнения некоторых базовых операций, чтобы продемонстрировать, как использовать комбинационную логику в собственных проектах. Глава также знакомит с разработкой `testbench`[2] на примере создания testbench с самопроверкой.

---

[1] ASIC (application-specific integrated circuit, «интегральная схема специального назначения») – интегральная схема, специализированная для решения конкретной задачи. FPGA (field-programmable gate array, «программируемая пользователем вентильная матрица») – разновидность программируемых логических интегральных схем, ПЛИС. Существуют и другие разновидности программируемых схем, но FPGA, как самые распространенные и универсальные, фактически стали синонимом ПЛИС. ASIC, подобно FPGA, разрабатываются на типовой основе, но затем отдаются в производство, в то время как FPGA выпускается в виде полуфабриката, который доводится до нужной функциональности программными методами непосредственно перед применением. Подробнее об ASIC и FPGA рассказано в главе 1. – *Прим. ред.*

[2] Testbench (дословно «испытательный стенд») – тестирующая программа или программно-аппаратный комплект, созданный для испытания запрограммированной в FPGA функциональности. В русскоязычной профессиональной среде прижился оригинальный англоязычный термин, потому в этой книге он приводится без перевода. – *Прим. ред.*

Глава 3. Подсчет нажатий на кнопку. В этой главе к проекту на основе комбинационной логики из предыдущей главы добавляются последовательностные элементы (встроенная память данных). В главе рассказывается о возможностях Artix-7 и других устройств FPGA для хранения данных, приводится пример простого проекта по подсчету нажатий на кнопку. Также пересказывается про использование тактовых генераторов и синхронизации, того немногого, что может полностью разрушить проект, если сделано неправильно.

Глава 4. Разработка калькулятора. В главе демонстрируется, как при создании более сложных схем неизбежно возникает необходимость отслеживать состояние устройства. В этой главе на примере классического инженерного устройства – контроллера светофора – рассказывается о конечных автоматах. Практическая часть главы посвящена улучшению калькулятора за счет разработки делителя с использованием схем с состояниями.

Глава 5. Ресурсы FPGA, и как их использовать. В этой главе после быстрого погружения в проектирование довольно сложных схем делается шаг назад и рассмотрены более подробно некоторые ресурсы FPGA. Чтобы продемонстрировать использование этих ресурсов в демонстрационных проектах, использованы некоторые периферийные устройства (микрофон PDM и датчик температуры I2C), которые есть на отладочной плате.

Глава 6. Математика, параллелизм и конвейеризация. В этой главе более подробно рассмотрено использование чисел с фиксированной и плавающей точкой, также конвейерное проектирование и параллелизм для повышения производительности.

Глава 7. Введение в AXI[1]. В главе рассказывается о том, как компания Xilinx приняла стандарт AXI для сопряжения своих IP[2] и разработала инструмент IP integrator для простого соединения IP-блоков с помощью графического интерфейса. В этой главе показано, как использовать AXI-интерфейс, на примере интеграции датчика температуры в проект с помощью IP integrator.

Глава 8. Много данных? MIG и DDR2. В этой главе рассказывается, как в Artix-7 обеспечен достаточный объем памяти и что делать, если требуется обеспечить доступ к мегабайтам или гигабайтам оперативной памяти. На отладочной плате размещена память DDR2. В главе рассказывается, как использовать Xilinx Memory Interface Generator для реализации интерфейса DDR2 и провести его моделирование, а также тестирование на плате.

Глава 9. Интерфейс VGA. В главе рассказывается о том, как устроен интерфейс VGA, и приводится простой способ отображения текста. До этого для вывода информации в проектах использовались светодиоды и семисегментный индикатор, что накладывало значительные ограничения. С помощью VGA становится возможно отображать аудиоданные и текст.

---

[1] AXI (Advanced eXtensible Interface) – стандарт высокопроизводительного интерфейса, разработанного фирмой ARM для связи между устройствами на одном кристалле. – *Прим. ред.*

[2] Сокращение IP (расшифровывающееся просто как intellectual property, «интеллектуальная собственность») означает в контексте FPGA специализированные области кристалла (IP-ядра, IP-блоки), добавленные для облегчения программирования некоторых распространенных функций. Об использовании IP речь идет на протяжении всей книги (см., например, главы 3,4,6, особенно подробно – в главе 7). – *Прим. ред.*

Глава 10. Свести все воедино. Глава посвящена добавлению в итоговый проект дополнительных периферийных устройств. В итоге благодаря подключению клавиатуры с помощью PS/2 будет создан комплексный проект, использующий VGA для отображения данных с микрофона и датчика температуры.

Глава 11. Темы повышенной сложности. В этой главе рассмотрены некоторые концепции SystemVerilog, которые были пропущены в других главах, но которые могут оказаться полезными. Продемонстрированы более продвинутые методы тестирования, и разобраны некоторые проблемы, которые могут возникать при проектировании, и способы их предотвращения.

Как получить максимальную пользу от этой книги

Эта книга не предполагает наличия знаний о FPGA, логическом проектировании или программировании. Для того чтобы начать изучение, понадобится компьютер с операционной системой Windows или Linux. В первой главе содержатся инструкции по установке необходимого программного обеспечения.

Программное обеспечение / аппаратное обеспечение, рассматриваемые в этой книге	Требования к ОС
Xilinx Vivado 2020.1	Windows 10 или Linux (Centos 7.4-7.7 или Ubuntu 18.04 или 20.04)
Nexys A7 board	Windows 10 или Linux (Centos 7.4-7.7 или Ubuntu 18.04 или 20.04)

Если вы используете цифровую версию этой книги, рекомендуется набирать код самостоятельно или получить доступ к коду через репозитарий GitHub (ссылка доступна в следующем разделе). Это поможет избежать возможных ошибок, связанных с копированием и вставкой кода.

Скачать файлы примеров кода

Файлы кодов примеров для этой книги можно загрузить из своей учетной записи на сайте www.packt.com. Если книга куплена в другом месте, можно посетить сайт поддержки www.packtpub.com/support и зарегистрироваться, чтобы получить файлы по электронной почте.

Файлы кодов примеров можно загрузить, выполнив следующие действия.

1. Авторизуйтесь или зарегистрируйтесь на сайте www.packt.com.
2. Выберите вкладку **Support**.
3. Перейдите по ссылке **Code Downloads.**
4. Введите название книги в поле поиска и следуйте инструкциям на экране.

После загрузки файла следует его разархивировать с помощью последней версии:

- WinRAR/7-Zip для Windows;
- Zipeg/iZip/UnRarX для Mac;
- 7-Zip/PeaZip для Linux.

Архив с кодами для книги также размещен на GitHub по адресу: https://github.com/PacktPublishing/Learn-FPGA-Programming.

В случае обновления кода он будет обновлен на существующем репозитарии GitHub.

По адресу https://github.com/PacktPublishing/ также размещены примеры кодов и проектов для других книг из обширного каталога Packt.

Скачать цветные изображения

PDF-файл с цветными изображениями скриншотов/вейвформ, используемых в этой книге, находится по адресу:

http://www.packtpub.com/sites/default/files/downloads/9781789805413_ColorImages.pdf.

Используемые обозначения

В этой книге используется ряд обозначений в тексте.

`Код в тексте`: моноширинный шрифт обозначает кодовые служебные слова в тексте, имена переменных, операторы языка, цитаты из кода. Например, «сигнал тактовой частоты `sys_clk_i`».

Имена папок, имена файлов, расширения файлов, имена путей, URL-адреса, пользовательский ввод, некоторые названия модулей и ники в Twitter также выделяются моноширинным шрифтом: «файл `logic_ex.xpr`».

Блок кода задается следующим образом:

```
always @(posedge CK) begin
    stage = D;
    Q = stage;
end
```

Чтобы обратить внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
module dff (input wire D, CK, output logic Q);
    initial Q = 1;
    always_ff @(posedge CK) Q <= D;
endmodule
```

Любой ввод или вывод командной строки записывается следующим образом:

```
`timescale 1ps/100fs
```

Жирный шрифт: обозначает новый термин, важное слово или слова, которые читатель должен найти в интерфейсе среды разработки. Например, слова в меню или диалоговых окнах отображаются в тексте следующим образом: «В окне проекта нажмите правой кнопкой мыши и выберите **Add Module**».

> **Подсказки и важные замечания**
>
> Выглядят таким образом.

Контакты для связи

Отзывы читателей всегда приветствуются.

Обратная связь общего характера: если у вас есть вопросы по любому аспекту этой книги, укажите название книги в теме сообщения и напишите по адресу customercare@packtpub.com.

Ошибки: мы будем очень признательны тем внимательным читателям, которые обнаружат в русском переводе этой книги какие-либо ошибки или опечатки и сообщат о них на e-mail: a.romanov@hse.ru.

Пиратство: если вы встретите в интернете незаконные копии наших произведений в любой форме, мы будем благодарны, если вы сообщите нам адрес местонахождения или название сайта. Пожалуйста, свяжитесь с нами по адресу copyright@packt.com со ссылкой на материал.

Если вы заинтересованы в том, чтобы стать автором: если у вас есть тема, в которой вы разбираетесь, и вы заинтересованы в том, чтобы участвовать в создании книги, посетите сайт authors.packtpub.com.

Отзывы

Пожалуйста, оставьте отзыв. Если вы прочитали и использовали эту книгу, почему бы не оставить отзыв на сайте, на котором вы ее приобрели? Потенциальные читатели могут ознакомиться с вашим непредвзятым мнением и использовать его для принятия решения о покупке, мы в Packt можем понять, что вы думаете о наших продуктах, а наши авторы могут увидеть ваш отзыв о своей книге. Спасибо!

Для получения дополнительной информации о компании Packt посетите сайт packt.com.

Благодарности

Конюхов Владислав Игоревич
Вадим Селезнев
Pavel Gurev
Алексей Яковлев

Отдельная благодарность (внесли значительный вклад в исправления):

Александр Щербенко (телеграмм @AlexSevere,
e-mail: khv_alexandr@mail.ru)

Раздел 1

Введение в FPGA

В этом разделе вы получите представление о том, что такое Field Programmable Gate Array (FPGA), что за технология лежит в ее основе, а также познакомитесь с архитектурой Artix-7.

В эту часть книги входит:

- *Глава 1 «Введение в FPGA и Xilinx Vivado»*.

Глава 1

Введение в FPGA и Xilinx Vivado

В данной главе мы рассмотрим **Field Programmable Gate Array (FPGA)** (программируемая логическая интегральная схема, ПЛИС) и технологию, лежащую в их основе. Эта технология позволяет таким компаниям, как Xilinx, производить перепрограммируемые микросхемы по технологии **Application Specific Integrated Circuit (ASIC).** Затем мы узнаем, как использовать FPGA на примере решения простой задачи. Если вы хотите ускорить математически сложные вычисления, как в задачах машинного обучения или искусственного интеллекта, или просто хотите сделать несколько проектов для развлечения, таких как ретро-вычисления или воспроизведение устаревших видеоигровых машин (https://github.com/MiSTer-devel/Main_MiSTer/wiki), эта книга станет началом вашего путешествия. Сейчас самое время, чтобы погрузиться в эту область, пусть даже только в качестве хобби. Платы для разработки дешевы и многочисленны, и поставщики начали предоставлять свои инструменты бесплатно для недорогих и небольших проектов.

В этой книге мы собираемся реализовать несколько примеров проектов, которые познакомят вас с разработкой на FPGA, а кульминацией станет проект, способный управлять монитором VGA.

К концу этой главы вы получите хорошее представление о FPGA и ее компонентах.

Основные темы, которые мы рассмотрим в этой главе:

- что такое ASIC;
- как создаются FPGA;
- что входит в состав FPGA;
- как использовать инструменты Xilinx Vivado для проектирования, тестирования и реализации проектов на FPGA.

Технические требования

Для работы с примерами в этой главе вам потребуется следующее аппаратное и программное обеспечение.

Аппаратура

В отличие от языков программирования SystemVerilog является языком описания аппаратуры, и для того, чтобы действительно увидеть результаты реализации проектов по этой книге, вам понадобится плата FPGA для загрузки проектов. Для целей этой книги рекомендуется использовать одну из двух плат для разработки, которые легко доступны. Можно использовать и другую плату, если она у вас уже есть. Но некоторые ресурсы платы могут быть не идентичны, или вам может потребоваться изменить файл ограничений (xdc), чтобы получить доступ к ресурсам другой платы.

- Информация о Nexys A7: https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/.
- Информация об обучающей плате Basys 3 Artix-7 FPGA: https://store.digilentinc.com/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/.

Nexys A7 предпочтительнее, поскольку она имеет внешние интерфейсы, которые будут обсуждаться в последующих главах и дадут вам опыт взаимодействия с внешним оборудованием. Рекомендуется использовать версию 100T на тот случай, если вы будете амбициозны и захотите изучить больше, поскольку разница в цене относительно невелика, и у нее вдвое больше ресурсов. За исключением памяти DDR, на плате Basys 3 можно реализовать большинство проектов, хотя для некоторых из них может потребоваться приобретение интерфейсных плат PMOD.

Программное обеспечение

Для работы вам потребуется следующее программное обеспечение:

- https://www.xilinx.com/products/design-tools/vivado.html;
- файлы кода для всех примеров из этой главы можно найти в репозитории GitHub этой книги по адресу https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH1.

Что такое ASIC?

Интегральные схемы специального назначения (ASIC) являются фундаментальными строительными блоками современной электроники – вашего ноутбука или ПК, телевизора, мобильного телефона, цифровых часов, практически всего, чем вы пользуетесь ежедневно. Это также фундаментальный строительный блок, на основе которого создается рассматриваемая нами FPGA. Если коротко, ASIC – это специально созданная микросхема, разработанная с использованием того же языка и методов, которые мы рассмотрим в этой книге.

FPGA появились благодаря тому, что технология создания ASIC следовала закону Мура (*Gordon E. Moore, Cramming more components onto integrated circuits, Electronics, Volume38, Number 8*, https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf) – идее о том, что количество транзисторов в чипе удваивается каждые 2 года. Это позволило создавать очень дешевую электронику при массовом производстве изделий, содержащих ASIC, и также привело к распространению более дешевых FPGA.

Почему ASIC или FPGA?

ASIC могут быть недорогими, если они производятся в больших количествах. Вы можете купить одноразовый калькулятор, флеш-накопитель (за копейки в расчете на гигабайт), недорогой мобильный телефон; все они работают как минимум на одной ASIC. Иногда ASIC необходимы, когда скорость имеет первостепенное значение или требуется очень большое количество логических элементов. Но в этих случаях они обычно используются только тогда, когда стоимость не является значимым фактором.

Можно разделить затраты на разработку устройства на базе ASIC или FPGA, на индивидуальную разработку нужной функциональности (NRE – Non-Recurring Engineering, «неповторяющиеся инженерные работы»), единовременные затраты на разработку чипа и на стоимость каждого чипа, исключая NRE. Эд Сперлинг в статье *CEO Outlook: It Gets Guch Harder From Here, Semiconductor Engineering, June 3, 2019,* https://semiengineering.com/ceo-outlook-the-easy-stuff-is-over/, утверждает следующее:

> *«NRE для 7-нм чипа составляет от 25 до 30 млн долл., включая набор масок и рабочую силу».*

Эти затраты включают в себя не только наборы масок или, иными словами, модели ASIC, использующиеся для нанесения материалов на кремниевые пластины, из которых создается чип. Это также команды инженеров по проектированию, реализации и верификации, которые могут состоять из сотен сотрудников. Обычно в стоимость ASIC включается и исправление ошибок. Это является одним из значимых факторов, поскольку большие и сложные устройства редко получаются без ошибок с первого раза.

Сравним это с FPGA. Достаточно сложные чипы могут быть разработаны одним человеком или небольшими командами. Большая часть NRE возлагается на поставщиков FPGA при их разработке, у которых хорошие объемы производства. То немногое, что остается от NRE, относится к инструментам и инженерным разработкам. Исправление ошибок ничего не стоит, за исключением времени, так как для перепрограммирования чипа не требуются наборы масок за миллионы долларов.

Компромиссом является стоимость конечного изделия для конкретного случая. Широко распространенные ASIC с низкой сложностью, такие как те, что находятся в карманном калькуляторе или цифровых часах, могут стоить копейки. Стоимость же микропроцессоров может исчисляться сотнями и тысячами долларов. Стоимость FPGA, даже самых недорогих Spartan-7, начинается от нескольких долларов, а самые сложные и быстрые могут достигать десятков тысяч долларов.

Еще один значимый фактор – стоимость инструментов. Как будет показано далее в этой главе, для небольших устройств компания Xilinx предоставляет систему проектирования Vivado в виде бесплатного пакета WebPack. Это ускоряет внедрение, и барьером для входа теперь являются компьютер и плата для разработки. Даже стоимость систем разработки для более сложных устройств составляет всего несколько тысяч долларов, если необходимо приобрести профессиональную копию Vivado. Инструменты ASIC могут стоить миллионы долларов и требуют многолетней практики разработки, поскольку риск неудачи очень высок. Как будет показано на примере проектов, разобранных в этой

книге, где ошибки иногда будут допущены намеренно, стоимость их исправления занимает всего несколько минут времени, потраченных в основном на то, чтобы понять, почему ошибки произошли.

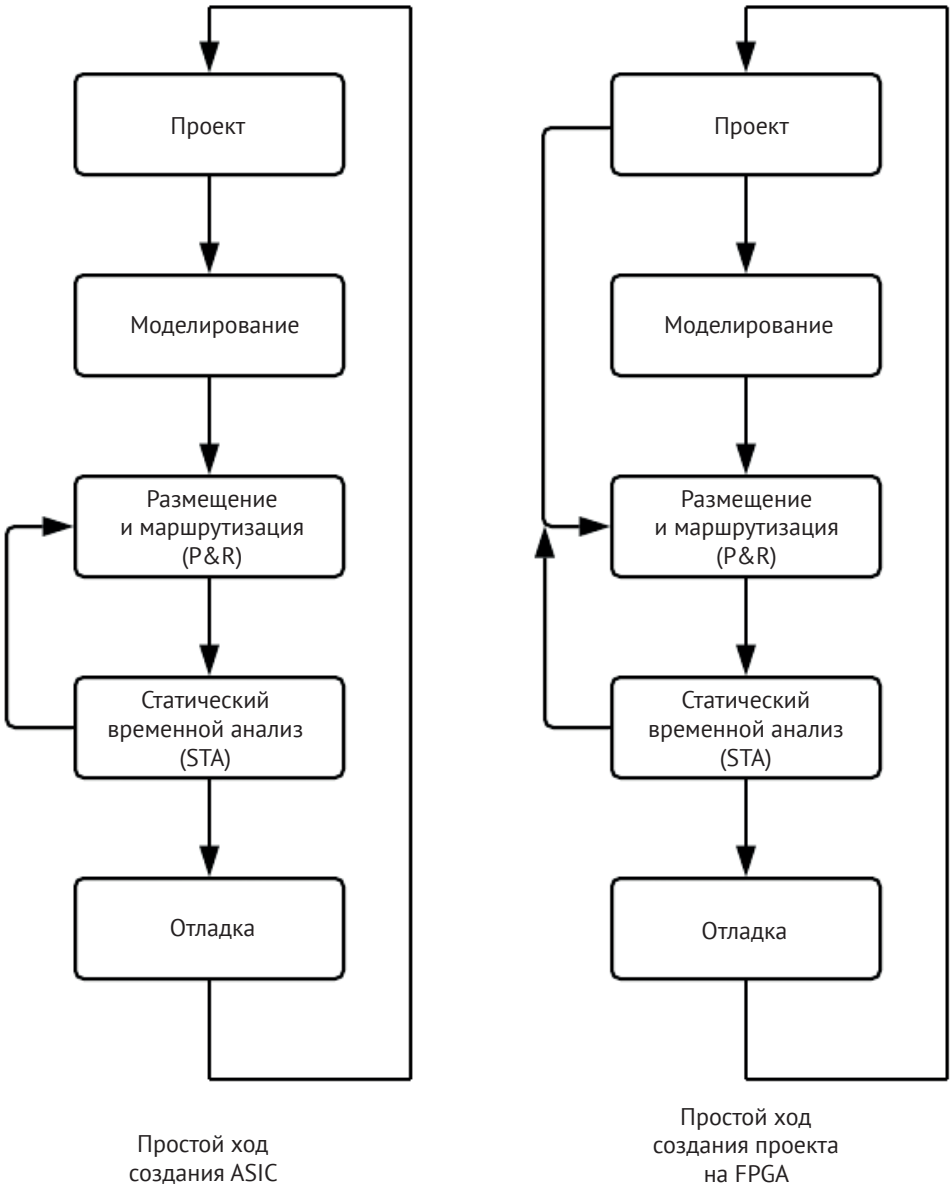


Рис. 1.1. Процессы разработки ASIC и FPGA

Ход создания ASIC и FPGA по сути одинаков. Процесс разработки ASIC, как правило, более линейный, поскольку есть всего один шанс сделать работающее устройство. При разработке FPGA такие вещи, как моделирование, могут

быть опциональными, хотя это настоятельно рекомендуется для сложных проектов. Одно из отличий заключается в том, что этап отладки с использованием ChipScope или аналогичных методов отладки на чипе для отслеживания внутренних сигналов может заменять моделирование. Главное же отличие состоит в том, что в ходе создания FPGA выполнение каждого этапа стоит только затраченного времени. В то же время любые изменения в реализованном проекте ASIC требуют определенного количества новых наборов масок, стоимость которых может исчисляться миллионами долларов.

Мы кратко разобрали, что такое ASIC и для каких целей могут выбираться ASIC и FPGA. Теперь давайте рассмотрим, как создается FPGA с использованием технологического процесса создания ASIC.

Как компания создает программируемое устройство, используя ASIC

Основой любой технологии ASIC является транзистор, причем в самых больших устройствах их количество достигает миллиардов единиц. Существует множество процессов ASIC, разработанных за многие годы, и все они основаны на двоичной логике, другими словами, на включенных или выключенных транзисторах. Эти включенные или выключенные транзисторы можно представить как логические значения «истина» и «ложь».

Основы алгебры логики были разработаны Джорджем Булем в 1847 году. Основополагающие принципы алгебры логики лежат в основе работы логических элементов, на чем строится вся цифровая логика. Код, который будет разрабатываться, будет достаточно высокого уровня, но важно понимать основы, что дает хорошую базу для первого проекта.

Базовые логические элементы

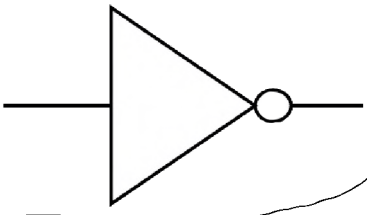
Существует четыре основных типа логических элементов. Обычно приводят таблицы истинности для этих элементов, чтобы разобраться с их функциональностью. Таблица истинности показывает, что будет на выходе для каждого набора входов схемы. Это показано на примере с логическим элементом НЕ (NOT).

> **Важное замечание**
>
> В этом разделе мы рассматриваем в основном только логические функции. Существуют эквивалентные побитовые функции, которые будут представлены позднее. Логические функции обычно используются в операторах if, а побитовые функции – в логических операциях.

Оператор присваивания

В SystemVerilog можно использовать оператор присваивания `assign`, чтобы присвоить значение, находящееся в правой части от знака равенства, его левой части. Он используется следующим образом:

```
assign out = in;
```



Вход	Выход
0	1
1	0

(FIFO, «первым пришел – первым ушел») для сглаживания нерегулярных выбросов данных[1] или для пересечения тактовых доменов[2], нужно проверить, есть ли данные, прежде чем принимать их из очереди для использования. FIFO имеют флаги, используемые для контроля состояния, два наиболее распространенных из них – полный и пустой. Мы можем решить задачу, проверив пустой флаг, как было показано ранее.

В последующих главах будет более подробно рассмотрено, как спроектировать FIFO, а также как его использовать.

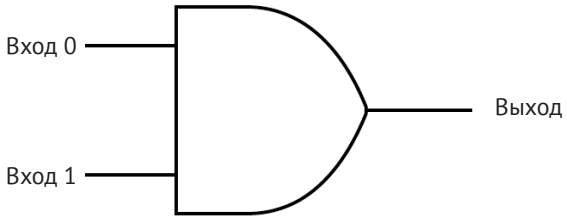
Логическое И (&&), побитовое И (&)

Часто необходимо проверить, являются ли активными одно или несколько условий одновременно. Для этого используют логический элемент `И (AND)`.

Функция на языке SystemVerilog может быть записана следующим образом:

```
assign out = in1 && in0; // логический оператор
```

Соответствующая таблица истинности выглядит так:



Графическое представление

Вход 0	Вход 1	Выход
0	0	0
0	1	0
1	0	0
1	1	1

Таблица истинности

Рис. 1.3. Представление логического элемента И (AND)

Продолжая пример с FIFO, можно извлекать данные из одного FIFO и помещать в другой:

```
if (!src_fifo_empty && !dst_fifo_full) ...
```

В этом случае вычисление логического выражения позволяет убедиться, что в исходном FIFO есть данные (там не пусто) и что место назначения не переполнено. Это можно сделать, проверив с помощью оператора `if`.

Логическое ИЛИ (||), побитовое ИЛИ (|)

Другая распространенная задача – проверить, установлен ли какой-либо один сигнал из группы сигналов для выполнения операции.

Функция в SystemVerilog может быть записана следующим образом:

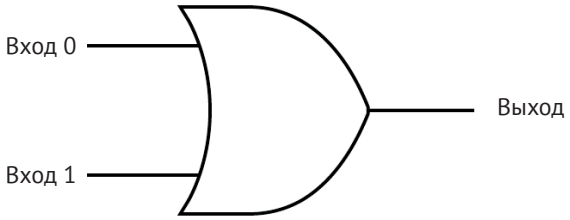
```
assign out = in1 ||  in0; // логический оператор
```

Соответствующая таблица истинности выглядит так:

---

[1] Подобнее см. главу 5. – *Прим. ред.*

[2] Пересечение тактовых доменов (clock domains crossing) – задача, возникающая при передаче битов данных между двумя частями устройства, управляющихся от разных источников синхросигналов. Подробнее см. главу 3. – *Прим. ред.*



Графическое представление

Вход 0	Вход 1	Выход
0	0	0
0	1	1
1	0	1
1	1	1

Таблица истинности

Рис. 1.4. Представление логического элемента ИЛИ (OR)

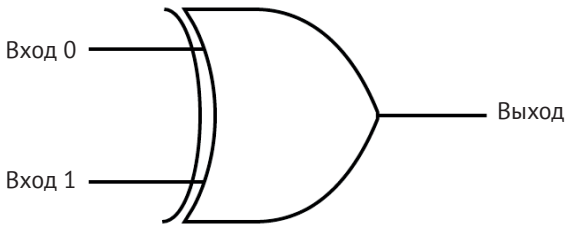
Далее рассмотрим функцию исключающего ИЛИ (`exclusive OR`).

Исключающее ИЛИ (XOR, ^)

Функция исключающего ИЛИ проверяет, установлен ли один из двух входов, но не оба. Функцию в SystemVerilog можно записать следующим образом:

```
assign out = in1 ^ in0; // логический оператор
```

Соответствующая таблица истинности выглядит так:



Графическое представление

Вход 0	Вход 1	Выход
0	0	0
0	1	1
1	0	1
1	1	0

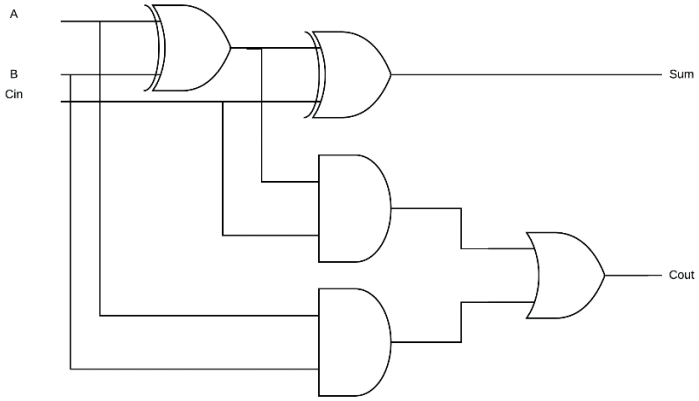
Таблица истинности

Рис. 1.5. Представление логического элемента Исключающее ИЛИ (XOR)

Эта функция используется при построении сумматоров, схем четности и для создания кодов для коррекции и проверки ошибок. В следующем разделе будет показано, как строится сумматор с использованием представленных логических элементов.

Более сложные операции

В предыдущих разделах были рассмотрены основные компоненты, из которых состоит любая цифровая схема. Разберем пример того, как можно объединить несколько логических элементов для выполнения задачи. Для этого введем понятие полного сумматора. Полный сумматор принимает три входа, A, B и перенос, и формирует результат на двух выходах: сумме и переносе. Таблица истинности сумматора имеет следующий вид:



A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Таблица истинности

Графическое представление полного однобитного сумматора

Рис. 1.6. Полный сумматор

Код SystemVerilog для полного сумматора, представленный в виде логических функций, будет выглядеть следующим образом:

```
assign Sum = A ^ B ^ Cin;
assign Cout = A & B | (A^B) & Cin;
```

Заметьте, что используются побитовые операторы `И (AND, &)` и `ИЛИ (OR, |)`, поскольку операции выполняются над битами. Этот простой, но важный пример показывает, что реальная функциональность может быть построена из базовых строительных блоков. Фактически все схемы в ASIC или FPGA построены таким образом, но благодаря распространению **языков проектирования высокого уровня** (**High-Level Design Languages, HDL**), таких как SystemVerilog, нет необходимости погружаться на этот уровень детализации, если только это действительно не необходимо.

Знакомство с FPGA

Массив логических элементов (вентилей) в терминах ASIC – это множество логических элементов с некоторым количеством настраиваемых соединений, которые могут быть сконфигурированы для конкретного приложения. Это позволяет получить более дешевый продукт, поскольку компании, разрабатывающей ASIC, нужно платить только за маски, необходимые для настройки. FPGA делает еще один шаг вперед, обеспечивая программируемость матрицы как части устройства. Это приводит к увеличению стоимости, поскольку потребитель платит за неиспользуемые соединения и память, необходимые для конфигурирования структуры FPGA, но позволяет и несколько снизить стоимость, поскольку эти части становятся стандартными устройствами, которые можно производить в больших количествах.

Если проанализировать функции из предыдущего раздела на примере сумматора, то можно отметить одну общую черту: все они могут быть получены с помощью таблицы истинности, которая становится ключевой при разработке на FPGA. Можно рассматривать эти таблицы истинности как представления функций в **постоянном запоминающем устройстве (Read Only Memory,**

ROM). Фактически можно рассматривать их как **программируемые ROM (PROM)** в случае создания FPGA.

Разберем пример основных логических функций. Можно воспроизвести любую из них с помощью **LUT (Lookup Table, таблицы поиска)**[1] с двумя входами, которая может выглядеть следующим образом:

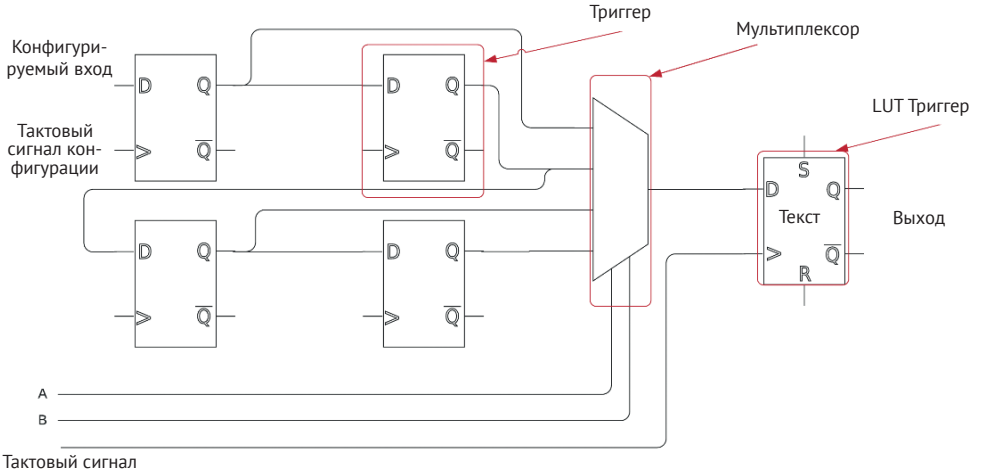


Рис. 1.7. Пример LUT с двумя входами

Хотя это очень упрощенный пример, здесь имеется четыре запоминающих элемента, в данном случае триггера, а в случае реальной FPGA, скорее всего, гораздо более простая структура, которая использует намного меньше транзисторов. Эти запоминающие элементы соединены друг с другом таким образом, что их конфигурация может быть изменена. Присоединение других таблиц поиска к цепочке позволяет конфигурировать несколько LUT при запуске или, в случае частичной реконфигурации, во время обычной работы. Окончательная структура LUT сформируется при добавлении триггера.

Преимущество простоты структуры заключается в возможности многократного повторения этой схемы. В случае современных FPGA они строятся из множества рядов логических элементов, подобных приведенному, что позволяет разработать, реализовать и проверить гораздо более простой элемент схемы, а затем воспроизвести его для создания устройств с большим количеством логических элементов. Это позволяет конструировать широкий спектр недорогих устройств с небольшим количеством элементов и более крупные устройства с большим количеством элементов. В некоторых проектах даже используют технологию **Stacked Silicon Interconnects (SSI)**, которая, по сути, является технологией объединения нескольких FPGA внутри одного корпуса.

[1] Lookup Tables, LUT, – метод реализации функции, в котором непосредственное вычисление заменяется поиском по таблице соответствия выходов входам. Позволяет заменить медленные вычисления на быстрый поиск в памяти (аналогично использованию готовых таблиц различных функций в эпоху, когда компьютеры еще массово не использовались). – *Прим. ред.*

В 1985 году компания Xilinx представила микросхему XC2064, которую можно считать первой FPGA, использующей массив из 64 LUT с тремя входами и с одним триггером. Прорывная идея в этой конструкции заключалась в том, что она была модульной и имела хорошие взаимосвязи между отдельными составляющими. Вся эта конструкция была приблизительно эквивалентна одному **настраиваемому логическому блоку (Combination Logic Block, CLB)** в Artix-7.

В основе FPGA лежит программируемая матрица. Матрица состоит из LUT с соответствующими триггерами, составляющими отдельные slice[1], из которых состоят CLB. Все эти блоки соединены между собой с помощью разветвленной сети каналов маршрутизации, что позволяет создавать практически любые конфигурации. FPGA также содержат множество других ресурсов, которые будут представлены по ходу этой книги: блочные ОЗУ (RAM), преобразователи из параллельного кода в последовательный и обратно (англ. *serial-deserial*, *SERDES*), элементы цифровой обработки сигналов (DSP) и множество типов программируемых входов/выходов.

Изучение Xilinx Artix-7 и устройств 7-й серии

FPGA, которые рассматриваются в этой книге, относятся к серии устройств Artix-7. Эти устройства обладают самой высокой производительностью на затраченный ватт мощности среди устройств Xilinx 7-й серии. При разумной цене они обладают большим количеством относительно высокопроизводительной логики для реализации проектов. Компоненты FPGA, которые здесь представлены, являются общими для устройств Spartan (младшего класса), Kintex (среднего класса) и Virtex (старшего класса) 7-й серии.

Комбинационные логические блоки

ASIC состоят из логических элементов, основанных на библиотеках, предоставляемых производителями ASIC, такими как TSMC или Tower. Эти библиотеки могут содержать все, начиная от логических элементов `И` (`AND`), `ИЛИ` (`OR`) и `НЕ` (`NOT`) и заканчивая более сложными математическими блоками и элементами хранения данных. При разработке FPGA используются те же уравнения алгебры логики, что и в ASIC. Но процесс синтеза будет нацелен на реализацию с помощью CLB, входящих в состав FPGA.

CLB состоит из пары `slice`, каждый из которых содержит четыре LUT с шестью входами и восемь триггеров. Vivado (или, по желанию, сторонний инструмент синтеза, например Synopsys Synplify) компилирует код SystemVerilog и сопоставляет его с элементами CLB. Чтобы полностью изучить детали того, как устроены CLB, рекомендуется прочитать «Руководство пользователя Xilinx UG474, 7 Series FPGA CLB» (https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf). На высоком уровне каждая LUT обеспечивает такую степень гибкости, что можно реализовать любую логическую функцию с 6 входами или две произвольно определенные функции с 5 входами, если они имеют общие входы. Также имеется специальная высокоскоростная логика переноса для арифметических операций, которая будет обсуждаться в последующих главах.

[1] Slice (букв. часть, доля, ломтик) – специфический для FPGA термин, в русскоязычной литературе в силу отсутствия устоявшегося термина принято не переводить. – *Прим. ред.*

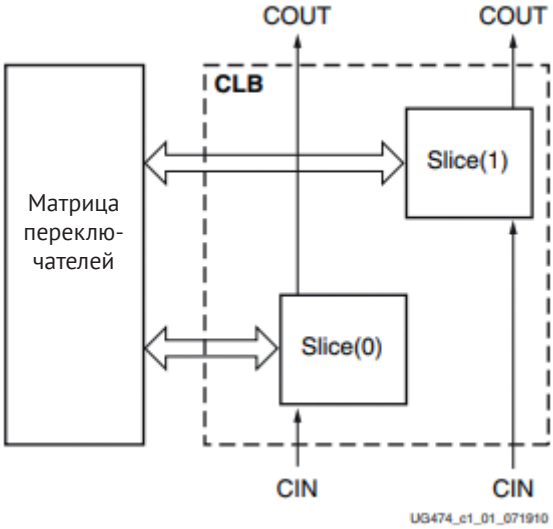


Рис. 1.8. Рисунок 1-1 из «Руководства пользователя CLB FPGA Xilinx UG474 7-й серии» (используется с разрешения)

`Slice` имеют два формата: `SLICEL` (логика) и `SLICEM` (память). `SLICEM` является надмножеством `SLICEL`. `SLICEM` добавляет возможность конфигурировать `Slice` в распределенное ОЗУ (RAM) или сдвиговый регистр. Существует примерно в три раза больше `SLICEL`, чем `SLICEM`. В таблице ниже показано их распределение для двух рассматриваемых в этой книге отладочных плат.

Плата	Устройство	Slices	SLICEL	SLICEM	LUT с 6 входами	Распределенное ОЗУ (Кб)	Регистр сдвига (Кб)	Триггеры
Basys 3	7A35T	5,200	3,600	1,600	20,800	400	200	41,600
Nexys A7	7A100T	15,850	11,100	4,750	63,400	1,188	594	126,800

Хотя теоретически возможно создать экземпляры компонентов нижнего уровня, такие как slice или LUT, и принудительно задействовать их функциональные возможности, это выходит за рамки данной книги, а такой метод не имеет широкого применения. В этой книге акцент сделан на использование CLB через синтез в Vivado разработанного проекта на SystemVerilog.

Встроенная память

Помимо `SLICEM`, составляющих CLB, которые могут использоваться в качестве памяти или регистров сдвига, FPGA содержат **блоки памяти с произвольным доступом (Block RAM, BRAM),** являющиеся более крупными элементами хранения данных. Все элементы 7-й серии имеют 36 Кбит BRAM, которые могут быть разделены на две 18 Кбит BRAMs. В следующей таблице показана BRAM, доступная на рекомендуемых отладочных платах.

Плата	Устройство	Количество блоков BRAM по 36 Кбит
Basys 3	7A35T	50
Nexys 7 A7	7A100T	135

Память BRAM может быть сконфигурирована следующим образом:

- полностью двухпортовая память – два порта, каждый порт на чтение и запись;
- простая двухпортовая память – один порт на чтение, один на запись. В этом случае память BRAM размером 36 Кбит может иметь длину слова до 72 бит, а память BRAM размером 18 Кбит – до 36 бит;
- однопортовая память.

Содержимое BRAM может быть загружено при инициализации и настроено через файл или начальный блок в коде. Это может быть полезно для реализации ПЗУ (ROM) или создания условий запуска.

BRAM в устройствах 7-й серии также содержат логику для реализации FIFO. Это экономит ресурсы CLB, уменьшает издержки на синтез и устраняет потенциальные проблемы с временными параметрами проекта. FIFO будет рассмотрено в одной из следующих глав.

Все 36 Кбит BRAM имеют выделенные функции **корректирующего кода (Error Correction Code, ECC).** Поскольку это больше относится к приложениям с высокой надежностью, таким как медицинские, автомобильные или космические, в данной книге на них подробно останавливаться нет необходимости.

Тактирование

В устройствах 7-й серии реализована разнообразная методология систем тактирования, которую можно подробно изучить в «Руководстве пользователя UG472 7 Series FPGA clocking resources» (https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf). Для большинства целей предоставленной в разделе PLL информации будет вполне достаточно, но в документе по ссылке можно найти гораздо больше подробностей.

Устройства ввода/вывода (I/Os)

В основном рассмотрение ограничено устройствами ввода/вывода, поддерживаемыми двумя целевыми платами для разработки. В целом устройства 7-й серии поддерживают множество интерфейсов от 3.3 В CMOS/TTL до LVDS[1] и интерфейсов памяти. Используемые платы определяют используемые типы устройств ввода/вывода в проектных файлах. Для получения дополнительной информации обо всех поддерживаемых типах ввода/вывода можно обратиться к «Руководству пользователя UG471 7 Series FPGA SelectIO resources».

DSP48E1

FPGA занимают заметное место в приложениях **цифровой обработки сигналов (Digital Signal Processing, DSP),** в которых используется большое ко-

[1] Low-voltage differential signaling, «низковольтная дифференциальная передача сигналов», – стандарт передачи на высоких частотах с помощью витой пары. Широко распространенный пример LVDS – проводной Ethernet. – *Прим. ред.*

личество функций **умножения-сложения (Multiply Accumulate, MAC)**. Одной из первых инноваций в FPGA было включение аппаратных умножителей, за которыми последовали блоки DSP, способные реализовать функции MAC.

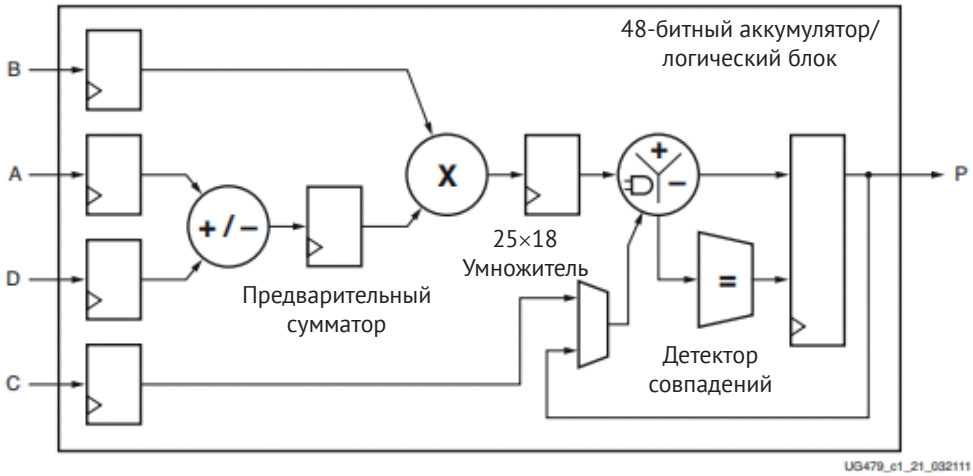


Рис. 1.9. Рисунок. 1-1 из «Руководства пользователя Xilinx UG479 серии 7 DSP48E1» (используется с разрешения)

Арифметические операции – одни из самых дорогих в FPGA. В ASIC самой громоздкой и медленной обычно является операция умножения, а самой компактной и быстрой – операция сложения. По этой причине в течение многих лет производители FPGA внедряли в свои матрицы аппаратные арифметические блоки. В FPGA все наоборот: более медленной операцией обычно является сложение, особенно при увеличении разрядности слагаемых. Причина этого заключается в том, что умножение превратилось в сложную конвейерную операцию. Блок DSP подробнее рассмотрен в следующих главах. «Руководство пользователя UG479 7 Series DSP48E1» (https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf) является хорошим справочником, если нужно углубиться в детали.

Архитектура ASMBL

Устройства 7-й серии – это четвертое поколение устройств, в которых компания Xilinx использует архитектуру Advanced Silicon Modular Block (ASMBL) для целей практического применения. Идея заключается в том, чтобы создать платформы FPGA, оптимизированные для различных целевых приложений. Рассматривая семейства 7-й серии, можно заметить, как различные конфигурации `slice` объединяются для достижения этих целей. Можно увидеть, как элементы, рассмотренные в этой главе, расположены друг за другом в виде колонок, чтобы предоставить ресурсы, которые будут использованы для будущих проектов.

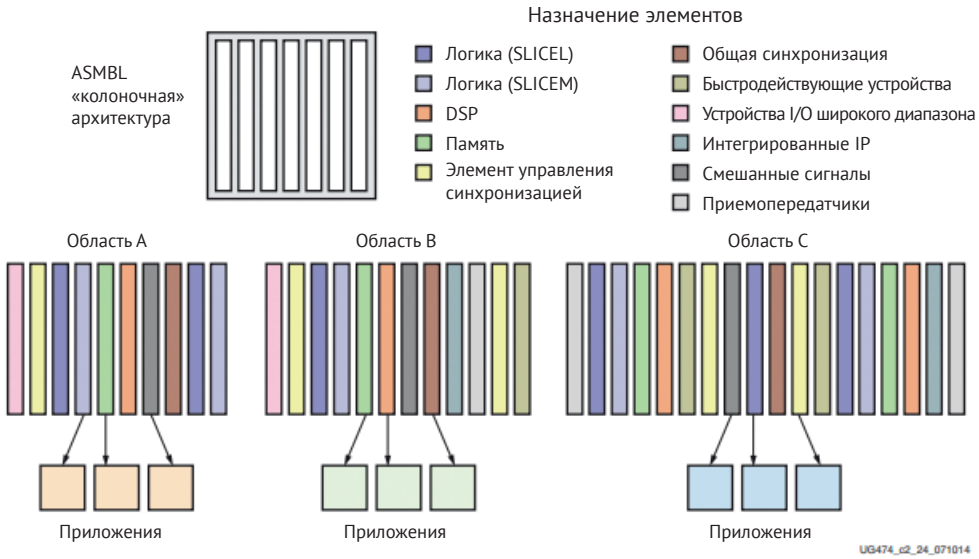


Рис. 1.10. Рисунок 2-1 из «Руководства пользователя CLB FPGA Xilinx UG474 серии 7» (использовано с разрешения)

Теперь, когда рассмотрение, из чего состоит Artix-7 и другие устройства 7-й серии, подошло к концу, нужно установить инструменты Xilinx, чтобы можно было приступить к первому проекту.

Знакомство с набором инструментов Vivado и отладочными платами

В этом разделе будут рассмотрены отладочные платы, рекомендуемые для проектов в данной книге. Также будет представлен очень простой проект с использованием Vivado для знакомства с инструментами проектирования, и будет показано, как программировать плату.

Отладочные платы

На рынке нет недостатка в отладочных платах для FPGA, которые можно легко приобрести. Одной из компаний, выпускающих очень доступные платы, является Digilent. Их платы имеют несколько приятных особенностей, самая важная среди которых – наличие встроенного контроллера UART с USB, который Xilinx Vivado распознает как кабель для программирования. Это делает настройку устройства очень простой. Рекомендуемые платы также имеют дополнительное преимущество: питание осуществляется по этому же USB-кабелю.

Nexys A7 100T (или 50T)

Nexys A7 – это плата, рекомендованная для данной книги. На ней есть все устройства, которые будут использованы в ходе работы далее.

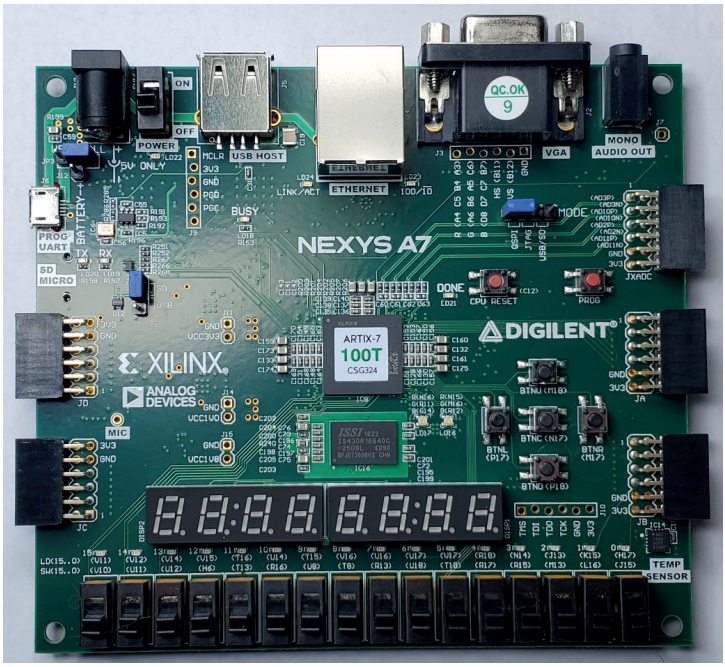


Рис. 1.11. Плата Digilent Nexys A7

Плата имеет следующие характеристики:

- Artix-7 XC7A100T или 50T;
- работа на частоте 450+ МГц;
- 128 МБ DDR2;
- последовательная флеш-память;
- встроенный USB UART для загрузки изображений и отладки с помощью ChipScope;
- устройство чтения карт памяти MicroSD;
- 10/100 Ethernet PHY;
- PWM-аудиовыход/вход для микрофона;
- датчик температуры;
- трехосевой акселерометр;
- 16 переключателей;
- 16 светодиодов;
- 5 кнопок;
- два трехцветных светодиода;
- два четырехзначных семсегментных индикатора;
- поддержка устройств USB;
- пять PMOD[1] (один XADC).

[1] Интерфейс PMOD (интерфейс периферийного модуля) – открытый стандарт, определенный производителем плат разработки Digilent Inc. для подключения различных устройств, от простых кнопок до аналогово-цифровых преобразователей и дисплеев. – *Прим. ред.*

Рассмотрим характеристики устройств, с которыми может быть заказана плата Nexys:

Устройство	XC7A100-1CSG324C	XC7A50T-1CSG324C
Логические slices	15,850	8,150
BRAM (Кбиты)	4,860	2,700
Блоки управления тактовыми сигналами	6	5
DSP	240	120

Одним из преимуществ выбора XC7A100T является наличие дополнительной оперативной памяти. Особенно в начале работы вам может потребоваться отладка микросхем с помощью ChipScope[1], а дополнительная оперативная память позволит сохранять более широкие шины или увеличить время хранения. Более подробно про ChipScope рассказано в следующей главе.

Basys 3

Альтернативной оценочной платой является Basys 3.

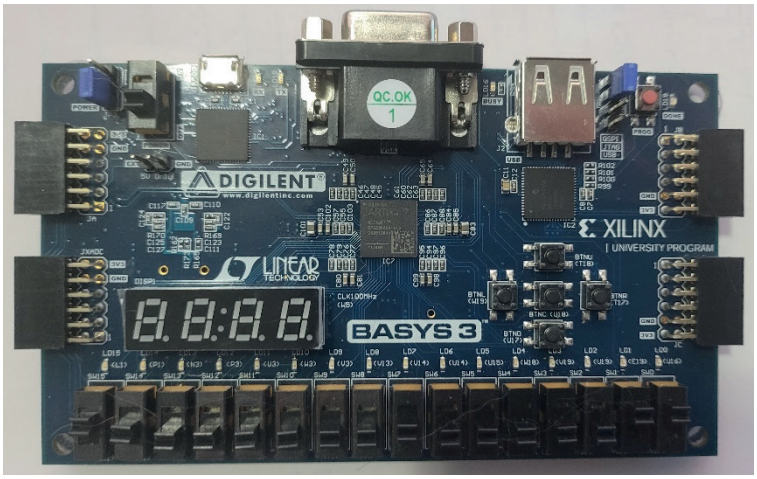


Рис. 1.12. Плата Digilent Basys 3

Эта плата имеет то же количество кнопок, светодиодов и переключателей, но только вдвое меньше семисегментных индикаторов. Демонстрационные примеры, приведенные в этой книге, могут работать на любой из этих плат, используя их ресурсы. На данной плате отсутствует оперативная память DDR2, поэтому ее использование для буфера кадров будет ограничено, о чем будет рассказано в следующей главе. На ней также отсутствуют датчик температуры, микрофон и аудио, которые будут рассмотрены во время работы с последова-

[1] ChipScope – виртуальный логический анализатор фирмы Xilinx, который может вызывать логические ресурсы внутри FPGA для захвата и анализа переменых в коде. – *Прим. ред.*

тельными интерфейсами. Но для преодоления этих ограничений можно приобрести дополнительные платы PMOD, обладающие этой функциональностью.

Плата имеет следующие характеристики:

- Artix-7 XC7A35T;
- работа на частоте 450+ МГц;
- последовательная флеш-память;
- встроенный USB UART для программирования платы и отладки с помощью ChipScope;
- устройство чтения карт памяти MicroSD;
- 10/100 Ethernet PHY;
- PWM-аудиовыход/вход для микрофона;
- 16 переключателей;
- 16 светодиодов;
- 5 кнопок;
- два трехцветных светодиода;
- одиночный четырехразрядный семисегментный индикатор;
- поддержка устройств USB;
- четыре PMOD (один двойного назначения, поддерживающий XADC).

Характеристики платы Basys 3 следующие:

Устройство	XC7A35T-1CSG324C
Логические секции	5,200
BRAM (Кбиты)	1,800
Блоки управления тактовыми сигналами	5
DSP	90

Важное замечание

На плате Basys 3 отсутствует память DDR2, акселерометры и интерфейс для аудио, которые будут рассмотрены в последующих главах. Для всего, кроме DDR2, имеются PMOD. Использовать Nexys A7 вместо Basys предпочтительней.

Таким образом, представлены два варианта плат, которые планируется использовать для апробации примеров из этой книги. Теперь нужно ознакомиться с инструментом Xilinx Vivado, который будет использоваться для проектирования, моделирования, сборки и отладки FPGA-проектов.

Знакомство с Vivado

После того как вы выбрали плату, лучший способ познакомиться с ней – это поработать над проектом.

Vivado – это инструмент Xilinx, который будет использоваться для сборки, тестирования, загрузки и отладки проектов. Он может быть запущен как инструмент командной строки или в режиме создания проекта с помощью гра-

фического интерфейса пользователя (GUI). Для целей обучения он будет использоваться в режиме работы над проектом с помощью GUI.

Установка Vivado

Для небольших устройств компания Xilinx предоставляет Vivado в свободном доступе в виде пакета WebPack, который содержит все возможности полной версии с ограничением поддержки некоторых устройств. Он доступен как для Windows, так и для Linux. В книге приведены скриншоты версии для Linux, но все апробировано на обеих ОС, поэтому можно использовать любую из них.

> **Важное замечание**
>
> Vivado WebPack принудительно передает информацию о работе данного САПР в Xilinx. В платной версии это можно отключить.

Для установки Vivado выполните следующие действия.

1. Создайте учетную запись на сайте https://www.xilinx.com/.
2. Посетите https://www.xilinx.com/support/download.html.
3. Загрузите Xilinx Unified Installer. Для этой книги используется версия 2020.1.
4. В Windows запустите файл .exe.

В Linux используйте следующие команды:

```
chmod +x Xilinx_Unified_2020.1_0602_1208_Lin64.bin; ./
Xilinx_Unified_2020.1_0602_1208_Lin64.bin
```

5. Введите данные своей учетной записи для установки.
6. Когда появится всплывающее окно, можно установить либо Vitis, либо Vivado. Для целей освоения этой книги нет необходимости использовать Vitis, но он включает в себя Vivado, поэтому, если вы любите преодолевать трудности и хотите попробовать Vitis, смело устанавливайте и его.
7. При выборе установки поддерживаемых устройств достаточно указать только 7-ю серию.
8. Выберите место для установки или используйте вариант по умолчанию.

Установка может занять некоторое время.

Структура каталогов

Установив Vivado, можно выполнить очень простой проект, чтобы познакомиться с Vivado и убедиться, что все настроено правильно. Структура каталогов, которую рекомендуется использовать, выглядит следующим образом:

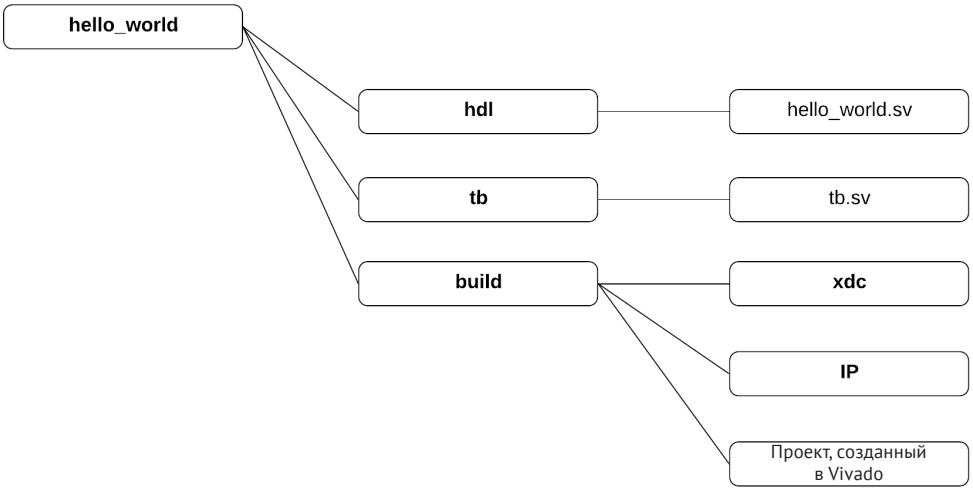


Рис. 1.13. Структура каталогов

Элементы, выделенные жирным шрифтом, являются каталогами. Для первого примера проекта требуется не так много кода. В итоге будет создано только три файла: исходный код HDL, тестовый файл `testbench` и файл ограничений.

Внутри каталога `hdl` создадим простой проект `logic_ex.sv` для запуска в Vivado[1]:

Logic_ex.sv

```
`timescale 1ns/10ps
module logic_ex
   {
   input  wire  [1:0]    SW,
   output logic [3:0]    LED
   );
  assign LED[0]  = !SW[0];
  assign LED[1]  = SW[1] && SW[0];
  assign LED[2]  = SW[1] || SW[0];
  assign LED[3]  = SW[1] ^ SW[0];
  endmodule // logic_ex
```

Сначала определим масштаб времени для работы симулятора. Значения 1 нс / 10 пс были довольно распространенным стандартом много лет назад, и для данного проекта это вполне подойдет. Если вы будете использовать вы-

[1] В коде дальнеших примеров постоянно встречаются записи в квадратных скобках (вроде `wire[1:0]` или `logic[3:0]`). Так задается размерность для векторов (в терминологии автора – упакованных значений многоразрядных переменных): например, запись `logic[3:0]` означает, что вектор `logic` имеет размерность 4 бита (с номерами 0, 1, 2, и 3), запись `wire[1:0]` – что `wire` имеет размерность 2 бита (номер 0 и 1). Отдельные биты вектора `logic` обозначаются как `logic[0]`, `logic[1]`, `logic[2]` и `logic[3]`. О массивах, в записи которых также присуствуют квадратные скобки, рассказывается в главе 2. – *Прим. ред.*

сокоскоростные приемопередатчики, то потребуется указывать еще меньшие периоды времени, такие как 1 пс / 1 фс.

> **Подсказка**
>
> Каждый модуль должен находиться в своем собственном файле, и этот файл должен быть назван так же, как и модуль. Это может облегчить жизнь при использовании некоторых инструментов, таких как коммерческие симуляторы или даже пользовательские сценарии.

Синтаксис для определения периода времени следующий:

```
`timescale  < time unit >/<time precision>
```

Здесь `time unit` определяет значение и единицу измерения задержек, `time precision` задает точность округления. Это значение обычно можно переопределить в симуляторе, и эти настройки не влияют на синтез. При использовании `` `timescale `` лучше всего установить его во всех файлах.

Написание	Сокращение (русск)	Единица измерения времени
s	с	Секунды
ms	мс	Миллисекунды
us (μs)	мкс	Микросекунды
ns	нс	Наносекунды
ps	пс	Пикосекунды
fs	фс	Фемтосекунды

Список портов состоит из одного порта, представляющего собой двубитное значение, которое будет подключено к двум крайним правым переключателям на плате. Также определим один выход с именем LED, который представляет собой четыре бита, соответствующие четырем светодиодам над четырьмя крайними правыми переключателями.

---

tb.sv

```
`timescale 1ns/ 100ps;
module tb;
  logic [1:0] SW;
  logic [3:0] LED;
  logic_ex u_logic_ex (.*);
  //logic_ex u_logic_ex (.SW, .LED);
  //logic_ex u_logic_ex (.SW(switch_sig), .LED(led_sig));
  //logic_ex u_logic_ex (.*, .LED(led_sig));
```

В приведенном примере объявлен модуль верхнего уровня под названием `tb`. Обратите внимание, что модуль `testbench` верхнего уровня не должен иметь никаких портов. В нем также объявлены две шины типа logic, которые будут подключены к модулю `hello world`.

В модуле tb создается компонент logic_ex и его экземпляр u_logic_ex. Существует несколько способов подключения портов. В незакомментированном примере используется обозначение .*, что соединит все порты с тем же именем, что и определенный сигнал в шаблонном модуле.

Во втором примере (закомментированном) используется .<name> (имя) порта, который нужно подключить. Для этого требуется, чтобы имя порта уже было определено.

Наконец, если есть сигнал с другим именем, можно использовать третий способ, который позволяет переименовать порт. Можно смешивать .* с переименованными портами, как показано в последнем примере.

Testbench обычно состоит из двух отдельных частей – генератора тестовых сигналов и блока их проверки:

```
// Stimulus
 initial begin
   $printtimescale(tb);
   SW = '0;
   for (int i = 0; i < 4; i++) begin
     $display("Setting switches to %2b", i[1:0]);
     SW = i[1:0];
     #100;
   end
 $display("PASS: logic_ex test PASSED!");
 $stop;
end
```

Блок тестовых сигналов прост, потому что прост проект, который тестируется. Можно полностью поместить его в начальный блок, который последовательно запускается после запуска симулятора. Сначала он выводит период времени, используемый в файле tb.sv. Затем вход SW в модуль logic_ex устанавливается равным 0. Использование константы '0 при присвоении SW информирует средства моделирования, что нужно установить все биты равными 0. Существует также соотвествующее значение константы '1, которое устанавливает все биты равными 1, или 'z[1]. Правила Verilog говорят, что присвоение SW = 0 эквивалентно SW = 32'b0, что приведет к предупреждению об изменении размеров данных. Во избежание появления предупреждений предпочтительнее использовать '0, '1 или 'z.

Важное замечание

SystemVerilog – это HDL (Hardware Description Language). Язык HDL должен быть способен моделировать параллельные операции, поскольку многие или все секции в FPGA будут все время работать параллельно. SW = ‘0 – это блокирующее присваивание. Таким образом, присваивание выполняется до перехода к следующему действию. Обсуждение блокирующего и неблокирующего присвоения будет при разборе темы про последовательностные схемы.

[1] Под состоянием z здесь имеется в виду высокоимпедансное третье состояние. – *Прим. ред.*

Затем блок цифровых сигналов повторяется четыре раза с помощью цикла `for`. В SystemVerilog есть возможность объявить переменную цикла внутри цикла `for`, в данном случае это `i`. Настоятельно рекомендуется объявлять ее таким образом, чтобы избежать предупреждений о многократном использовании, когда используется одно и то же именование переменных в нескольких циклах `for`.

Внутри цикла `for` выводится текущее состояние переключателей с помощью системной функции `$display`. Поскольку требуется вывести только 2 бита, которые инкрементируются без ведущих нулей, то в маске вывода указывается `2%b`. Затем `SW` присваиваются значения младших двух битов `i`. Хотя в этом нет необходимости, в цикл добавлена еще задержка на 100 нс с использованием выражения `#100`.

Вызов системной функции `$stop` приводит к завершению моделирования.

> **Важное замечание**
>
> Задержка происходит в наносекундах из-за настроек периода времени, который определен в `testbench`.

Рассмотрим блок проверки. В любом хорошем `testbench` блок проверки должен быть самопроверяемым. Это означает, что в конце теста должно выводиться сообщение о том, пройден тест или нет, а если не пройден, то почему. Это также означает, что разработка `testbench` часто может быть такой же трудоемкой или даже более сложной, чем разработка кода проекта для реализации на FPGA. Но это выходит за рамки данной книги. Все коммерческие симуляторы, включая симулятор Vivado, также поддерживают универсальную методику верификации (Universal Verification Methodology), которая представляет собой набор классов и функций SystemVerilog, специально созданных для верификации HDL-проектов.

```
 always @(SW, LED) begin
    if (!SW[0] !== LED[0]) begin
      $display("FAIL: NOT Gate mismatch");
      $stop;
    end
    if (&SW[1:0] !== LED[1]) begin
      $display("FAIL: AND Gate mismatch");
      $stop;
    end
    if (|SW[1:0] !== LED[2]) begin
      $display("FAIL: OR Gate mismatch");
      $stop;
    end
    if (^SW[1:0] !== LED[3]) begin
      $display("FAIL: XOR Gate mismatch");
      $stop;
    end
  end
endmodule
```

В отличие от блока генерации тестовых сигналов требуется, чтобы этот блок реагировал на события в проекте. Для этого используется блок `always`,

чувствительный только к изменениям на входах `SW` и выходах LED проекта. Это простой случай, когда каждый бит LED сопоставляется соответствующим битам **`SW`**, проходящим через соответствующие логические элементы. Для этого используется оператор `!==`, который является оператором сравнения на неравенство, но при этом учитывает неопределенные состояния (x) в тех случаях, когда в проекте есть ошибки. Более сложные `testbench` будут рассмотрены в следующих главах.

В примере также используются операторы приведения (`&`, `|` и `^`), которые применяются к двум битам `SW`. Запись `&SW[1:0]` эквивалентна `SW[0]&SW[1]`.

Выполнение примера

На этом этапе следует скопировать файлы для этой книги с GitHub или клонировать репозитарий.

Загрузка проекта

Загрузите проект в Vivado.

1. В Windows найдите, где установлен Vivado, и дважды щелкните по иконке Vivado. В Linux процедура выглядит следующим образом[1]:

```
Source <Vivado Install>/settings64.sh (or.csh)
Vivado
```

2. Выполните *шаги 2 и 3* при первом запуске Vivado.
3. Откройте **Xhub Stores**.

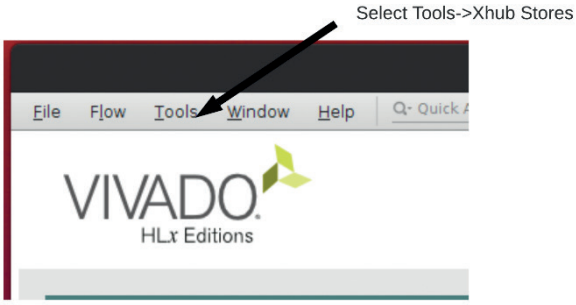


Рис. 1.14. Xhub Stores

Xilinx Xhub Stores – это удобный способ добавления скриптов, конфигурационных файлов для плат и тестовых проектов в Vivado.

4. Инсталлируйте конфигурационный файл платы, используемой в тестовом проекте.

Выберите вкладку **Boards**, а затем перейдите к Digilent Artix A7 100T, или 35T, или Basys 3. Можно увидеть, что существует довольно много коммерческих плат, которые предоставляют свои файлы для установки:

[1] Во всех современных версиях Vivado необходимо использовать `settings64.sh`. – *Прим. конс.*

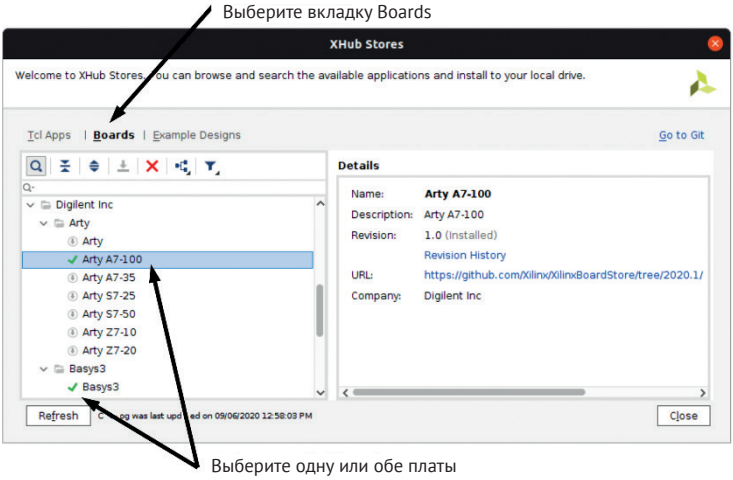


Рис. 1.15. Добавление плат Digilent

5. Выберите открытый проект и перейдите к `CH1/build/logic_ex/logic_ex.xpr` для платы Nexys A7 или к `CH1/build/logic_ex/logic_ex_basys3.xpr` для платы Basys 3, как показано на следующем скриншоте:

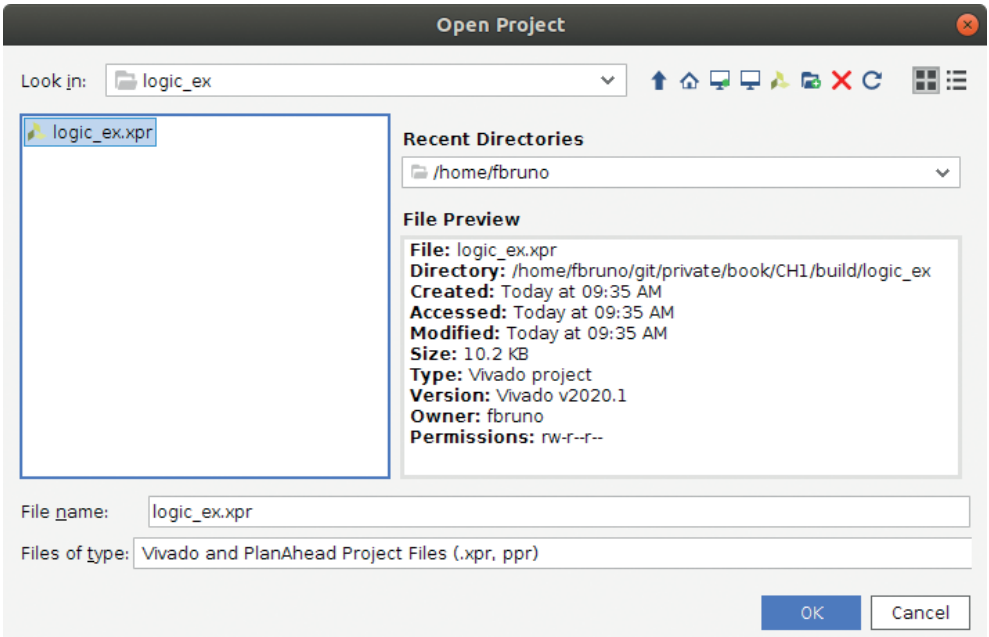


Рис. 1.16. Откройте окно проекта

После открытия отобразится следующее окно:

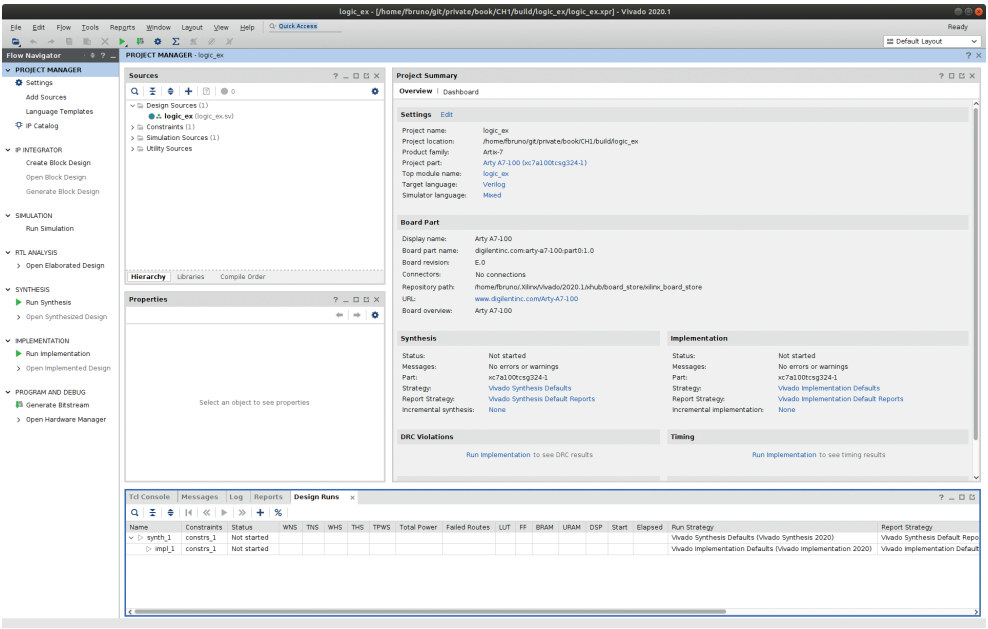


Рис. 1.17. Главный экран Vivado для проекта logic_ex

Окно проекта Vivado предоставляет легкий доступ к процессу проектирования и всю информацию, относящуюся к проекту. С левой стороны располагается **Flow Navigator**. Здесь представлены все шаги, которые нужно будет выполнить в процессе тестирования и создания файла прошивки для программирования FPGA. В настоящее время выделен пункт **PROJECT MANAGER**, **обеспечивающий** легкий доступ к исходным кодам и окну Project Summary, которое должно быть пустым, так как проект загружен в первый раз. При последующих загрузках проекта в нем будет отображаться информация из предыдущего запуска.

> **Важное замечание**
>
> Все проекты в этой книге поставляются в комплекте с предварительно настроенными файлами проекта. Инструкции по настройке первого проекта в режиме консоли или режиме графического интерфейса приведены в приложении. Это поможет вам в дальнейшем настраивать собственные проекты.

Изучим исходные коды проекта.

Имеется файл проекта `logic_ex.sv`. Также есть файл ограничений *.xdc и файл `testbench (tb.sv)`, создающий экземпляр `logic_ex.sv` в разделе исходных файлов для симуляции. Можно дважды щелкнуть по любому из этих файлов и просмотреть их в текстовом редакторе, встроенном в Vivado. В настоящее время проект настроен так, чтобы ссылаться на файлы в их текущем местоположении в структуре каталогов, поэтому файл можно редактировать в любом предпочитаемом редакторе.

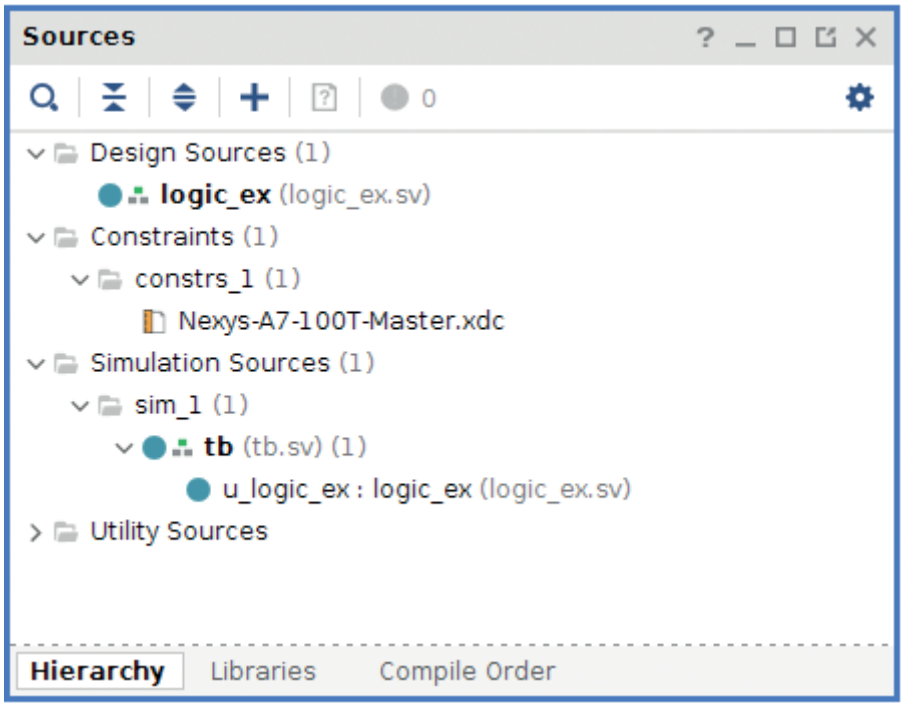


Рис. 1.18. Источники проекта

Посмотрев на **Project Summary**, можно увидеть, что проект в настоящее время настроен для работы с платой Nexys-A7-100T A7-100.

Запуск симуляции

Запустите симулятор Vivado, чтобы проверить корректность проекта.

Для этого нажмите **Run Simulation | Run Behavioral Simulation** в разделе **PROJECT MANAGER.** Можно увидеть, что доступны и некоторые другие опции, которые выделены серым цветом. Эти опции позволяют осуществлять запуск после синтеза или после имплементации, с временными характеристиками или без них. Поведенческое моделирование (иногда его еще не совсем корректно называют симуляцией, делая кальку с английского языка) выполняется относительно быстро и точно отображает функциональность проекта, если код написан правильно. Не рекомендуется запускать моделирование после синтеза или имплементации, если только не происходит отладка сбоев платы и нет нужды в точном тестировании реализованной версии проекта, поскольку в таком случае моделирование значительно замедлится.

Запуск поведенческого моделирования является первым шагом в общей последовательности процесса разработки проекта. Окно моделирования занимает весь главный экран Vivado.

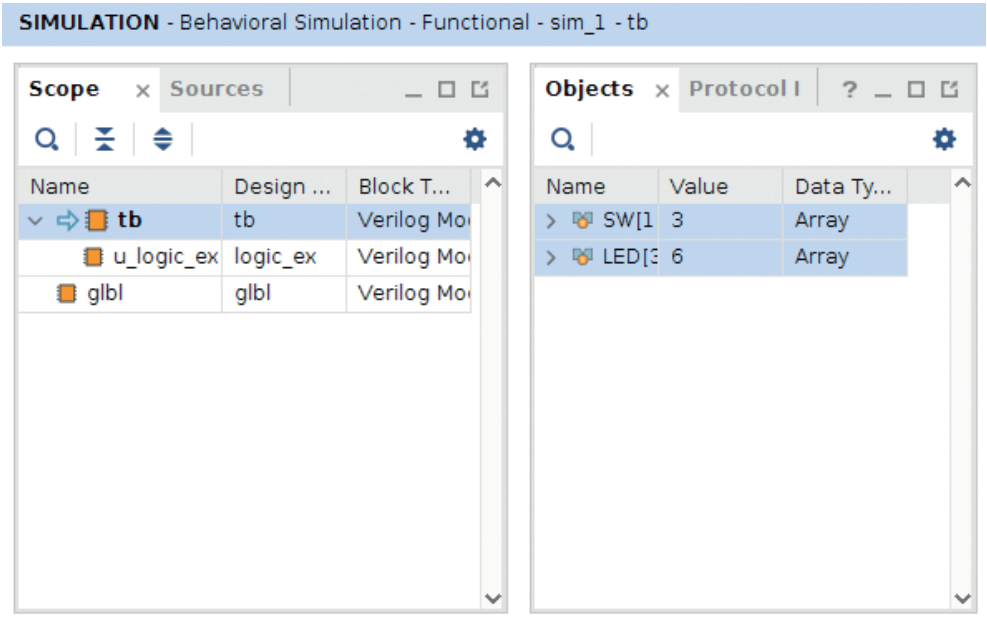


Рис. 1.19. Настройка моделирования проекта

Экран **Scope** предоставляет доступ к объектам внутри каждого модуля. В данном случае внутри `testbench` (`tb`) отображаются два сигнала `SW[1:0]` и `LED[3:0]`. Они добавлены в вейвформы (временны́е диаграммы).

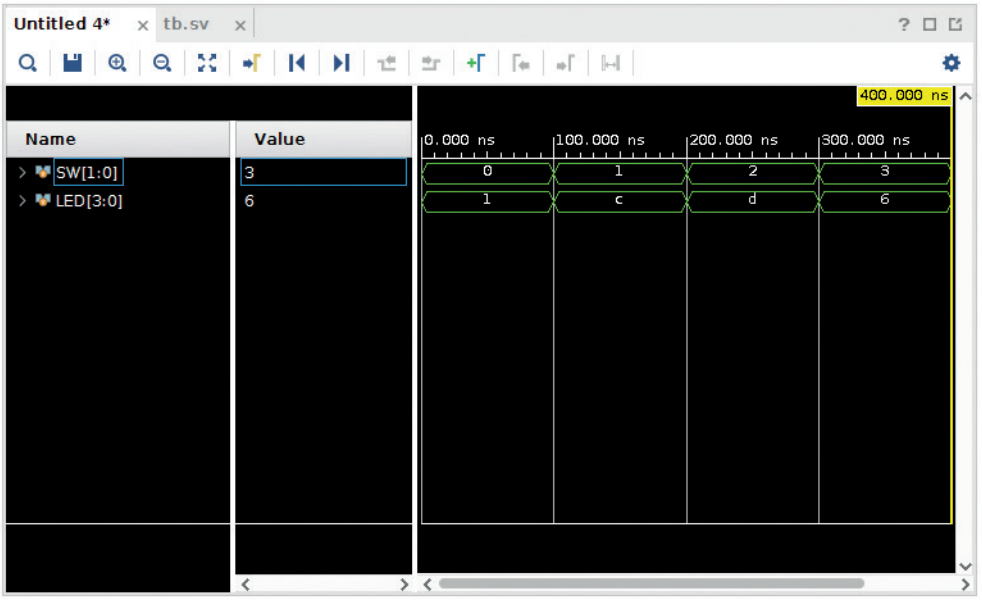


Рис. 1.20. Вейвформы

Вейвформы позволяют проанализировать сигналы в проекте и их поведение в процессе моделирования. Это наиболее часто используемая функция моделирующего САПРа при отладке проекта. На рисунке сигнал SW инкрементируется с помощью цикла `for` в `testbench`. Соответственно, происходит изменение значения LED. Текущее отображение представлено в шестнадцатеричном формате, но можно изменить его на двоичное или, нажав на символ > справа от сигнала, отобразить отдельные биты сигнала. Также следует обратить внимание на то, что каждое изменение сигналов соответствует временному сдвигу на 100 нс. Это связано с командой установки задержки модельного времени `#100`, которая используется для движения по шкале времени, и с настройкой временно́го масштаба (timescale).

Последнее окно является самым важным для `testbench` с самопроверкой:

```
Tcl Console   Messages   Log

Time resolution is 1 ps
source tb.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a w
#   }
# }
# run 1000ns
Timescale of (tb) is 1ns/100ps.
Setting switches to 00
Setting switches to 01
Setting switches to 10
Setting switches to 11
PASS: logic_ex test PASSED!
$stop called at time : 400 ns : File "/home/fbruno/git/private/book/CH1/tb/tb.sv" Line 20
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:06 ; elapsed = 00:00:06 . Memory (MB): peak = 8121.176 ; g
current_wave_config {}
WARNING: [Wavedata 42-16] Error Unable to get wave configuration ''.
Untitled 4
add_wave {{/tb/SW}} {{/tb/LED}}

Type a Tcl command here
```

Рис. 1.21. Консоль TCL

Консоль `TCL`[1] отображает все сообщения из функций `$display` или конструкций `assertion`. В данном случае отображается вывод функции `$printtimescale(tb)` как `1 ns/ 100 ps`. Также выводятся значения, на которые установлены переключатели, и можно видеть те же значения во временных диаграммах. В конце есть сообщение **`PASS: logic_ex test PASSED!`**, что является результатом тестирования. Поэкспериментируйте с `testbench`. Меняйте операторы или переставляйте их, чтобы убедиться, что тест провалится, если это сделать. Такие упражнения дадут вам уверенность в том, что `testbench` работает правильно.

[1] TCL (Tool Command Language) – встроенный интерпретируемый язык программирования. В консоль tcl выводятся служебные сообщения, сопровождающие процесс выполнения действий в Vivado. – *Прим. ред.*

Цель верификации состоит не в том, чтобы убедиться, что проект работает, а в том, чтобы попытаться сделать так, чтобы выявить ошибки в его работе. Здесь простой случай, поэтому на самом деле сейчас это невозможно, но следует убедиться, что протестированы все непредвиденные ситуации, чтобы удостовериться, что проект работает корректно.

> **Подсказка**
>
> Рекомендуется выработать для себя и применять везде набор правил о том, как обозначать прохождение и провал тестов. Данный тест прост. Но гораздо более обширный набор тестов для реального проекта может содержать случайные импульсы и множество целевых тестов. Принятие такого соглашения, как отображение слов PASS и FAIL, позволяет легко обрабатывать результаты тестов.

Реализация

Теперь, когда есть уверенность в том, что проект работает так, как задумано, пришло время собрать его и проверить на плате.

Сначала проанализируйте файл `.xdc`. Кликните на **Project Manager** в **Flow Navigator**, затем разверните закладку ограничений (constraints) и дважды кликните на файле `xdc`.

Для Nexys-A7-100T необходимо раскомментировать следующие строки, чтобы установить настройки напряжений на пинах платы:

```
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

Здесь `set_property` – это команда языка `TCL`, которая устанавливает заданное свойство проекта, используемое Vivado. В предыдущей команде устанавливаются `CFGBVS` и `CONFIG_VOLTAGE` в значения, необходимые для Artix-7 FPGA.

Следующий блок кода задает расположение переключателей и светодиодов (для удобства размещены подряд):

```
set_property -dict { PACKAGE_PIN J15 IOSTANDARD LVCMOS33 }
[get_ports { SW[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 }
[get_ports { SW[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 }
[get_ports { LED[0] }]; #IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 }
[get_ports { LED[1] }]; #IO_L24P_T3_RS1_15 Sch=led[1]
set_property -dict { PACKAGE_PIN J13   IOSTANDARD LVCMOS33 }
[get_ports { LED[2] }]; #IO_L17N_T2_A25_15 Sch=led[2]
set_property -dict { PACKAGE_PIN N14   IOSTANDARD LVCMOS33 }
[get_ports { LED[3] }]; #IO_L8P_T1_D11_14 Sch=led[3]
```

Команды `set_property` создают `TCL`-словарь `(-dict)`, содержащий `PACKAGE_PIN` и `IOSTANDARD` для каждого порта в проекте. `TCL`-команда `get_port` используется для того, чтобы установить порт в проекте. Символ # – это комментарий в `TCL`.

Расположение выводов и стандарты вводов/выводов определяются производителем платы. Здесь используется стандарт вводов/выводов 3.3 В.

Шаги для генерации загрузочного файла (bitstream) следующие.

1. **Синтез:** переводится код SystemVerilog в промежуточный логический формат для оптимизации.
2. **Имплементация: проект размещается на ресурсах FPGA чипа**, оптимизируются результаты размещения и производится маршрутизация.
3. **Создание битового потока (загрузочного файла для программирования FPGA)**: генерируется файл для загрузки на плату.

Эти действия можно выполнять по отдельности. Это делают, если нужно просмотреть промежуточные результаты, чтобы увидеть, какие ресурсы платы потребляются или какие получаются временные характеристики проекта, или если вы проектируете под специализированную плату и вам нужно выполнить планирование выводов. В данном случае можно нажать непосредственно на **Generate Bitstream** и позволить программе выполнить все шаги автоматически. Разрешите системе использовать настройки по умолчанию. После завершения откройте результаты синтеза.

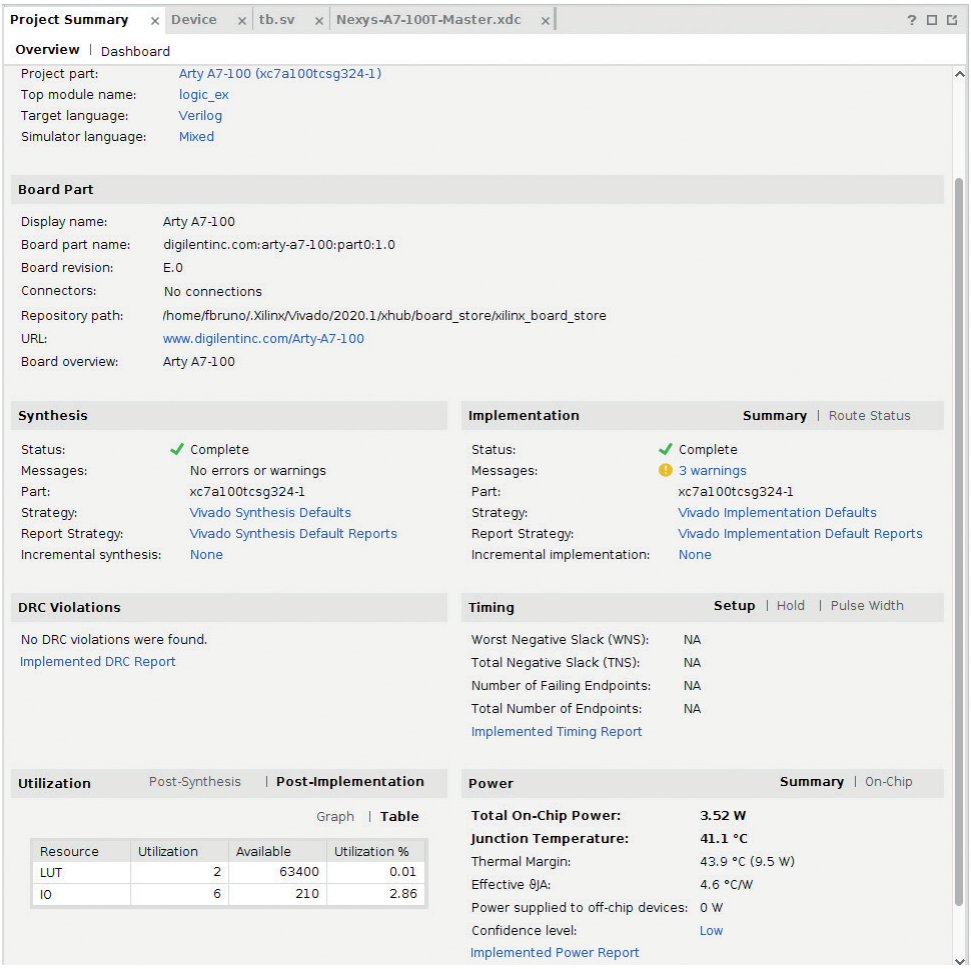


Рис. 1.22. Результаты проекта

В окне результатов синтеза проекта можно увидеть результаты трассировки. Задействовано 2 LUT и 6 устройств ввода/вывода (SW + LED). Синхронизация отсутствует, поскольку этот проект является полностью комбинационным, иначе в отчете было бы больше информации о временны́х параметрах.

Если перейти на вкладку **Device**, то можно получить представление о том, как используются ресурсы FPGA-чипа.

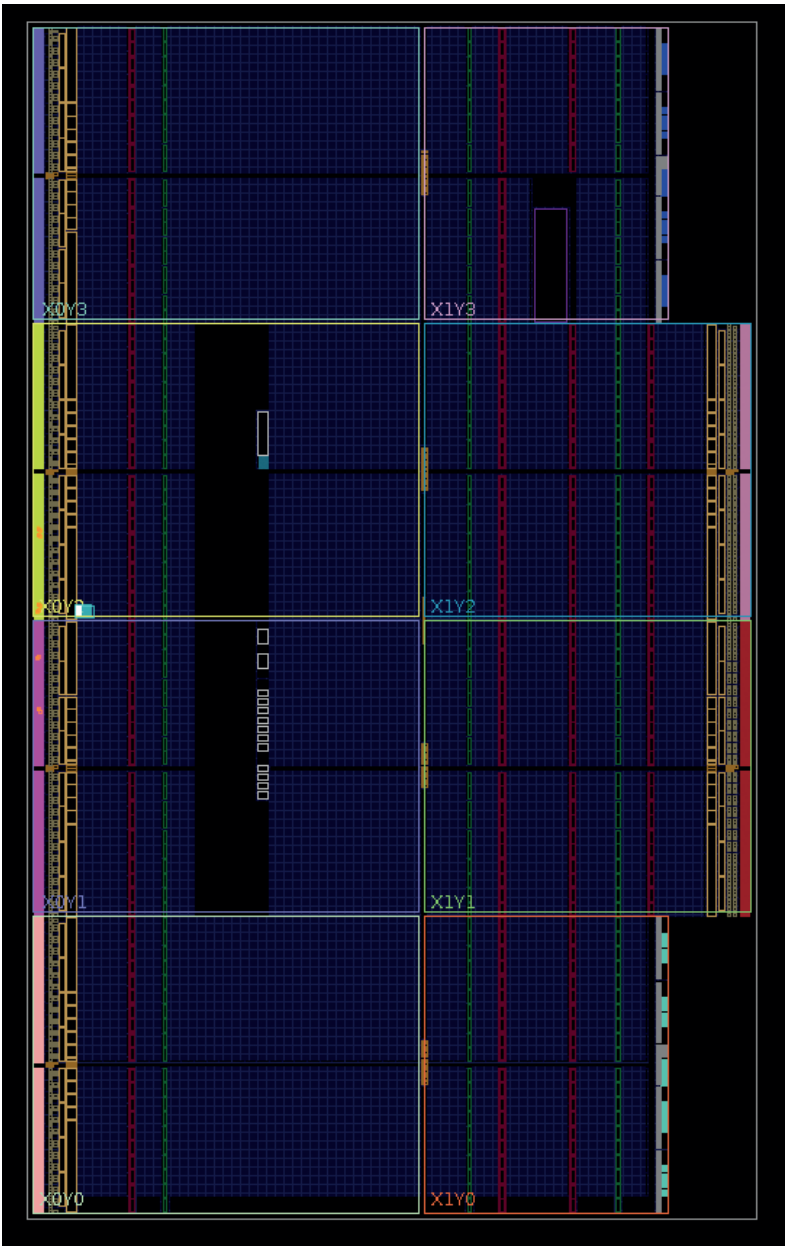


Рис. 1.23. Визуализация ресурсов FPGA-чипа

На рисунке ресурсов FPGA-чипа можно увидеть маленькую белую точку в середине с левой стороны. Она обозначает место размещения используемых LUT.

Программирование платы

Теперь пришло время увидеть плату в действии. Для этого выполните следующие действия.

1. Убедитесь, что отладочная плата подключена к сети и включена.
2. Теперь кликните на **Open hardware manager**, расположенную последней среди опций в **Flow Navigator**. В главном окне откроется менеджер оборудования.
3. Кликните на **Open target | Autoconnect**.
4. Выберите программируемое устройство. Загрузочный файл должен быть выбран автоматически. На плате на несколько секунд погаснут все световые сигналы, а затем, если два левых переключателя опущены вниз, будет следующий результат:

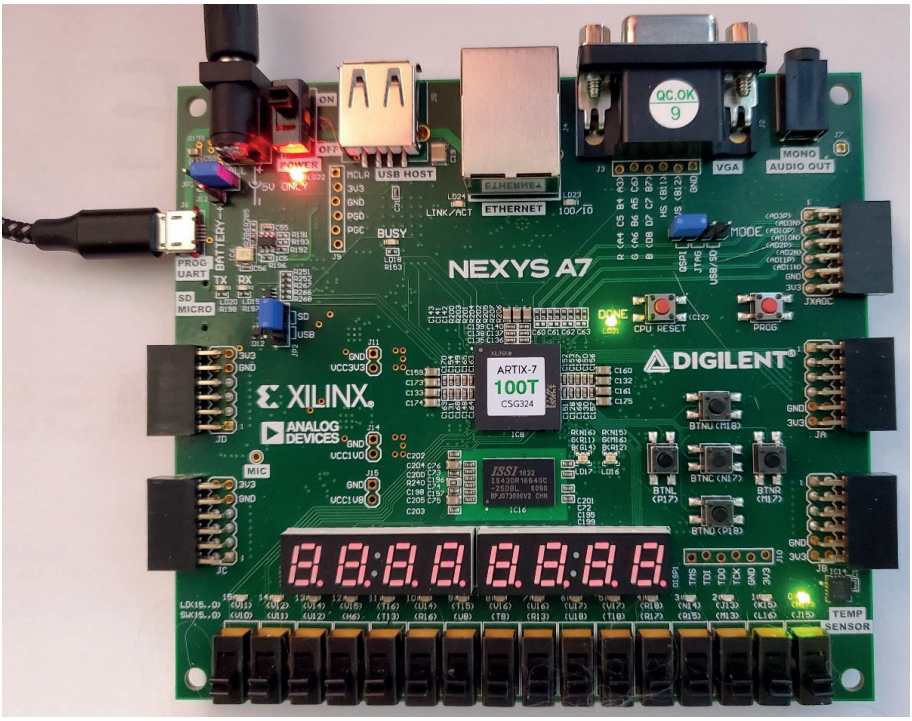


Рис. 1.24. Окончательный вид платы

5. Установите переключатели в состояния 00, 01, 10, 11, где 0 – положение вниз, а 1 – положение вверх. Убедитесь, что свечение светодиодов соответствует результатам моделирования. Соответствует ли это тому, как должна работать плата, по вашему мнению? Есть ли иногда мерцание при переключении переключателей? На последний вопрос ответ будет дан в главе 3 «Подсчет нажатий на кнопку».

Поздравляем! Ваш первый проект на плате FPGA выполнен. Вы сделали первый шаг на этом пути и перенастроили аппаратное обеспечение FPGA для выполнения некоторых простых задач. По мере изучения книги задачи будут становиться все сложнее и интереснее, и вскоре вы сможете на их основе создавать свои собственные проекты.

Выводы

Эта глава знакомит с основами ASIC и FPGA, с тем, как они создаются и в каких ситуациях какие из них имеют больший смысл с точки зрения денежных затрат. Вы научились использовать плату FPGA и программировать ее. Это является введением к остальной части книги, где будет использоваться отладочная плата и ваши навыки программирования в различных задачах и проектах. В конечном итоге эти навыки станут основой для разработки ваших собственных проектов как для работы, так и для развлечения.

В следующей главе будет использоваться созданный тестовый проект, поскольку она посвящена более глубокому рассмотрению проектирования комбинационных схем.

Вопросы

1. Когда можно использовать FPGA?
   a) Создается прототип приложения, которое в конечном итоге может стать ASIC.
   b) Планируются очень небольшие объемы производства.
   c) Необходимо оборудование, на котором можно будет в будущем легко изменить алгоритм работы.
   d) Все вышеперечисленное.
2. Когда следует использовать ASIC?
   a) Необходимо разработать специализированное приложение, которое должно быть создано в небольшом количестве, и бюджет ограничен.
   b) Требуется разработать калькулятор, который будет выпускаться серийно и для которого требуется специализированный процессор.
   c) Нужно разработать что-то чрезвычайно маломощное, и стоимость не имеет значения.
   d) Создается спутник для получения изображений земной поверхности, и необходимо иметь возможность обновлять алгоритмы в течение всего срока службы спутника.
   e) a и b.
3. В этой главе был рассмотрен полный сумматор. Полусумматор – это схема, которая может складывать два входа без учета бита переноса. Заполните таблицу истинности для выходов суммы и переноса полусумматора.

A	B	Сумма	Перенос
0	0		
0	1		
1	0		
1	1		

4. Измените код и testbench для проверки следующих логических элементов: NAND (НЕ И), NOR (НЕ ИЛИ) и XNOR (НЕ Исключающее ИЛИ). Подсказка: вы можете поменять значение унарного оператора на противоположное, добавив перед ним оператор ~, другими словами, NAND (НЕ И) то же самое, что ~&. Загрузите получившиеся схемы на плату и проанализируйте их работу.

Задание повышенной сложности

1. Откройте CH1/build/challenge.xpr.
2. Измените строки в challenge.sv, чтобы реализовать полный сумматор:

```
assign LED[0]  = ; // Напишите код для суммы
assign LED[1]  = ; // Напишите код для переноса
```

3. Измените tb_challenge.sv, чтобы провести его моделирование:

```
if () begin // Изменить для проверки
```

Подсказка: можно обратиться к следующим главам книги, чтобы посмотреть дополнительную информацию, или произвести быстрый поиск в сети Интернет.

Дополнительное чтение

Для получения дополнительной информации обратитесь к следующим ссылкам.

- Конфигурируемый логический блок FPGA 7-й серии: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.
- Ресурсы синхронизации FPGA 7-й серии: https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf.
- 7 Series DSP48E1 Slice: https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf.
- Справочное руководство Nexys A7: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual.
- Справочное руководство Basys 3: https://reference.digilentinc.com/reference/programmable-logic/basys-3/reference-manual.

Раздел 2

Введение в проектирование, моделирование и синтез на Verilog RTL

В этом разделе вы научитесь программировать на языке SystemVerilog на среднем уровне. Вы разработаете, промоделируете и при желании реализуете на реальной плате от спецификации до работающего устройства целый ряд проектов.

В эту часть книги входят:

- *Глава 2 «Комбинационная логика»;*
- *Глава 3 «Подсчет нажатий на кнопку»;*
- *Глава 4 «Разработка калькулятора»;*
- *Глава 5 «Ресурсы FPGA, и как их использовать»;*
- *Глава 6 «Математика, параллелизм и конвейеризация».*

Глава 2

Комбинационная логика

Проекты обычно включают комбинационную и последовательностную логику. Комбинационная логика состоит просто из логических элементов, как это было показано в главе 1 «Введение в FPGA и Xilinx Vivado». Последовательностная схема определяет свое состояние обычно в зависимости от фронта тактового импульса, но она также может быть и основана на уровнях тактового сигнала, чему будет посвящен раздел о том, чего не следует делать при проектировании последовательностных схем.

В этой главе рассмотрена разработка с нуля полного модуля SystemVerilog, способного выполнять некоторые базовые операции реального мира, которые вы можете однажды использовать в своих проектах.

В этой главе рассмотрены следующие основные темы:

- создание модулей SystemVerilog;
- представление типов данных;
- упаковка кода с помощью функций;
- проект – создание комбинационной логики.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH2.

Создание модулей SystemVerilog

В основе каждого проекта лежат составляющие его модули. От `testbench`, используемого для проверки проекта, до любых компонентов реализации – все составляющие объявляются где-то в виде модуля. Для тестового проекта, который рассматривается в этой главе, будет создан набор модулей, представляющих функции, доступ к которым можно получить с помощью кнопок и переключателей на отладочной плате. Эти переключатели будут использоваться для установки значений, а пять кнопок – для выполнения операций.

Рассмотрим основные части объявления модуля:

```
module project_2
#(parameter SELECTOR,
  Parameter BITS = 16)
(input wire [BITS-1:0]          SW,
 input wire                     BTNC,
 input wire                     BTNU,
 input wire                     BTNL,
 input wire                     BTNR,
 input wire                     BTND,
 output logic signed [BITS-1:0]);
```

Создан модуль под названием `project_2`, который является верхним уровнем проекта. Первая секция внутри скобок в записи `#()` – это список параметров, позволяющий задать параметры, которые можно будет использовать в перечне портов или модулей. Параметры можно определить в любом месте модуля, и они также могут быть переопределены во время синтеза проекта. Но параметры должны быть определены до их использования.

Создание многократно используемого кода с помощью параметров

Параметры могут быть использованы для переопределения информации в реализации модуля. Эта информация может использоваться в модуле для управления размерностями данных, как в случае с параметром `BITS`, который имеет значение по умолчанию 16, если его не переопределить. Параметры также могут управлять реализацией логики или модулей, как будет показано при изучении оператора `case`. Также можно создать параметр `SELECTOR`, который не имеет значения по умолчанию. Это хороший способ удостовериться, что что-то определено в реализации, поскольку нет значения по умолчанию. Так как, если параметр не переопределен, это приведет к ошибке.

Параметрами могут быть целые числа (integers), строки (strings) или даже типы (types):

```
#(parameter type SW_T = logic unsigned [15:0], ...
 (input  SW_T       SW, ...
```

Здесь объявлен тип `SW_T`, который по умолчанию является `logic unsighed [15:0]`, и создан порт `SW`, использующий этот тип. При реализации модуля можно передать новый тип, тем самым переопределив значение по умолчанию и обеспечив более широкое использование проекта.

> **Подсказка**
>
> Хорошей практикой является сохранение параметров, предназначенных для переопределения, в списке параметров, а в самом модуле использование определения localparams для параметров, которые не могут быть переопределены. Параметры являются отличным способом выразить замысел проекта. Если вернуться к проекту через длительное время, *магические числа*, такие как 3,14, будут иметь гораздо меньше смысла, чем параметр Pi.

Рассмотрим типы данных, которые используются в SystemVerilog для передачи данных.

Знакомство с типами данных

Все языки компьютерного программирования нуждаются в переменных. Это место в памяти или регистры, хранящие значения, к которым может обращаться выполняемая программа. **Языки проектирования аппаратуры (Hardware Design Languages, HDL)** немного отличаются, так как они предназначены для того, чтобы описывать аппаратуру. Здесь существуют эквиваленты переменных с точки зрения запоминающей/последовательностной логики, которые будут обсуждаться в следующей главе, но также нужны проводники (wires) для перемещения данных по аппаратуре, которая разрабатывается, используя ресурсы маршрутизации FPGA, даже если эти данные нигде не хранятся:

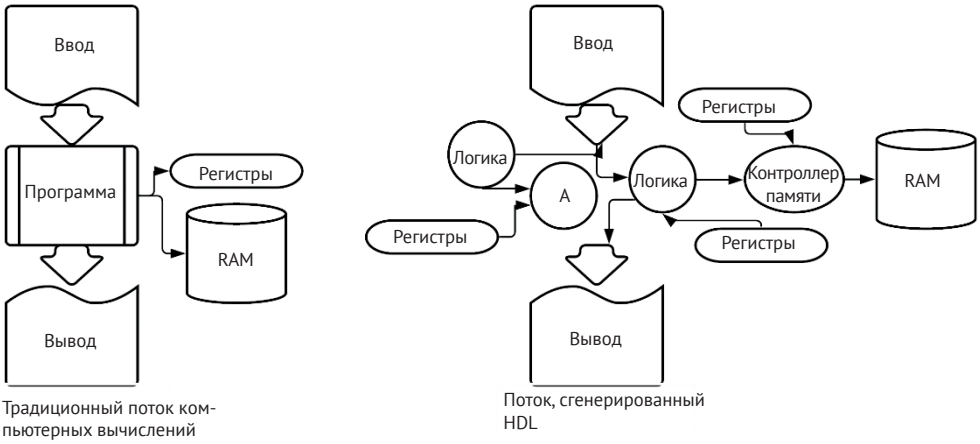


Рис. 2.1. Выполнение программы в сравнении с реализацией на HDL

В традиционном потоке имеется компьютер с процессором и памятью. Программа выполняется линейно, хотя в современных компьютерах последние годы уровень параллелизма постоянно возрастает. Когда ведется разработка на SystemVerilog, то используются типы данных для создания аппаратуры, которая будет хранить или физически перемещать данные от таблиц поиска (**LUT**) к другим LUT. Если требуется использовать внешнюю память, о которой будет рассказано в главе 8 «Много данных? MIG и DDR2», то необходимо реализовать аппаратное обеспечение для связи с памятью.

Представление встроенных типов данных

SystemVerilog имеет множество встроенных типов данных, но наиболее интересными для проектирования являются логический и битовый типы:

- **`logic`** (логический): этот тип данных использовался в предыдущей главе. Логический тип может представлять `0`, `1`, `x` (неопределенное состояние) или `z` (третье состояние, высокоимпедансное);

> **Важное замечание**
>
> Если вы когда-либо использовали Verilog, вы знаете о типе `reg`. Этот тип был непонятным для новых разработчиков на HDL, поскольку они видели `reg` и думали, что это сокращение от register. На самом деле тип `reg` – это любой сигнал, исходящий из блока `always`, хотя блоки `always` могут использоваться для генерации комбинационной логики, как это будет показано далее. Хотя `reg` все еще можно использовать для обратной совместимости, лучше использовать `logic` или `bit`, которые можно использовать как в операторах присваивания `assign`, так и в блоках `always`. Тип `logic` также позволяет передавать неопределенные значения `x` по схеме. Это может быть полезно для отладки условий запуска.

- **`bit`** (битовый): битовый тип использует меньше памяти для хранения данных, чем логический, и может хранить только 0 или 1. Это позволяет использовать меньше памяти и потенциально ускорить моделирование за счет неопределенных значений.

Существует также четыре других, менее используемых типа с двумя состояниями:

- `byte`: 8 бит;
- `shortint`: 16 бит;
- `int`: 32 бита;
- `longint`: 64 бита.

> **Важное замечание**
>
> Различия между bit и logic связаны исключительно с тем, как они ведут себя при моделировании. Оба типа будут генерировать одинаковые логические элементы и элементы памяти в аппаратуре. Все остальные типы различаются только размером или представлением знака по умолчанию.

Это все основные типы, используемые в SystemVerilog. Но что делать, если нужно работать с данными разных размеров или с большим количеством данных, чем могут обработать эти типы?

Создание массивов

Причина, по которой `byte`, `shortint`, `int` и `longint` используются не так часто, заключается в том, что обычно размер сигналов определяют по мере необходимости, например:

```
bit [7:0] my_byte; // определено 8-битное значение
```

Здесь `my_byte` определяется как упакованное 8-битное значение. Можно также создать неупакованную версию:

```
bit my_byte[8]; // определено 8-битное значение
```

Упакованные версии имеют преимущество при формировании массивов, а неупакованные имеют преимущество по скорости обращения к памяти при моделировании, о чем будет рассказано в главе 5 «Ресурсы FPGA, и как их использовать».

Массивы также могут иметь несколько измерений:

```
bit [2:0][7:0]   my_byte[1024][768]; // определен массив 8-битных значений
//   3    4              1     2        Упорядочение массива
```

Упорядочение массива определено в предыдущем коде. Ниже перечислены допустимые способы доступа к массиву:

```
my_array[0][0]                Возвращает значение [2:0][7:0].
my_array[1023][767][2]           Возвращает 8-битное значение.
```

Для определения массива можно использовать диапазон, например `[7:0]`, или количество элементов, например `[1024]`.

Обращение к элементам массива

SystemVerilog предоставляет системные функции для доступа к информации о массивах. Это позволяет создавать многократно используемый код.

> **Важное замечание**
>
> Параметр dimension (размерность) является необязательным и по умолчанию равен 1.

Это становится еще более важным, когда нужно использовать параметры данного типа данных.

`$dimension(my_array)`	4.
`$left(my_array, [dimension])`	`[1]=1023, [2]=767, [3]=2, [4]=7.`
`$right(my_array, [dimension])`	`0` для всех размерностей
`$high(my_array, [dimension])`	Наибольшее значение в диапазоне размерности
`$low(my_array, [dimension])`	Наименьшее значение в диапазоне размерности
`$size(my_array, [dimension])`	`=$high(my_array, [dimension]) – $low (my_array, [dimension]) + 1`
`$increment(my_array, [dimension])`	Возвращает 1, если `$left >= $right`, в противном случае -1. Удобно для циклов for
`$bits()`	Возвращает количество битов, используемых конкретной переменной или выражением. Удобно для передачи информации о размере экземплярам модулей
`$clog2()`	Возвращает размер массива, который может содержать заданное количество элементов, а не значение, которое было передано. Например, `$clog2(4)` возвращает 2, которое может хранить четыре значения от `0` до `3`

Эти системные функции позволяют определить параметры массива.

Присвоение значений элементам массива

Когда требуется присвоить значение сигналу, определенному как массив, нужно правильно определить его размер, чтобы избежать ошибок. Если не указать размер, то, как сказано в **справочном руководстве по языку Verilog (Language Reference Manual, LRM)**, по умолчанию он будет равен 32 битам.

Существует три способа присвоения без указания знака: `'1` устанавливает все биты равными 1, `'0` устанавливает все биты равными 0 и `'z` устанавливает все биты равными `z`. Если это одномерный массив, то можно использовать `n'b` для указания двоичного значения в `n` бит, `n'd` для указания десятичного значения в `n` бит или `n'h` для указания шестнадцатеричного значения в `n` бит:

```
logic [63:0] data;
assign data = '1; // эквивалентно data = 64'hFFFFFFFFFFFFFFFF;
assign data = '0; // эквивалентно data = 64'd0;
assign data = 'z; // эквивалентно data = 64'hzzzzzzzzzzzzzzzz;
assign data = 0; // data[31:0] = 0, биты data[63:32] не проинициализированы
```

Важно помнить, что `n` в этих случаях – это количество битов, а не количество цифр.

Работа с цепями с тремя состояниями

Есть еще один тип данных, который заслуживает упоминания, хотя он и будет в этой книге использоваться только в более сложных проектах. Это **`wire`** (провод). Тип `wire` может принимать 120 различных возможных значений, т. е. четыре основных значения – 0, 1, x, z, и различные варианты значения напряжения питания. У типа `wire` есть так называемая функция разрешения. Сигналы типа `wire` – это единственные сигналы, которые могут быть подключены к нескольким цепям управления. Как использовать этот тип данных, будет показано на примере работы с протоколом **последовательного периферийного интерфейса (Serial Peripheral Interface, SPI)** и при реализации доступа к памяти DDR2 на плате Nexys A7.

FPGA, как правило, не имеют внутренних возможностей для работы в трех состояниях. В примере на рис. 2.2 показаны два устройства с подключенными буферами устройств ввода/вывода (I/O) в трех состояниях.

```
logic [1:0] in;
logic [1:0] out;
logic [1:0] enable;
tri1 R_in;
assign R_in = (enable[0]) ? out[0] : ‘z;
assign R_in = (enable[1]) ? out[1] : ‘z;
assign in[0] = R_in;
assign in[1] = R_in;
```

Код выше демонстрирует, как создаются два буфера с тремя состояниями. `tri1` – это конструкция, используемая только в целях моделирования, в которой сигнал объявляется как имеющий три состояния со слабой подтяжкой[1] к 1.

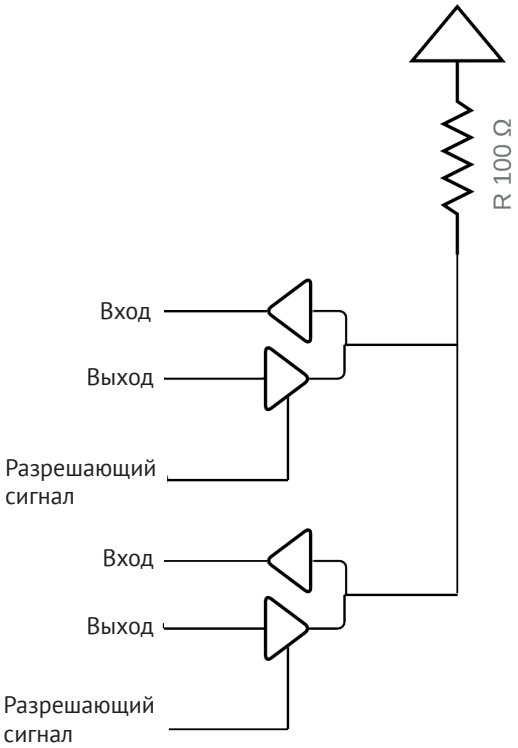


Рис. 2.2. Пример цепи с тремя состояниями

Работа со знаковыми и беззнаковыми числами

В Verilog был только один знаковый тип данных – `integer`. SystemVerilog позволяет однозначно определять беззнаковые `(unsigned)` и знаковые `(signed)` форматы для любого встроенного типа:

```
bit signed [31:0] signed_vect; // создание 32-битного знакового значения
bit unsigned [31:0] unsigned_vect; // создание 32-битного беззнакового значения
```

При выполнении знаковых арифметических операций важно убедиться в корректности выбранной размерности аргументов. Также в компьютерных вычислениях со знаковыми числами для получения правильного результата необходимо убедиться, что все задействованные операнды являются знаковыми.

[1] Тип `wire` имеет несколько вариантов установки сигнала «по умолчанию». «Слабая подтяжка к 1» (*weak pullup to 1,* `weak1`) эквивалентна выходу с «открытым коллектором (стоком)» и подтягивающим резистором. Такие выходы можно объединять между собой («монтажное ИЛИ») и все вместе «перетянуть» к нулю сигналом `strong0` («сильный 0») с любого из выводов. При подаче третьего состояния (`z`) вся шина останется в единичном состоянии. При подаче «слабого 0» (`weak0`, обеспечивается заземленным резистором) на шину в состоянии `weak1` состояние окажется неопределенным. – *Прим. ред.*

Важное замечание

Цифровая логика, такая как в компьютерных процессорах или в реализации на FPGA, использует дополнительный код для представления знаковых чисел. Это означает, что, чтобы сделать число отрицательным, нужно просто инвертировать его и прибавить 1. Например, чтобы получить -1 в дополнительном коде, предполагая, что для представления числа имеется 4 бита, следует взять 4'b0001, инвертировать его, получив 4'b1110, и прибавить 1, в результате чего будет получено значение 4'b1111. Третий бит используется для хранения знака, поэтому если он равен 0, то число положительное, а если 1 – число отрицательное. Это также означает, что граничные значения, которые можно представить с помощью 4 бит, – это 4'b0111 (или +7) и 4'b1000 (или -8).

Добавление битов к сигналу с помощью операции конкатенации[1]

В SystemVerilog имеется широко используемая операция конкатенации, представляющая собой объединение (склеивание) вместе битов или сигналов для создания больших векторов или их копирования. При преобразовании беззнакового целого числа в знаковое обычно требуется использовать оператор конкатенации `{}` для добавления `1'b0` в бит знака, чтобы результирующий сигнал оставался беззнаковым. Оператор конкатенации можно использовать для объединения нескольких сигналов вместе, например `{1'b0, unsigned_vect}`. Он также может быть использован для размножения сигналов. Например, `{2{unsigned_vect}}` будет эквивалентен `{unsigned_vect, unsigned_vect}`.

Преобразование знаковых и беззнаковых чисел

Беззнаковое число можно преобразовать в знаковое с помощью ключевого слова `signed'` и выполнить обратную операцию с помощью ключевого слова `unsigned'`:

```
logic unsigned [15:0] unsigned_vect = 16'hFFFF;
logic unsigned [15:0] final_vect;
logic signed [16:0] signed_vect;
logic signed [15:0] signed_vect_small;
assign signed_vect = signed'({1'b0, unsigned_vect}); // +65535
assign signed_vect_small = signed'(unsigned_vect); // -1
assign unsigned_vect = unsigned'(signed_vect);
assign final_vect = unsigned'(signed_vect_small); // 65535
```

Беззнаковое 16-битное число может принимать значения от 0 до 65535. 16-битное знаковое число может принимать значения от −32768 до 32767, по-

[1] Конкатенация (букв. объединение) — операция соединения объектов последовательной структуры (список, последовательность бит, массив и пр.). В общем случае результатом конкатенации двух объектов A и B является объект C = AB, полученный поочередным добавлением всех элементов объекта B, начиная с первого, в конец объекта A. — *Прим. ред.*

этому если присвоить число, большее чем 32767, то в знаковом числе того же размера будет установлен бит знака, в результате чего число станет отрицательным.

Это эквивалентно системным функциям Verilog, т. е. функциям `$signed()` и `$unsigned()`. Но предпочтительней использовать операторы преобразования.

> **Важное замечание**
>
> При приведении знаковых значений к беззнаковым или беззнаковых к знаковым следует обращать внимание на размерность числа. Например, чтобы сохранить положительный характер беззнакового числа, обычно используется оператор конкатенации {}, как, например, `signed({1'b0, unsigned_vect})`. Это означает, что результирующий сигнал будет иметь размерность на 1 бит больше. При переходе от знакового значения к беззнаковому необходимо следить за тем, чтобы число было положительным, иначе результат операции присваивания будет некорректным. Пример таких некорректных присваиваний можно увидеть в предыдущем коде, где `signed_vect_small` становится равным –1, а не 65535, а `final_vect` становится 65535, хотя `signed_vect_small` равно –1.

Создание типов, определяемых пользователем

Можно создавать собственные типы с помощью `typedef`. Частым примером, используемым в SystemVerilog, является создание пользовательского типа для ускорения моделирования. Это можно сделать с помощью определения:

```
`ifdef FAST_SIM
   typedef bit bit_t
`else
   typedef logic bit_t
`endif
```

Если `FAST_SIM` определено, то в любой момент, когда используется тип `bit_t`, среда моделирования будет использовать тип `bit`, в противном случае – тип `logic`. Это может ускорить процесс моделирования.

> **Подсказка**
>
> Хорошей идеей является принятие соглашения об именовании при создании типов, в данном случае использование суффикса `_t`. Это помогает идентифицировать определяемые пользователем типы и избежать путаницы при их использовании в проекте.

Доступ к сигналам при использовании значений перечисляемых типов

Когда речь идет о читабельности, предпочтительнее использовать переменные с именами, которые имеют больше смысла и являются самодокументирующими.

Для этого можно использовать перечисляемые типы, например, таким образом:

```
enum bit   [1:0]   {RED, GREEN, BLUE} color;
```

В данном случае создается переменная `color`, состоящая из значений `RED`, `GREEN` и `BLUE`. Моделирующие САПР будут отображать эти значения в своих временных диаграммах. Более подробно перечисляемые типы обсуждаются в главе 3 «Подсчет нажатий на кнопку».

Упаковка кода с помощью функций

Часто приходится иметь дело с кодом, который нужно повторно использовать в одном модуле или который является общим для группы модулей. Такой код можно упаковать в функцию:

```
function [4:0] func_addr_decode(input [31:0] addr);
   func_addr_decode = '0;
   for (int i = 0; i < 32; i++) begin
      if (addr[i]) begin
         return(i);
      end
   end
endfunction
```

Здесь создана функция `func_addr_decode`, которая возвращает 5-битное значение. `function` принимает 32-битный входной сигнал, называемый адресом (address). Функции могут иметь несколько выходов, но в данной книге такая возможность не используется. Чтобы вернуть значение функции, можно присвоить результат имени функции или использовать оператор `return`.

Создание комбинационной логики

Два основных способа создания комбинационной логики – это операторы присваивания `assign` и блоки `always`. Операторы присваивания `assign` удобны при создании сугубо комбинационной логики с помощью лаконичного кода. Это не означает, что результирующая логика обязательно будет примитивной. Например, можно создать массивное вычисление умножения-сложения с помощью одной строки кода или большие комбинационные структуры, используя оператор присваивания и вызов функции:

```
assign mac = (a * b) + old_mac;
assign addr_decoder = func_addr_decode(current_address);
```

Блок `always` позволяет определить более сложную функциональность в одном процессе. Блоки `always` уже были в предыдущей главе. Использовался список чувствительности на определенные сигналы для задач тестирования. Подобные списки позволяют блоку `always` срабатывать только при изменении сигнала из списка. Вернемся к `testbench`, который был представлен в главе 1 «Введение в FPGA и Xilinx Vivado»:

```
always @(SW, LED) begin
```

В этом примере блок `always` будет срабатывать только при изменении SW или LED из одного состояния в другое.

> **Важное замечание**
>
> Списки чувствительности на сигналы не синтезируются и полезны только при тестировании. При описании синтезируемого кода в блоке `always` рекомендуется использовать `always_comb`.

Когда разрабатывается синтезируемый код с использованием блока `always`, рекомендуется использовать структуру `always_comb`. Этот тип кода является синтезируемым и рекомендуется для комбинационной логики. Причина в том, что `always_comb` выдаст предупреждение или ошибку, если была непреднамеренно создана защелка.

> **Важное замечание**
>
> Замечание о триггерах-защелках: это один из типов запоминающих элементов. Защелки чувствительны к уровню сигнала, что означает, что они *прозрачны,* когда тактовый сигнал имеет высокий уровень, а когда он переходит в низкий уровень, значение сохраняется. Защелки применяются в цифровых схемах, особенно в мире ASIC, но в FPGA их следует избегать любой ценой, поскольку они почти всегда приводят к проблемам синхронизации и случайным сбоям. Тем не менее в рамках учебного проекта, создаваемого в этой главе, будет показано, как работает защелка и к чему может привести ее использование.

Существует несколько различных операций, которые могут выполняться в блоке `always`. Поскольку генерируется комбинационная логика, то необходимо убедиться, что предусмотрены все возможные пути выполнения любой из команд, указанных внутри `always`. Это будет обсуждено позже.

Операторы присваивания

В SystemVerilog существует два основных типа присваиваний: блокирующие и неблокирующие. Поскольку разработка ведется на языке HDL, необходимо иметь возможность моделировать создаваемую аппаратуру. Все аппаратные средства, которые можно спроектировать в FPGA, будут работать параллельно.

Создание нескольких присваиваний с использованием оператора неблокирующего присваивания

Где бы ни были в коде описаны различные блоки `always`, в аппаратуре все они будут выполняться одновременно. Поскольку это фактически невозможно на обычном компьютере, выполняющем команды последовательно или (в лучшем случае) параллельно на нескольких потоках, нужен способ моделирования параллельного поведения аппаратуры. В симуляторах это достигается с помощью планировщика, который делит время симуляции на дельта-циклы. Таким образом, если запланировано выполнение нескольких заданий, они все равно будут выполняться последовательно друг за другом. Это делает обработку блокирующих и неблокирующих присваиваний критически важной.

Неблокирующее присваивание – такое присваивание, которое должно произойти внутри дельта-цикла, когда время симулятора идет вперед. Неблокирующие присваивания обсуждаются более подробно в главе 3 «Подсчет нажатий на кнопку».

Блокирующие присваивания

Блокирующие присваивания выполняются немедленно. За редким исключением, обычно только для `testbench`, все присваивания в блоке `always_comb` будут блокирующими.

В SystemVerilog существует несколько блокирующих присваиваний:

=	Присвоить то, что справа, тому, что слева
+=	Увеличить на значение справа и присвоить
–=	Уменьшить на значение справа и присвоить
*=	Умножить на значение справа и присвоить
/=	Разделить на значение справа и присвоить
%=	Разделить на значение справа и присвоить результат выполнения операции деления по модулю
&=	Применить логическое И со значением справа и присвоить
|=	Применить логическое ИЛИ со значением справа и присвоить
^=	Применить логическое Исключающее ИЛИ со значением справа и присвоить
<<=	Применить побитовый сдвиг влево и присвоить (эквивалентно умножению на 2^значение справа)
>>=	Применить побитовый сдвиг вправо и присвоить (эквивалентно делению на 2^значение справа)
<<<=	Применить арифметический сдвиг влево и присвоить (эквивалентно умножению на 2^значение справа); бит знака сохраняется
>>>=	Применить арифметический сдвиг вправо и присвоить (эквивалентно делению на 2^значение справа); бит знака сохраняется

Для инкрементирования или декрементирования сигналов также существуют сокращенные формы.

Инкрементирование сигналов

Список сокращений для инкрементирования:

- инкрементирование до операции (префиксное), ++i, увеличивает значение i перед его использованием;
- инкрементирование после операции (постфиксное), i++, увеличивает значение i после его использования;
- декрементирование до операции (префиксное), --i, уменьшает значение i перед его использованием;
- декрементирование после операции (постфиксное), i--, уменьшает значение i после его использования.

Рассмотрим теперь, как использовать переменные для принятия решений.

Принятие решений: if-then-else

Одной из основ любого языка программирования является управление потоком операций. В случае HDL это генерация фактической логики, которая будет реализована в структуре FPGA. Оператор ветвления `if-then-else` можно рассматривать как мультиплексор, а условное выражение оператора `if` – как линии выбора. Рассмотрим простейший пример:

```
if     (add == 1)  sum = a + b;
else               sum = a - b;
```

Код выше позволяет выбрать, будет ли `b` прибавляться к `a` или вычитаться из `a` в зависимости от уровня сигнала `add`. Упрощенная схема того, что может быть сгенерировано, показана на следующей диаграмме:

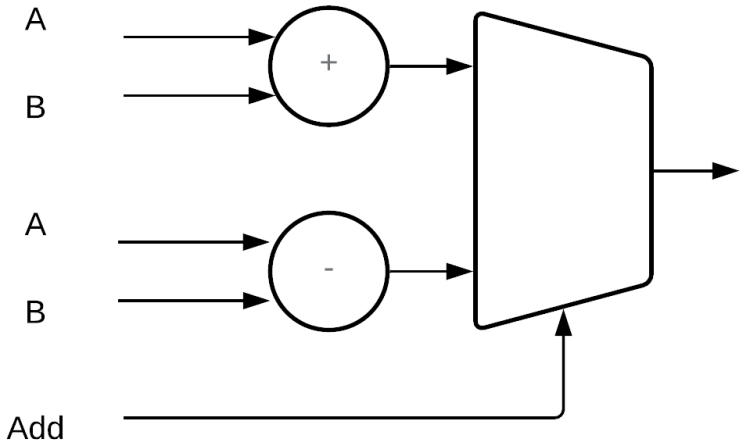


Рис. 2.3. Представление if-then-else

По всей вероятности, логика будет реализована гораздо менее затратным способом. Рекомендуется анализировать результаты ваших проектов по мере их создания, чтобы понять, какого рода оптимизация применяется при компиляции проектов.

Сравнение значений

SystemVerilog поддерживает обычные операции равенства, такие как `==` и `!=`. Эти операторы проверяют, равны или не равны две стороны сравнения соответственно. Поскольку имеется дело с аппаратным обеспечением и существует возможность наличия неопределенных значений (x), у этих операторов есть недостаток, заключающийся в том, что такие значения могут вызвать совпадение, даже если оно не предполагалось, перейдя в условие `else`. Это обычно является большой проблемой при моделировании. Поэтому существуют версии этих операторов, устойчивые к неопределенным значениям; это `===` и `!==`. В `testbench` рекомендуется использовать эти операторы, чтобы избежать непредвиденных совпадений.

Сравнение операторов равенства с подстановочными знаками

Также можно сравнивать диапазоны значений. Это возможно с помощью операторов `=?=` и `!?=`. Они позволяют использовать подстановочные знаки в условии совпадения. Например, допустим, имеется 32-битная шина и нужно обрабатывать адресацию с нечетным выравниванием:

```
if (address[3:0] =?= 4'b00zz) slot = 0;
else if (address[3:0] =?= 4'b01zz) slot = 1;
```

Операторы подстановочных знаков позволяют это сделать. В примере младшие два бита игнорируются.

Операторы if с уникальностью или приоритетностью

Обычно, когда используют оператор `if`, думают о каждой оценке `if` как об отдельном сравнении, которое зависит от предыдущих `if`. Этот тип оператора `if` является приоритетным, что означает, что первое совпадение `if` будет иметь значение «истина» (true). В простом примере, представленном ранее, просматривается один и тот же адрес и игнорируются два младших бита. Часто во время оптимизации программа понимает, что операторы `if` не могут перекрываться, и оптимизирует логику соответствующим образом. Но если известно, что операторы `if` не могут перекрываться, можно использовать ключевое слово `unique` (уникальный), чтобы сообщить Vivado, что каждый оператор `if` не пересекается с предыдущими или последующими. Это позволит программе лучше оптимизировать результирующую логику. При этом следует соблюдать осторожность. Рассмотрим, что произойдет, если попытаться сделать следующее:

```
unique   if (address[3:0] =?= 4'b00zz) slot = 0;
else     if (address[3:0] =?= 4'b01zz) slot = 1;
else     if (address[3:0] =?= 4'b1000) slot = 2;
else     if (address[3:0] =?= 4'b1zzz) slot = 3;
```

Здесь два последних оператора `else if` пересекаются. Если указать в этом случае `unique`, то, скорее всего, возникнет несоответствие между результатами моделирования и синтеза. Если во время моделирования `address[3:0]` будет равен `4'b1000`, то симулятор выдаст предупреждение о нарушении условия уникальности `unique`. Синтезирующий САПР проведет оптимизацию неправильно, и схема не будет работать так, как задумано. Такая ситуация будет показана на примере далее, когда будет нарушено условие уникальности `unique` в операторе case.

Этот тип `if` на самом деле является приоритетным, и если бы была необходимость, то программу можно было бы записать, например, таким образом:

```
priority if (address[3:0] =?= 4'b00zz) slot = 0;
```

Указывать приоритетность на самом деле не обязательно, кроме как для обеспечения ясности намерений. Потому что программа обычно может определить, можно ли оптимизировать `if` как уникальный. Если нет, то он будет рассматриваться как приоритетный.

Важное замечание

`unique` и `priority` являются мощными инструментами, поскольку они могут значительно сократить аппаратные затраты и улучшить временные характеристики полученной комбинационной схемы. Но следует соблюдать осторожность, поскольку их неправильное определение может привести к логическим ошибкам. Во время моделирования происходит проверка, чтобы условия не были нарушены, но это позволяет обнаружить только те случаи, которые возникают во время моделирования.

Оператор выбора case

Оператор выбора `case` обычно используется для выполнения большого количества сравнений. Существует три варианта этого оператора: `case`, `casex` и `casez`. Оператор `case` используется, когда нет необходимости в использовании групповых шаблонов. Если нужно использовать шаблоны, как это было рассмотрено ранее, рекомендуется использовать `casez`. Существует два основных способа использования операторов `case`. Первый (наиболее традиционный):

```
casez (address[3:0])
   4'b00zz:   slot = 0;
   4'b01zz:   slot = 1;
   4'b1000:   slot = 2;
   4'b1zzz:   slot = 3;
endcase
```

Как и в случае с оператором `if`, для управления программой можно использовать уникальность `unique` или приоритетность `priority`. Кроме того, можно задать передачу управления по умолчанию. Это должно быть сделано, если используется уникальность `unique`.

Существует еще один способ использования оператора `case`, который может быть особенно полезен:

```
priority case (1'b1)
   address[3]: slot = 0;
   address[2]: slot = 1;
   address[1]: slot = 2;
   address[0]: slot = 3;
endcase
```

В данном случае происходит определение ведущей единицы. Поскольку может быть установлено несколько битов, указание модификатора `unique` может вызвать проблемы с оптимизацией. Если бы проект имел унитарное (one-hot) кодирование[1] адреса, то указание `unique` создало бы более оптимизированное решение.

[1] Унитарный код (англ. *unitary* или *one-hot encoding*) — способ двоичного представления состояния объекта в виде последовательности битов, содержащей только одну логическую единицу, положение которой зависит от кодируемой величины. Наглядная модель унитарного кода — шкальный индикатор, высота столбика в котором пропорциональна представляемой величине. — *Прим. ред.*

> **Важное замечание**
>
> Существуют различные способы кодирования данных. Двоичное кодирование может устанавливать несколько битов одновременно и обычно представляет собой возрастающее значение. При унитарном кодировании (one-hot encoding) устанавливается один бит за один раз. Это упрощает декодирование. Существует также способ кодирования, который будет рассмотрен при обсуждении **FIFO (First-In-First-Out)**, называемый кодом Грея, невосприимчивый к проблемам синхронизации при использовании соответствующих ограничений.

Для более простых случаев выбора в SystemVerilog имеется простой оператор условного присваивания.

Условный оператор

В SystemVerilog есть более сокращенный формат для условного выбора результата в следующей форме:

```
Out =   (sel)  ?   ina  :   inb;
```

Когда `sel` имеет высокий уровень, `ina` будет присвоен `out`; в противном случае `inb` будет присвоен `out`.

> **Подсказка**
>
> Запись `sel ? ...` является более короткой формой для `sel == 1'b1 ? .....`

В этом разделе рассмотрены основные типы данных и массивы, а также способы их использования. В следующем разделе будет показано, как применять пользовательские типы данных.

Использование пользовательских типов данных

SystemVerilog предоставляет множество способов создания определяемых пользователем типов. Определяемые пользователем типы также могут храниться в массивах.

Создание структур

Структуры (structures) позволяют группировать сигналы, которые должны быть вместе. Например, если бы нужно было создать 16-битное значение, состоящее из двух 8-битных значений, `h` и `l`, можно было бы сделать что-то вроде этого:

```
typedef struct packed {bit  [7:0]  h;  bit  [7:0]  l;}  reg_t;
reg_t  cpu_reg;
assign   cpu_reg.h   =   8'hFE;
```

Значение ключевых слов:

- `typedef` означает, что создается тип, определяемый пользователем;
- `struct` – создается структура;
- `packed` – структура в памяти будет упакованной[1].

> **Подсказка**
>
> Структуры и объединения могут быть упакованными или неупакованными, но, поскольку упакованные обычно имеют больше смысла в контексте аппаратного обеспечения, в данной книге будут использоваться именно они.

Обращение к частям структуры осуществляется с помощью добавления к сигналу части структуры (в данном случае `h`), отделенной точкой.

Создание объединений

Объединение (`union`) позволяет создать переменную с несколькими представлениями. Это полезно, если нужно несколько способов доступа к одним и тем же данным. Например, когда микропроцессоры перешли от 8 к 16 битам, потребовались способы доступа к части регистра для старых операций:

```
union packed {bit [15:0] x; cpu_reg cr;} a_reg;
always_comb begin
   a_reg.x = 16'hFFFF;
   a_reg.cr.h = '0;
end
```

В примере выше создано объединение 16-битного регистра и структуры, состоящей из двух 8-битных значений. Здесь после первого блокирующего присваивания `a_reg` устанавливает все биты равными 1. После второго присваивания старшие 8 бит установлены равными 0, что означает, что `a_reg` равен `16'h00FF`.

Проект 1. Создание комбинационной схемы

В этой главе представлены типы сигналов и способы создания комбинационной схемы. Этот проект будет содержать несколько компонентов, которые нужны для создания небольшого калькулятора. Он довольно простой и будет обладать следующими возможностями:

- находить позицию ведущей единицы для входного вектора, установленного при помощи переключателей;
- складывать, вычитать или перемножать два числа;
- подсчитывать количество установленных переключателей.

[1] В упакованной структуре не добавляются дополнительные биты для выравнивания полей по границам байтов или слов. Упакованные структуры занимают меньше места (ровно столько, сколько занимают их поля). Неупакованные структуры занимают больше места за счет выравнивания, но работают быстрее при симуляции. – *Прим. конс.*

На следующей схеме показано, как выглядит плата Nexys A7:

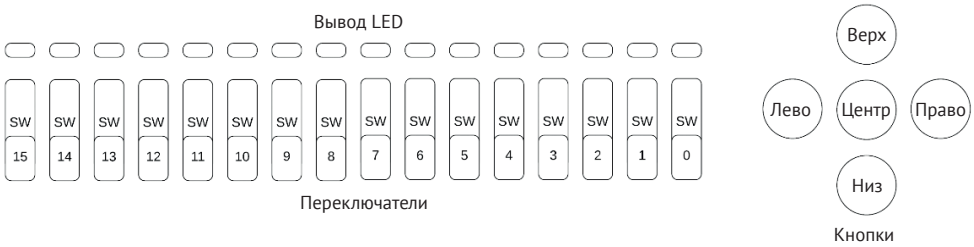


Рис. 2.4. Вводы/выводы платы Nexys A7

В проекте из предыдущей главы было изучено, как использовать переключатели для ввода и светодиоды для вывода. В этом проекте будут задействованы все переключатели, показанные на рис. 2.4, для вычисления количества единиц и детектора ведущей единицы. Для реализации детектора ведущей единицы устройство будет определять положение самого левого переключателя из 16 установленных.

Для арифметических операций разделим переключатели на две группы. Переключатели 7:0 будут предназначены для ввода B, а переключатели 15:8 – для ввода A. Выходные данные будут отображаться в виде числа в дополнительном коде, используя все 16 светодиодов над переключателями, как показано на схеме. То есть –1 будет означать, что все светодиоды горят, а 0 – что все светодиоды выключены.

Testbench

Поскольку будет происходить сборка отдельных компонентов, требуется разработать универсальный `testbench`, который позволит тестировать каждый компонент по отдельности, а затем все вместе. Для этого будут использоваться параметры. В этом `testbench` будет три параметра:

- `SELECTOR` используется для модуля ведущей единицы, чтобы определить один из четырех способов поиска ведущей единицы. Он также используется для выбора между сложением и вычитанием для модуля `add_sub`;
- `UNIQUE_CASE` определяет, будет ли генерироваться уникальное значение регистра или случайное число, в котором может быть несколько установленных битов;
- `TESTCASE` позволяет тестировать отдельные компоненты (`LEADING_ONES`, `NUM_ONES`, `ADD`, `SUB` и `MULT`) или все вместе (`ALL`).

Чтобы изменить эти параметры в `testbench`, нужно выбрать **Settings | Simulation | Generics/Parameters**:

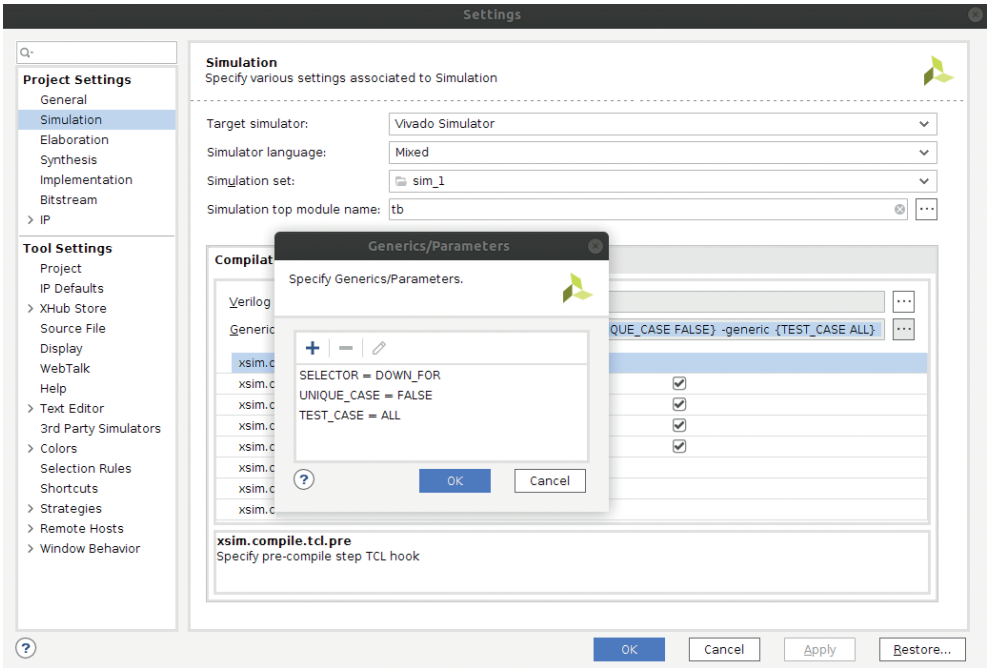


Рис. 2.5. Установка параметров симуляции

Аналогично, чтобы изменить параметры для реализации, нужно выбрать **Settings | General | Generics/Parameters**:

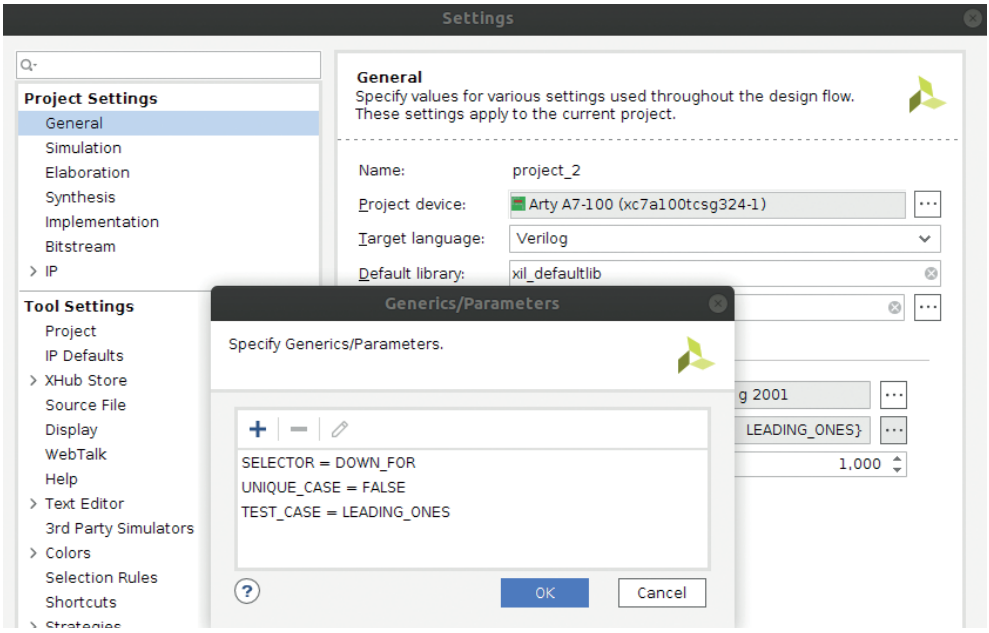


Рис. 2.6. Установка параметров, используемых при сборке проекта

Существует множество способов разработки `testbench`. Среди разработчиков принято использовать отдельные файлы для отдельных тестов и сценарий оболочки для многократного вызова симулятора. Если вам интересно изучить этот тип `testbench`, ознакомьтесь с графическим ускорителем с открытым исходным кодом GPLGPU на GitHub: `https://github.com/asicguy/gplgpu`. Для текущего проекта будет применяться более простой способ: использование параметров для выбора тестовых примеров.

В целом существует три способа тестирования проекта.

Моделирование с помощью целевого тестирования

Этот тип тестирования используется, когда есть конкретный тестовый случай, который нужно проверить. Например, может быть проверка того, что произойдет, когда в детекторе ведущей единицы не установлены никакие биты, все биты установлены в состояние единиц или установлены наибольшее и наименьшее значения в случае математических операций. Их также можно использовать для завершения рандомизированного тестирования.

Моделирование с использованием рандомизированного тестирования

Рандомизированное тестирование используется в основном в `testbench` с самопроверкой. Для этого можно использовать две системные функции:

- `$random()`, которая возвращает 32-битное случайное число. Она возвращает новое случайное число каждый раз, когда ее вызывают;
- `$urandom_range(a,b)`, которая возвращает число, находящееся между a и b. В нашем случае используется `$urandom_range(0,4)` для установки одной из четырех кнопок.

Далее будет описано, как производить моделирование с помощью ограниченной рандомизации.

Моделирование с использованием ограниченной рандомизации

В SystemVerilog встроен очень надежный набор возможностей для тестирования. Этот тип тестирования можно представить так, как будто имеется процессор с некоторым количеством допустимых инструкций, и необходимо рандомизировать `testbench` так, чтобы он использовал эти инструкции и убедиться, что все они будут использованы в определенный момент. Это выходит за рамки данной книги, но в разделе «Дополнительное чтение» приведены ссылки, с которыми можно ознакомиться.

Реализация детектора ведущей единицы с использованием оператора case

Первый модуль представляет собой детектор ведущей единицы. Он будет реализован несколькими различными способами, чтобы показать их преимущества, недостатки и потенциальные проблемы.

Первое, что нужно решить, – является ли входящий сигнал унитарным кодом. Если он представляет собой унитарный код, можно получить оптимизированный результат, используя ключевое слово `unique`:

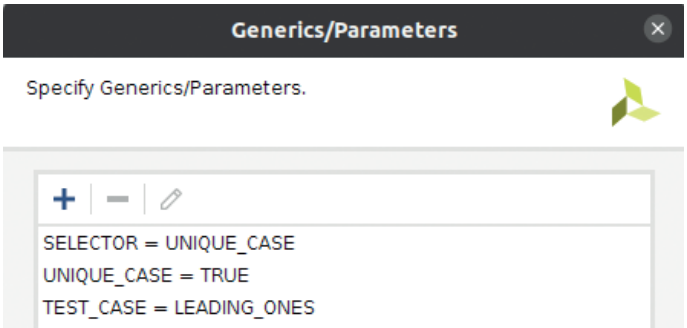


Рис. 2.7. Тестирование ведущего бита с помощью оператора case

Убедитесь, что параметры симуляции установлены так, как показано на скриншоте выше.

Управление реализацией с помощью generate

Уделите немного времени и изучите файл `leading_ones.sv`. В нем показано, как оператор генерации `generate` можно использовать для выборочного создания кода. Формат оператора `generate` – это `generate <условие (condition)>`, как показано ниже:

```
generate
  if (SELECTOR == "UNIQUE_CASE") begin : g_UNIQUE_CASE
```

В данном случае условие представляет собой оператор `if` и используется для выборочной реализации одного из четырех блоков `always`. Операторы выбора `case` и циклы `for` также являются допустимыми условиями, которые будут разобраны по мере изучения материала этой книги. Именно здесь параметры особенно полезны для управления тем, что будет создано.

> **Подсказка**
>
> Хорошим стилем программирования является использование меток внутри блоков `generate`. В будущих версиях `SystemVerilog` это будет обязательным требованием.

Обратите внимание, что оператор `case` по умолчанию закомментирован. На данном этапе оставьте его как есть и запустите тест:

```
  WARNING: 100000ns : none of the conditions were true for unique case from
File:/home/fbruno/git/books/Learn-FPGA-Programming/CH2/hdl/leading_ones.sv:17
```

Почему было сгенерировано предупреждение? Создавая уникальный случай, следует убедиться, что не только происходит лишь одно совпадение, но и что вообще хоть одно совпадение будет. Идея в том, чтобы `LED = 0`, когда не

установлен SW, поэтому раскомментируем условие по умолчанию. Теперь если запустить моделирование снова, то тест будет пройден.

> **Важное замечание**
>
> Параметры могут управлять тем, как реализуется логика или как выполняется код testbench. В testbench есть строка `if (UNIQUE_CASE == «TRUE») begin`, которая управляет тем, как выполняется код для ограничения количества устанавливаемых единиц.

Теперь разрешите неуникальные значения, чтобы посмотреть, как среда моделирования их обрабатывает. Измените UNIQUE_CASE на "FALSE":

```
Setting switches to 0011010100100100
WARNING: 0ns : Multiple conditions true
     condition at line:21 conflicts with condition at line:20
for unique case from File:/home/fbruno/git/books/Learn-
FPGA-Programming/CH2/hdl/leading_ones.sv:17
```

Это только первый случай, но на самом деле их много. Если `testbench` обнаруживает случаи, нарушающие предположение об уникальности, будут выведены предупреждения, которые сигнализируют о том, что у проекта могут быть проблемы.

Итак, давайте посмотрим, что произойдет, если создать загрузочный файл (bitstream). Убедитесь, что в **Settings | General | Top Module Name** установлено значение `leading_ones`, а в `SELECTOR` в разделе **Generics/Parameters** – значение `UNIQUE_CASE`. Затем нажмите на **Generate Bitstream**.

> **Важное замечание**
>
> Настройки **Generics/Parameters** выполняются в двух местах в Vivado. Общие настройки применяются при создании проекта. Параметры моделирования применяются только при моделировании.

Посмотрите на вкладку Project Summury. В нижней левой части окна посмотрите на использование ресурсов после синтеза проекта. По умолчанию результат отображается в виде графика, но можно выбрать табличный вариант, чтобы получить точные значения. В данном случае получен такой результат:

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	7	63400	0.01
IO	21	210	10.00

Рис. 2.8. Использование ресурсов после синтеза проекта

Для этой реализации потребовалось 7 LUT. Но что произойдет, если запустить проект на плате? Откройте менеджер оборудования (hardware manager) и цель (target), а затем выберите **Program device**.

Ожидается значение в унитарном коде, поэтому попробуйте устанавливать по одному биту за раз, начиная с 0, так чтобы одновременно был включен только один переключатель в унитарной кодировке. Правильно ли загораются светодиоды? Вы должны увидеть двоичное значение для установленного переключателя плюс один, поэтому включенный SW0 будет показывать 5'b00001, SW1 будет показывать 5'b00010, а SW15 – 5'b10000. Теперь попробуйте установить несколько переключателей, например 15 и 0. Что вы получили? В данном случае получено значение 5'b10001. Теперь попробуйте другие варианты. Вы заметите, что некоторые комбинации все равно случайно дают правильное значение.

Теперь попробуйте пересобрать проект без ключевого слова `unique`. Установите `SELECTOR` на `"CASE"`, а затем сгенерируйте загрузочный файл.

Посмотрев на отчет по этой сборке, можно увидеть, что обработка приоритетности потребовала почти в два раза большего количества LUT. В данном случае потребовалось 13 LUT. Проверьте работу проекта на плате.

Попробуйте объединить несколько переключателей. Всегда ли получается положение переключателя +1 для самого верхнего переключателя?

В этом разделе показано, что уникальность `unique` допускает оптимизацию. Оператор `unique case` потребовал почти в два раза меньше ресурсов, чем оператор выбора `case` без уникальности `unique`. У оператора выбора `case` недостатком является то, что приходится указывать все возможные случаи, поэтому его нельзя использовать повторно для произвольного количества случаев. Давайте рассмотрим другой, более масштабируемый способ обработки детектора ведущей единицы: с помощью использования цикла `for`.

Проектирование многоразового детектора ведущей единицы с помощью цикла for

Цикл `for` позволяет быстро создавать повторяющуюся логику. В случае с детектором ведущей единицы легко представить, как это можно сделать с помощью цикла `for`. Есть два способа добиться этого, оба из которых будут рассмотрены в данном разделе.

Установка SELECTOR = DOWN_FOR

Первый способ прост и повторяет шаги решения этой задачи с помощью оператора `case`:

```
always_comb begin
   LED = '0;
   for (int i = $high(SW); i >= $low(SW); i--) begin
      if (SW[i]) begin
         LED = i + 1;
         break;
      end
   end
end
```

Системные функции `$high` и `$low` для того, чтобы сделать код универсальным и пригодным для многократного использования. Цикл прерывается при первом обнаружении единицы.

> **Подсказка**
>
> Прерывание (`break`) цикла `for` является синтезируемым. Важно учитывать, можно ли развернуть цикл или есть ли способ описать цикл так, чтобы прерывание не требовалось. Если можно придумать относительно простой способ сделать это, то, скорее всего, проблем с синтезом не возникнет.

Например, можно развернуть цикл, представив его следующим образом:

```
Logic [3:0] SW;
always_comb begin
   LED = '0;
   if      (SW[3]) LED = 4;
   else if (SW[2]) LED = 3;
   else if (SW[1]) LED = 2;
   else if (SW[0]) LED = 1;
   else LED = 0;
end
```

Теперь рассмотрим другой способ представления цикла `for`, который удовлетворяет требованию развертывания.

Установка SELECTOR = UP_FOR

Продвигаясь от младшего бита к старшему при поиске единицы, можно гарантированно найти старший бит, как последнюю найденную единицу. Это также позволяет понять, что прерывание может быть синтезировано, поскольку есть способ представить цикл `for` так, чтобы в прерывании не было необходимости.

Подсчет количества единиц

С поиском ведущей единицы связан подсчет количества единиц в векторе. Это можно легко сделать с помощью цикла `for`:

```
always_comb begin
   LED = '0;
   for (int i = $low(SW); i <= $high(SW); i++) begin
      LED += SW[i];
   end
end
```

Установите `SELECTOR` на `NUM_ONES` и `TEST_CASE` на `NUM_ONES` и запустите моделирование, чтобы убедиться в работоспособности схемы. Убедитесь, что `SELECTOR` установлен на `NUM_ONES` на вкладке **`General`** и что имя верхнего модуля установлено на `num_ones`. Затем сгенерируйте загрузочный файл и запустите его на плате.

Проверьте проект на плате, переключая переключатели один за другим в любом порядке. В результате светодиоды должны загораться в соответствии с двоичным счетом, т. е. 16'b0, 16'b1, 16'b10, 16'b11 и т. д.

Реализация сумматора/вычитателя (adder/subtractor)

Рассмотрим модуль add_sub. В математике существует множество способов реализации сумматора или вычитателя. Многие компании продают инструменты для проектирования арифметических блоков с высокой производительностью или с малым количеством логических элементов. Для FPGA в 99 % случаев лучше позволить синтезатору оптимизировать проект. Результат будет вполне удовлетворительный. С помощью параметра SELECTOR можно выбрать, какая операция сложения или вычитания будет выполняться.

Сложение

Установите SELECTOR на ADD и TEST_CASE на ADD и запустите моделирование, чтобы убедиться, что все работает. Убедитесь, что SELECTOR установлен на ADD на вкладке **General** и что имя верхнего модуля иерархии установлено в add_sub. Затем сгенерируйте загрузочный файл и запустите его на плате.

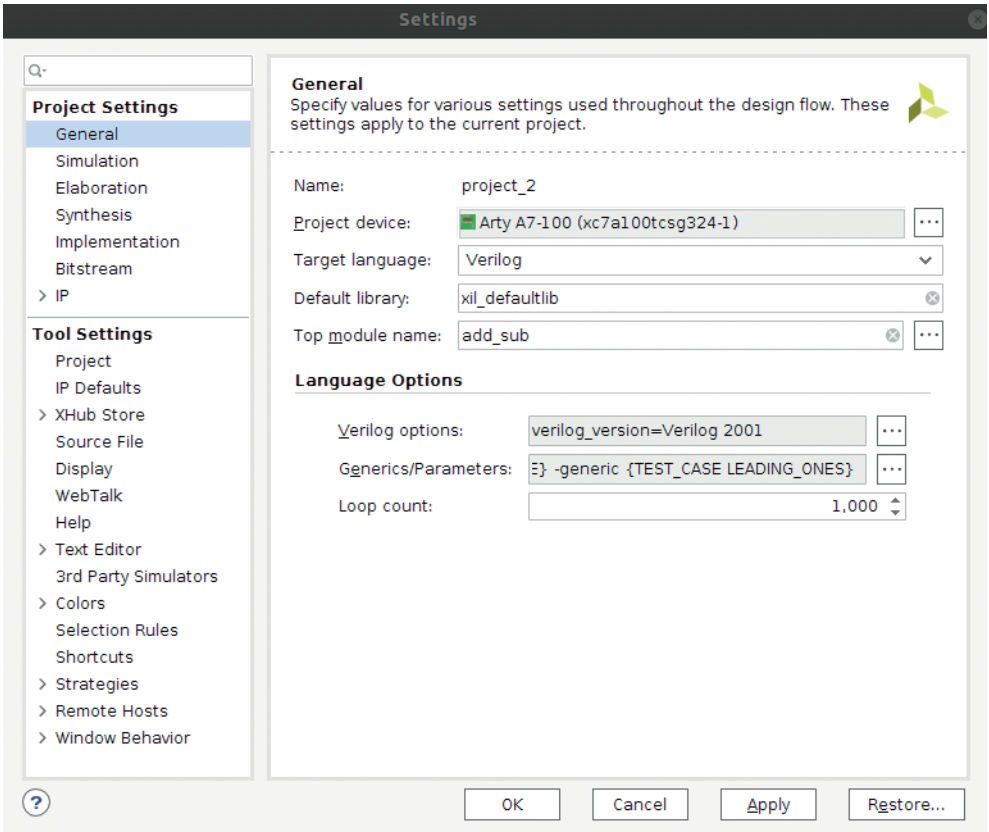


Рис. 2.9. add_sub выбран как верхний модуль иерархии

После загрузки битового потока на плату попробуйте несколько комбинаций битов в нижних 8 и верхних 8 битах. В частности, если установить бит 0

и бит 8 равными 1, то на светодиоде должен установиться бит 1, т. е. значение `16'h2`. Теперь попробуйте установить бит 0 и бит 15 в 1. Какой будет результат?

Может показаться немного странным, что загорелось так много светодиодов, но можно заметить, что горят только старшие биты. Это потому, что было указано `8'h80` + `8'h1`. Поскольку числа представлены в дополнительном коде, в десятичной системе это будет –128 + 1 или –127, что в шестнадцатеричной системе будет равно `16'hFF81`.

Вычитание

Установите `SELECTOR` на `SUB` и `TEST_CASE` на `SUB` и запустите моделирование, чтобы убедиться, что все работает. Убедитесь, что `SELECTOR` установлен на `SUB` на вкладке **General** и что имя верхнего модуля иерархии установлено на `add_sub`. Затем сгенерируйте загрузочный файл и запустите его на плате.

Теперь происходит вычитание младших 8 бит из старших 8 бит. Попробуйте установить бит 15 в нулевое состояние. Все светодиоды должны гореть, т. е. показывать –1.

> **Важное замечание**
>
> Чтобы получить –1 в двоичном формате, нужно инвертировать число и добавить 1; например:
> `-16’b0000000000000001 = 16’b11111111111111110 + 1 = 16’b1111111111111111111111111.`

Обратите внимание, что для сумматора и вычитателя, независимо от того, что складывается со знаковыми числами, старшие 8 бит всегда будут либо все 0, либо все 1.

Умножение

Последний модуль, который будет рассмотрен, – умножитель. Синтезирующий САПР выберет самый простой способ реализации, и, поскольку размер умножителя всего 8×8, по умолчанию он будет реализован с помощью LUT.

Установите `SELECTOR` на `MULT` и `TEST_CASE` на `MULT` и запустите моделирование, чтобы убедиться, что все работает.

Процесс моделирования автоматизирован. Но также можно использовать команду `add_force` в среде моделирования. Пример такого использования показан на следующем скриншоте. Команда `force` (принудительно) переопределяет значение сигнала в среде моделирования. Когда моделирование закончилось, было принудительно установлено значение `0x1234` на SW-вход умножителя. После этого нужно увеличить время моделирования, что можно сделать с помощью команды `run 10ns`.

Команда `force` хорошо подходит для тех случаев, когда требуется изолировать определенный сценарий или провести эксперимент со сценарием `what if` (что, если) во время выполнения. Выполнять моделирование только таким образом не рекомендуется, поскольку при следующем моделировании придется повторять все действия. Поэтому размещение тестов в `testbench` SystemVerilog является лучшим решением в долгосрочной перспективе.

Когда работа над проверкой отдельного сценария закончена, можно использовать команду `remove_forces` для сигнала, чтобы вернуть управление `testbench`.

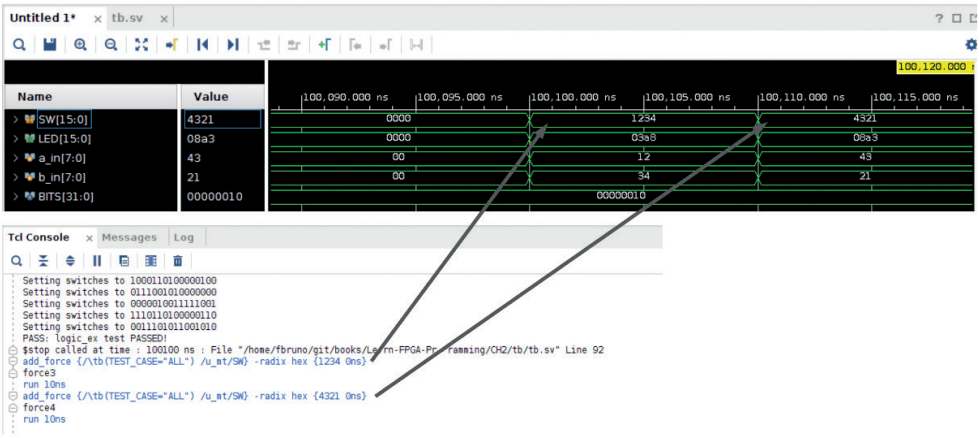


Рис. 2.10. Оператор force в симуляции

Убедитесь, что `SELECTOR` установлен на `MULT` на вкладке **`General`**, а имя модуля верхнего уровня иерархии установлено на `mult`, затем сгенерируйте загрузочный файл и запустите его на плате.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	61	63400	0.10
IO	32	210	15.24

Рис. 2.11. Использование умножителя

На рисунке выше показаны ресурсы, использованные при построении умножителя.

> **Подсказка**
>
> При сложении двух знаковых чисел размера n получится значение размера n.
>
> При сложении двух беззнаковых чисел размера n получится значение размера n+1. Умножение двух чисел размера n приведет к значению размера 2×n.

Объединяем все вместе

Создадим простое АЛУ (арифметико-логическое устройство) верхнего уровня, чтобы можно было собрать все созданные модули вместе. Проанализируйте на `project_2`. На плате есть пять кнопок. Они будут использоваться для управления тем, результат какой операции будет на выходе.

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	134	63400	0.21
IO	37	210	17.62

Когда процесс загрузки прошивки на плату закончится, светодиоды погаснут. Нажмите несколько переключателей и выберите нужную функцию, нажав на соответствующую кнопку. Поздравляем – простой калькулятор готов! Обратите внимание на то, что, если кнопку отпустить, светодиоды погаснут.

Добавление защелки

Поскольку пока не используются генераторы тактовых импульсов, добавим защелку. В данном конкретном случае переключатели статичны, поэтому использование защелки не должно вызвать никаких проблем:

```
always_latch begin
//always_comb begin
   //LED = '0;
```

Замените `always_comb` на `always_latch` и закомментируйте `LED = ' 0`. Затем запустите программу заново. Что происходит, когда прошивка загружена на плату и вы пытаетесь выбрать операцию? Операция будет не такой, как ожидалось, и светодиоды будут вести себя почти случайным образом. Именно по этой причине не следует использовать защелки. Если схема ведет себя не так, как предполагалось, следует проанализировать журналы компиляции и убедиться, что не была синтезирована защелка.

Выводы

В этой главе рассказано о том, как создавать комбинационную логику и различные модули, а также как проводить их моделирование с помощью `testbench` с самопроверкой. Также были изучены различные методы оптимизации, которые можно выполнить с оператором `case`, и показано, как в некоторых случаях можно получить значительную экономию ресурсов; но также было показано, как можно столкнуться с проблемами, если проектные предположения неверны. Затем были упомянуты защелки и проблемы, которые они вызывают, даже когда они вроде бы должны быть безопасны.

На данном этапе хочется надеяться на то, что у вас появилась определенная уверенность в том, как создавать комбинационные схемы и проводить их моделирование. Следующая глава знакомит с последовательностными схемами, а именно с использованием регистров для хранения значений и выполнения операций. Простой калькулятор будет расширен и будет показано, как его можно улучшить за счет использования элементов хранения.

Вопросы

1. Упакованный массив используется для реализации памяти. Правда или нет?
2. Когда оператор `break` может быть использован в цикле `for`?
   a) В любом случае.
   b) Если есть возможность переписать цикл for таким образом, чтобы прерывание не требовалось.
   c) Только если можно изменить направление цикла, т. е. выполнить его снизу вверх, а не сверху вниз.

3. Определите размер сигналов `add_unsigned, add_signed и mult`:

```
Logic unsigned [7:0] a_unsigned;
logic unsigned [7:0] b_unsigned;
logic signed [7:0] a_signed;
logic signed [7:0] b_signed;
assign add_unsigned = a_unsigned + b_unsigned;
assign add_signed = a_signed + b_signed;
assign mult = a_unsigned * b_unsigned;
```

4. Деление – очень дорогостоящая операция. Посмотрите на поддерживаемые Vivado конструкции в руководстве Vivado Synthesis manual («Дополнительное чтение»). Можно ли легко заменить операцию умножения операцией деления? Что можно сделать без модификации кода?

Задание повышенной сложности

Проанализируйте модуль add_sub:

```
logic signed [BITS/2-1:0]       a_in;
logic signed [BITS/2-1:0]       b_in;
...
  {a_in, b_in} = SW;
```

Если бы требовалось заменить `a_in` и `b_in` на пользовательский тип, который инкапсулирует оба значения, что следовало бы использовать: структуру или объединение? Измените код так, чтобы он использовал пользовательский тип, а затем запустите симуляцию и опробуйте его на плате.

Дополнительное чтение

Обратитесь к следующим ссылкам для получения дополнительной информации о том, что было рассмотрено в этой главе.

- Информация об UVM[1]: https://www.accellera.org/downloads/standards/uvm.
- Руководство по работе в Vivado:

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug901-vivado- synthesis.pdf.

[1] Universal Verification Methodology (UVM) — стандартизированная методика верификации цифровых схем. — *Прим. ред.*

Глава 3

Подсчет нажатий на кнопку

В этой главе будет рассказано, как сохранять состояние проекта путем добавления последовательностных элементов. Ограничиваясь комбинационной логикой и не имея возможности хранить информацию, на самом деле можно достичь немногого. Для того чтобы получить полезное устройство, нужен счетчик команд, регистры и долговременное хранение информации. Что представлял бы из себя мобильный телефон без возможности хранить номера, электронные письма или фотографии?

В этой главе будут рассмотрены следующие основные темы:

- что такое последовательностные элементы и как их использовать;
- проект – подсчет нажатий на кнопоку;
- синхронизация.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH3.

Что такое последовательностный элемент?

В главе 1 «Введение в FPGA и Xilinx Vivado» была рассмотрена защелка и показано, что она не подходит для целей хранения значений. Для хранения информации разработчики FPGA используют регистр или триггер (flip-flop). Прежде чем создать первый триггер, нужно кратко ознакомиться с генераторами тактовых импульсов.

Синхронизация проекта

Для реализации цифровой логики обычно требуется хотя бы один источник синхронизации в проекте, а иногда и несколько. Источник синхронизации часто называют генератором тактовых импульсов. Он обычно представляет собой внешний кварцевый генератор, который синтезирует колебания

с определенной частотой, представляющие собой последовательность из нулей и единиц в схеме. Иногда в этой книге вход тактового генератора будет использоваться напрямую. Но, если нужна определенная частота, более высокая или низкая, чем на этом входе, используют другие варианты, такие как **блок фазовой автоподстройки частоты**, или **ФАПЧ (Phase Locked Loop, PLL**), и **блок управления тактовой частотой (Mixed Mode Clock Manager, MMCM)**, которые будут обсуждаться в главе 5 «Ресурсы FPGA, и как их использовать».

Изображая временные диаграммы, обычно рисуют тактовые импульсы в виде периодического сигнала прямоугольной формы, как показано на рис. 3.1:

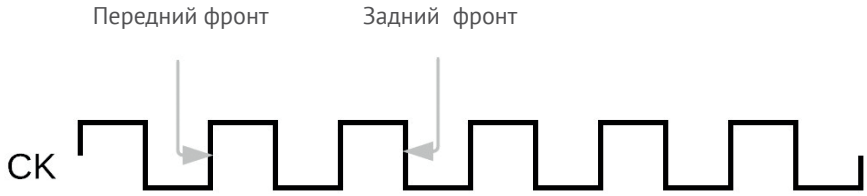


Рис. 3.1. Тактовые импульсы

Vivado необходимо сообщить о созданных тактовых импульсах, чтобы программа могла правильно отсчитывать время в проектах. До сих пор в этой книге про синхронизацию ничего не говорилось, поскольку не было эталона для измерения времени. Добавьте следующий текст в XDC-файл:

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } \
   [get_ports { clk }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name clk -period 10.00 -waveform {0 5} \
   [get_ports {clk}];
```

Чтобы создать генератор тактовых импульсов в проекте, используется команду `create_clock` языка **TCL**. Нужно указать период (`-period`) тактовых импульсов в наносекундах, а также можно указать, как будет выглядеть форма сигнала (`-waveform`). Примените это к порту в проекте с помощью `get_ports` и задайте генератору тактовых импульсов имя с помощью `-name`. В более сложных проектах можно определить несколько генераторов тактовых импульсов на данном выводе; например, можно использовать высокую тактовую частоту для повышения производительности и низкую для экономии энергии, а для генерации тактовых импульсов использовать **фазовую автоподстройку частоты (ФАПЧ).** При применении нескольких генераторов тактовых импульсов анализатор синхронизации должен убедиться в том, что схема соответствует временным требованиям и имеет безопасные пересечения тактовых доменов.

> **Подсказка**
>
> В большинстве случаев не нужно беспокоиться об указании формы сигнала (`-waveform`). Но если в проекте есть несколько сдвинутых по фазе тактовых импульсов одинаковой частоты, то нужно указать параметр формы сигнала, чтобы анализ синхронизации был выполнен правильно.

Пока что проект будет простой, и в нем будет генерироваться только один тактовый сигнал.

Базовый регистр

Если бы сейчас было проектирование ASIC, то, скорее всего, в библиотеке было бы несколько различных типов регистров, поскольку они могут быть оптимизированы по площади в зависимости от функциональности. Например, **Т-триггеры** (**toggle flip-flops, FF**), устройства, которые переключаются, если управляющий сигнал установлен в 1 и на них подается тактовый сигнал. Поскольку компания Xilinx ориентируется на все возможные типы схем, регистры основаны на так называемых **D-триггерах (D flip-flops, DFF)**[1].

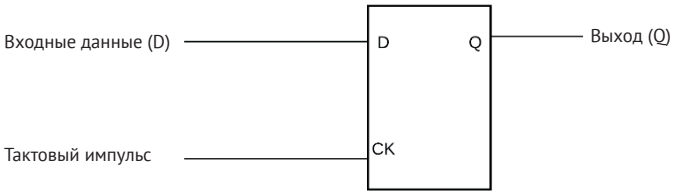


Рис. 3.2. Простой D-триггер

Простой DFF на рис. 3.2 принимает данные на вход D и сохраняет их каждый такт, выставляя их на выходе Q. Этот тип запоминающего элемента должен непрерывно получать данные на вход, поскольку каждое изменение на входе зеркально отражается на выходе.

Создание триггеров

В SystemVerilog можно создать триггер одним из двух способов: используя `always_ff @ (edge sensitivity list)` или `always @ (edge sensitivity list)`.

---

[1] Русскоязычная терминология в отношении триггеров отличается от англоязычной. Для избежания недопонимания со стороны учившихся по отечественным пособиям следует пояснить, что триггер-защелку (англ. *latch*), о котором шла речь ранее, работающий по уровню тактового сигнала, в отечественной литературе называют статическим D-триггером; триггер (англ. *flip-flop,* FF), сохраняющий данные по фронту тактового сигнала, – динамическим D-триггером (у автора это D flip-flop, DFF); а Т-триггер (*toggle flip-flop*), переключающийся в противоположное состояние по фронту тактового сигнала, – счетным триггером. Подробнее можно посмотреть, например, в книге: Микушин А. В., Сажнев А. М., Сединин В. И. Цифровые устройства и микропроцессоры. СПб, БХВ-Петербург, 2010. – *Прим. ред.*

Два ключевых слова SystemVerilog, которые описывают событие, происходящее на фронте сигнала, – это `posedge`, передний (положительный) фронт тактового сигнала (*rising edge*), и `negedge`, задний (отрицательный) фронт (*falling edge*). В общем случае в данной книге будет использоваться только передний фронт, но в некоторых особых обстоятельствах иногда может понадобиться и задний фронт сигнала.

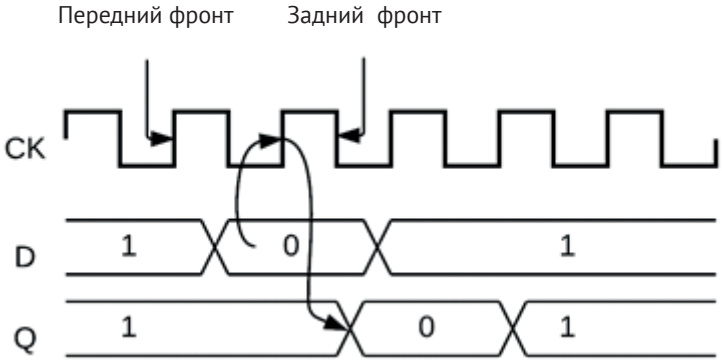


Рис. 3.3. Синхронизация по переднему фронту (posedge) DFF

На диаграмме фронты тактового сигнала обозначены как `posedge` и `negedge`.

> **Подсказка**
>
> В целом придерживайтесь одного фронта тактового сигнала и используйте его последовательно. В проектах, описанных в данной книге, используется только передний фронт. Это поможет избежать проблем с синхронизацией в проекте.

Рассмотрим, как построить DFF в `SystemVerilog`:

CH3/simple_ /hdl/simple_ .sv

```
module dff (input wire D, CK, output logic Q);
   always_ff @(posedge CK) Q <= D;
endmodule
```

Преимущество команды `always_ff` в том, что она передает замысел проекта. Vivado принимает эту конструкцию, но не генерирует ошибку, даже если FF не указано. Другие же средства проектирования могут генерировать ошибку, поэтому в большинстве случаев рекомендуется использовать `always_ff`. Если вы хотите провести моделирование или запуск этого проекта на плате Nexys A7, то проект находится по ссылке
https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH3/simple_ff.

Для моделирования, сборки и тестирования проекта следуйте шагам, описанным в главе 2 «Комбинационная логика».

Рис. 3.4. Простая синхронизация триггера

Из осциллограммы выше видно, что Q изменяется с каждым изменением входа D. Это позволяет хранить данные, но, если нужно использовать DFF для хранения данных, необходимо самостоятельно реализовать логику передачи входного значения триггера на его выход.

Когда использовать always@() для генерации триггера

У `always_ff` есть ограничение, заключающееся в том, что любой генерируемый им сигнал не может управляться ничем другим. FPGA поддерживают использование начального оператора для определения начального значения выхода FF. В примере `simple_ff.sv` начальное значение для Q не определяется кодом. Это видно на рис. 3.4, где условием запуска является неопределенное состояние 'x. Это означает, что инструмент синтеза может использовать либо 0, либо 1. Можно попробовать создать начальное значение для FF:

CH3/simple_init_ /hdl/simple_init_ .sv

```
module dff (input wire D, CK, output logic Q);
   initial Q = 1;
   always_ff @(posedge CK) Q <= D;
endmodule
```

Но если попытаться запустить симулятор, то в панели сообщений будет следующее уведомление об ошибке:

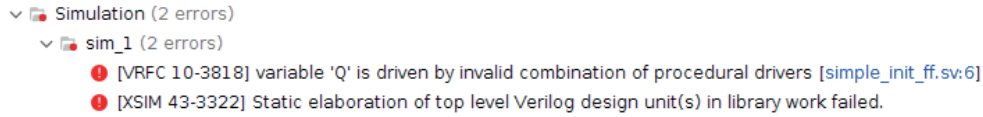


Рис. 3.5. Сбой моделирования с использованием always_ff с начальным значением

Исправить это можно, изменив `always_ff` на `always`, тогда схема будет работать правильно. Внесите изменения и запустите моделирование снова:

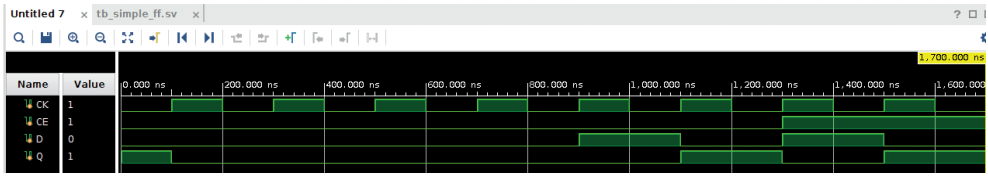


Рис. 3.6. Начальное значение Q

При предыдущем запуске моделирования Q имеет начальное значение 1, но на первом такте в Q сразу же загружается D, равное 0.

Использование неблокирующих присваиваний

Можно заметить, что впервые встретилось присваивание с помощью <=, т. е. неблокирующее присваивание. Причина использования неблокирующего присваивания заключается в том, что теперь, когда были введены последовательностные элементы, необходимо следовать параллелизму, характерному для **языков описания аппаратуры (HDL)**, и рассмотреть планирование как способ его моделирования.

До сих пор все программы выполнялись как в любой обычной программе, в виде серии шагов, выполняемых последовательно. Давайте посмотрим, как это будет работать, если применить такой же подход к блоку кода, реализующему регистр:

CH3/blocking/hdl/blocking.sv

```
always @(posedge CK) begin
   stage = D;
   Q = stage;
end
```

Получилось, что был введен промежуточный элемент хранения под названием `stage` (стадия). Что произойдет, если запустить моделирование приведенного кода?

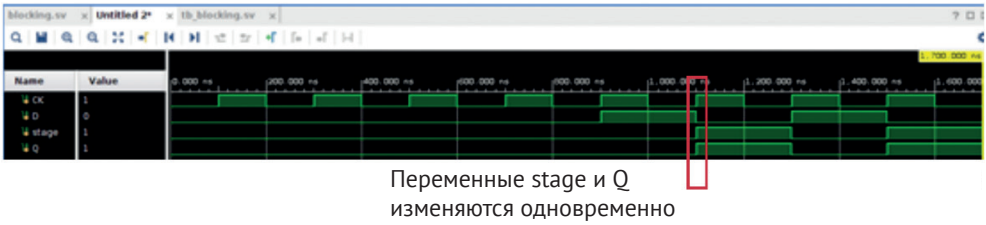


Рис. 3.7. Моделирование блокирующих присваиваний в блоке always, реализующем последовательностную схему

Важно отметить, что элементу `stage` сразу же присваивается значение D, а затем Q сразу же становится равен значению `stage`. `stage` фактически становится соединителем в окончательной реализации схемы.

Что произойдет, если попробовать изменить BLOCK на "FALSE" в `testbench`?

```
always @(posedge CK) begin
   stage <= D;
   Q <= stage;
End
```

Если запустить моделирование, то окажется, что `stage` ведет себя как стадия конвейера, как и предполагалось.

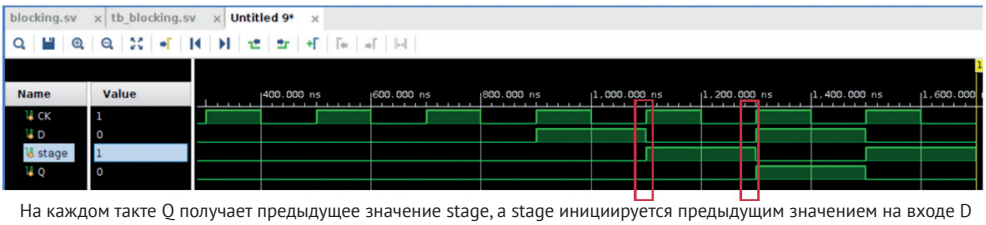


Рис. 3.8. Моделирование неблокирующих присваиваний в блоке always, реализующем последовательностную логику

Обратите внимание на различия. Если посмотреть на предыдущий код с использованием неблокирующих присваиваний, то он интерпретируется следующим образом:

- запланировать присвоение D следующему значению `stage`, но текущее значение `stage` оставить неизменным;
- запланировать присвоение текущего значения `stage` следующему значению Q.

Фактически симулятор будет проходить через весь код проекта, планируя присваивания. Эти запланированные события называются дельта-циклами. Как только это будет выполнено, время начнет идти вперед. Затем присваивания вступят в силу, и все повторится снова. Именно так и планировалось, что будет работать схема.

> **Подсказка**
>
> Все комбинационные блоки (always_comb) в проекте должны использовать блокирующие присваивания. Все последовательностные блоки (always @(posedge) или always_ ff) должны использовать неблокирующие присваивания. Несоблюдение этого правила может привести к несоответствию результатов моделирования/синтеза.

Проект для демонстрации работы блокирующих и неблокирующих присваиваний можно найти по ссылке `CH3/blocking/build/blocking.xpr`.

Регистры в Artix 7

В предыдущем примере рассмотрена простейшая версия регистра. Такие регистры прекрасно отображаются в Artix 7. Но регистры Artix 7 предлагают гораздо больше функциональных возможностей, которые будут рассмотрены в данной книге.

Для каждой LUT существует два FF, один целевой FF и один, который может быть сконфигурирован как FF или защелка. Как показано в главе 2 «Комбинационная логика», защелки в лучшем случае ненадежны, поэтому они не рассматриваются в данной книге. Кроме того, если выбраны защелки, остальные четыре FF не могут быть использованы, что еще больше ограничивает количество доступных ресурсов. Ниже приведен пример регистров **блока комбинационной логики (CLB):**

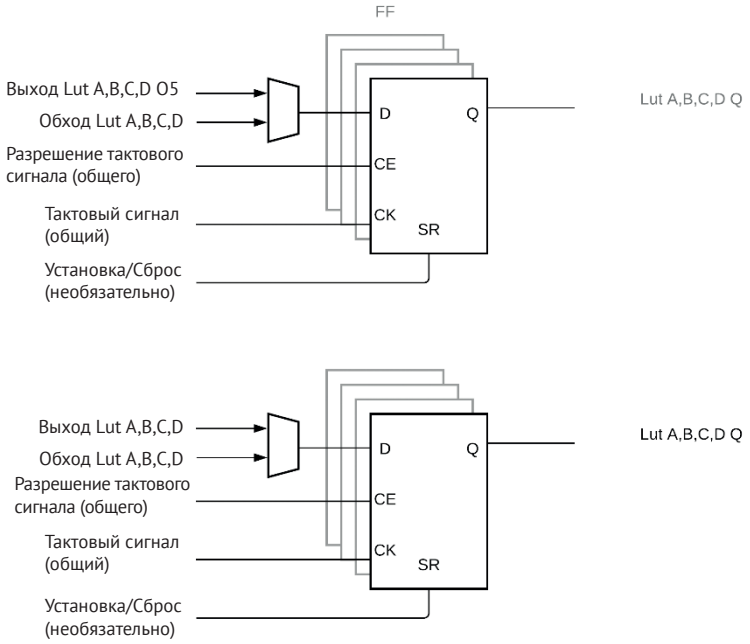


Рис. 3.9. Регистры CLB в Artix 7

Для каждой группы FF теперь есть общий сигнал разрешения тактового сигнала, общий тактовый сигнал и общая линия установки или сброса. Все входы D выбираются индивидуально из **таблиц поиска (LUT)**, или же LUT могут быть обойдены.

Как удерживать состояние схемы с помощью входа разрешения тактового сигнала

Простые DFF, которые были рассмотрены ранее, изменяют выход при каждом изменении входа. Разрешение тактовой частоты (`CE`) позволяет FF сохранять свое значение, пока нет необходимости менять данные. Рассмотрим, как это можно использовать на практике:

```
module dff (input wire D, CK, CE, output logic Q);
   initial Q = 1;
   always @(posedge CK) if (CE) Q <= D;
endmodule
```

На вейвформе показано, как использование входа `CE` влияет на выход `Q` у FF:

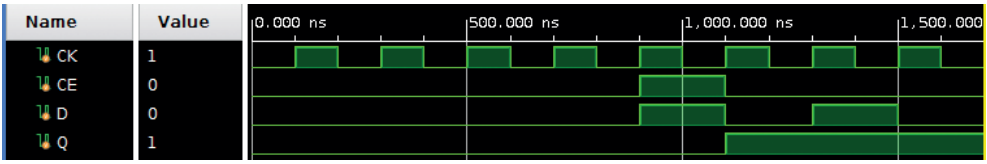


Рис. 3.10. FF с активацией тактовой частоты

Обратите внимание, когда D в первый раз переключается в 1 вместе с CE, Q также становится равен 1. Как только CE переключается в 0, значение Q остается равным 1 независимо от состояния входа D.

Сброс триггера

Ранее было показано, как можно использовать начальное значение для того, чтобы схема находилась в известном состоянии. Это отлично работает при начальной загрузке, но что, если нужно разработать схему, которая время от времени сбрасывается? Компания Xilinx разработала **программируемую пользователем вентильную матрицу (field-programmable gate array, FPGA)** Artix 7 с конфигурируемой системой установки/сброса. Вход **Установка/Сброс (Set/Reset, SR)** регистра LUT может быть настроен на установку или сброс и как синхронный или асинхронный.

Первый выбор следует сделать относительно асинхронности или синхронности установки/сброса FF. Синхронность означает, что сигнал сброса создается генератором тактового сигнала, управляющим FF, поэтому сброс будет должным образом синхронизирован в проекте. Ограничение заключается в том, что сброс произойдет только по приходе фронта тактового сигнала. Если наличие тактового сигнала в схеме не гарантировано, то необходимо использовать асинхронный сброс. Асинхронный сброс должен быть спроектирован таким образом, чтобы он срабатывал асинхронно и не конфликтовал с входом тактового сигнала. Это устраняет потенциальные проблемы с синхронизацией в проекте.

Xilinx рекомендует ограничить сброс только основными сигналами, чтобы ускорить анализ временных характеристик и сэкономить ресурсы маршрутизации за счет уменьшения количества цепей с ветвлениями.

CH3/simple_ _async/hdl/simple_ _async.sv, ASYNC = "TRUE"

```
always @(posedge CK, posedge SR) begin
   if (SR) Q <= '0;
   else if (CE) Q <= D;
end
```

В коде выше приведен пример FF с асинхронным сбросом. Это сброс, потому что Q получает значение '0, когда SR становится 1. Это была бы установка, если бы значение изменилось на'1. Из вейвформы следует, что сброс немедленно влияет на выход Q.

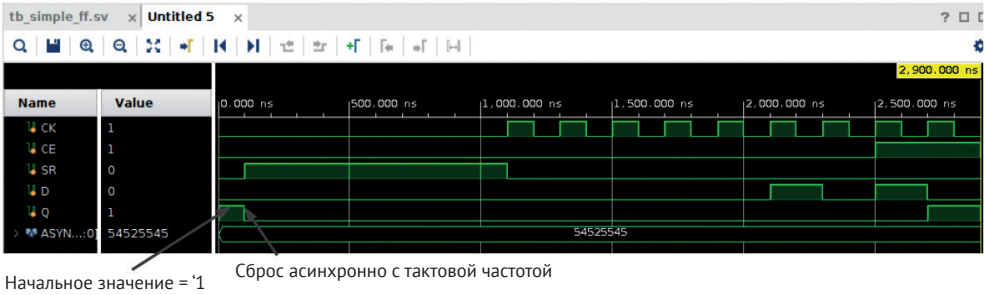


Рис. 3.11. Моделирование асинхронного сброса

Ниже показан пример FF с синхронным сбросом:

CH3/simple_ _async/hdl/simple_ _async.sv, ASYNC != "TRUE"

```
always_ff @(posedge CK) begin
   if (SR) Q <= '0;
   else if (CE) Q <= D;
end
```

Если проанализировать асинхронную версию триггера, то можно увидеть, что в списке чувствительности присутствует только `posedge SR`. Это говорит о том, что триггер среагирует на сигнал сброса только по приходе фронта тактового сигнала. Из вейвформы следует, что, в отличие от асинхронного, синхронный сброс не действует до тех пор, пока не начнет поступать тактовый сигнал:

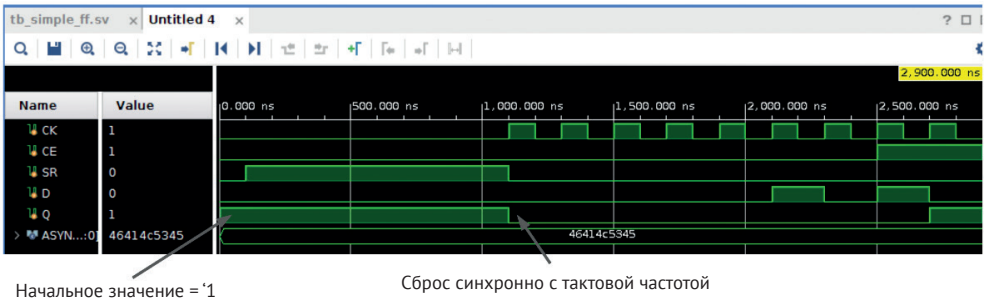


Рис. 3.12. Моделирование синхронного сброса

> **Подсказка**
>
> При необходимости сброса рекомендуется использовать синхронный сброс, за исключением случаев, когда нет гарантии что будет работать генератор тактового сигнала. Ограничьте количество сигналов, подлежащих сбросу, и Vivado позаботится об анализе синхронизации.

Теперь, когда основы работы с регистрами изучены, давайте займемся проектом этой главы.

Проект 2. Подсчет нажатий на кнопку

Основная задача разрабатываемой схемы в данной главе – это подсчитывать нажатия на кнопку и отображать их количество в читабельной форме с помощью семисегментного индикатора.

Семисегментный индикатор

В предыдущих главах двоичные числа отображались с помощью светодиодов на плате. Возможно, вы задавались вопросом, почему не использовался ряд несветящихся восьмерок. Причина в том, что с семисегментным индикатором связана синхронизация, для выполнения которой нужны регистры.

Рассмотрим, как подсвечивать семь сегментов. На следующей схеме показано, какой сегмент управляется тем или иным катодом.

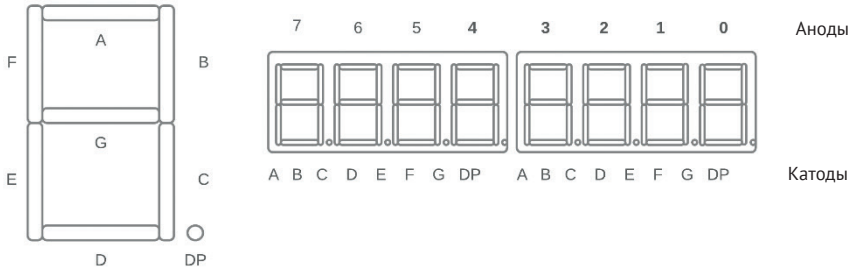


Рис. 3.13. Семисегментный индикатор

Если посмотреть на схему выше, то можно увидеть восемь сигналов, которые определяют, горит ли определенный светодиод или нет. Чтобы составить изображение, нужно придумать модуль, принимающий **двоично-десятичное (binary coded decimal, BCD)** или шестнадцатеричное число и преобразует его в формат, который может обрабатывать индикатор. Существует несколько вариантов, как это реализовать. Можно создать один конвертер, который будет работать со всем индикатором, или сделать по одному конвертеру для каждой цифры. В любом случае нужен проект.

> **Важное замечание**
>
> До сих пор использовались двоичные числа, отображая по одному биту на светодиод. Шестнадцатеричные числа представляют собой 4-битные двоичные числа от 4'b0000 = 4'h0 до 4'b1111 = 4'hF. Числа BCD – это способ представления десятичных чисел в компьютерной памяти с помощью значений от 4'b0000 = 4'd0 до 4'b1001 = 4'd9; значения выше 9 не используются.

Независимо от того, нужно ли отобразить число в формате BCD или в шестнадцатеричной системе счисления, можно создать модуль, принимающий 4-битное число и кодирующий биты, которые нужно отобразить, в данном случае на 8-битной шине катодов. **Десятичная (разрядная) точка (decimal point, DP)** является отдельным сигналом, чтобы она согласовывалась с другими данными.

CH3/counting_buttons/hdl/cathode_top.sv

```
always_ff @(posedge clk) begin
   cathode[7] <= digit_point;
   case (encoded)
     4'h0: cathode[6:0] <= 7'b1000000;
     4'h1: cathode[6:0] <= 7'b1111001;
     ...
     4'hE: cathode[6:0] <= 7'b0000110;
     4'hF: cathode[6:0] <= 7'b0001110;
   endcase
end
```

Если посмотреть на рис. 3.13, то можно увидеть, как происходит отображение на семисегментном индикаторе. Каждый сегмент загорается при подаче на него 0. Это называется активным низким сигналом. Из кода в `cathode_top` следует, например, что при `encoded`, равном 4'h0, на сегменты A-F поступает сигнал низкого уровня, поэтому они будут светиться, а на сегмент G поступает сигнал высокого уровня, поэтому он будет выключен. Если посмотреть на рис. 3.13, то можно увидеть, что это приведет к отображению 0.

Рассмотрим вейвформу, которую нужно реализовать для управления семисегментным индикатором. Обратите внимание, что версия Basys 3 имеет только 4x7 сегментов, поэтому потребуется сделать модуль кодирования параметризуемым.

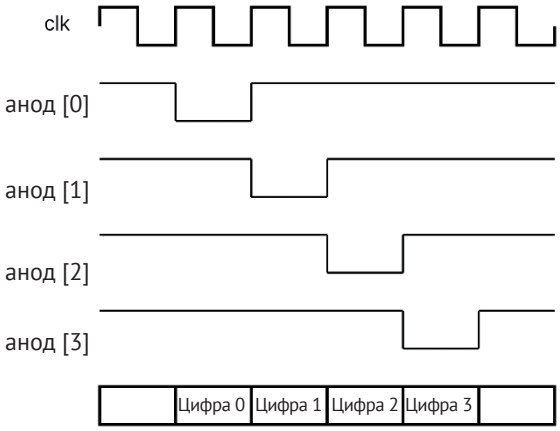


Рис. 3.14. Временная диаграмма семисегментного индикатора

Нужно генерировать анодный сигнал, подавая значение 0 на все позиции анодов[1] с частотой обновления 1/8 или 1/4, если используется плата Basys 3. Создадим два счетчика:

CH3/counting_buttons/hdl/seven_segment.sv

```
localparam INTERVAL = int'(1000000000 / (CLK_PER *REFR_RATE));
```

Сначала создадим локальный параметр `INTERVAL`, используемый для определения интервала времени, через который будут циклически перебираться аноды. Обратите внимание, что можно вычислять параметры.

В данном случае берется $1*10^9$ нс в секунде и делится на период тактовой частоты (тактовая частота 100 МГц имеет период 10 нс, тактовый сигнал подается непосредственно на плату Nexys-A7-100T A7 100T), умноженный на частоту обновления. Поскольку результатом является значение с плавающей точкой, его надо привести к целому значению с помощью операции приведения типа `int'()`.

[1] Если рассматривать семисегментные индикаторы без обвязки, то на общие аноды необходимо подавать высокий уровень (при этом катоды-сегменты для их зажигания подключаются к низкому уровню). Но цепи управления в данном случае устроены так, что на них подается инвертированный сигнал. – *Прим. ред.*

```
initial begin
   refresh_count = '0;
   anode_count   = '0;
end
always @(posedge clk) begin
   if (refresh_count == INTERVAL) begin
     refresh_count <= '0;
      anode_count   <= anode_count + 1'b1;
   end else refresh_count <= refresh_count + 1'b1;
   anode <= '1;
   anode[anode_count] <= '0;
   cathode            <= segments[anode_count];
end
```

В соответствии с приведенным кодом генерируются два счетчика, первый из которых – счетчик обновления (`refresh_count`), определяющий время, когда будет поступать сигнал на каждый анод. INTERVAL для этого счетчика устанавливается на основе частоты обновления. Когда счетчик обновления достигает значения INTERVAL, его надо сбросить и увеличить на единицу второй счетчик, `anode_count`. Он определяет, на какой анод будет подан сигнал (сигнал активного низкого уровня, т. е. 0) для обновления семисегментного индикатора. Счетчик `anode_count` не имеет ограничений, потому что он будет считать либо до 4, либо до 8, так что ему можно позволить счет без ограничений. Если бы количество секций в индикаторе не являлось степенью 2, надо было бы ограничить счетчик так же, как и `refresh_count`.

Обнаружение нажатия на кнопку

Кнопки на плате подключены так, чтобы при нажатии выдавать 1, т. е. обычно они передают в FPGA значение 0[1]. Это означает, что для обнаружения нажатия кнопки нужно детектировать положительный фронт импульса.

Анализ синхронизации

Рассмотрим взаимосвязь между тактовыми импульсами и тем, как анализируются временны́е параметры схемы. Существует два основных ограничения синхронизации между данными и тактовыми импульсами. Эти ограничения должны быть соблюдены для обеспечения надежной работы проекта.

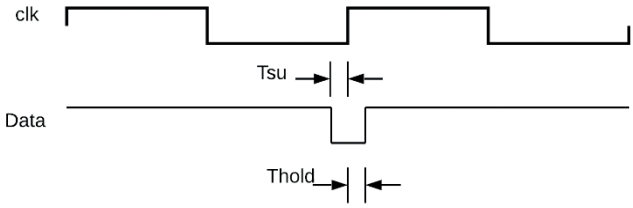


Рис. 3.15. Ограничения синхронизации

[1] В зависимости от платы поведение кнопки может отличаться. Довольно распространенной является схема подключения кнопок с противоположным поведением, когда к ним подключено питание, и в не нажатом состоянии они передают логическую единицу. При нажатии кнопка замыкает цель на землю и передает логический 0. Чтобы понимать, как работают кнопочные переключатели на конкретной плате, рекомендуется изучить схему платы в мануале.

Первое ограничение – это время установки (setup) сигнала, или T_{su} на диаграмме временны́х ограничений. Это период времени, в течение которого сигнал должен быть стабильным перед фронтом тактового сигнала. Если сигнал меняется в пределах или после **Tsu**, то устройство может работать неправильно. В синхронной схеме **Tsu** обычно нарушается только в том случае, если частота тактового сигнала слишком высока для того, чтобы самые длинные тактовые импульсы синхронизировали каналы передачи данных в проекте. Время установки в синхронной цифровой схеме можно исправить, снизив тактовую частоту или изменив ее так, чтобы уменьшить длительность временных интервалов.

Второе ограничение, время удержания (hold), или T_{hold}, – период времени, во время которого сигнал должен оставаться стабильным после прихода импульса тактового сигнала. Обычно это не является проблемой в устройствах с одной **большой областью с логическими ресурсами (Super Logic Region, SLR[1]),** таких как то, которое используется для апробации примеров в данной книге. Но это становится проблемой, когда в схеме имеется несколько источников тактовых сигналов и не соблюдается надлежащая синхронизация между ними. Проблема с временем удержания заключается в том, что ее невозможно устранить путем снижения тактовой частоты, ее можно только предотвратить. Когда такая проблема возникает в синхронной схеме, причиной обычно является локальная маршрутизация линий передачи тактовых сигналов в схемах с большой перегруженностью маршрутов.

> **Важное замечание**
>
> Компания Xilinx создает очень большие устройства, состоящие из нескольких матриц FPGA, соединенных с подложкой. Каждая из таких матриц называется SLR. Это позволяет создавать устройства с высокой плотностью элементов, которые по количеству логических элементов могут конкурировать с ASIC.

Когда же имеется абсолютно асинхронный сигнал, такой как BTNC[2] от кнопки без надлежащей синхронизации, невозможно гарантировать удовлетворение требований соблюдения времени установки и удержания.

Проблемы, возникающие из-за асинхронных сигналов

Представьте как случайный асинхронный сигнал, такой как `BTNC`, может привести к нарушению времени установления и удержания. Чтобы продемонстрировать проблему, было добавлено временное ограничение `create_clock` на вход `BTNC`, которое можно раскомментировать и запустить. Частота тактового сигнала не имеет значения, просто ограничение создано так, чтобы фронты изменялись в зависимости от их взаимодействия с `clk`.

---

[1] Super Logic Region, SLR, – область FPGA, в которой все логические элементы размещены на одном кристалле чипа. Если это не так (пример – FPGA Virtex-7 2000T, состоящий из четырех отдельных кремниевых кристаллов в одном чипе), то могут возникнуть проблемы с цепями, пересекающими границы SLR (см. также замечание автора далее). – *Прим. ред.*

[2] BTNC расшифровывается как *button connected*. – *Прим. ред.*

CH3/counting_buttons/build/Nexys-A7-100T-Master.xdc

```
create_clock -add -name BTNC -period 99.99 [get_ports {BTNC}];
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets BTNC_IBUF]
```

Также нужно установить атрибут `CLOCK_DEDICATED_ROUTE` на `FALSE`, поскольку вывод `BTNC` для центральной кнопки не был установлен в качестве выделенного тактового вывода. Обычно требуется, чтобы внешний тактовый сигнал на FPGA поступал на вывод, который подключается к выделенным ресурсам синхронизации FPGA. `CLOCK_DEDICATED_ROUTING` снимает это ограничение, так что становится возможным для маршрутизации тактовых сигналов использовать внутренние ресурсы чипа. Правда, может появиться расфазировка тактовых сигналов, что может стать проблемой для связанных тактовых сигналов, но в данном случае этого не произойдет. Давайте посмотрим на отчет о временных параметрах схемы после компиляции.

Использование асинхронного сигнала напрямую

Установите `ASYNC_BUTTON` в подсчитываемых кнопках на `NOCLOCK`. Оставьте тактовые ограничения, чтобы представить асинхронный сигнал, поступающий на вход. BTNC будет использоваться как вход непосредственно в регистр, синхронизируемый по `clk`.

После сборки проекта, если проанализировать взаимодействие тактовых импульсов, можно увидеть, что путь BTNC к `clk` по-прежнему небезопасен.

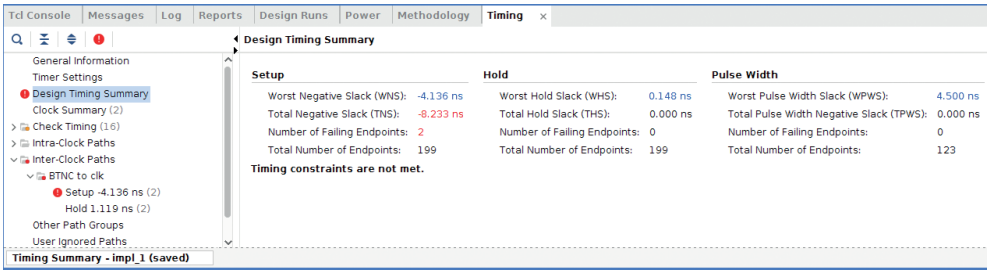


Рис. 3.16. Асинхронный вход BTNC

Также можно увидеть, что требования синхронизации (timing requirement) равны 0 пс.

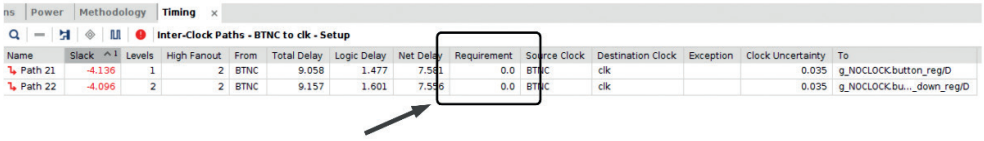


Рис. 3.17. Требования синхронизации

Это признак того, что сигналы исходят от асинхронных тактовых генераторов. Анализ синхронизации работает следующим образом: когда имеются тактовые сигналы с разной частотой, инструмент синтеза сравнивает их фронты друг с другом, чтобы найти наихудшее выравнивание для синхронизации.

В данном случае Vivado нашел выравнивание, при котором сигнал должен удовлетворять требованию в 0 нс.

Также можно вызвать окно взаимодействия тактовых сигналов (clock interaction):

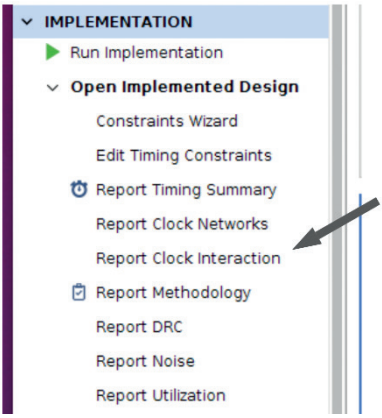


Рис. 3.18. Взаимодействие тактовых сигналов

Панель взаимодействия тактовых сигналов показывает, что пути от BTNC до `clk` синхронизированы, но они небезопасны.

Если попробовать сделать это на плате, то можно будет наблюдать, что значение на семисегментном индикаторе будет увеличиваться на единицу не при каждом нажатии центральной кнопки. Счет ведется в шестнадцатеричном формате. Схему можно сбросить с помощью красной кнопки сброса на плате.

Проблема с нажатием кнопок

Способ, с помощью которого можно заставить схему работать, заключается в правильной синхронизации сигнала BTNC. Поскольку сигнал BTNC нажимается во временном масштабе человека, можно считать его статическим сигналом, за исключением момента, когда кнопка нажата и отпущена. При работе с подобным сигналом можно использовать двухступенчатый синхронизатор и дополнительный триггер FF для определения фронта.

На рис. 3.19 показано, что на тактовом цикле 0 возможно нарушение синхронизации на триггере FF0. На диаграмме синхронизации это обозначено буквой X, но на самом деле это происходит, когда FF0 переходит в метастабильное состояние. Метастабильность означает, что выход Q триггера находится в неопределенном состоянии из-за нарушения времени установки или удержания. То есть триггер FF1 распознает состояние как 0 или 1, но не гарантирует, какое именно. Вот почему триггер FF0 управляет одним и только одним триггером. Если бы он управлял двумя или более, существовала бы вероятность, что разные тригеры получили бы разные значения. К тому времени, когда FF1 выведет свое значение Q, его можно будет использовать в тактовом домене `clk`, но поскольку нет возможности использовать FF0 и нужно обнаружить фронт, то нужен FF2, чтобы удерживать старое значение BTNC:

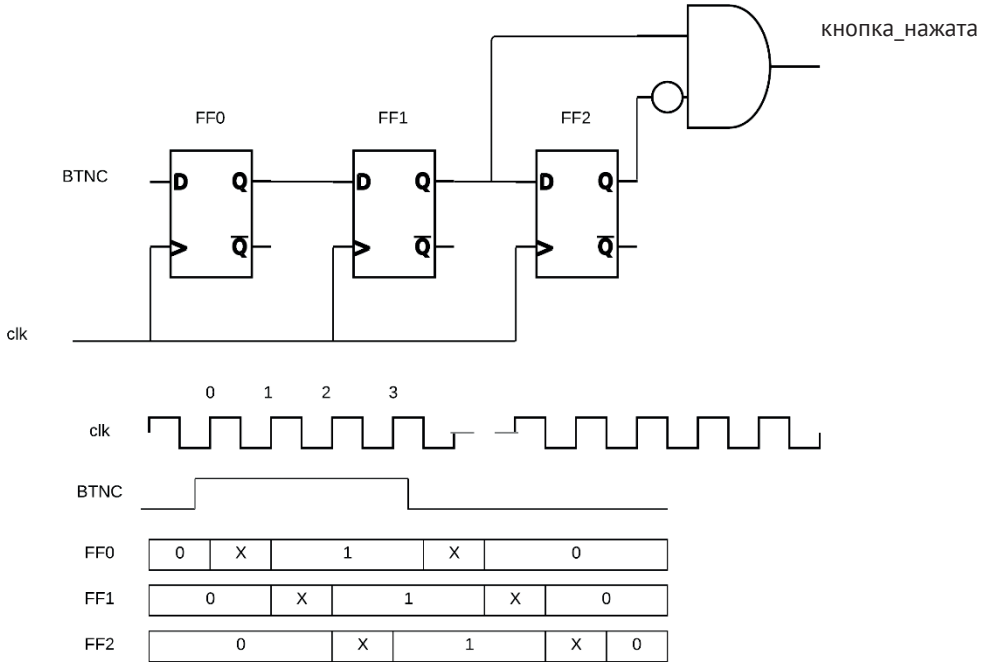


Рис. 3.19. Синхронизатор с нажатой кнопкой

```
logic [2:0] button_sync;
always @(posedge clk) begin
   button_sync <= button_sync << 1 | BTNC;
  if (button_sync[2:1] == 2'b01) button_down <= '1;
   else button_down <= '0;
end
```

Напомним, что << – это оператор сдвига влево, поэтому нет возможности рассматривать `button_sync <= button_ sync << 1 | BTNC` как регистр сдвига, где бит 0 – это асинхронный FF, поэтому он не используется в операции сравнения.

> **Важное замечание**
>
> Метастабильность относится к статистике и анализу среднего времени между отказами. Для FF1 или FF2 все еще возможно распространение метастабильности FF0, но это статистически маловероятно. Явление метастабильности часто является предметом вопросов на собеседованиях.

Остается еще одна проблема, которую необходимо решить.

Разработка безопасной реализации

Закомментируйте ограничения BTNC, добавленные в предыдущем разделе, и установите `ASYNC_BUTTON = "SAFE"`. После запуска проекта откройте отчет о временных параметрах схемы.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.498 ns	Worst Hold Slack (WHS):	0.169 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	168	Total Number of Endpoints:	168	Total Number of Endpoints:	125

All user specified timing constraints are met.

Рис. 3.20. Временны́е соотношения правильно синхронизированного проекта

Если посмотреть на панель взаимодействия тактовых сигналов, то можно увидеть, что все безопасно. Теперь попробуем запустить проект на плате.

Можно заметить, что иногда все еще наблюдаются некоторые странности при подсчете. Не учтена еще одна вещь. Нажатия осуществляются на механические переключатели, а они страдают от явления, называемого дребезгом.

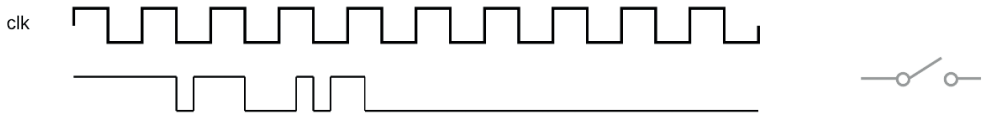


Рис. 3.21. Переключатель без противодребезговой защиты

На рис. 3.19 показано изображение электромеханического переключателя и то, как он колеблется в течение некоторого периода времени, прежде чем успокоиться. Требуется создать схему, которая будет ждать в течение определенного периода времени после того, как обнаружит нажатие на переключатель, сбрасывая саму себя, если обнаружит дребезг.

```
always @(posedge clk) begin
   button_down <= '0;
   button_sync <= button_sync << 1 | BTNC;
   if (button_sync[2:1] == 2'b01) counter_en <= '1;
   else if (~button_sync[1])      counter_en <= '0;
   if (counter_en) begin
      counter <= counter + 1'b1;
      if (&counter) begin
         counter_en <= '0;
         counter    <= '0;
         button_down <= '1;
      end
   end
end
```

Приведенная схема запускает таймер при обнаружении нажатия кнопки, т. е. по нисходящему фронту. Затем она ждет 256 тактовых циклов, чтобы убедиться, что переключатель не дребезжит, и ждет следующего фронта импульса, сбрасывая счетчик, если дребезг обнаружен.

Попробуйте запустить проект с антидребезговой защитой на плате, и вы больше не увидите никаких странностей с подсчетом нажатий кнопки.

Окончательный результат синтеза проекта: 68 LUT, 133 FF и **Worst Negative Slack**[1] **(WNS)** 6,732 пс. Существует много возможностей для увеличения частоты тактового сигнала. Изначально планировалось использовать тактовую частоту 100 МГц (с периодом 10 нс). При WNS, равном 6,732, можно уменьшить период тактовой частоты до 10 – 6,732 нс, что составит примерно 300 МГц.

Для анализа временны́х соотношений в проекте существует еще несколько метрик, помимо ограничений:

- **WNS:** самая длинная задержка пути в проекте в наносекундах. Если схема нарушает временны́е соотношения, этот показатель будет отрицательным;
- **Total Negative Slack, TNS:** сумма всех нарушенных задержек прохождения сигналов в проекте (0, если WNS положительный);
- **Worst Hold Slack, WHS:** должно быть положительным или равным 0 для работающей схемы;
- **Total Hold Slack, THS:** должно быть положительным или равным 0 для работающей схемы.

Рассмотрим теперь, как можно перейти к десятичному представлению цифр на семисегментном индикаторе.

Переход на десятичное представление

До сих пор счетчик считал в шестнадцатеричном формате. Но большинству людей удобнее работать с десятичными числами. Как это обычно бывает, существует несколько способов решения задачи. Можно считать в двоичной системе и переводить в десятичную или просто считать в десятичной.

Для этого проекта будем считать в десятичной системе счисления. Если представлять десятичную систему счисления на восьми семисегментных индикаторах, то максимальное число, которое можно представить, будет 99 999 999. Если считать в двоичном формате и конвертировать в десятичный, то это составит число 2^{32} = 4 294 967 296, которое не поместится на индикаторе (впрочем, вы бы вряд ли провели остаток жизни, нажимая на кнопки, чтобы насчитать так много).

```
// Decimal increment function
function [NUM_SEGMENTS-1:0][3:0] dec_inc;
   input [NUM_SEGMENTS-1:0][3:0] din;
   bit [3:0]                     next_val;
   bit                           carry_in;
   carry_in = '1;
   for (int i = 0; i < NUM_SEGMENTS; i++) begin
     next_val = din[i] + carry_in;
     if (next_val > 9) begin
```

[1] Slack (букв. провисание) – в данном случае временной параметр цепи, характеризующий разность между временем, необходимым для предустановки/удержания сигнала на входе элемента, и временем, которое реально для этого имеется. Worst Negative Slack – наихудшее значение этого параметра, чем оно ближе к нулю, тем хуже. При отрицательном значении WNS схема неработоспособна (см. также далее по тексту). – *Прим. ред.*

```
            dec_inc[i] = '0;
            carry_in   = '1;
         end else begin
            dec_inc[i] = next_val;
            carry_in   = '0;
         end
      end // for (int i = 0; i < NUM_SEGMENTS; i++)
   endfunction // dec_inc
```

Функция выше принимает данные в **двоично-десятичной кодировке (BCD).** Цикл `for` увеличивает каждый разряд, считая до тех пор, пока не достигнет 10, в этом случае он сбрасывает этот разряд до 0 и передает перенос в следующий разряд.

Измените режим с HEX на DEC и выполните сборку.

Окончательный результат синтеза проекта: 97 LUT, 133 FF и WNS 1,490. За более читабельный дисплей приходится платить: примерно на 50 % большие затраты LUT и около 5 нс задержки, но при этом временные параметры по прежнему легко укладываются в синхронизацию на частоте 100 МГц.

Знакомство с ILA

Компания Xilinx предоставляет в составе Vivado очень мощное средство для отладки, которое называется **интегрированным логическим анализатором (Integrated Logic Analyzer, ILA).** ILA дает возможность добавить логический анализатор, который можно вставить в проект. Самый простой способ использования ILA – начать с маркировки сигналов для отладки (debug). Сделать это можно, добавив атрибут `mark_debug` следующим образом:

```
(* mark_debug = "true" *) logic button_down;
(* ASYNC_REG = "TRUE", mark_debug = "true" *) logic [2:0]
button_sync;
```

Чтобы применить атрибуты к сигналам, используют стиль комментариев SystemVerilog `(*  *)`. На примере `button_sync` показано, как можно применить несколько атрибутов.

Обе схемы `SAFE` и `DEBOUNCE` имеют уже установленный `mark_debug`. Выберите `SAFE` и выполните следующие шаги по настройке и запуску ILA.

Маркировка сигналов для отладки

Если открыть файл на `CH3/counting_buttons/hdl/counting_buttons.sv`, то там будет несколько сигналов, помеченных для отладки. Не стоит помечать все сразу, но чем больше будет добавлено представляющих интерес сигналов, тем больше шансов найти их после синтеза.

Далее выполните следующие действия.

1. Выберите **Run Synthesis** и **Open Synthesized Design**. Обычно этот шаг совмещен с генерацией загрузочного файла, но в данном случае нужно запустить синтез отдельно, чтобы можно было настроить отладку.
2. Выберите **Set Up Debug**…:

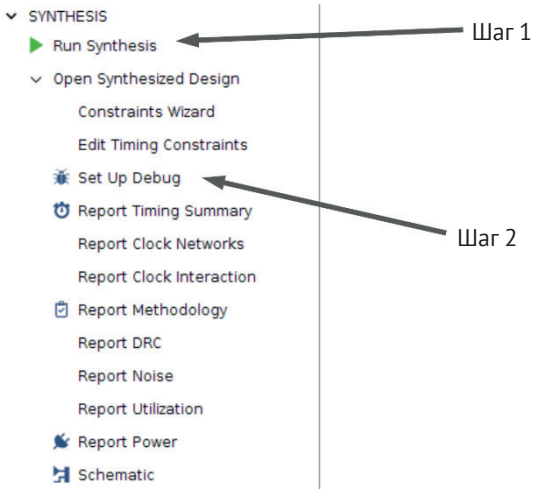


Рис. 3.22. Настройка ILA

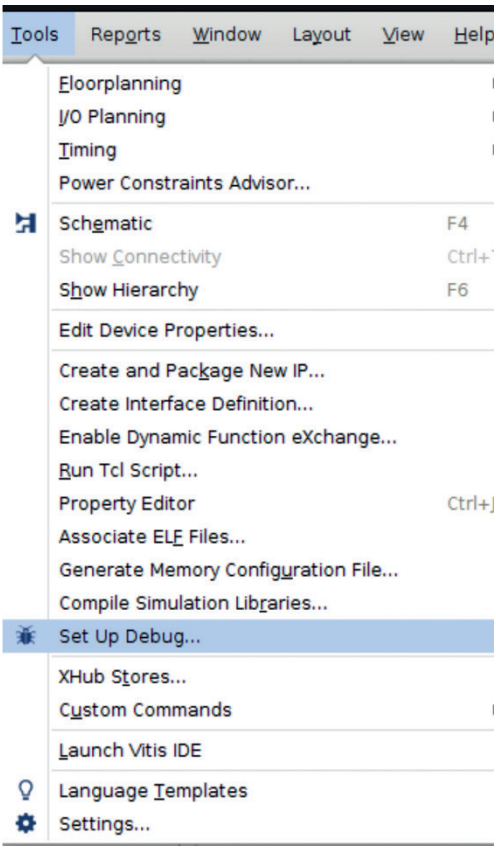


Рис. 3.23. Настройка этапа отладки

3. Появится окно. Выберите **Next** (далее).
4. Выберите **Continue Debugging** (продолжить отладку, так необходимые сигналы уже добавлены через `mark_debug`). Добавьте регистры синхронизации и выход нажатой кнопки. Если посмотреть на версию DEBOUNCE, то в ней еще есть счетчик.
5. Выберите кнопку **Find Nets to Add**... (найти цепи для добавления). Нажмите **OK**. В результате будут показаны все цепи, которые можно проанализировать в проекте:

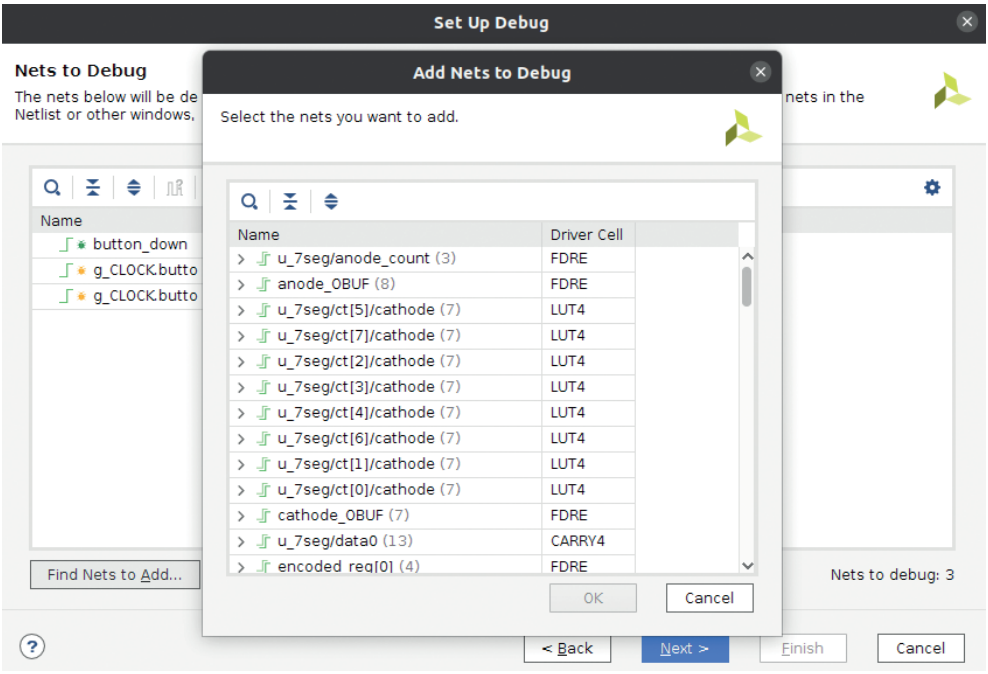


Рис. 3.24. Добавление цепей для отладки с помощью ILA

6. Добавлено достаточно сигналов для анализа, поэтому просто нажмите **Cancel** (отмена).

Не стесняйтесь добавить еще несколько сигналов, если хотите, или поэкспериментируйте позже. Следует помнить, что для внутреннего хранения временных диаграмм (waves) в FPGA имеется только блочная оперативная память, поэтому чем больше сигналов будет добавлено, тем меньше глубина дискретизации будет. Используемые FPGA обладают большим количеством ресурсов, поэтому сейчас беспокоиться не о чем.

7. Вернувшись на экран отладки настроек, нажмите кнопку **Next**.

Оставьте основные параметры ILA по умолчанию. Глубина дискретизации равна 1024, на данный момент этого достаточно. Используемая частота тактового сигнала достаточно низкая, поэтому стадии конвейера не нужны. ILA имеет некоторые расширенные возможности отладки, но в данной книге они не рассматриваются.

Программирование устройства

Как обычно, нужно собрать проект, чтобы можно было увидеть результаты его выполнения. Выберите **Finish** и сгенерируйте загрузочный файл.

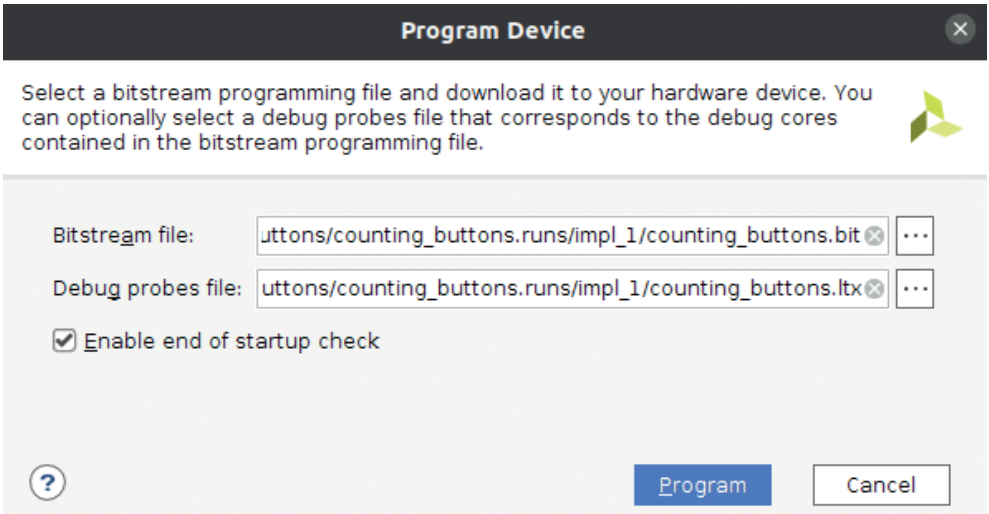


Рис. 3.25. Программирование устройства

Можно заметить, что отладочные пробы (debug probes) теперь настроены. Выберите **Program**.

В результате этих действий будет открыт просмотр ILA:

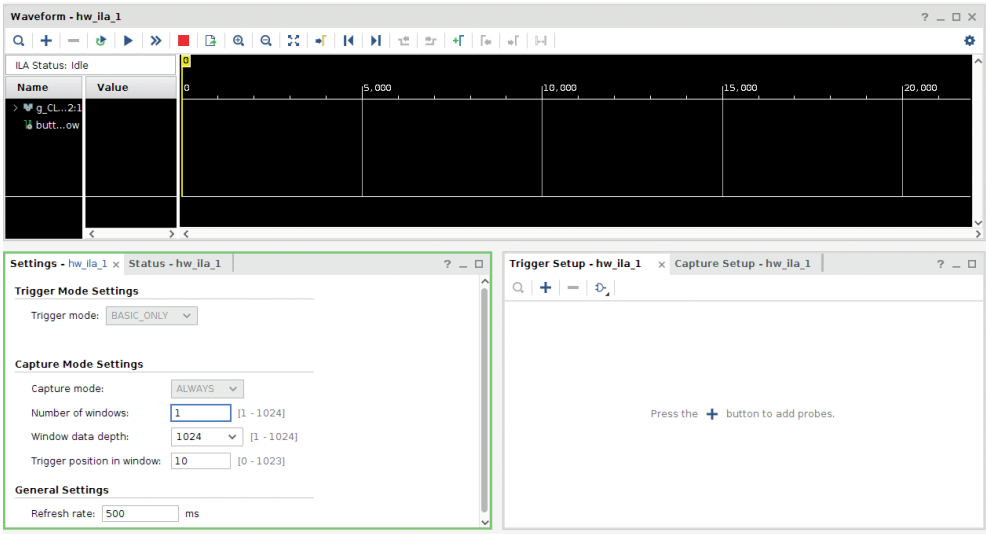


Рис. 3.26. Вид ILA

Панель настроек **Settings** позволяет ограничить глубину данных. Иногда нужно выбрать большое количество отсчетов для захвата, но, если требуется

ограничить количество отсчетов для конкретного запуска, это можно сделать с помощью этой панели. Пункт **Trigger position** (Состояние триггера) устанавливает, сколько отсчетов будет захвачено до триггера. В данном случае установлено 10 отсчетов до триггера, а остальные – после.

В настройках триггера надо добавить условие, на которое будет срабатывать триггер.

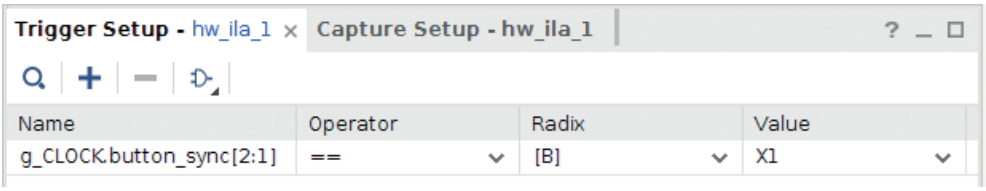


Рис. 3.27. Настройка триггера

Выбрано срабатывание триггера всякий раз, когда бит под номером 1 в переменной `button_sync` становится равным 1.

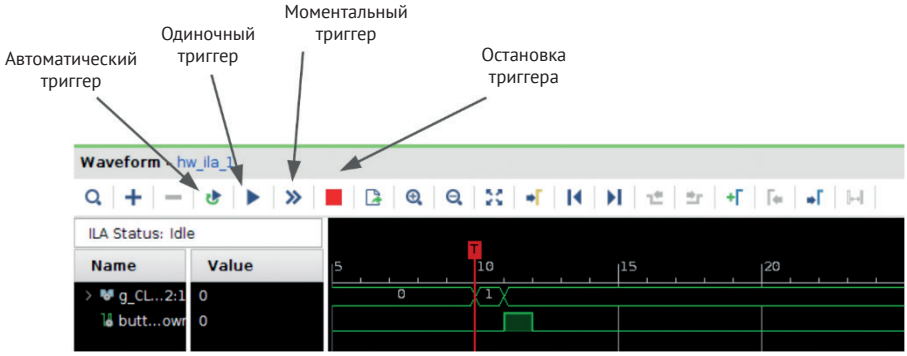


Рис. 3.28. Запуск триггера

Существует несколько вариантов захвата сигналов в ILA. **Моментальный триггер (Immediate Trigger, >>)** будет фиксировать сигналы в их текущем состоянии. **Автоматический триггер** (**Auto Trigger**, стрелка вправо с зеленым кругом) автоматически переключит триггер после запуска. **Одиночный триггер** (**Single Trigger**) нужен для захвата одного события. Наконец, если не получается достичь условия срабатывания триггера, можно остановить его и установить другой с помощью кнопки **Остановка триггера (Stop Trigger).**

Попробуйте использовать **Single Trigger**, затем нажмите центральную кнопку, и вы увидите то, что показано на рис. 3.28. В данном примере не удалось зафиксировать аномалию, но это, по крайней мере, показывает, как использовать ILA для отладки возможных проблем проекта непосредственно на плате.

Что насчет симуляции?

В этой главе основное внимание уделено возможностям лабораторной отладки, а не запуску моделирования. Это делается не всегда. Но создание модели

для тестирования семисегментного индикатора, а также моделирование дребезга кнопки заняло бы слишком много времени и сил. В конце концов, это FPGA, и можно программировать и выполнять отладку проекта столько раз, сколько потребуется. Это не самый распространенный путь, но в данном проекте он сработал хорошо.

Подробное изучение синхронизации

В проекте 2 используется тактирование схемы от внешнего источника.

В последующих проектах будет осуществляться взаимодействие между несколькими тактовыми доменами. Например, в главе 5 «Ресурсы FPGA, и как их использовать», будет осуществляться взаимодействие между основной логикой и контроллером памяти DDR, работающим в другом тактовом домене.

Зачем использовать несколько тактовых сигналов?

При проектировании на FPGA обычно нужно учитывать несколько факторов синхронизации. Иногда необходимо использовать определенный тактовый сигнал для интерфейса. Например, если ведется разработка схемы, которая взаимодействует с Ethernet 10 Гбит, то где-то в проекте нужно использовать тактовые сигналы с частотой, кратной 156,25 или 322,27 МГц, в зависимости от того, происходит ли взаимодействие с уровнем PCS или PMA[1]. Это связано с тем, что данные должны передаваться на этой частоте и поступать на этой частоте.

В других случаях может потребоваться высокая производительность или низкое энергопотребление. Увеличение тактовой частоты может повысить пропускную способность или увеличить количество вычислений в секунду, если данные нужны быстро или выполняется много операций. Более высокая тактовая частота требует больших затрат энергии. Если ведется разработка схемы, которая должна работать в условиях пониженного энергопотребления, то все равно может потребоваться получать данные с большей скоростью, но при этом можно будет экономить энергию за счет более медленного выполнения операций.

Двухступенчатый синхронизатор

Основным элементом для реализации синхронизации является двухступенчатый синхронизатор. Определяя две ступени в тактовом домене, можно применить атрибут ASYNC_REG к FF. Этот атрибут сообщает Vivado, что данные регистры используются для синхронизации и должны быть расположены как можно ближе друг к другу. Простой двухступенчатый синхронизатор выглядит следующим образом:

```
(* ASYNC_REG = "TRUE" *) logic [1:0] sync;
always @(posedge dst_clk) sync <= sync << 1 | async;
```

Этот код создает два триггера в домене `dst_clk` и указывает Vivado обрабатывать их как таковые.

---

[1] PCS и PMA – составляющие уровня сетевой модели Ethernet (стандарт 802.3), отвечающие за передачу/прием сигналов на физическом уровне. – *Прим. ред.*

Синхронизация управляющих сигналов

Часто в схеме могут использоваться сигналы состояния, имеющие более низкую частоту и поступающие из одного тактового домена в другой. Для таких сигналов следует выполнять квитирование (handshake) между обеими сторонами интерфейса либо убедиться, что изменения сигналов достаточно редкие, чтобы не беспокоиться о квитировании.

Для такого типа интерфейса можно использовать синхронизатор переключения (toggle synchronizer). Он устанавливает сигнал, пересекающий тактовые домены, а затем генерирует импульс, когда обнаруживается фронт сигнала в синхронизированном домене. Это нужно, чтобы убедиться, что переключения происходят медленнее, чем можно генерировать импульсы на принимающей стороне. Один из способов предотвратить такую ситуацию – послать аналогичный сигнал переключения обратно в качестве подтверждения:

```
logic async_toggle;
(* ASYNC_REG = "TRUE" *) logic [2:0] sync;
logic sync_pulse;
always @(posedge src_clk)
   if (ctrl_in) async_toggle <= ~async_toggle;
always @(posedge dst_clk) sync <= sync << 1 | async_toggle;
assign sync_pulse = ^sync[2:1];
```

Следующий рисунок иллюстрирует вейвформу синхронизатора переключения:

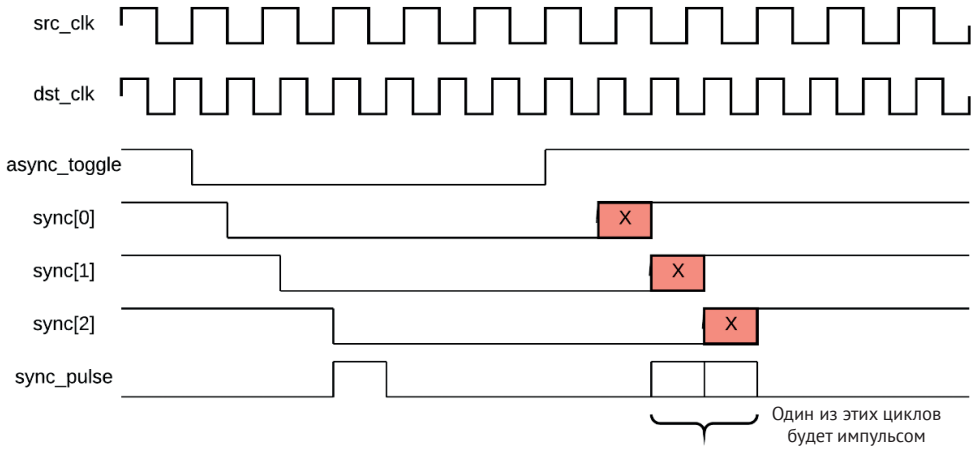


Рис. 3.29. Вейвформа синхронизатора переключения

На вейвформе показано, что восходящий фронт `async_toggle` может нарушить синхронизацию на `sync[0]`. Это показано символом X. Хотя `sync[0]` может стать метастабильным, `sync[1]` случайным образом зарегистрирует 0 или 1, и это безопасно перейдет в `sync[2]`. Но, поскольку это будет сделано случайно, неизвестно, какой из двух выделенных циклов сгенерирует импульс на `dst_clk`.

Передача данных

Если имеются данные, удовлетворяющие критериям предыдущего синхронизатора (т. е. данные будут стабильны в течение времени удержания в `dst_clk`), то можно передать их вместе с управляющим сигналом. Исходные данные захватываются и удерживаются в `src_clk` в течение времени, необходимого для синхронизации. Если тактовые соотношения известны, то это можно сделать, выждав достаточно длительный период времени, или можно передать сигнал подтверждения обратно из `dst_clk` при получении импульса синхронизации.

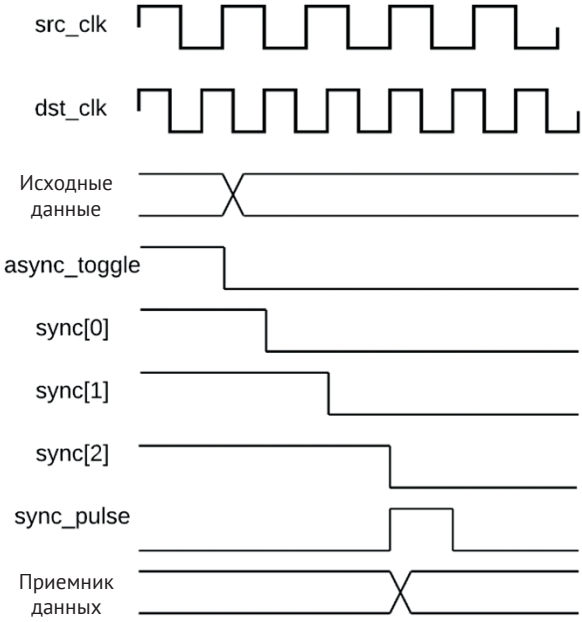


Рис. 3.30. Передача данных через тактовые домены

Это делается путем защелкивания сигнала в регистре в том же цикле, в котором происходит переключение сигнала при прохождении через синхронизатор.

```
logic async_toggle;
logic [31:0] async_data;
(* ASYNC_REG = "TRUE" *) logic [2:0] sync;
logic sync_pulse;
logic [31:0] sync_data;
always @(posedge src_clk)
   if (ctrl_in) begin
      async_toggle <= ~async_toggle;
     async_data   <= ...;
   end
always @(posedge dst_clk) begin
   sync <= sync << 1 | async_toggle;
   if (sync_pulse) sync_data <= async_data;
end
assign sync_pulse = ^sync[2:1];
```

c) FIFO или двухступенчатый синхронизатор с данными или без данных.
d) Синхронизаторы не нужны!

4. Когда следует использовать always @(posedge clk), а не always_ff @(posedge clk)?

   a) Когда лень печатать.
   b) Когда нужно использовать оператор initial для предварительной загрузки регистра.
   c) Когда нужно сбросить регистр синхронно или асинхронно.

5. Когда нужно добавить модуль для защиты от дребезга?

   a) Когда происходит пересечение тактовых доменов.
   b) Когда происходит пересылка данных из одного FF в другой.
   c) Когда нужно взаимодействовать с электромеханическими кнопками или переключателями.

Задание повышенной сложности

В главе 2 на основе *комбинационной логики* была создана схема, которая может выполнять некоторые простые операции и отображать данные в двоичном виде на светодиодах.

1. Измените код, чтобы использовать модуль семисегментного индикатора для вывода данных вместо светодиодов или в дополнение к ним. Выводите данные в шестнадцатеричном или десятичном формате. Если вы выберете десятичный формат, то нужно будет либо использовать функцию для сложения в десятичной системе счисления, либо выполнить преобразование перед выводом данных на семисегментный индикатор.
2. Запустите код на плате, чтобы проверить его работу.

Дополнительное чтение

Для получения дополнительной информации обратитесь к следующим источникам.

- Для получения дополнительной информации об ограничениях проектирования, которые могут быть применены в Vivado: https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0004-vivado-applying-design-constraints-hub.html.
- Дополнительная информация о свойствах, используемых в Vivado: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug912- vivado-properties.pdf.

Глава 4

Разработка калькулятора

В этой главе будут использованы полученные знания о комбинационной логике и последовательностных схемах в SystemVerilog для обсуждения проектирования конечных автоматов[1]. Будут рассмотрены классические конструкции конечных автоматов и разработан контроллер светофора – типовой проект по **электротехнике (Electrical Engineerin, EE)**.

Ранее был создан контроллер для семисегментного индикатора, который можно использовать для отображения числовых значений, и показано, как безопасно работать с кнопками и переключателями. Теперь воспользуемся этими знаниями: покажем, как разработать конечный автомат для выполнения вычислений, и разработаем первый действительно полезный проект – простой калькулятор, способный вводить два 16-битных числа, складывать, вычитать, умножать и делить их, выводя результат на семисегментный индикатор.

После завершения этой главы вы должны уметь конструировать простые конечные автоматы, использовать простые конечные автоматы для реализации алгоритмов и понимать основы компьютерной математики.

В этой главе будут рассмотрены следующие основные темы:

- представление основных типов конечных автоматов на HDL: Мили (Mealy) и Мура (Moore);
- реализация контроллера светофора с использованием конечного автомата;
- проектирование простого конечного автомата для калькулятора;
- проектирование и моделирование целочисленного делителя;
- запуск на плате.

---

[1] Конечный автомат (*finite-state* machine, или просто *state machine*) – устройство, состояние выхода которого в каждый момент времени зависит не только от состояния входов (как в комбинационной схеме или просто логическом элементе), но и от предыдущего/предыдущих состояний. Конечный автомат может описываться перечнем событий (состояний входов), состояний выхода/выходов и таблицы переходов между ними. Простейший конечный автомат – самоблокирующаяся кнопка (кнопка с фиксацией), связанная со светодиодом. Одно и то же воздействие на вход (нажатие кнопки) в зависимости от предыдущего состояния включает или выключает светодиод. Пример более сложного конечного автомата упоминает автор – это светофор. – *Прим. ред.*

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH4.

Реализация первого конечного автомата

В общем случае **конечный автомат** принимает на вход ряд событий и, основываясь на них, проходит через набор состояний, которые могут отображаться на одном или нескольких выходах. Конечный автомат может быть как простым, так и довольно сложным. В предыдущей главе была разработана простая схема для управления семисегментным индикатором. Контроллер семисегментного индикатора содержал два счетчика, которые циклически пропускали ноль через аноды и представляли комбинации состояний на входах катодов для каждого разряда. Можно было бы разработать для этого конечный автомат, но проще было сделать так, как это уже реализовано.

Прежде чем погрузиться в проект калькулятора, необходимо рассмотреть два способа кодирования конечных автоматов и две традиционные реализации конечных автоматов.

Разработка последовательного конечного автомата

Первый способ кодирования конечного автомата заключается в его записи в одном блоке `always`, управляемом тактовым генератором.

Такой конечный автомат будет выглядеть примерно так:

```
enum bit {IDLE, DATA} state;
initial state = IDLE; // Определение начального состояния
always @(posedge clk) begin
   case (state)
     IDLE: begin
         dout_en <= '0;
         if (start) begin
            dout_en <= '1;
            state <= DATA;
         end
       end
       DATA: begin
          dout_en <= '1;
          if (done) begin
             dout_en <= '0;
            state <= IDLE;
         end
       end
   endcase
end
```

Из приведенного кода следует, что определены состояния (states) с помощью регистра перечисления `enum` под названием `state`. Затем происходит присваивание начального значения `initial` – значения, которое будет загружено при запуске проекта на FPGA.

> **Подсказка**
>
> Используйте типы `enum` для определения конечных автоматов. Хорошие имена помогают передать замысел проекта, и при моделировании на вейвформах (waves) будут отображаться имена состояний, а не числа.

Основной код находится в блоке `always`. Используется оператор `case` для определения состояний конечного автомата, `IDLE` и `DATA`. У конечного автомата два входа, `start` и `done`, и один выход, `dout_en`. Этот тип кода является лаконичным и быстрым, поскольку все выходы конечного автомата рассчитываются за один полный цикл тактового сигнала. Но у него есть потенциальный недостаток – каждый выход будет выходом триггера. В некоторых случаях может потребоваться, чтобы выход срабатывал сразу после изменения входа, а это невозможно, если описать конечный автомат таким образом.

В показанной версии конечного автомата просто использовалась одна переменная состояния. Другим способом кодирования конечного автомата является разделение переменной состояния на текущее и следующее состояния. Рассмотрим это более подробно в следующем разделе.

Разделение комбинационной и последовательностной логики в конечном автомате

Конечный автомат можно разделить на две части: комбинационную часть, основанную на текущем состоянии, которое порождает следующее состояние; оно защелкивается в регистре состояния, представляющем собой последовательностную часть конечного автомата.

```
enum bit {IDLE, DATA} current_state, next_state;
initial current_state = IDLE; // определение начального состояния
always @(posedge clk) current_state <= next_state;
always_comb begin
  current_state = next_state; // предотвращение возникновения защелки
  dout_en = '0; // avoid a latch
  case (current_state)
    IDLE: begin
      if (start) begin
        dout_en = '1;
        next_state = DATA;
      end
    end
    DATA: begin
      dout_en = '1;
```

```
        if (done) begin
          dout_en = '0;
          next_state = IDLE;
        end
      end
    endcase
  end
```

У представленного программного кода много общего с предыдущим примером. Но есть и некоторые функциональные различия, заключающиеся в том, что `dout_en` создается комбинационным образом. Чтобы добиться соответствия, нужно защелкнуть `dout_en` на триггере.

Любой из этих способов приемлем для построения конечного автомата. На самом деле у обоих есть преимущества, и можно смешивать их использование в зависимости от ситуации. Прежде чем обсуждать два классических типа конечных автоматов, следует определить, чего требуется достичь в этом проекте.

Разработка интерфейса калькулятора

Рассмотрим еще раз, что имеется на отладочных платах для подачи и отображения данных. И на плате *Basys 3*, и на *Nexys A7* имеется массив из 16 светодиодов, 16 переключателей и 5 кнопок:

Рис. 4.1. Кнопки и переключатели на плате FPGA

Можно использовать переключатели для ввода 16-битного значения в шестнадцатеричном или **двоично-десятичном формате (binary coded decimal, BCD).** Также на плате имеется 5 кнопок. Определим функции, которые они будут выполнять:

- кнопка влево (Left): A+B;
- кнопка вправо (Right): A–B;
- кнопка вверх (Up): A*B;
- центральная кнопка (Center): = (равенство);
- кнопка вниз (Down): очистка.

Используя семисегментные индикаторы, можно обрабатывать вывод результата и, используя светодиод, показывать знаковый бит.

Ввод/вывод (I/O) можно реализовать с помощью конечного автомата, приведенного на схеме ниже:

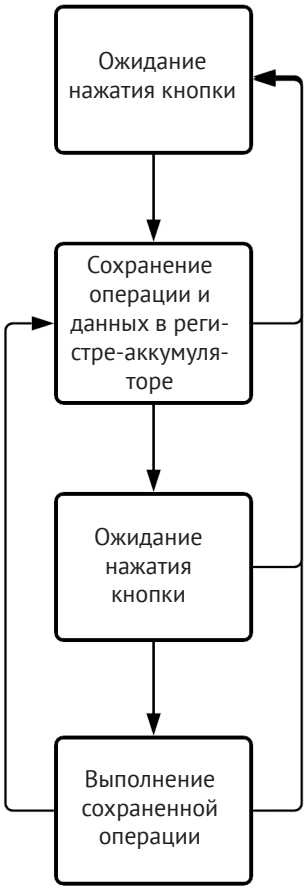


Рис. 4.2. Конечный автомат для реализации калькулятора

Теперь давайте обсудим два классических типа конечных автоматов и то, как можно было бы реализовать калькулятор, используя любой из них.

Проектирование конечного автомата Мура

В 1956 году Эдвард Ф. Мур в своей статье *Gedanken-Experiments on Sequential Machines* представил концепцию конечного автомата, выходы которого зависят только от состояния автомата.

На практике это означает, что текущее состояние генерирует выходы. Входы управляют только логикой следующего состояния. Такой тип конечного автомата может достигать высоких тактовых частот, поскольку выходы изменяются только за полный цикл тактового сигнала. Такие конечные автоматы могут быть большими, поскольку логика принятия решений не влияет на выходные данные напрямую, а, по сути, приводит к возникновению новых состояний, увеличивая пространство состояний.

Следующий программный код находится в файле `CH4/hdl/calculator_moore.sv`.

```
typedef enum bit [2:0]
            {
             IDLE,
             WAIT4BUTTON,
             ADD,
             SUB,
             MULT,
             } state_t;
```

Анализируя версию калькулятора, реализованную с помощью автомата Мура, легко выделить отдельные состояния для каждой операции. Как было указано, каждая операция выполняется за полный тактовый цикл. Это могло бы быть преимуществом, если бы требовалось увеличить скорость работы проекта, но, как было показано ранее, операция сложения требует достаточно много времени:

```
IDLE: begin
  accumulator <= '0;
  last_op     <= buttons; // операция к исполнению
  accumulator <= switch;
  if (start) state <= buttons[DOWN] ? IDLE : WAIT4BUTTON;
end
WAIT4BUTTON: begin
  op_store    <= buttons;
  if (start) begin
    case (1'b1)
      last_op[UP]:     state <= MULT;
      last_op[DOWN]:   state <= IDLE;
      last_op[LEFT]:   state <= ADD;
      last_op[RIGHT]:  state <= SUB;
      default:         state <= WAIT4BUTTON;
    endcase // case (1'b1)
  end else state <= WAIT4BUTTON;
end
MULT: begin
  last_op     <= op_store; // хранение последней операции
  accumulator <= accumulator * switch;
  state       <= WAIT4BUTTON;
end
```

Здесь не рассмотрен весь конечный автомат. Уделите несколько минут и проанализируйте программный код по следующей ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH4/hdl/Calculator_moore.sv.

После нажатия кнопки значения переключателя и операция загружаются в `accumulator` и `last_op`. Когда нажимается следующая кнопка, выполняется предыдущая операция с аккумулятором и вторыми значениями переключателя. Схема ведет себя как карманный калькулятор, где после нажатия X + Y = результат появляется на экране. Кроме того, можно связать операции, например X + Y – Z =.

Если проанализировать конструкцию конечного автомата, то можно увидеть, что входные данные влияют только на значения следующего состояния и что каждая операция с данными требует свое состояние.

Давайте посмотрим, как это будет выглядеть, если реализовать проект с помощью конечного автомата Мили.

Реализация конечного автомата Мили

В 1955 году Джордж Х. Мили представил концепцию конечного автомата, выходы которого определяются текущим состоянием и входами. Проанализируйте пример реализации конечного автомата Мили для калькулятора.

Код автомата находится в файле `CH4/hdl/Calculator_mealy.sv`:

```
typedef enum bit
               {
                IDLE,
                WAIT4BUTTON
                } state_t;
```

Из определений состояний видно, что пространство состояний значительно сократилось. Количество состояний изменилось с пяти на два. Как удалось этого добиться?

```
case (state)
  IDLE: begin
    last_op     <= buttons; // операция к исполнению
    accumulator <= switch;
    if (start) state <= buttons[DOWN] ? IDLE : WAIT4BUTTON;
  end
  WAIT4BUTTON: begin
    if (start) begin
      last_op <= buttons; // хранение последней операции
     case (1'b1)
        last_op[UP]:    accumulator <= accumulator * switch;
        last_op[DOWN]:  state       <= IDLE;
        last_op[LEFT]:  accumulator <= accumulator + switch;
        last_op[RIGHT]: accumulator <= accumulator - switch;
        default:        state       <= WAIT4BUTTON;
      endcase // case (1'b1)
    end else state <= WAIT4BUTTON;
  end
endcase // case (state)
```

Теперь больше нет ограничения на использование входа только для изменения состояния. Теперь можно использовать входы для прямого воздействия на операции. В конечном автомате Мура были состояния, предназначенные для операций. Теперь же автомат будет просто оставаться в состоянии `WAIT4BUTTON`.

Знание конструкций конечных автоматов Мили и Мура по большей части относится к академическим знаниям. Если вы решите продолжить карьеру в области FPGA, то на собеседовании вас наверняка спросят, в чем разница между этими автоматами. Но спрашивают об этом в основном потому, что при приеме на первую работу мы, инженеры, не очень-то знаем, о чем еще спрашивать, помимо академических знаний. Что вам действительно понадобится в дальнейшем, так это некоторые практические советы.

Практическое проектирование конечных автоматов

Реальность проектирования конечных автоматов такова, что нет особой необходимости беспокоиться о формальности кода и о том, является ли он кодом Мили или Мура. Следует использовать то, что удобно и соответствует решению, над которым ведется работа. На практике обычно используют смесь

стилей в зависимости от задачи. В некоторых случаях это может быть автомат Мура, а может быть и автомат гибридного типа, использующий разделенную конструкцию текущего и следующего состояния.

Ознакомившись с основами проектирования конечных автоматов, давайте рассмотрим практический проект, сердцем которого является конечный автомат.

Проект 3. Создание простого калькулятора

Теперь, когда основы конечных автоматов рассмотрены и показано ядро разрабатываемого калькулятора, перейдем к его реализации. Первый вопрос, который возникает: как хранить данные в проекте? Ранее использовался формат BCD при инкрементировании значений. Для этого было представлено простое решение.

Если бы надо было сохранить внутренние данные в формате BCD, пришлось бы разработать для BCD оператор сложения, вычитания и умножения. Это сложнее, чем просто реализовать оператор инкремента. В качестве альтернативы можно рассмотреть возможность сохранения внутреннего представления в двоичном виде, но с преобразованием в десятичную систему для отображения. Такой подход имеет дополнительное преимущество: можно использовать обычные операторы сложения, вычитания и умножения SystemVerilog в двоичном представлении, а затем создать функцию для преобразования.

Файлы проекта находятся по следующим адресам:

- Nexys A7: `CH4/build/calculator/calculator.xpr`;
- Basys 3: `CH4/build/calculator/calculator_basys3.xpr`.

Прежде чем пойти по этому пути, следует рассмотреть пакеты (библиотеки, package) SystemVerilog.

Инкапсуляция для повторного использования

SystemVerilog предоставляет возможность создания пакетов для инкапсуляции кода, который необходимо повторно использовать в нескольких модулях. Это также обеспечивает удобный способ повторного использования кода для нескольких приложений.

Следующий код можно найти в файле CH4/hdl/calculator_pkg.sv:

```
`ifndef NUM_SEGMENTS
`define NUM_SEGMENTS 8
`endif
`ifndef _CALCULATOR_PKG
`define _CALCULATOR_PKG
package calculator_pkg;
  localparam NUM_SEGMENTS = `NUM_SEGMENTS;
  localparam UP           = 3'd0;
  localparam DOWN         = 3'd1;
```

Создан пакет как пустой модуль `package <package name>;`.

Пакет (**package**) может содержать параметры, функции, задачи и определяемые пользователем типы. В нашем случае определены `localparam` для облегчения идентификации кнопок и функция для преобразования двоичного представления в BCD:

```
function bit [NUM_SEGMENTS-1:0][3:0] bin_to_bcd;
   // будет реализована поддержка 4 или 8 разрядов[1]
   input [31:0] bin_in;
   bit [NUM_SEGMENTS*4-1:0] shifted;
  shifted    = {30'b0, bin_in[31:30]};
   for (int i = 29; i >= 1; i--) begin
     shifted = shifted << 1 | bin_in[i];
     for (int j = 0; j < NUM_SEGMENTS; j++) begin
       if (shifted[j*4+:4] > 4) shifted[j*4+:4] += 3;
     end
   end
   shifted = shifted << 1 | bin_in[0];
   for (int i = 0; i < NUM_SEGMENTS; i++) begin
     bin_to_bcd[i] = shifted[4*i+:4];
   end
   endfunction // bin_to_bcd
 endpackage // calculator_pkg
`endif
```

В функции преобразования есть довольно сложные места. Это часто является проблемой при проектировании. Множество циклов `for` необходимо развернуть, чтобы создать логику, которая будет выполнять реальную задачу в FPGA. Даже при частоте 100 МГц это будет непростой задачей.

> **Подсказка**
>
> Если вы ведете разработку в режиме без проекта (без графического интерфейса), т. е. используется цикл проектирования, не связанный с проектом, то в начало файла нужно добавить следующее:
>
> ```
> `ifndef _CALCULATOR_PKG
> `define _CALCULATOR_PKG
> ```
>
> Это предотвратит переопределение пакета, которое может вызвать предупреждения или ошибки.

У пакетов SystemVerilog есть одно серьезное ограничение. Нельзя передавать параметры. Вот почему пришлось определить `NUM_SEGMENTS` в самом пакете.

Это можно обойти, используя макроопределение `define`:

```
`ifndef NUM_SEGMENTS
`define NUM_SEGMENTS 8
`endif
```

Затем можно присвоить определение `localparam`:

```
localparam NUM_SEGMENTS = `NUM_SEGMENTS;
```

[1] В оригинале у автора путаница между сегментами (отдельными светящимися линиями, составляющими рисунок цифры) и разрядами (группами из семи сегментов, представляющими отдельные цифры многоразрядного числа). NUM_SEGMENTS следует понимать не как номер сегмента, а как номер разряда, переключаемого в соответствии с диаграммой на рис. 3.14. – *Прим. ред.*

В консольном режиме можно переопределить параметр в TCL-скриптах; в проектном режиме – переопределить определяемые параметры в настройках **PROJECT MANAGER**. Теперь это нужно делать в нескольких местах, как это было показано ранее для синтеза и симуляции.

Проектирование модуля верхнего уровня иерархии

Рассмотрим модуль верхнего уровня иерархии для калькулятора.

Следующий код находится в файле `CH4/hdl/calculator_top.sv`:

```
`ifndef NUM_SEGMENTS
`define NUM_SEGMENTS 8
`endif
module calculator_top
  #(
   parameter BITS         = 32,
   parameter NUM_SEGMENTS = `NUM_SEGMENTS,
   parameter SM_TYPE      = "MEALY" // Автомат Мили или Мура
   )(
   input wire                          clk,
   input wire [15:0]                   SW,
   input wire [4:0]                    buttons,
   output logic [NUM_SEGMENTS-1:0]     anode,
   output logic [7:0]                  cathode
   );
```

Можно выбрать размер встроенной памяти данных, установив параметр `BITS`. Также можно задать количество сегментов (как это обсуждалось в разделе об упаковке кода) и тип конечного автомата.

Будет использоваться контроллер семисегментного индикатора `seven_segment`, разработанный в главе 3 «Подсчет нажатий на кнопку», а также автомат для защиты кнопок от дребезга:

```
generate
  if (USE_PLL == "TRUE") begin : g_USE_PLL
    sys_pll u_sys_pll
      (
       .clk_in1  (clk),
     .clk_out1 (clk_50)
      );
    end else  begin : g_NO_PLL
      assign clk_50 = clk;
    end
endgenerate
```

В проекте также присутствует фазовая автоподстройка частоты (PLL), и создан внутренний тактовый генератор `clk_50`. Генерация тактовых сигналов с помощью PLL будет рассмотрена в следующем разделе. Для первого запуска установите параметр `USE_PLL = "FALSE"`:

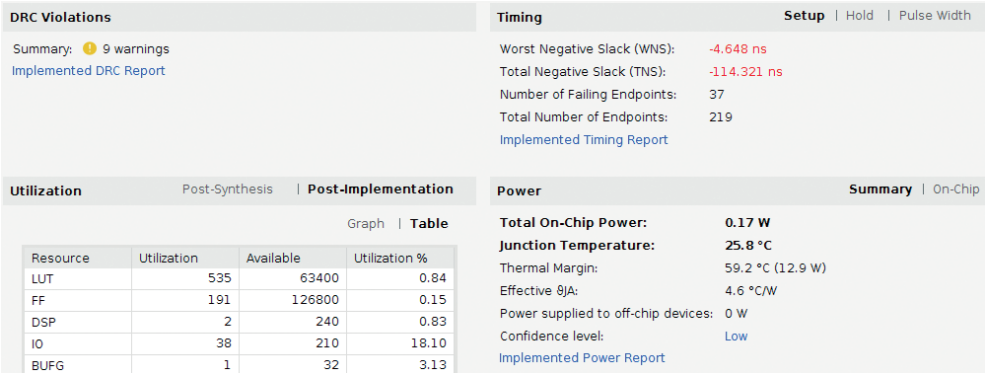


Рис. 4.3. Схема калькулятора на основе автомата Мили, 100 МГц

Из этого запуска видно, что проект не соответствует временным требованиям на частоте 100 МГц. Можно посмотреть на неудавшиеся маршруты в анализаторе времени и заметить, что преобразователь двоичного кода в BCD не соответствует требованиям по времени:

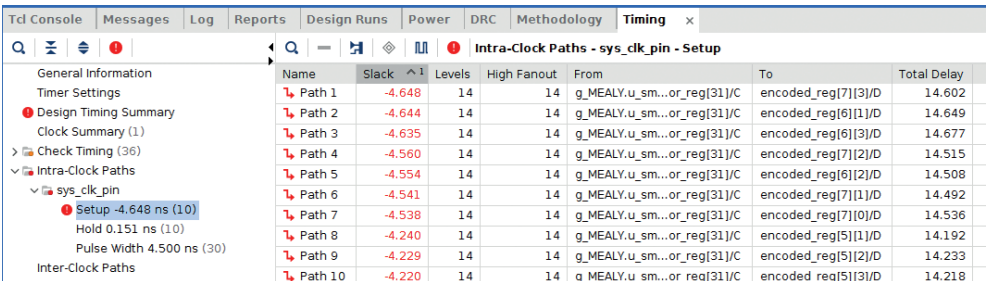


Рис. 4.4. Временны́е соотношения автомата Мили

У конечного автомата Мура имеется больше времени для выполнения операций, но он, скорее всего, не сильно повлияет на неудавшиеся маршруты, поскольку они проходят через преобразователь. Попробуйте осуществить запуск еще раз с SM_TYPE, установленным на MOORE. Вы увидите, что результаты очень похожи. В проекте с автоматом Мура есть немного дополнительных элементов хранения и немного лучшие временны́е параметры.

Изменение тактовой частоты с помощью PLL или MMCM

Artix 7 содержит **блоки управления тактовой частотой (Clock Management Tiles, CMTs)**, которые являются основой для генерации тактовых сигналов в используемых устройствах. Nexys A7 100T имеет 6 CMT. Nexys A7 50T и Basys 3 имеют по 5 CMT.

CMT содержит компонент фазовой автоподстройки частоты (**Phase Locked Loop, PLL**) и компонент управления тактовой частотой (**Mixed Mode Clock Manager, MMCM**). PLL и MMCM могут использоваться для синтеза частоты, т. е. для создания тактовых сигналов с большей или меньшей частотой из внешних тактовых сигналов:

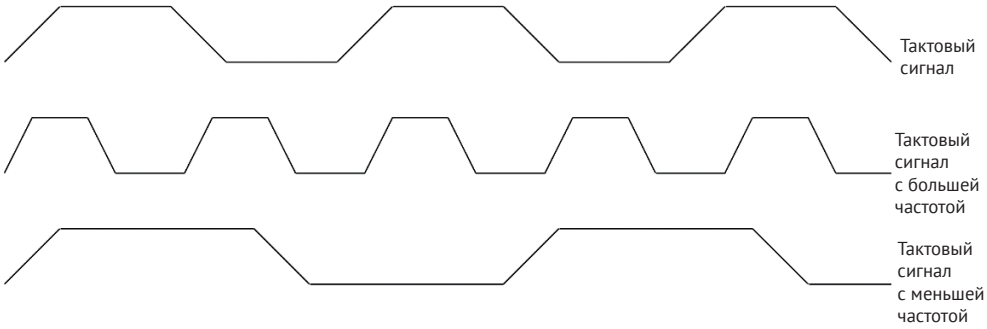


Рис. 4.5. Синтез частоты PLL/MMCM

Из диаграммы выше видно, что можно принимать внешний тактовый сигнал на вход. Для используемой платы это будет тактовая частота 100 МГц, и можно генерировать тактовые сигналы с большей или меньшей частотой в зависимости от потребностей. Существуют и другие применения CMT, которые не будут рассмотрены в этой книге из-за ограниченности интерфейсов плат.

Давайте построим собственный PLL для уменьшения внутренней тактовой частоты до 50 МГц, чтобы можно было соблюсти временные параметры в проекте. Для этого ознакомьтесь с каталогом IP. Компания Xilinx и ее партнеры предоставляют IP для своих устройств FPGA, некоторые из которых бесплатны, а другие могут быть лицензированы. Рекомендуем вам ознакомиться с тем, что доступно, особенно с бесплатными вариантами. Давайте посмотрим, как это сделать.

1. Выберите каталог IP.

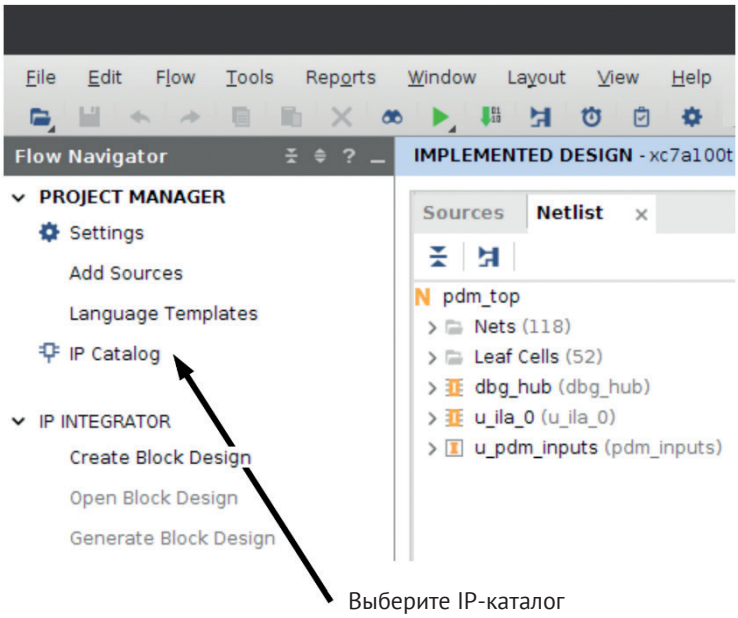


Рис. 4.6. IP-каталог

2. Выберите мастер управления тактовой частотой.

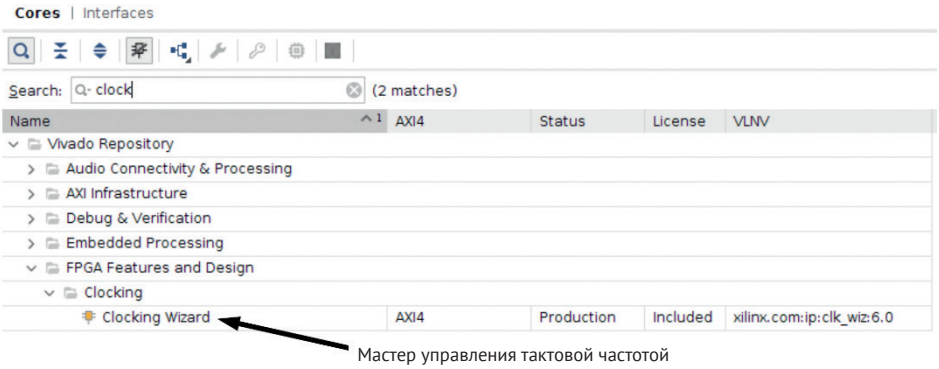


Рис. 4.7. Мастер управления тактовой частотой

3. Первым шагом к созданию IP-блока с помощью мастера является его настройка. Выберите кнопку **IP Location** и обязательно укажите `CH5/build/IP`. Затем измените название компонента на `sys_pll`. Убедитесь, что **CLK_IN1** установлен на **sys clock**.

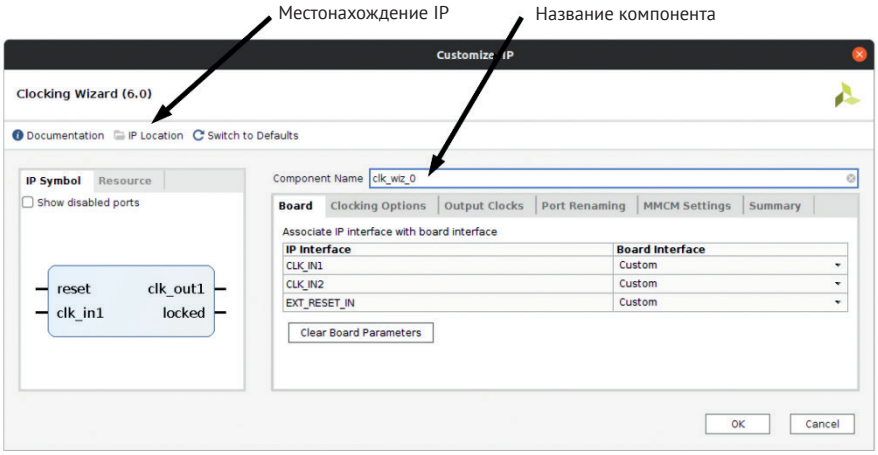


Рис. 4.8. Настройка IP-блока

4. Откройте вкладку **Output clocks** и установите **clk_out1** на 50 МГц. Так как это четное деление 100 МГц, то можно установить частоту точно. В иных случаях будет сделано все возможное, но частота будет немного смещена. Не забудьте снять выбор с **reset** и **locked**, так как эти выходы в данном случае не нужны.

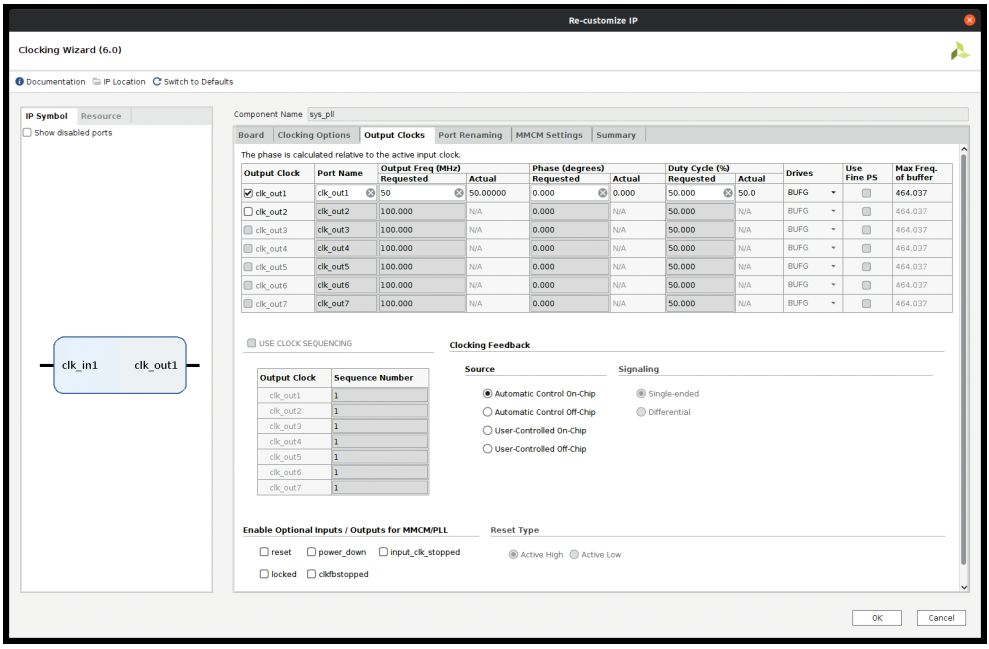


Рис. 4.9. Вкладка Output clocks

5. В проекте уже есть этот компонент, поэтому мастер можно закрыть. Если требуется добавить компонент в проект, нажмите **OK**, и появится запрос на добавление в проект и генерацию выходных портов.
6. Теперь установите параметр **USE_PLL** в **PROJECT MANAGER | Settings | General** на значение **TRUE**, как это уже делалось ранее.
7. Установите для параметра **PLL** значение **TRUE** и выполните следующую сборку.

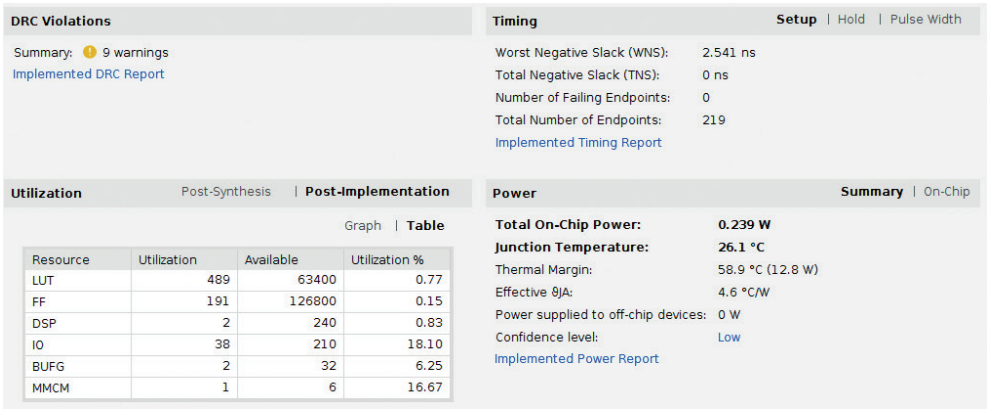


Рис. 4.10. Конечный автомат Мили на частоте 50 МГц

Теперь, когда тактовая частота была снижена, средства трассировки не так сильно стремятся уложиться в неоправданные временные рамки. Можно увидеть результат этого по уменьшению количества LUT в данном запуске. Временные параметры соблюдаются легко. Попробуйте запустить проект на плате. Попробуйте выполнить операции сложения и умножения.

Теперь перезагрузите схему и вычтите что-нибудь из 0. Вы увидите, что на семисегментном индикаторе высвечивается что-то похожее на случайные значения. В схеме нет поддержки отрицательных чисел. Это то, о чем стоит подумать в задании повышенной сложности этой главы.

Следует отметить, что разработанный калькулятор довольно прост. Он поддерживает сложение, вычитание и умножение. Почему же не было включено деление? Вспомните, как вы учили деление столбиком. Это процесс перестановки и вычитания чисел, пока есть достаточно большое значение, из которого можно вычитать. Оказывается, операции сложения/вычитания и умножения очень просты по сравнению с делением, поэтому они заложены в структуру FPGA. Чтобы выполнить деление, следует изучить, как это можно сделать.

Разработка блока деления

Существует два классических алгоритма целочисленного деления: **восстанавливающий** (с восстановлением остатка) и **невосстанавливающий** (без восстановления остатка). Разница между этими двумя методами заключается в том, что при выполнении тестового вычитания на каждом проходе либо восстанавливается, либо сохраняется отрицательный результат в зависимости от используемого метода. При невостанавливающем делении в конце выполняется этап коррекции.

Рассмотрим, как реализовать невосстанавливающий делитель для разрабатываемого калькулятора.

Построение конечного автомата невосстанавливающего делителя

Первым шагом является создание диаграммы состояний для разрабатываемого конечного автомата. Алгоритм невостанавливающего деления можно найти во многих местах в сети Интернет. Предлагаемая схема состояний представлена далее.

Из схемы конечного автомата видно, что это одна из самых сложных конструкций конечного автомата среди тех, которые рассматривались до сих пор. Здесь есть несколько проверок и условий ветвления.

Как и во многих других конечных автоматах, начальным состоянием является ожидание (Idle). После получения команды на деление инициализируются внутренние переменные (шаг 1). Затем автомат входит в основной цикл на шаге 2. Проверяется знак остатка (2a), а затем сдвигается остаток и частное (2b). Если остаток отрицательный, прибавляется делитель, в противном случае происходит его вычитание (шаг 3). Затем подается инвертированный знак остатка обратно в младший бит частного и уменьшается счетчик (шаг 4). Если счетчик не равен нулю, происходит возврат к шагу 2 (шаг 5), в противном случае окончательно проверяется знак остатка (шаг 6). Если он отрицательный, снова прибавляется делитель, в противном случае алгоритм завершает свою работу.

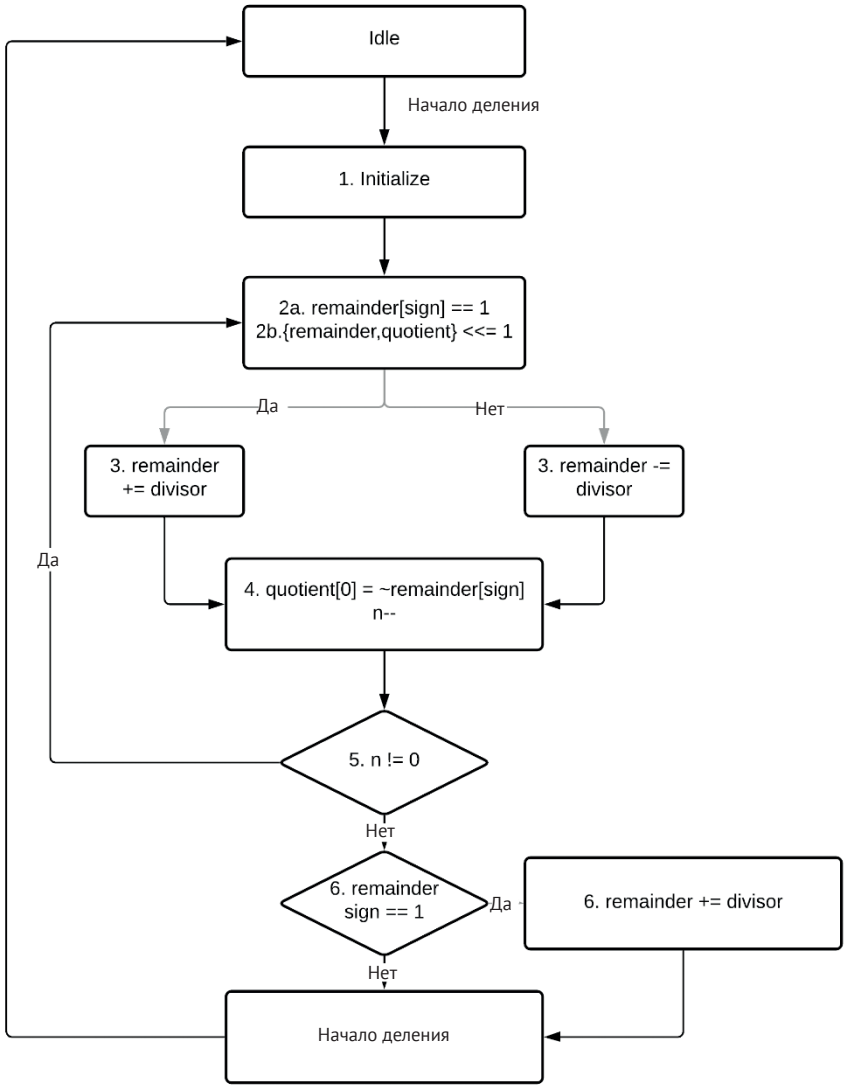


Рис. 4.11. Алгоритм деления без восстановления остатка

Проанализируем исходный код делителя. Следующий код находится в файле `CH4/hdl/divider_nr.sv`:

```
module divider_nr #(parameter BITS      = 16)
  (
   input wire                       clk,
   input wire                       start, //начало
   input wire unsigned [BITS-1:0]   dividend, //делимое
   input wire unsigned [BITS-1:0]   divisor, //делитель
   output logic                     done,  //флаг завершения операции деления
   output logic unsigned [BITS-1:0] quotient, //частное
   output logic unsigned [BITS-1:0] remainder //остаток
   );
```

Первое, что следует отметить, – это то, что были добавлены два сигнала для управления работой конечного автомата, `start` (начало) и `done` (сделано). Существует несколько вариантов для выполнения деления. Можно либо всегда брать количество тактов, равное количеству битов делимого, либо корректировать делимое, чтобы удалить ведущие 0, так как в результате эти биты всегда будут равны 0 в делителе. Поскольку был уже разработан детектор ведущей единицы, его можно использовать для ускорения операции деления, где это возможно. Для текущей задачи это не так важно. Тем не менее с целью снизить затраты времени на разработку можно повторно использовать этот модуль.

При получении сигнала запуска (start) делитель начнет работу.

```
always @(posedge clk) begin
  done <= '0;
  case (state)
    IDLE: begin
      if (start) state <= INIT;
    end
    INIT: begin
      state         <= LEFT_SHIFT;
      quotient      <= dividend << (BITS - num_bits_w);
      int_remainder <= '0;
      num_bits      <= num_bits_w;
    end
```

Обратите внимание, что `done` имеет значение по умолчанию `'0`, поэтому он будет в состоянии логической единицы только в тех состояниях, где это явно указано. Переходя в состояние `INIT`, делитель сдвигается влево, чтобы удалить ведущие 0, и устанавливается `num_bits`, чтобы знать, как долго будет работать делитель. Также промежуточный остаток устанавливается равным 0:

```
leading_ones
  #(
   .SELECTOR  ("DOWN_FOR"),
   .BITS (BITS)
   )
u_leading_ones
  (
   .SW        (dividend),
   .LED       (num_bits_w)
   );
```

Модуль `leading_ones` собран как модуль для платы, поэтому используются имена портов, которые есть на плате. Чтобы сделать этот модуль действительно переносимым, следует переименовать порты в нечто более соответствующее детектору ведущей единицы, например `vector_in` и `ones_position`:

```
LEFT_SHIFT: begin
  {int_remainder, quotient} <= {int_remainder, quotient} << 1;
  if (int_remainder[$left(int_remainder)])
    state   <= ADJ_REMAINDER0;
  else
    state   <= ADJ_REMAINDER1;
end
```

Количество битов возвращается с помощью разработанной ранее функции `leading_ones`. `LEFT_SHIFT` (сдвиг влево) представляет состояния 2a и 2b на схеме состояний – сдвиг влево промежуточных результатов. Когда происходит сравнение по знаковому биту `int_remainder[$left(int_ remainder)]`, следует помнить, что `$left` вернет или старший бит, или знаковый бит внутреннего остатка. Также необходимо отметить, что поскольку используется неблокирующий метод, то можно сдвигать и проверять предыдущее значение в одном и том же тактовом цикле.

> **Подсказка**
> if (A) – это сокращение для if (A!=0).

Имеется два состояния для обновления внутреннего остатка, `AJD_REMAINDER[2]`. Эти два состояния представляют собой третьи шаги в схеме состояний:

```
UPDATE_QUOTIENT: begin
   state       <= TEST_N;
   quotient[0] <= ~int_remainder[$left(int_remainder)];
   num_bits    <= num_bits - 1'b1;
 end
 TEST_N: begin
   if (|num_bits)
     state <= LEFT_SHIFT;
   else
     state <= TEST_REMAINDER1;
 end
```

Далее происходит обновление младшего значащего бита частного, корректируется количество обрабатываемых битов, а затем происходит проверка, завершен ли сдвиг:

```
TEST_REMAINDER1: begin
   if (int_remainder[$left(int_remainder)])
     state   <= ADJ_REMAINDER2;
   else
     state   <= DIV_DONE;
 end
 ADJ_REMAINDER2: begin
   state     <= DIV_DONE;
   int_remainder <= int_remainder + divisor;
 end
 DIV_DONE: begin
   done <= '1;
   state    <= IDLE;
 end
```

После завершения сдвига происходит проверка и обновление промежуточного остатка, если это необходимо. В конечном итоге устанавливается сигнал `done`, чтобы показать, что результат готов.

Закончив разработку проекта, проведем его моделирование.

Моделирование делителя

Ниже представлен testbench делителя. Это позволит проверить алгоритм деления до его реализации.

Следующий код находится в файле CH4/tb/tb_divider_nr.sv:

```
for (int i = 0; i < 100; i++) begin
  dividend <= $random;
  divisor  <= $random;
  start <= '1;
  @(posedge clk);
  start <= '0;
  while (!done) @(posedge clk);
  repeat (5) @(posedge clk);
end
```

Основная часть кода создает случайные делитель и делимое. Проект запускается, после чего происходит цикл ожидания, пока сигнал done не перейдет в высокий уровень, прежде чем вводить следующие значения:

```
always @(posedge clk) begin
  if (done &&
      (quotient != dividend/divisor) &&
      (remainder != dividend%divisor)) begin
    $display("failure!");
    $display("quotient:   %d", quotient);
    $display("remainder:  %d", remainder);
    $display("expected Q: %d", dividend/divisor);
    $display("expected R: %d", dividend%divisor);
    $stop;
  end
end
```

Testbench сверяет частное и остаток деления со значениями, возвращаемыми оператором деления SystemVerilog (/) и оператором деления по модулю (%).

> **Важное замечание**
>
> В SystemVerilog есть оператор деления. Но он может быть синтезирован только для операндов, которые являются степенями 2, или если он возвращает целое значение.
>
> Существует также оператор деления по модулю (%), который возвращает остаток от деления. Он также может быть синтезирован, только если он возвращает целое значение или если правый операнд является степенью 2.

Определение размера промежуточного остатка

Обратите внимание на то, что промежуточный остаток объявлен на 1 бит больше, чем фактический остаток:

```
logic signed [BITS:0] int_remainder;
```

Можно констатировать, что возникли трудности при проверке проекта. Изначально внутренний остаток объявлен как `BITS-1:0`. Не было учтено, что имеется дело с беззнаковыми числами до 65535. Поскольку происходит сложение или вычитание делителя из `int_remainder`, то имеется 16-битное беззнаковое значение +/– 16-битное беззнаковое значение. Это означает, что нужен 17-битный внутренний остаток, чтобы хранить бит знака.

Вот и закончена работа над простым калькулятором. Он был доработан за счет добавления сложной функции деления. Перейдем теперь к другому важному проекту из инженерного курса – контроллеру светофора.

Проект 4. Управление перекрестком с помощью светофоров

Классической задачей для начинающих инженеров является разработка контроллера светофора. Файлы проекта Xilinx для Nexys A7 можно найти в `CH4/build/traffic_light/ traffic_light.xpr`. На плате Basys 3 нет трехцветных светодиодов, поэтому данный проект не может быть выполнен на этой плате.

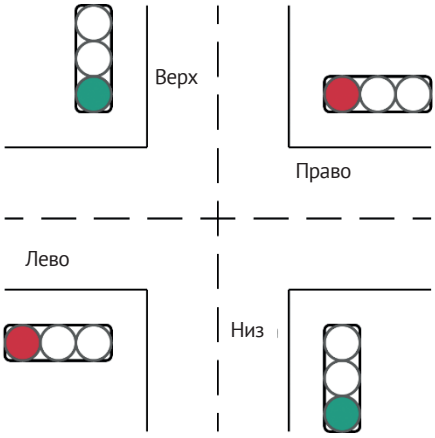


Рис. 4.12. Перекресток, регулируемый светофорами

На представленной схеме показан основной сценарий. Имеется перекресток с четырьмя светофорами и четырьмя датчиками, обозначенными как Верх, Низ, Лево, Право.

Некоторые основные правила таковы:

- когда загорается зеленый свет, он остается зеленым минимум 10 с;
- если автомобиль проезжает на зеленый свет, он игнорируется;
- когда автомобиль стоит на красный свет, датчик сигнализирует, что этот светофор должен быть переключен на зеленый после того, как другой светофор будет зеленым в течение 10 с;
- при переходе от зеленого к красному свет становится желтым на 1 с.

Задача сформулирована. Первым шагом, как всегда, является создание графа состояний.

Определение графа состояний

Зачастую можно сразу же приступить к разработке кода, но граф состояний может быть хорошим способом документирования намерений и поиска потенциальных проблем на раннем этапе проектирования:

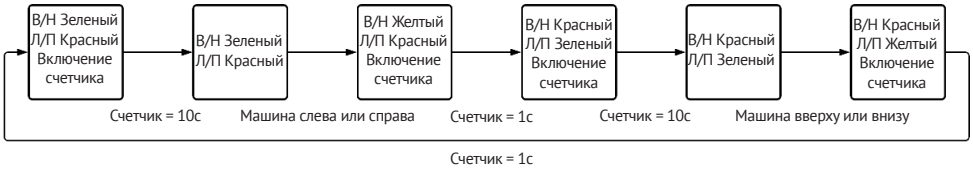


Рис. 4.13. Диаграмма состояний контроллера светофора (В – верх, Н – низ, Л – лево, П – право)

Конечный автомат выглядит относительно простым. Он представляет собой последовательный поток данных с двумя входами и счетчиками, которые удерживают состояние в течение минимального времени. Как запустить светофоры – понятно, но как использовать плату для отображения их состояний?

Отображение состояний светофоров

На используемых платах есть трехцветные светодиоды. С помощью них можно отображать практически любой цвет спектра, используя **широтно-импульсную модуляцию (ШИМ).** У каждого такого светодиода есть три вывода для управления красным, зеленым и синим цветом[1].

Широтно-импульсная модуляция

В цифровой логике на самом базовом уровне информация передается в виде цепочек из единиц и нулей. Когда раньше светодиоды зажигались, то это означало, что была подана `'1`, чтобы светодиод горел, и `'0` – чтобы он был выключен. В случае трехцветных светодиодов имеется контроль над тремя выводами, по одному для красного, зеленого и синего. Если установить любой из них в `'1`, то будет ярко светиться соответствующий цвет. Рекомендуется подавать сигнал со скважностью менее 100 %, чтобы светодиод не слепил. В случае с контроллером светофора скважность выставлена на 50 %:

```
always @(posedge clk) begin
  light_count <= ~light_count;
  R <= '0;
  G <= '0;
  B <= '0;
  if (light_count) begin
```

Создавая однобитовый сигнал, называемый `light_count`, и используя оператор `if`, можно добиться того, чтобы светодиод горел только 50 % времени.

[1] Разумеется, в реальных моделях светофора применять RGB-светодиод там, где гораздо проще применить трехцветный красный-желтый-зеленый, нецелесообразно. Но автор, очевидно, подстраивался под возможности платы, которую использует, и, кроме того, преследует цель проиллюстрировать различные методы обращения с оборудованием. – *Прим. ред.*

Можно заметить одну потенциальную проблему с трехцветным светодиодом. Имеется красный и зеленый цвет для светофора, но нет желтого. Если вспомнить теорию цвета, то можно показать, что при смешивании красного и зеленого получится желтый, как показано в примере ниже.

```
GREEN: begin
  G[0] <= '1;
end
YELLOW: begin

  R[0] <= '1;
  G[0] <= '1;
end
RED: begin
  R[0] <= '1;
End
```

На самом деле нецелесообразно создавать достаточно большое пространство состояний, чтобы реализовать задержку в 1 или 10 с. Для этого потребуется разработать схему задержки. Один из способов – создать счетчик и использовать его в конечном автомате.

Реализация задержек с помощью счетчика

Последний элемент, который нужен для того, чтобы реализовать проект, – это счетчик. Если посмотреть на диаграмму состояний на рис. 4.13, можно увидеть, что происходит счет 1 или 10 с. Можно было бы сделать два отдельных счетчика, но можно и иначе – создать один счетчик, достаточно большой, чтобы считать до 10 с и использовать его для обоих случаев:

```
localparam COUNT_1S  = int'(100000000 / CLK_PER);
localparam COUNT_10S = 10 * int'(100000000 / CLK_PER);
bit [$clog2(COUNT_10S)-1:0]        counter;
```

Размер счетчика установлен на 10 с, и имеется два параметра, определенных для итогового подсчета. Когда нужно посчитать, счетчик включается с помощью `enable_count`, и проверяется итоговое значение с помощью оператора `if`. Если счетчик не используется, он сбрасывается в 0.

```
always @(posedge clk) begin
  lr_reg         <= lr_reg << 1 | SW[0];
  ud_reg         <= ud_reg << 1 | SW[1];
  enable_count   <= '0;
  if (enable_count) begin
    counter <= counter + 1'b1;
  end else begin
    counter <= '0;
  end
  case (state)
    INIT_UD_GREEN: begin
      up_down      <= GREEN;
      left_right   <= RED;
      enable_count <= '1;
      if (counter == COUNT_10S) state <= UD_GREEN_LR_RED;
    end
```

Проанализируйте конечный автомат и убедитесь, что его поток данных соответствует графу состояний. Запустите его на плате.

Теперь у вас должен получиться исправный контроллер светофора. Поэкспериментируйте с переключателями и убедитесь, что они будут оставаться в заданном состоянии, пока автомобиль не будет обнаружен. Убедитесь, что освещение работает правильно. В настоящее время разработать контроллер светофора стало намного проще. Ранее это приходилось делать на макетной плате с дискретными электронными компонентами.

Выводы

В этой главе было показано, как можно использовать знания о последовательностных и комбинационных элементах SystemVerilog для разработки конечных автоматов. Были рассмотрены два классических типа конечных автоматов, а затем разработан простой калькулятор, используя полученные знания. Также были затронуты некоторые основы математики и то, как разработать целочисленный делитель с помощью SystemVerilog.

Было показано повторное использование модулей проекта, реализовав package для калькулятора. Код детектора ведущей единицы, разработанный ранее, также был переиспользован.

В главе была кратко рассмотрена реализация конечного автомата и показано на высоком уровне, как можно управлять тактовой частотой с помощью PLL, чтобы проект работал на плате.

С этими знаниями теперь можно приступить к расширению калькулятора. В настоящее время он работает только с беззнаковыми числами. Но будет не так уж сложно сделать его способным работать и со знаковыми числами.

В следующей главе будут рассмотрены некоторые ресурсы платы и вы увидите, как захватывать аудиоданные и воспроизводить их. Будет рассказано о датчике температуры, разработан термостат и вывод температуры на семисегментный индикатор. Также вы узнаете, как работать с методами обработки данных и сгладить показания датчика, чтобы сделать их немного стабильнее.

Вопросы

1. В модуле делителя выполняется сдвиг промежуточных результатов. Почему был использован следующий код:

```
{int_remainder, quotient} <= {int_remainder, quotient} <<
1;
```

а не такой:

```
{int_remainder, quotient} <<= 1;
```

a) Такой код лучше передает замысел проекта.
b) <<= – это блокирующее присваивание, и он используется в блоке синхронизации, что нарушает принципы безопасной практики проектирования.
c) Когда пользуется функция конкатенации {}, то нельзя использовать <<=.

2. Что из перечисленного ниже является синтезируемым в SystemVerilog?

```
logic [15:0] A, B;
```

   a) A / B.
   b) A / 4.
   c) A % B.
   d) 5 % 4.
3. Поэкспериментируйте с цветами в проекте контроллера светофора. Придумайте новые цвета, увеличив размер счетчика и включая RGB-выходы в разное время. Цветовое пространство практически не ограничено.
4. Разработанный калькулятор в настоящее время не реализует функцию деления. Можете ли вы модифицировать его для поддержки деления? На плате Basys 3 вам нужно будет заменить одну из используемых кнопок. На плате Nexys A7 вы можете использовать кнопку сброса процессора или переназначить кнопки.

Задание повышенной сложности

Разработанный простой калькулятор в настоящее время работает только с беззнаковыми числами. В случае сложения, вычитания и умножения двоичное число уже представлено в дополнительном коде. Вспомните, что для получения дополнительного кода числа нужно инвертировать его и прибавить единицу. Можете ли вы модифицировать код проекта так, чтобы можно было работать с отрицательными числами? Возможно, вы захотите использовать один из светодиодов для отображения знака результата.

Задание еще более высокой сложности

Отрицательные числа при делении обрабатывать сложнее. Можете ли вы модифицировать невосстанавливающий делитель, чтобы он также работал с отрицательными числами?

Дополнительное чтение

Обратитесь к следующим ссылкам для получения дополнительной информации о том, что было рассмотрено в этой главе:

- https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual.;
- https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf.

Глава 5

Ресурсы FPGA, и как их использовать

В этой главе будут более подробно рассмотрены некоторые из ресурсов, лежащих в основе FPGA. С некоторыми из них вы уже познакомились вкратце, например с **запоминающими устройствами с произвольным доступом (RAM)** и арифметическими блоками DSP, в то время как другие были пока обойдены стороной, например модуль фазовой подстройки частоты PLL, один из которых был использован для решения проблемы с тактовой частотой в проекте калькулятора. Изучение этих новых ресурсов будет опираться на предыдущий опыт.

К концу этой главы у вас сложится представление о том, как взаимодействовать с внешними компонентами. Вы познакомитесь с несколькими форматами данных, широтно-импульсной модуляцией (PWM) и частотно-импульсной модуляцией (PDM). Вы увидите простую последовательную шину I2C в действии, а также узнаете, как реализовать хранение данных в виде FIFO.

В этой главе будут рассмотрены следующие основные темы:

- PDM-микрофон;
- моделирование микрофона;
- встроенная память;
- обработка данных датчика температуры I2C;
- сглаживание данных;
- FIFO.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH5.

Проект 5. Слушать и учиться

Для этого проекта требуется микрофон на плате Nexys A7. Чтобы запустить его на плате Basys 3, необходимо установить дополнительный микрофон `pmod`, подключить его и соответствующим образом изменить файл XDC.

Плата Nexys A7 снабжена цифровым микрофоном, который можно использовать для обнаружения шума, речи и тому подобного в окружающей среде, где находится плата. В данном проекте этот микрофон будет использоваться для записи звука. Для этого нужно изучить форматы данных и способы их выборки. Также будет возможно воспроизведение записанного звука.

Что такое цифровой PDM-микрофон?

Цифровой микрофон должен принимать аналоговые аудиоданные и преобразовывать их в цифровые данные, пригодные для использования электроникой. Сигнал захватывается однобитным АЦП (ADC), который кодирует его выход с помощью **частотно-импульсной модуляции (Pulse Density Modulation, PDM[1])**. Когда импульсы более плотные в течение определенного периода времени, они представляют большие значения. На рис. 5.1 приведен сигнал из `testbench` в виде синусоиды. Это пример того, как может выглядеть PDM-форма данного сигнала.

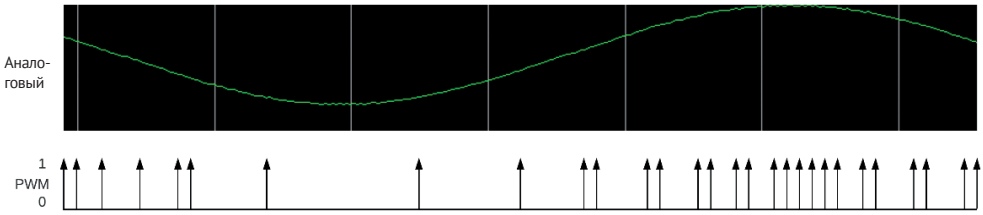


Рис. 5.1. Пример PDM-формы сигнала

Преимущество этого типа сигнала в том, что для передачи информации нужен всего один провод, поскольку обычно частота звука ограничена примерно 24 кГц, а тактовая частота будет на порядки выше.

Рассмотрим, как взаимодействовать с микрофоном. Проект находится по адресу `CH5/build/pdm_audio/pdm_audio.xpr`.

---

[1] Pulse Density Modulation, PDM (частотно-импульсная модуляция), – способ оцифровки сигнала, при котором аналоговое значение амплитуды представляется в виде количества импульсов высокой частоты (единицы мегагерц) в единицу времени (плотности импульсов). Ближайший родственник всем известной PWM (широтно-импульсной модуляции, ШИМ), но, в отличие от нее, вместо фиксированной частоты с переменной шириной импульса использует намного более высокую частоту с изменением количества импульсов фиксированной ширины в единицу времени. PDM представляет собой однобитное преобразование, результат которого записывается в виде простой последовательности нулей и единиц во времени. Традиционная PCM (импульсно-кодовая модуляции, ИКМ) использует целое двоичное слово для записи мгновенного значения амплитуды, потому для нее модуль оцифровки получается дороже и сложнее, оцифровка происходит с меньшей частотой дискретизации, чем при PDM, а шумы отфильтровываются хуже. – *Прим. ред.*

Следующий код можно найти в файле `CH5/hdl/pdm_top.sv`:

```
module pdm_top
  #(parameter CLK_FREQ = 100)
  (
   input wire   clk, // 100МГц тактовая частота
   // Microphone interface
   output logic m_clk,
   output logic m_lr_sel,
   input wire   m_data,
   output logic R,
   output logic G,
   output logic B,
   output logic [0:0] LED
   );
```

Из кода следует, что используется тактовая частота 100 МГц, поскольку понадобится источник тактовой частоты для связи с микрофоном. Здесь есть два выхода на микрофон и один вход для получения данных от микрофона (`m_data`). Выходы тактируются сигналом (`m_clk`) с частотой данных, умноженной на количество циклов выборки; в данном случае 12 кГц * 128 выборок = 1,536 МГц. Последний сигнал `m_lr_sel` выбирает, какие данные будут представлены по переднему или заднему фронту тактового сигнала. Это необходимо для того, чтобы два устройства могли использовать одну шину данных для создания левого и правого канала, как показано на рис. 5.2.

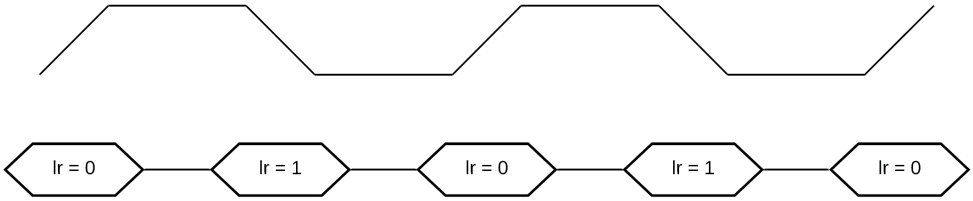


Рис. 5.2. Синхронизация микрофона – левый/правый канал

Также выведен один из трехцветных светодиодов для отображения информации об амплитуде в качестве обратной связи.

Тактовая частота микрофона должна быть в диапазоне от 1 до 3,3 МГц. Следуя указаниям по проектированию из руководства по Nexys A7, мы построили схему для генерации сигналов синхронизации устройства.

Следующий код находится в файле `CH5/hdl/pdm_inputs.sv`:

```
module pdm_inputs
  #(
   parameter          CLK_FREQ    = 100,    // МГц
   parameter          SAMPLE_RATE = 2400000 // Гц
   )
  (
   input wire         clk, // 100МГц
   // Интерфейс микрофона
   output logic       m_clk,
   output logic       m_clk_en,
   input wire         m_data,
```

```
    // Выводы амплитуды
    output logic [6:0] amplitude,
    output logic       amplitude_valid
    );
```

Модуль `pdm_inputs` генерирует тактовые сигналы для микрофона и получает данные обратно:

```
localparam CLK_COUNT = int'((CLK_FREQ*1000000)/(SAMPLE_
RATE*2));
...
    if (clk_counter == CLK_COUNT - 1) begin
      clk_counter <= '0;
      m_clk       <= ~m_clk;
      m_clk_en    <= ~m_clk;
    end else
      clk_counter <= clk_counter + 1;
```

Что касается генерации нового тактового сигнала, существует несколько вариантов. Лучше всего использовать PLL или MMCM для генерации точного тактового сигнала, который можно использовать. В случае с низкой тактовой частотой это невозможно. Что можно сделать, имея достаточно высокую частоту в 100 МГц, так это создать счетчик, который считает до значения, представляющего половину тактового периода генерируемых частот, а затем можно создать чистый тактовый генератор с помощью триггера. Нет необходимости иметь генератор непосредственно в разрабатываемой схеме, поэтому создается импульс `m_clk_en`, который даст знать, когда можно захватывать данные по переднему фронту тактового импульса.

Для квантования данных PDM нужно создать набор перекрывающихся окон, как показано на рис. 5.3. Перекрывающиеся окна позволяют выполнять подвыборку результатов для увеличения разрешения выборок, удваивая частоту выборки:

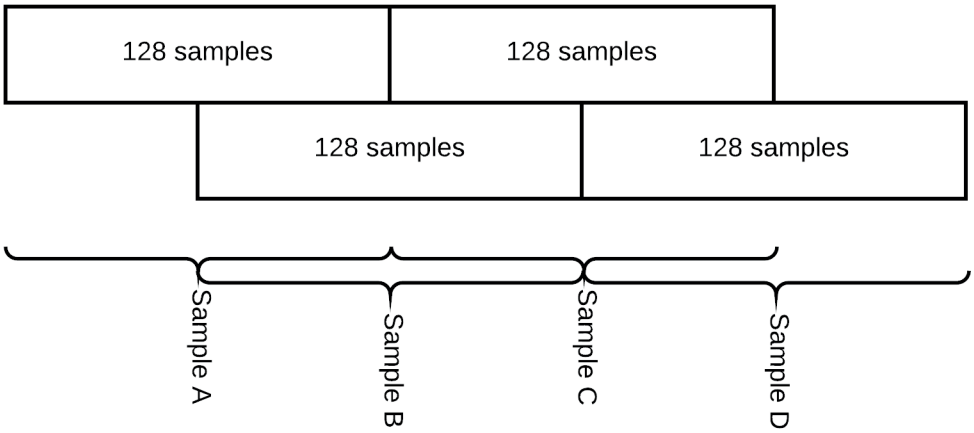
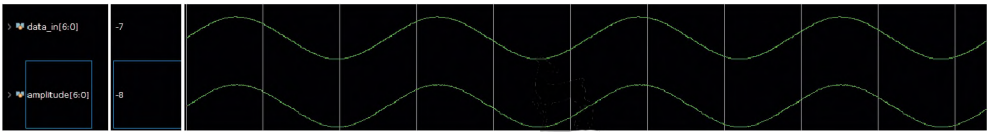


Рис. 5.3. Выборка

Ниже приведен код схемы, которая собирает отсчеты 12 кГц, чередуя их для создания выходного сигнала 24 кГц:


data_in[6:0]
-7
amplitude[6:0]
-8

szynalski.com/tone-generator/, в качестве синусоиды для генерации тестового тона с помощью компьютера.

Рис. 5.5. Захват волны в chipscope

На рис. 5.5 показан захват голоса с помощью микрофона. Было обнаружено, что нужно находиться очень близко, чтобы захватить что-нибудь полезное.

Одна из весьма полезных настроек в chipscope при очень низкой тактовой частоте – это включение контроля захвата (capture control):

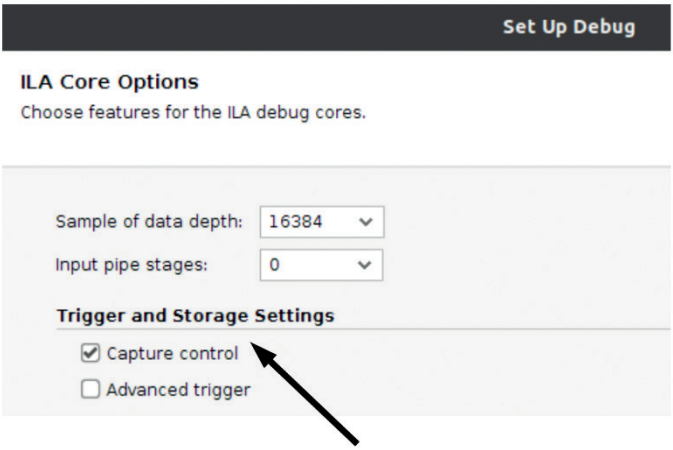


Рис. 5.6. Контроль захвата сигнала

Когда контроль захвата сигнала включен, можно получить доступ к другой панели в ILA, которая позволяет ограничить время захвата данных в буфер:

Рис. 5.7. Настройки захвата данных

Величина `amplitude_valid` установлена на частоту 24 кГц. Настройка захвата (capture setup) позволяет захватывать только то, что действительно требуется захватить для извлечения актуальной информации об амплитуде. Если посмотреть на предыдущую форму волны на рис. 5.5, то центральная точка находится в `~7'd64`, что соответствует 0.

Таким образом показано, что можно получать данные PDM и генерировать значения амплитуды на частоте 24 кГц, но пока это никак не используется. Следующим шагом будет добавление памяти.

Встроенная память

Теперь, когда удалось получить некоторые аудиоданные, нужно что-то с ними сделать. В настоящее время регистр данных постоянно перезаписывается с частотой 24 кГц. Можно многое сделать с аудиоданными, но сначала нужно их сохранить. Можем создать оперативную память (RAM) и, нажав на кнопку, начать записывать данные, зажигая светодиод, когда запись завершится.

RAM-разработка в коде против реализации с помощью шаблона или реализации из IP-каталога

Существует несколько вариантов создания оперативной памяти. Наиболее гибким методом создания оперативной памяти является ее описание с помощью кода SystemVerilog. Преимуществом этого метода является кросс-платформенная поддержка большинством производителей FPGA и разными устройствами. Компания Xilinx предоставляет набор шаблонов, которые могут помочь вести разработку.

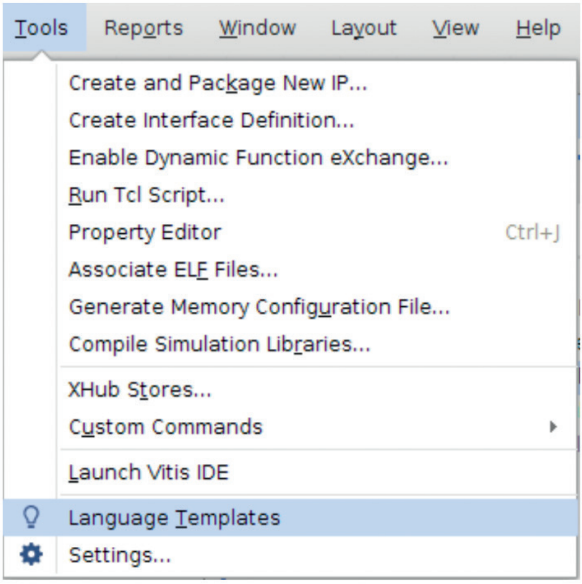


Рис. 5.8. Шаблоны кода

Если вы выберете **Tools | Language Templates** (Инструменты | Языковые шаблоны), то можно получить доступ к различным примерам кода, которые

помогут начать разработку. Прежде чем приступить к разработке кода для создания оперативной памяти, рассмотрим основные типы памяти.

Основные типы оперативной памяти

Существует три основных типа оперативной памяти:

- **однопортовая (Single Port, SP)**;
- **простая двухпортовая (Simple Dual Port , SDP)**;
- **полная двухпортовая (True Dual Port, TDP)**.

Проанализируйте схемы, чтобы увидеть, как выглядят порты ядер памяти. На схемах показано, как ОЗУ будут подключаться к проекту (рис. 5.9):

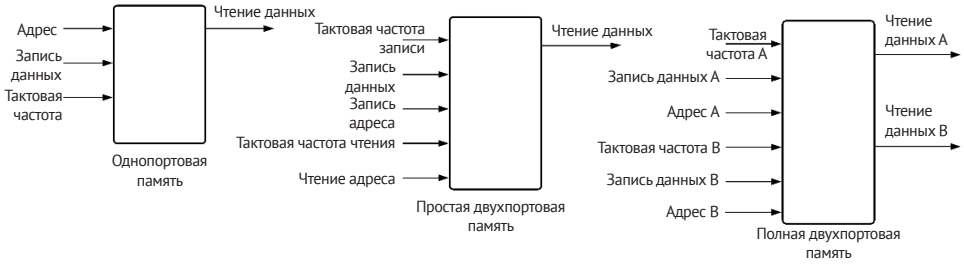


Рис. 5.9. Типы оперативной памяти

Блоки оперативной памяти внутри Artix 7 довольно сложные. Они поддерживают конфигурируемую ширину данных, опцию `byte enable` и коды исправления ошибок. Рекомендуется прочитать руководство по ресурсам памяти 7-й серии (ссылка в разделе «Дополнительное чтение»). Существуют способы обойти, казалось бы, непреодолимые инженерные проблемы, например как эффективно реализовать ОЗУ объемом 64 КБ, поняв, что различные конфигурации ОЗУ могут, по сути, решить эту проблему.

Однопортовые ОЗУ

Однопортовое ОЗУ – это самый простой тип. Это ОЗУ имеет один адресный порт, что означает, что оно может читать из памяти и записывать в память в одно место в течение тактового цикла. Этот тип ОЗУ легко создать с помощью SystemVerilog:

```
localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0]         memory[MEM_DEPTH];
logic [$clog2(MEM_DEPTH)-1:0] address;
logic [MEM_WIDTH-1:0]         write_data, read_data;
logic wren;
initial memory = '{default: '0};
always @(posedge clk) begin
  if (wren) memory[address] <= write_data;
  read_data <= memory[address];
end
```

Код выше показывает, как сконфигуровать однопортовую оперативную память. Параметр `WIDTH` в 8 бит, параметр `DEPTH` установлен в 256 8-битных байтов. Запись в память происходит при единичном значении параметра `wren`.

Чтение из памяти происходит на основе адресного входа на каждом цикле. Начальная инициализация данной памяти необязательна. Блочные ОЗУ и память SLICEM могут иметь начальное значение, тогда как UltraRAM – нет.

Необходимо учитывать несколько важных моментов.

- Память должна быть объявлена как неупакованная.

```
logic [MEM_WIDTH-1:0]         memory[MEM_DEPTH];
```

- Можно проинициализировать все типы памяти, за исключением UltraRAM, которая отсутствует в Artix 7. Для сохранения портативности не следует инициализировать очень большие блоки ОЗУ.
- Блочные ОЗУ должны иметь синхронное чтение данных. Распределенные ОЗУ (SLICEM) могут иметь асинхронные порты чтения. Наличие асинхронного порта чтения значительно сокращает время доступа к памяти.

> **Подсказка**
>
> Использование ‘{} облегчает инициализацию разреженных распакованных массивов. Можно инициализировать отдельные места, используя их адрес и значения по умолчанию для всех остальных. Например, запись `{0: 8’hFF, 7’h40, default: ‘0}; позволяет проинициализировать память по адресу 0 значением 0xFF, по адресу 1 – значением 0x40, а остальные адреса – нулями.

Однопортовые запоминающие устройства могут одновременно считывать или записывать только по одному адресу в единицу времени. Простая двухпортовая память немного более гибкая, поскольку обеспечивает независимый контроль над адресами чтения и записи, поэтому можно читать данные из одного места, а записывать в другое. Это особенно полезно при создании FIFO.

Простая двухпортовая оперативная память

Простая двухпортовая оперативная память управляется двумя тактовыми сигналами – тактовым сигналом для чтения и отдельным сигналом для записи. Это частный случай **полного двухпортового ОЗУ**, в котором один порт записи и один порт чтения используют тактовые сигналы с одинаковой или разной частотой. Такие типы ОЗУ часто используются для обеспечения гибкости в случае FIFO или когда данные должны быть считаны с адресов, отличных от тех, где в данный момент происходит запись. Эти типы ОЗУ также довольно просты для разработки.

```
localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0]         memory[MEM_DEPTH];
logic [$clog2(MEM_DEPTH)-1:0] wr_address, rd_address;
logic [MEM_WIDTH-1:0]         write_data, read_data;
logic wren;
initial memory = '{default: '0};
always @(posedge wr_clk)
  if (wren) memory[wr_address] <= write_data;
always @(posedge rd_clk) read_data <= memory[rd_address];
```

Простое двухпортовое ОЗУ очень похоже на однопортовое ОЗУ; его главное отличие состоит в разделении адресов записи и чтения. Обратите внимание на то, что `wr_clk` и `rd_clk` могут быть сигналами с одинаковой или разной тактовой частотой.

Эти первые два типа памяти используются чаще всего. Но есть еще один тип памяти, который используется реже.

Полная двухпортовая оперативная память

Полное двухпортовое ОЗУ обеспечивает полный доступ для чтения/записи с обоих портов. Существуют некоторые ограничения, относящиеся к тому, что произойдет, если при чтении или записи возникнут конфликты адресов. Обычно используют макрос `xilinx xpm_memory` при разработке полной двухпортовой оперативной памяти, но она также может быть определена с помощью SystemVerilog:

```
localparam MEM_DEPTH = 256;
localparam MEM_WIDTH = 8;
logic [MEM_WIDTH-1:0]         memory[MEM_DEPTH];
logic [$clog2(MEM_DEPTH)-1:0] address_a, address_b;
logic [MEM_WIDTH-1:0]         write_data_a, read_data_a;
logic [MEM_WIDTH-1:0]         write_data_b, read_data_b;
logic                         wren_a, wren_b;
initial memory = '{default: '0};
always @(posedge clk_a) begin
  if (wren_a) memory[address_a] <= write_data_a;
  read_data_a <= memory[address_a];
end
always @(posedge clk_b) begin
  if (wren_b) memory[address_b] <= write_data_b;
  read_data_b <= memory[address_b];
end
```

Приведенный фрагмент кода выглядит очень похоже на две однопортовые памяти. Разница в том, что оба блока always ссылаются на одно и то же хранилище, обеспечивая двойные порты в памяти.

Таким образом, было показано, как сконфигурировать все три типа памяти. Компания Xilinx предоставляет макросы для таких распространенных функций, как Clock Domain Crossing (CDC), FIFO и ОЗУ, в рамках своих функций **Xilinx Parameterized Macro (XPM)**.

> **Важное замечание**
>
> Vivado отобразит определенную в коде память на основе ее размера. При этом можно принудительно задать конкретную реализацию с помощью атрибута ram_style, например:
>
> `(* ram_style = «block» *) logic [7:0] memory[256]; // Блочная оперативная память`
>
> `(* ram_style = «distributed» *) logic [7:0] memory[256]; // Память на основе SLICEM`
>
> `(* ram_style = “registers” *) logic [7:0] memory[256]; // Использование регистров`

Рассмотрим предоставляемые Xilinx шаблоны.

Создание памяти с помощью шаблонов xpm_memory

Все функции XPM находятся в `<vivado install>/data/ip/xpm`. Рассмотрим, как реализован шаблон `xpm_memory`. В этом файле есть шесть вариантов описания памяти:

- `xpm_memory_dpdistram`: двухпортовая распределенная оперативная память;
- `xpm_memory_dprom`: двухпортовое ПЗУ;
- `xpm_memory_sdpram`: простое двухпортовое ОЗУ;
- `xpm_memory_spram`: однопортовое ОЗУ;
- `xpm_memory_sprom`: однопортовое ПЗУ;
- `xpm_memory_tdpram`: полное двухпортовое ОЗУ.

Если нужно использовать полное двухпортовое ОЗУ, то следует изучить этот файл и использовать `xpm_memory_tdpram`.

Использование IP-каталога для создания памяти

Последний вариант – использовать IP-каталог для создания памяти с определенной функциональностью. Не рекомендуется использовать этот вариант, поскольку он ограничивает возможности по работе с новыми семействами FPGA или с FPGA других производителей без формирования новых компонентов.

В предыдущем разделе было рассмотрено, как сконфигурировать или создать экземпляр памяти. В следующем разделе предстоит сделать выбор, каким образом будет происходить захват аудиоданных.

Захват аудиоданных

Теперь, зная как создать оперативную память, можно определить ее для захвата аудиоданных:

```
// ОЗУ для хранения аудиоданных
logic [6:0]                  amplitude_store[RAM_SIZE];
logic [$clog2(RAM_SIZE)-1:0] ram_wraddr;
logic [$clog2(RAM_SIZE)-1:0] ram_rdaddr;
logic                        ram_we;
logic [6:0]                  ram_dout;
always @(posedge clk) begin
  if (ram_we) amplitude_store[ram_wraddr] <= amplitude;
  ram_dout <= amplitude_store[ram_rdaddr];
end
```

Чтобы активировать захват, можно воспользоваться одной из кнопок:

```
// Захват аудиоданных
always @(posedge clk) begin
  button_csync <= button_csync << 1 | BTNC;
  ram_we <= '0;
  for (int i = 0; i < 16; i++)
    if (clr_led[i]) LED[i] <= '0;
  if (button_csync[2:1] == 2'b01) begin
    start_capture <= '1;
    LED <= '0;
```

```
    end else if (start_capture && amplitude_valid) begin
      LED[ram_wraddr[$clog2(RAM_SIZE)-1:$clog2(RAM_SIZE)-4]] <=
'1;
      ram_we <= '1;
      ram_wraddr                  <= ram_wraddr + 1'b1;
     if (&ram_wraddr) begin
        start_capture <= '0;
        LED[15]       <= '1;
      end
    end
  end // always @ (posedge clk)
```

Код выше ожидает нажатие центральной кнопки, а затем запускает счетчик для захвата выборок RAM_SIZE. Хорошо бы дать пользователю обратную связь, поэтому старшие биты адреса используются для того, чтобы зажигать светодиоды по одному.

После захвата аудиоданных нужно что-нибудь с ними сделать, чтобы продемонстрировать, что в памяти что-то появилось. Для этого можно воспользоваться аудиовыходом платы Nexys A7.

```
// Воспроизведение аудио
always @(posedge clk) begin
  button_usync <= button_usync << 1 | BTNU;
  m_clk_en_del <= m_clk_en;
  clr_led      <= '0;
  if (button_usync[2:1] == 2'b01) begin
    start_playback <= '1;
    ram_rdaddr     <= '0;
  end else if (start_playback && m_clk_en_del) begin
    clr_led[clr_addr] <= '1;
    AUD_PWM_en <= '1;
    if (amplitude_valid) begin
      ram_rdaddr <= ram_rdaddr + 1'b1;
      amp_counter <= 7'd1;
      amp_capture <= ram_dout;
      if (ram_dout != 0) AUD_PWM_en <= '0;
    end else begin
      amp_counter <= amp_counter + 1'b1;
      if (amp_capture < amp_counter) AUD_PWM_en <= '0;
    end
    if (&ram_rdaddr) start_playback <= '0;
  end
end
assign AUD_PWM = AUD_PWM_en ? '0 : 'z;
```

В приведенном примере происходит ожидание нажатия кнопки, после чего осуществляется последовательный проход по памяти и выводятся данные. При этом светодиоды выключаются один за другим, чтобы показать, что происходят какие-то действия.

Есть один момент, который следует отметить при управлении динамиком. Его выход является выходом с открытым стоком. Это означает, что для того, чтобы подать сигнал на управляющую схему, на него надо подавать низкий уровень для 0, но, когда нужно, чтобы на выходе была «1», его надо перевести в разомкнутое состояние, а подтягивающий резистор на плате поднимет его до нужного уровня. Формирование линии с тремя состояниями происходит следующим образом.

1. Определите выход как `wire`.
   ```
   output wire          AUD_PWM
   ```
2. Определите внутренний управляющий сигнал.
   ```
   logic                AUD_PWM_en;
   ```
3. Определите выход с тремя состояниями.
   ```
   assign AUD_PWM = AUD_PWM_en ? '0 : 'z;
   ```

Когда `AUD_PWM_en` установлен в логическую 1, на выходе будет 0. Когда он установлен в логический 0, на выходе будет третье состояние.

Теперь, когда можно управлять разъемом для наушников, нужно посмотреть на формат вывода. Так же как на входе используется PDM, на выходе используется **PWM** (**широтно-импульсная модуляция, ШИМ**). В PDM генерировалась строка из единиц и нулей, которые можно было считать в течение определенного периода времени, чтобы определить амплитуду сигнала.

Теперь нужно превратить амплитуду в PWM-сигнал. Это довольно простой процесс. Это можно сделать, создав 7-битный счетчик[1], и, сравнивая значение счета с амплитудой, отправлять единицу до тех пор, пока значение счета меньше амплитуды:

```
if (button_usync[2:1] == 2'b01) begin
start_playback <= '1;
ram_rdaddr     <= '0;
end else if (start_playback && m_clk_en_del) begin
clr_led[clr_addr] <= '1;
AUD_PWM_en <= '1;
if (amplitude_valid) begin
ram_rdaddr <= ram_rdaddr + 1'b1;
amp_counter <= 7'd1;
amp_capture <= ram_dout;
if (ram_dout != 0) AUD_PWM_en <= '0; // Активация подтягивания сигнала к 1
end else begin
amp_counter <= amp_counter + 1'b1;
if (amp_capture < amp_counter) AUD_PWM_en <= '0;
// Активация подтягивания сигнала к 1
end
if (&ram_rdaddr) start_playback <= '0;
end
```

Теперь пришло время собрать и опробовать созданный проект на плате. Нажмите на *центральную* кнопку. Светодиоды должны включиться один за другим. После завершения нажмите кнопку *верх*, и светодиоды один за другим погаснут. Если подключить наушники, то можно услышать некоторый шум. Если постукивать по микрофону, можно услышать постукивания при воспроизведении.

В проекте 5 было показано, как захватить данные PDM и сохранить их в оперативной памяти. Также было показано, как воспроизводить данные с помощью PWM через аудиопорт. Это всего лишь введение, и у вас есть много возможностей для усовершенствования проекта. Имея FPGA с сотнями блоков DSP и оперативной памятью, можно добавить аудиоэффекты, воспроизводить звуки в обратном направлении, усиливать звук, фильтровать и т. д.

---

[1] Счетчик 7-битный, так как одна выборка аудиоданных соответствует 128 тактам рабочей частоты. – *Прим. ред.*

Проект 6. Использование датчика температуры

На плате Nexys A7 установлен датчик температуры Analog Devices ADT7420. Для связи с этой микросхемой используется стандартный интерфейс I2C. Этот двухпроводной интерфейс применяется в основном для низкоскоростных устройств. Его преимущество в том, что он позволяет подключать несколько микросхем через один и тот же интерфейс и обращаться к ним по отдельности. В данном случае он будет использоваться для простого считывания текущей температуры с устройства и отображения значения на семисегментном индикаторе.

Первый шаг – это разработка интерфейса I2C. В главе 7 «Введение в AXI» будет рассмотрено проектирование интерфейса I2C общего назначения, а пока воспользуемся тем, что ADT7420 работает в режиме, в котором можно получать данные о температуре, считывая два расположения фронта в информационном сигнале. Сначала давайте посмотрим на временную диаграмму для шины I2C и на цикл чтения, который будет использоваться:

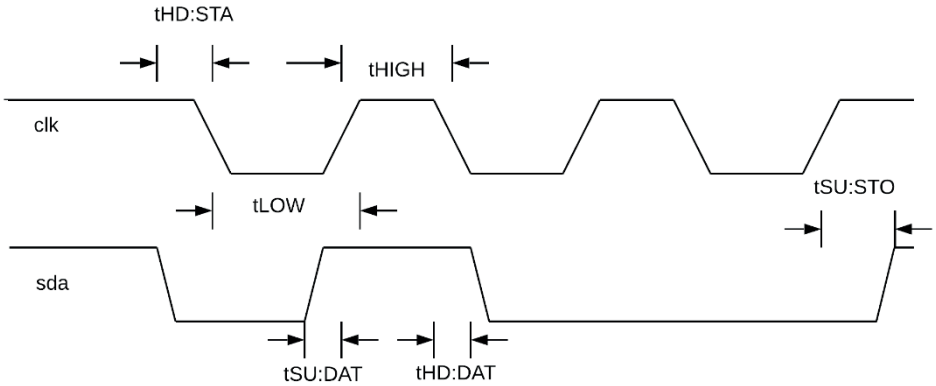


Рис. 5.10. Временная диаграмма I2C

Из временной диаграммы следует, что на ней обозначены интервалы времени установки и удержания сигнала, которые уже встречались ранее в уже выполненных проектах. Также есть минимальная длительность тактового импульса[1], которую нужно поддерживать. Можно определить параметры для обеспечения этих требований.

Следующий код находится в файле `CH5/hdl/i2c_temp.sv`:

```
localparam TIME_1SEC = int'(INTERVAL/CLK_PER); // Количество тактов в 1 с
 localparam TIME_THDSTA = int'(600/CLK_PER);
 localparam TIME_TSUSTA = int'(600/CLK_PER);
 localparam TIME_THIGH  = int'(600/CLK_PER);
 localparam TIME_TLOW   = int'(1300/CLK_PER);
 localparam TIME_TSUDAT = int'(20/CLK_PER);
 localparam TIME_TSUSTO = int'(600/CLK_PER);
 localparam TIME_THDDAT = int'(30/CLK_PER);
```

[1] Требования к временным параметрам шины и их обозначения (например, время установки и время удержания импульса «старт» tSU:STA и tHD:STA, длительности низкого и высокого состояния tHIGH и tLOW тактового импульса и др.) взяты автором из документации на датчик AD7420 (см. ссылку в конце главы). – *Прим. ред.*

Рекомендуется посмотреть на конечный автомат в модуле `i2c_temp`. Этот конечный автомат управляет доступом к датчику температуры на плате Nexys A7. Алгоритм работы автомата довольно прост.

1. Подождать 1 с.
2. Отправить команду Start по линиям SDA/SCL. Затем передать команду на чтение данных с датчика температуры и прочитать два 8-разрядных регистра, содержащих текущую температуру в градусах Цельсия.
3. Повторять до тех пор, пока не будет передана нужная информация и не будут получены данные обратно.
4. Передать команду Stop и вернуться к шагу 1.

Для доступа к датчику температуры определено три состояния шины:

- передача предопределенных сигналов запуска, адреса, чтения и остановки;
- переход шины в третье состояние во время циклов ACK и циклов данных;
- захват данных с шины SDA.

Конечный автомат обеспечивает взаимодействие с датчиком температуры и возвращает данные. Теперь нужно найти способ отобразить эти данные так, чтобы человек мог их понять.

Обработка данных

Требуется определить, как лучше всего отобразить температуру. ADT7420 возвращает данные в виде 16-битного значения:

```
[15:7] Integer  // целая часть
[6:3] fraction * 0.0625  // дробная часть
[2:0] Don't Care // незначащая часть
```

Можно использовать функцию `bin_to_bcd` для целочисленной части, чтобы сгенерировать данные для семисегментного индикатора, но что можно сделать для вычисления дробной части? Имеется всего 16 значений, поэтому можно создать таблицу поиска и просто искать младшие 4 цифры[1]. Это фактически создает ПЗУ, которое может быть проиндексировано. ПЗУ создается во многом так же, как и ОЗУ:

```
logic [15:0] fraction_table[16];
initial begin
  for (int i = 0; i < 16; i++) fraction_table[i] = i*625;
end
```

Затем можно преобразовать температуру, основываясь на выходе микросхемы датчика температуры:

```
// преобразовать температуру
always @(posedge clk) begin
  convert_frac <= convert;
```

[1] Данные по температуре датчика ADT7420 в представленном виде (13 информационных бит — режим по умолчанию) имеют разрешение дробной части в 1/16 (0,0625) градуса Цельсия. Для перевода дробной части в десятичный вид необходимо ее умножить на коэффициент 10/16 и еще на 1000 для представления в виде целого числа. Отсюда коэффициент 625 в коде формирования таблицы для определения дробной части. – *Прим. ред.*

```
    if (convert) begin
      encoded_int  <= bin_to_bcd(temp_data[15:7]); // целая часть
      fraction     <= bin_to_bcd(fraction_table[temp_data[6:3]]);
     decimal      <= 8'b00010000;
    end
  end // always @ (posedge clk)
  assign encoded = {encoded_int[3:0], encoded_frac[3:0]};
```

Одним из недостатков преобразования температуры каждую секунду при такой точности дробной части является то, что отображение на дисплее может довольно сильно меняться в зависимости от условий окружающей среды. Можно применить к данным своего рода фильтр, чтобы получать среднюю температуру за определенный период времени.

Теперь, когда было изучено, как обрабатывать данные, давайте узнаем больше о том, как можно фильтровать и улучшать качество этих данных.

Сглаживание данных

В системе счисления по основанию 10 можно довольно просто разделить на число, кратное 10. Каждое следующее число, кратное десяти, представляет собой просто сдвиг вправо на одну цифру:

```
12345/10  = 1234.5 Усеченное = 1234, Округленное = 1235
12345/100 = 123.45 Усеченное = 123,  Округленное = 123
```

Аналогично в двоичном исчислении каждый сдвиг вправо – это деление на 2:

```
10110>>1 = 1011.0 Усеченное = 1011, Округленное = 1011
10110>>2 = 101.10 Усеченное = 101,  Округленное = 110
```

Как это помогает в фильтрации данных? Если требуется отфильтровать период из 2, 4, 8, 16, 32… 2^n образцов, то в двоичной/шестнадцатеричной форме операция деления практически не требуется, так как это просто сдвиг и возможное округление.

Используя приведенный факт, можно создать простой фильтр, суммируя данные о температуре за определенный период времени, как это показано на рисунке ниже.

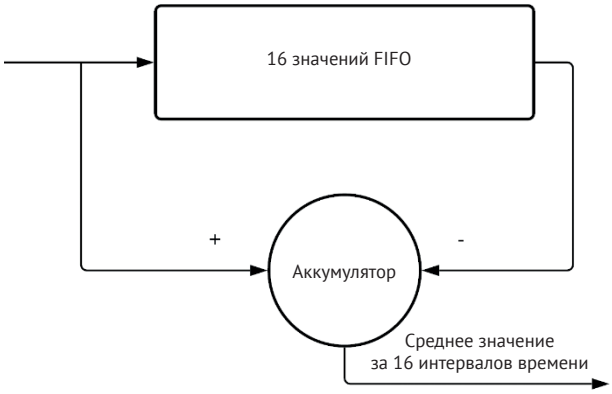


Рис. 5.11. Простой фильтр скользящего среднего

Способ создания фильтра скользящего среднего заключается в том, что сохраняется текущее среднее значение за определенный период времени. В предыдущем случае было выбрано 16 циклов, хотя любой показатель, равный степени двойки, является хорошим выбором. Вариант, не равный степени двойки, возможен, но пока не будет обсуждено, как улучшить этот фильтр в главе 6 «Математика, параллелизм и конвейерное проектирование», лучше его не рассматривать.

Принцип работы фильтра заключается в том, что входящие данные добавляются в аккумулятор и вычитается выходной результат за 16 тактов до этого. Тогда аккумулятор будет содержать сумму входящих данных за последние 16 тактов. Если разделить это число на 16, то будет получено среднее значение за последние 16 циклов.

Более глубокое погружение в FIFO

Ключевым элементом FIFO является оперативная память, указатели чтения и записи, а также логика генерации флагов. В синхронном FIFO это очень просто реализовать:

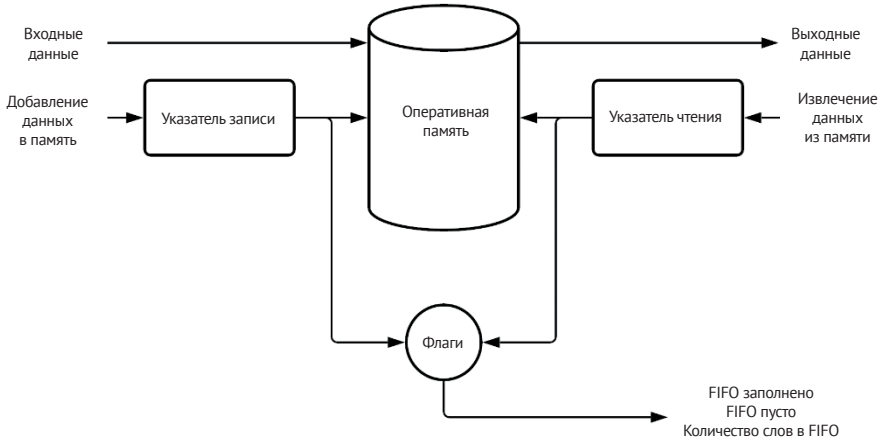


Рис. 5.12. Синхронный FIFO

Указатель записи увеличивается на единицу при каждом добавлении (push), а указатель чтения – при каждом извлечении (pop).

Генерация флагов сводится к сравнению указателей чтения и записи друг с другом. Когда имеется дело с синхронным FIFO, эти сравнения просты, поскольку все генерируется одним и тем же тактовым генератором и имеет четкую временную привязку. А как насчет асинхронного FIFO, т. е. FIFO с разными генераторами тактового сигнала для чтения и записи?

Вспомните рассуждения о синхронизации и многоразрядных шинах. Что произойдет, если попытаться сравнить адрес чтения и адрес записи в разные такты?

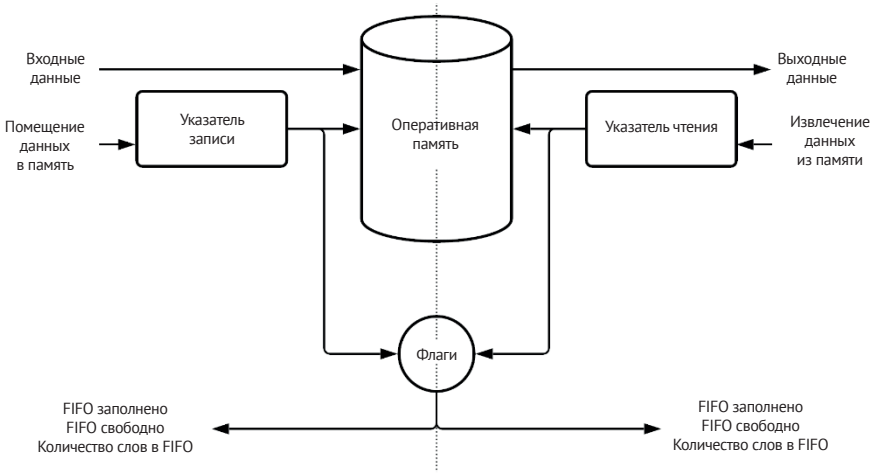


Рис. 5.13. Асинхронный FIFO (нефункциональный)

Разница между синхронным FIFO и асинхронным FIFO показана пунктирной линией в середине рис. 5.13. Каждая половина представляет собой независимый тактовый домен. На рис. 5.12 все находится в одном тактовом домене.

Рассмотрим указатель чтения и записи в разных тактовых доменах. Предположим, что генераторы тактовых сигналов не связаны между собой и что глубина FIFO равна 16 слов (ячеек памяти) с 4-битными адресами:

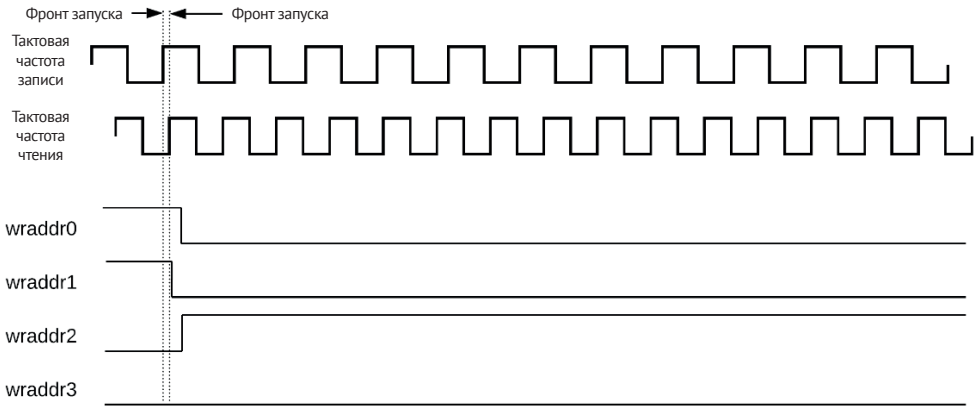


Рис. 5.14. Адресация в FIFO

Глядя на представленную диаграмму, можно понять, что, когда в счетчике одновременно изменяется несколько битов, а тактовые сигналы асинхронны, невозможно определить, каким будет захваченный адрес.

Эту проблему можно решить с помощью кодирования указателей адресов кодом Грея.

Код Грея

Чтобы получить код Грея, двоичный счет увеличивается путем добавления 1 к младшему биту. В результате получается следующий счет:

00 - 01 - 10 - 11.

Код Грея позволяет во всем двоичном представлении числа изменять только один бит за раз, например как в следующей последовательности:

00 - 01 - 11 – 10.

> **Важное замечание**
>
> Счетчики с кодировкой Грея имеют ограниченный диапазон, так как они всегда должны изменять только один бит за раз. Степень 2 всегда безопасна для реализации, но и другие комбинации, такие как $2^n + 2^m$, также работают.

Давайте рассмотрим FIFO с использованием кода Грея:

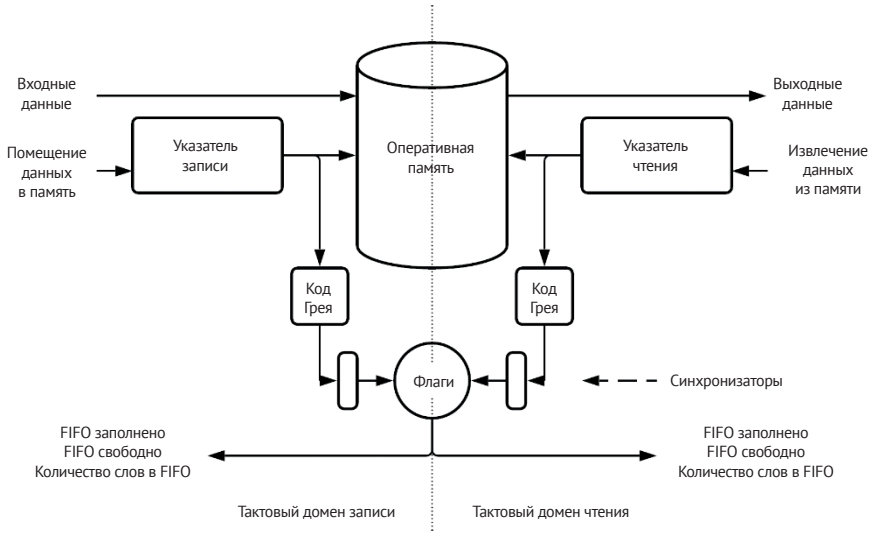


Рис. 5.15. Асинхронный FIFO с кодированием Грея

Добавив модуль кода Грея и модули синхронизации в каждый тактовый домен, можно сравнивать закодированные кодом Грея значения друг с другом или преобразовывать их обратно в двоичный код на требуемой тактовой частоте.

Поскольку коды Грея позволяют изменять только один бит за раз, будет гарантированно захвачено либо старое, либо новое значение и не будет зафиксировано переходное значение, которое может привести к ошибке пустого или полного FIFO или к ошибке в количестве слов.

Ограничения

Бывает необходимо использовать ограничение `set_max_delay` между указателем записи и первым регистром в блоке сихронизации.

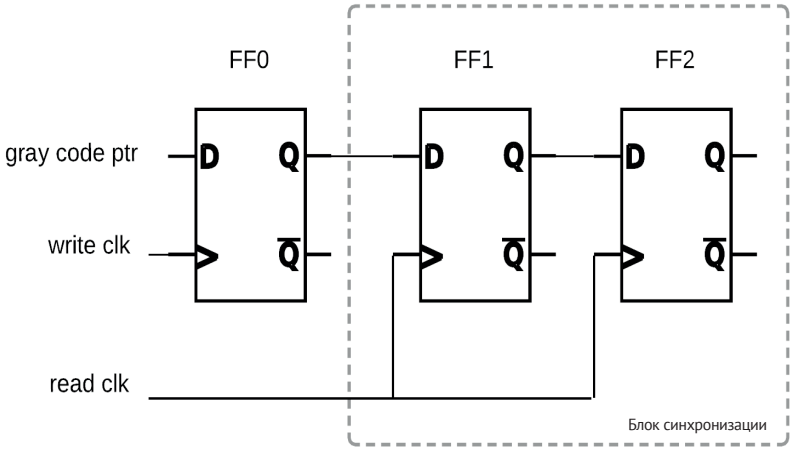


Рис. 5.16. Использование set_max_delay

Взглянув на код Грея и логику блока синхронизации на стороне записи (аналогичная схема есть на стороне чтения), можно увидеть, где нужно применить временное ограничение `set_max_delay`:

```
Set_max_delay -datapath_only <delay> -from [get_pins FF0/Q] -to
[get_pins FF1/D] -datapath_only
```

Для обеспечения абсолютной безопасности задержка должна быть установлена равной периоду тактовой частоты назначения или меньше. Она может составлять до двух тактовых импульсов назначения, но для безопасности используйте от 1 до 1,5 периода тактовой частоты назначения.

Генерация FIFO

Будет использоваться синхронный FIFO, но понимание того, как работает асинхронный FIFO, крайне важно, если вы решите строить карьеру, работая с FPGA. Почти наверняка вам зададут вопрос об этом на собеседовании.

Преимущество работы с Vivado заключается в том, что компания Xilinx создала макрос `xpm_fifo_(sync | async)`. Его реализация описана в `i2c_temp.sv`. Если открыть файл, то можно увидеть, что есть много портов, которые не используются. Данные в FIFO будут отправляться при каждом сигнале преобразования. Также есть небольшой конечный автомат, который генерирует данные для преобразования и отправки на интерфейс семисегментного индикатора.

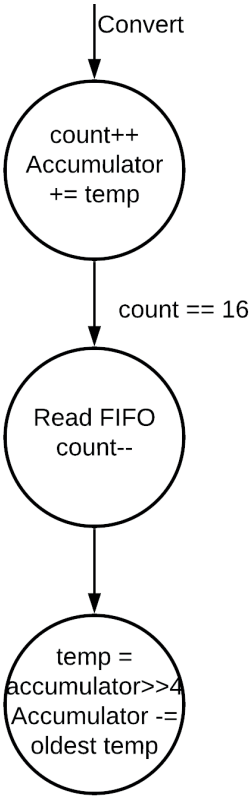


Рис. 5.17. Простой конечный автомат фильтра показаний температуры

Взгляните на конечный автомат, он очень прост. С помощью FIFO создается буфер для хранения 16 отсчетов. По мере поступления каждого нового отсчета увеличивается значение аккумулятора. Достигнув 16 отсчетов, значение аккумулятора делится на 16, и это является средней температурой за предыдущие 16 с временны́х интервалов.

```
always @(posedge clk) begin
  rden <= '0;
  rden_del       <= rden;
  smooth_convert <= '0;
  if (convert) begin
    smooth_count              <= smooth_count + 1'b1;
    accumulator               <= accumulator + temp_data;
  end else if (smooth_count == 16) begin
   rden <= '1;
    smooth_count            <= smooth_count - 1'b1;
  end else if (rden) begin
    accumulator             <= accumulator - dout;
  end else if (rden_del) begin
    smooth_convert          <= '1;
    smooth_data             <= accumulator >> 4;
  end
end
```

Проанализируйте представленный код и посмотрите, можете ли вы выделить последовательность состояний. Он разработан не так, как это делалось раньше, но вы должны быть в состоянии понять его последовательность. После того как вы проанализируете код, соберите его и опробуйте на плате.

Кое-что при выполнении проекта может удивить, а именно то, что в начале на выходе остается 0 в течение длительного периода времени, если быть точным – первые 16 с. Это происходит потому, что требуется дождаться первого заполнения FIFO. Что произойдет, если выводить значение аккумулятора/16 каждый цикл? Подумайте об этом немного или измените код и посмотрите.

Если всегда выводить значение аккумулятора/16, то получилось бы, что в начале работы схемы температура постепенно повышалась вверх от 1/16 текущей температуры до средней температуры в комнате. Для чего-то это некритично, и в данном случае любой метод будет приемлем, но что, если нужно чтобы всегда было текущее значение? Для этого придется дождаться главы 6 «Математика, параллелизм и конвейерное проектирование», где обсуждается представление чисел с фиксированной точкой[1].

Выводы

В этой главе было рассмотрено, как осуществить простую коммуникацию с внешним миром. Разработанная схема собирает данные с микрофона, сохраняет их и воспроизводит; также была изучена шина I2C – распространенный способ коммуникации с низкоскоростными устройствами. Было показано, как можно вывести число с фиксированной точкой на семисегментном индикаторе; проведено знакомство с FIFO и показано, как можно фильтровать данные, чтобы устранить шумы, возникающие при изменении температуры.

Интерфейс I2C используется для связи со многими низкоскоростными устройствами, такими как АЦП и ЦАП, и очень важен для многих проектов FPGA. На данном этапе вы должны чувствовать себя уверенно в данных вопросах. Создание более общей версии интерфейса будет рассмотрено в одной из последующих глав. Если вас интересуют аудиоданные, вы должны уже обладать определенной уверенностью в захвате, манипулировании и генерации аудио.

В следующей главе будут разобраны некоторые математические операции и разобрано, как можно без лишних сложностей вывести данные датчика температуры с помощью чисел с фиксированной точкой. Будут рассмотрены числа с плавающей точкой и операции, которые можно выполнять над аудиоданными.

Вопросы

1. В чем преимущества шины I2C?

   a) С ее помощью можно быстро перемещать большие объемы данных.
   b) Для связи нужны только два провода.
   c) Можно подключить несколько устройств, используя только два провода.

[1] В русском языке для разделения целой и дробной части используют запятую, а в английском языке – точку. Поскольку данная книга является переводом с английского языка, было решено оставить термины «фиксированная точка» и пр. – *Прим. ред.*

d) Все вышеперечисленное.
e) Только (b) и (c).

2. Какой порядок предпочтителен, когда необходима память?

   a) Использование IP-каталога, описание в коде, использование `xpm_memory`.
   b) Использование `xpm_memory`, использование IP-каталога, описание в коде.
   c) Описание в коде, использование `xpm_memory`, использование IP-каталога.
   d) Использование IP-каталога, использование `xpm_memory`, описание в коде.

3. Выражение `assign data = (data_en) ? 'z : '0;`

   a) Определяет множитель.
   b) Определяет регистр.
   c) Определяет ввод–вывод с тремя состояниями.

4. Код Грея используется в FIFO.

   a) Всегда.
   b) Для передачи информации счетчика между тактовыми доменами в асинхронных FIFO.
   c) Только в синхронных FIFO.

5. Какой тип памяти создает следующий код?

```
always @(posedge clk) begin
   if (wren) store[addr] <= din;
   dout <= store[addr];
end
```

   a) Простая двухпортовая память.
   b) Полная двухпортовая память.
   c) Однопортовая память.
   d) ПЗУ.

Дополнительное чтение

Обратитесь к следующим ссылкам для получения дополнительной информации о том, что было рассмотрено в этой главе.

- https://reference.digilentinc.com/reference/programmable- logic/nexys-a7/reference-manual.
- **Спецификация датчика температуры:** https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf.
- **Более подробная информация про FIFO**: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.
- https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

Глава 6

Математика, параллелизм и конвейеризация

Микропроцессоры – это специально разработанные ASIC, которые могут иметь очень высокую производительность при работе на очень высоких частотах – до 5 ГГц на момент создания этой книги. Многие такие процессоры являются процессорами общего назначения. Это означает, что они должны быть сбалансированы для выполнения широкого спектра задач. В отличие от них FPGA Artix 7, на которую ориентируется данная книга, может работать на частоте до 300–400 МГц. FPGA более высокого класса могут работать на частоте до 800 МГц. В отличие от микропроцессоров FPGA могут быть ориентированы на конкретное приложение. Благодаря этому можно использовать такие методы проектирования, как параллелизм, т. е. повторение логики для выполнения большего количества задач за один цикл тактового сигнала, чем это может сделать микропроцессор. Можно также использовать конвейеризацию для достижения высокой пропускной способности.

В этой главе будут более подробно рассмотрены числа с фиксированной точкой применительно к датчику температуры, а также числа с плавающей точкой, и будет показано, почему иногда лучше использовать числа с плавающей точкой вместо чисел с фиксированной точкой. Затем будут описаны ограничения использования чисел с плавающей точкой в FPGA (или в общем случае). Также будут изучены параллелизм и конвейерные конструкции на примере FFT[1] IP компании Xilinx, будет показано, как можно применить их для обработки аудиопотока.

[1] FFT (Fast Fourier Transform) – быстрое преобразование Фурье, БПФ. – *Прим. ред.*

К концу этой главы вы будете хорошо разбираться в математике с фиксированной и плавающей точкой. Вы получите представление о потоковой передаче AXI и о том, как с ее помощью можно соединить несколько компонентов. Это позволит показать особенности конфейеризации, один из двух способов повышения производительности FPGA. В данной главе также будет кратко описано, как FPGA используются в параллельных системах.

В этой главе будут рассмотрены следующие основные темы:

- арифметика с фиксированной точкой;
- числа с плавающей точкой одинарной и двойной точности;
- арифметика с плавающей точкой;
- конвейерные конструкции;
- параллельные конструкции.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH6.

Числа с фиксированной точкой

В этой книге было разобрано уже немало примеров с двоичными и BCD-числами. Двоичные числа отлично подходят для математики, потому что с такими числами сложение, вычитание и умножение реализуются дешево и просто. Деление тоже не так уж плохо, но требует больше времени. BCD-числа на самом деле использовались только для вывода на семисегментный индикатор.

В предыдущей главе нужно было ввести на семисегментный индикатор числа с фиксированной точкой. Вспомните формат данных, поступающих с датчика температуры:

```
[15:7] Integer  // целая часть
[6:3] fraction * 0.0625  // дробная часть
[2:0] Don't Care // не имеет значения
```

Если рассмотреть математические операции с такими числами, то можно узнать, что сложение двух чисел увеличивает разрядность результата на 1 бит, а для умножения двух чисел нужно сложить их разрядности. Вопрос в том, где будет находиться фиксированная точка в обоих случаях.

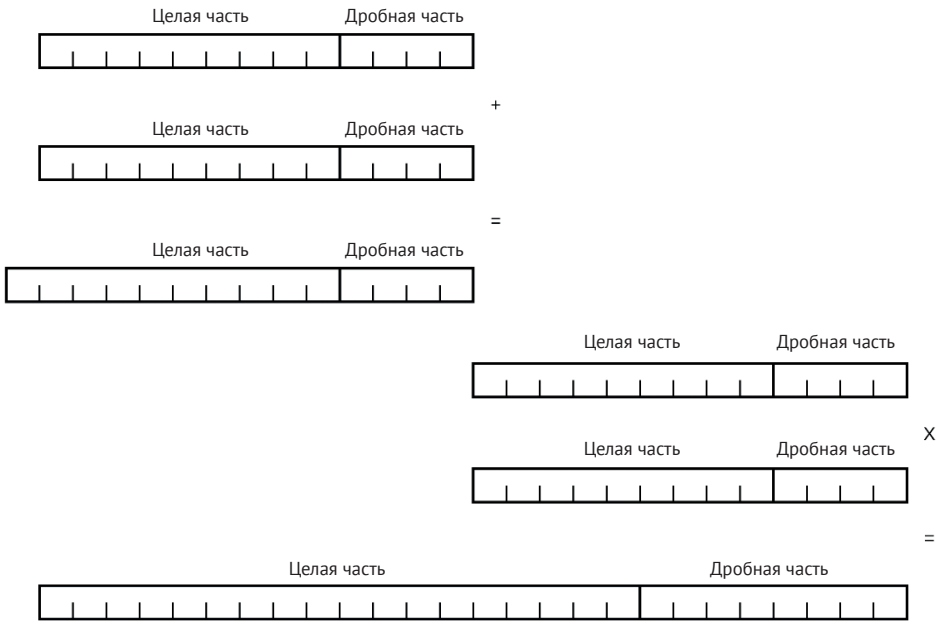


Рис. 6.1. Сложение/вычитание и умножение чисел с фиксированной точкой

Важно помнить, что при сложении двух чисел с фиксированной точкой разделяющая точка остается в том же месте. При умножении складывается количество битов как в целых частях, чтобы получить результирующую целую часть, так и в дробных частях, чтобы получить результирующую дробную часть. Это видно из диаграммы выше. Если перемножить два числа с разрядностью 9.4, в результате будет число с разрядностью 18.8.

Важно помнить, что при сложении необходимо следить за тем, чтобы точки обоих чисел были совмещены. При умножении такого требования нет.

Преимущество работы с числами с фиксированной точкой заключается в том, что такая же логика используется для математических операций в двоичном формате с фиксированной точкой. Разница лишь в том, где именно размещается десятичная точка.

Теперь, когда вы научились использовать арифметику чисел с фиксированной точкой, применим ее в датчике температуры.

Проект 7. Использование чисел с фиксированной точкой для обработки данных с датчика температуры

Давайте посмотрим, как можно оптимизировать усреднение температуры, чтобы справиться с первыми 16 с, когда температура вычисляется неправильно. Это происходит потому, что происходит деление недопустимого значения температуры на первые 15 тактов.

Бывают случаи, когда не может быть ни задержки, ни неточных результатов. Типичный вопрос на собеседовании: как убедиться, что выходной сигнал такого фильтра действителен в течение времени запуска? Так что вам его, возможно, когда-нибудь придется решать.

Использование арифметики чисел с фиксированной точкой для очистки времени запуска

Рассмотрим, как выглядит коэффициент масштабирования чисел с фиксированной точкой. Задача состоит в том, чтобы отмасштабировать результат до одного значения с датчика. Для этого значение надо масштабировать с помощью дроби. В следующей таблице показаны первые 15 циклов плюс устойчивое состояние в аккумуляторе. Следующая таблица содержит значения дроби, которые нужно вычислить:

Цикл	Значение accumulator	Усредняющий множитель
1	T0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1
2	T0 + T1 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	½
3	T0 + T1 + T2 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/3
4	T0 + T1 + T2 + T3 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	¼
5	T0 + T1 + T2 + T3 + T4 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/5
6	T0 + T1 + T2 + T3 + T4 + T5 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/6
7	T0 + T1 + T2 + T3 + T4 + T5 + T6 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/7
8	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/8
9	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + 0 + 0 + 0 + 0 + 0 + 0 + 0	1/9
10	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + 0 + 0 + 0 + 0 + 0 + 0	1/10
11	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + 0 + 0 + 0 + 0 + 0	1/11
12	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + 0 + 0 + 0 + 0	1/12
13	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + 0 + 0 + 0	1/13
14	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + 0 + 0	1/14
15	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + T14 + 0	1/15
16+	T0 + T1 + T2 + T3 + T4 + T5 + T6 + T7 + T8 + T9 + T10 + T11 + T12 + T13 + T14 + T15	1/16

Нужно умножить значение в аккумуляторе на дробное значение и сдвинуть его, чтобы получить масштабированный результат. Если это сделать, то вместо скачков температура останется довольно постоянной.

Для преобразования дробных значений в двоичное представление можно использовать следующий сайт: https://www.exploringbinary.com/binary-converter/. Для перевода дробей в десятичное представление сформирована следующая таблица преобразования:

```
divide[0]    = 17'b1_00000000_00000000; // 1
divide[1]    = 17'b0_10000000_00000000; // 1/2
divide[2]    = 17'b0_01010101_01010101; // 1/3
divide[3]    = 17'b0_01000000_00000000; // 1/4
divide[4]    = 17'b0_00110011_00110011; // 1/5
divide[5]    = 17'b0_00101010_10101010; // 1/6
```

```
divide[6]    = 17'b0_00100100_10010010; // 1/7
divide[7]    = 17'b0_00100000_00000000; // 1/8
divide[8]    = 17'b0_00011100_01110001; // 1/9
divide[9]    = 17'b0_00011001_10011001; // 1/10
divide[10]   = 17'b0_00010111_01000101; // 1/11
divide[11]   = 17'b0_00010101_01010101; // 1/12
divide[12]   = 17'b0_00010011_10110001; // 1/13
divide[13]   = 17'b0_00010010_01001001; // 1/14
divide[14]   = 17'b0_00010001_00010001; // 1/15
divide[15]   = 17'b0_00010000_00000000; // 1/16
divide[16]   = 17'b0_00010000_00000000; // 1/16
```

Используя знания, полученные из предыдущей главы, можно создать ПЗУ с помощью инициализации начальных значений памяти. Первое, что для этого нужно сделать, – это решить, сколько битов точности необходимо. **DSP 48** может выполнять умножение чисел с разрядностью 18x25 в дополнительном коде. Поэтому выбран 17-битный беззнаковый коэффициент масштабирования. Увеличение коэффициента может повлиять на количество используемых множителей или на скорость работы; уменьшение коэффициента не влияет на количество необходимых ресурсов, но может снизить точность.

Представленная таблица имеет формат 1.16, и значения в ней являются усеченными, а не округленными. Вы можете рассмотреть возможность округления значений. Округление двоичного числа осуществляется простым добавлением старшего бита, который нужно усечь, к битам, которые нужно оставить.

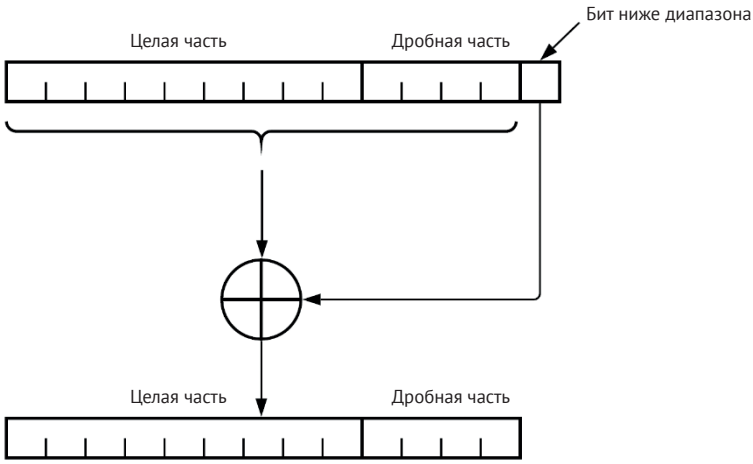


Рис. 6.2. Округление

0.00000, округленное до 4 бит, будет равно 0.0000.

0.00001, округленное до 4 бит, будет равно 0.0001.

0.11111, округленное до 4 бит, будет равно 1.0000.

Итак, имеются коэффициенты масштабирования, но как их использовать?

Изменим созданный конвейер так, что теперь значение аккумулятора вычисляется на каждом такте, при этом на каждом такте применяется новый коэффициент масштабирования, и данные выводятся на семисегментный индикатор. Раньше данные выводились только после накопления на протяжении 16 циклов.

```
always @(posedge clk) begin
  rden <= '0;
  smooth_convert <= '0;
  convert_pipe   <= convert_pipe << 1;
  if (convert) begin
    convert_pipe[0]           <= '1;
    smooth_count              <= smooth_count + 1'b1;
    accumulator               <= accumulator +
                                 temp_data[15:3];
  end else if (smooth_count == 16) begin
    rden                     <= '1;
   smooth_count             <= smooth_count - 1'b1;
  end else if (rden) begin
    accumulator              <= accumulator - dout;
  end else if (convert_pipe[2]) begin
    if (~sample_count[4]) sample_count <= sample_count + 1'b1;
      smooth_data             <= accumulator *
                                 divide[sample_count];
  end else if (convert_pipe[3]) begin
    smooth_convert           <= '1;
    smooth_data              <= smooth_data >> 16;
  end
end
```

Теперь конвейер работает всегда, поскольку есть `convert_pipe`. Что было изменено, так это то, что теперь происходит масштабирование значения аккумулятора на определенный коэффициент масштабирования. Вспомните обсуждение умножения, которое было ранее. Здесь добавлены 16 бит дробной части к масштабированному значению, поэтому в конце его нужно удалить. Это можно сделать, добавив этап `convert_pipe[3]`.

Можно запустить симуляцию и убедиться, что данные не меняются. Измените `testbench` для ввода постоянного значения 25 °C, 0x19 в шестнадцатеричном формате. После этого создайте загрузочный файл и опробуйте его на плате. Вместо нуля на протяжении 16 с ноль будет выводиться одну секунду, а затем начнется отображение данных. Данные должны быть достаточно постоянны в течение всего времени работы платы.

На основании этого проекта можно заметить, что арифметика с фиксированной точкой не требует больших затрат и очень проста в реализации. Конвейер занимает только четыре тактовых цикла для расчета сглаженной температуры за последние 16 циклов измерений.

Преобразование температуры с помощью арифметики с фиксированной точкой

Проект датчика температуры теперь удовлетворяет всем требованиям. Можно отображать температуру с разрешением до 1/16 °C, и при добавлении осред-

нения можно выводить ее в усредненном виде. Но в нем все еще не хватает кое-чего. Если вы находитесь за пределами США, то, скорее всего, вам неважно, какова температура в чем-либо, кроме градусов Цельсия, но в США люди упорно придерживаются имперских единиц измерений. Необходимо добавить конвертацию по Фаренгейту, чтобы можно было определить температуру в градусах по Фаренгейту.

Рассмотрим формулу, которая используется для перевода градусов Цельсия в градусы Фаренгейта:

$$T_{Fahrenheit} = \left(T_{Celsius} \times \frac{9}{5} \right) + 32.$$

Рис. 6.3. Формула для преобразования градусов Цельсия в градусы Фаренгейта

Формула несложная. Можно реализовать делитель и умножитель, но, поскольку 9/5 – константа, можно создать ее представление с фиксированной точкой, а затем умножать на константу. Вспомните, сколько времени требуется делителю, чтобы выполнить операцию деления. Умножение же можно выполнить за один цикл. Это подчеркивает одну важную вещь, о которой следует помнить. Часто существует несколько способов решения задачи. Первое или наиболее очевидное решение не всегда является наилучшим, поэтому полезно помнить о других способах решения задачи.

Как это обычно бывает, имеется выбор, где можем выполнить эту операцию. Чтобы все было просто и компактно, можно сделать это после того, как будет уменьшен размер промежуточного результата. Это приведет к уменьшению размера множителя.

Также понадобится способ выбора единиц измерения, Цельсия или Фаренгейта, поэтому добавлен `SW[0]` для управления градусами Цельсия/Фаренгейта и `LED[0]` для индикации отображения в градусах Фаренгейта.

Измените конвейер, чтобы можно было применять преобразование:

```
end else if (convert_pipe[3]) begin
  smooth_data    <= smooth_data >> 16;
  smooth_convert <= ~SW;
end else if (convert_pipe[4]) begin
  smooth_convert          <= SW;
  smooth_data             <= ((smooth_data * NINE_FIFTHS) >>
16) + (32 << 4);
end
```

Основное изменение заключается в том, что в `convert_pipe [3]` происходит преобразование данных, представленных в градусах по Цельсию в формат BCD в градусах по Фаренгейту с помощью `smooth_convert <= ~ SW. convert_pipe [4]`. Обратите внимание, что в этом случае используется больше преимуществ DSP 48, чем ранее, поскольку умножение и сложение выполняются в одном тактовом цикле.

Соберите проект и проверьте вывод температуры на семисегментный индикатор в формате градусов по Фаренгейту.

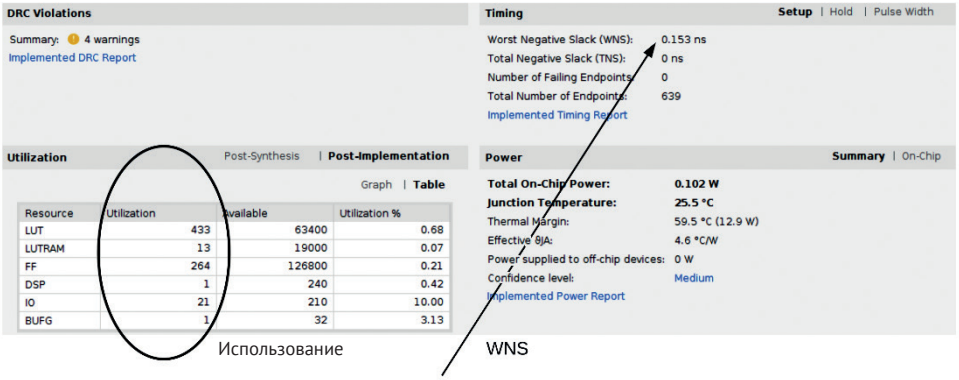


Рис. 6.4. i2c_temp, использование чисел с фиксированной точкой для преобразования температуры в градусы по Фаренгейту

При положительном значении WNS схема комфортно работает и использует очень мало ресурсов FPGA. Давайте посмотрим на конвейер преобразования, отображенный на вейформе.

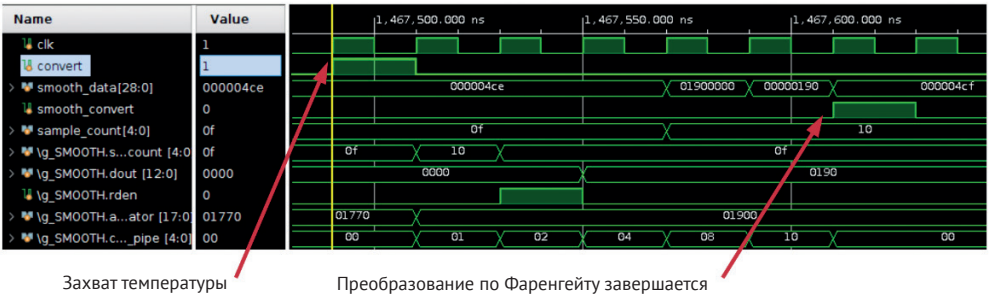


Рис. 6.5. Моделирование проекта считывания температуры по I2C с преобразованием в градусы по Фаренгейту

Из рисунка следует, что при реализации в формате использования чисел с фиксированной точкой конвейер короткий и потребляет небольшое количество ресурсов устройства.

Глядя на проект, видно, что не потребовалось модифицировать интерфейс I2C. На самом деле можно создать IP-ядро более общего назначения, которое можно использовать для подключения к другим устройствам I2C. Это будет рассмотрено в главе 7 «Введение в AXI».

А как насчет чисел с плавающей точкой?

Вы наверняка слышали о числах с плавающей точкой. Если числа с фиксированной точкой могут представлять только очень ограниченный, определенный диапазон дробных значений, то числа с плавающей точкой могут представлять числа от очень маленьких до очень больших, хотя их точность ограничена в зависимости от используемого стандарта. **Институт инженеров электротех-**

ники и электроники (Institute of Electrical and Electronics Engineers, IEEE) стандартизировал ряд форматов чисел с плавающей точкой. Компании, производящие графические карты, такие как Nvidia, также внесли свой вклад в стандарт за прошедшие годы.

Арифметические операции с плавающей точкой являются ресурсозатратными по сравнению с операциями с фиксированной точкой. Чтобы получить представление об этом, достаточно привести тот факт, что только в процессоре Pentium компания Intel стандартизировала интеграцию своего сопроцессора с плавающей точкой в основной микропроцессор. До появления процессора Pentium каждый x86-процессор, от 8086 до 80486, имел соответствующий сопроцессор x87 (8087, 80287 и т. д.), который обеспечивал операции с плавающей точкой.

Одной из причин выбора компанией Xilinx блоков DSP48 в своих разработках было желание улучшить поддержку операций с плавающей точкой в FPGA. Плавающая точка уже не является таким препятствием для использования, как это было раньше, хотя работа с ней по-прежнему в целом медленнее и сложнее, чем с фиксированной точкой.

Давайте рассмотрим IEEE-представление чисел с плавающей точкой одинарной и двойной точности[1]:

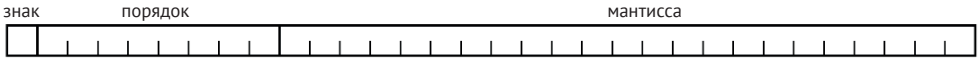


Рис. 6.6. IEEE-представление числа с плавающей точкой одинарной точности

Бит **знака** такой же, как и в числе с дополнительным кодом, т. е. «1» означает отрицательное значение, а «0» – положительное. Мантисса – это беззнаковое 24-битное число. Можно заметить, что определено только 23 бита. 24-й бит – это подразумеваемая 1, где число представлено как `1.дробная часть (мантисса)`. Это становится возможным, так как значение 0 представлено 32-битным полем, где все биты установлены в 0.

Порядок смещен на 127, поэтому фактический порядок равен –127, что дает значение от –126 до +127. Значения –127 и +128 зарезервированы.

Из этого можно сделать вывод, что плавающая точка – отличный выбор, когда у нас есть числа, расположенные относительно близко друг к другу, но с большим потенциальным диапазоном значений. Плавающая точка покрывает эти случаи, но за счет затрат дополнительных ресурсов и времени обработки. Реализация математических операций здесь более ресурсоемкая и медленная, чем с числами с фиксированной точкой, но если в одном наборе вычислений работа происходит в микроскопическом масштабе, а в другом – в галактическом, то можно использовать формат с одинарной точностью.

Двойная точность расширяет порядок до 11 бит, а мантиссу – до 52 бит.

В прошлом, если требовалось разработать что-то с использованием плавающей точки, нужно было создать свои собственные операторы с плавающей точкой. Пример оператора с плавающей точкой можно найти в проекте GPLGPU по адресу https://github.com/asicguy/gplgpu/tree/master/hdl/math.

---

[1] Подробнее о записи чисел с плавающей точкой (запятой) см. статьи в «Википедии»: «Число с плавающей запятой», «Число одинарной точности» и «Число двойной точности». – *Прим. ред.*

Рассмотрим основные компоненты для вычислений с плавающей точкой. Для этого анализа сосредоточимся на проектах GPLGPU с плавающей точкой, которые были реализованы в ASIC в 1998 году и повторно реализованы в FPGA в начале 2000-х годов. В зависимости от скорости, на которую вы ориентируетесь, конвейеризация может быть более или менее одинаковой, но это хороший базовый уровень и отправная точка для обсуждения.

Сложение и вычитание с плавающей точкой

Если обычно умножение является более медленным оператором для двоичных чисел или чисел с фиксированной точкой, то в случае чисел с плавающей точкой сложение/вычитание может потребовать больше времени. Причина этого в том, что требуется выравнивать десятичную точку, как если бы вычисления выполнялись вручную. Вспомним, что в мантиссе подразумевается 1 в целой части. Это означает, что после завершения сложения или вычитания нужно скорректировать порядок так, чтобы конечная мантисса имела вид `1.x`.

Умножение с плавающей точкой

Умножение с плавающей точкой не такое сложное. Достаточно сложить порядки и умножить мантиссы.

Обратное значение для числа с плавающей точкой

Целочисленное деление – это серия вычитаний, которые выполняются восстанавливающим и невосстанавливающим делением. Это позволяет получить точный ответ, хотя для больших целых значений могут потребоваться сотни таких циклов.

Подобно алгоритмам деления целых чисел, понадобится аналогичный алгоритм для умножения чисел с плавающей точкой[1]. Для этого подходит, например, метод **Ньютона–Рафсона**. Он состоит из подбора начального приближения из таблицы поиска, которое вычисляется предварительно, а затем путем последовательных приближений сводится к решению. Возможно, вы помните или слышали об ошибке деления в Pentium. Эта ошибка возникала из-за некорректных значений в таблице, которая использовалась при делении.

Более практичная библиотека операций с плавающей точкой

Для первого знакомства рекомендуется изучить или использовать функции в GPLGPU. Они лицензированы под GPL v3. Но давайте изучим, что есть у Xilinx для выполнения операций с плавающей точкой:

[1] Операция деления в этом подходе заменяется операцией умножения на обратное значение. – *Прим. ред.*

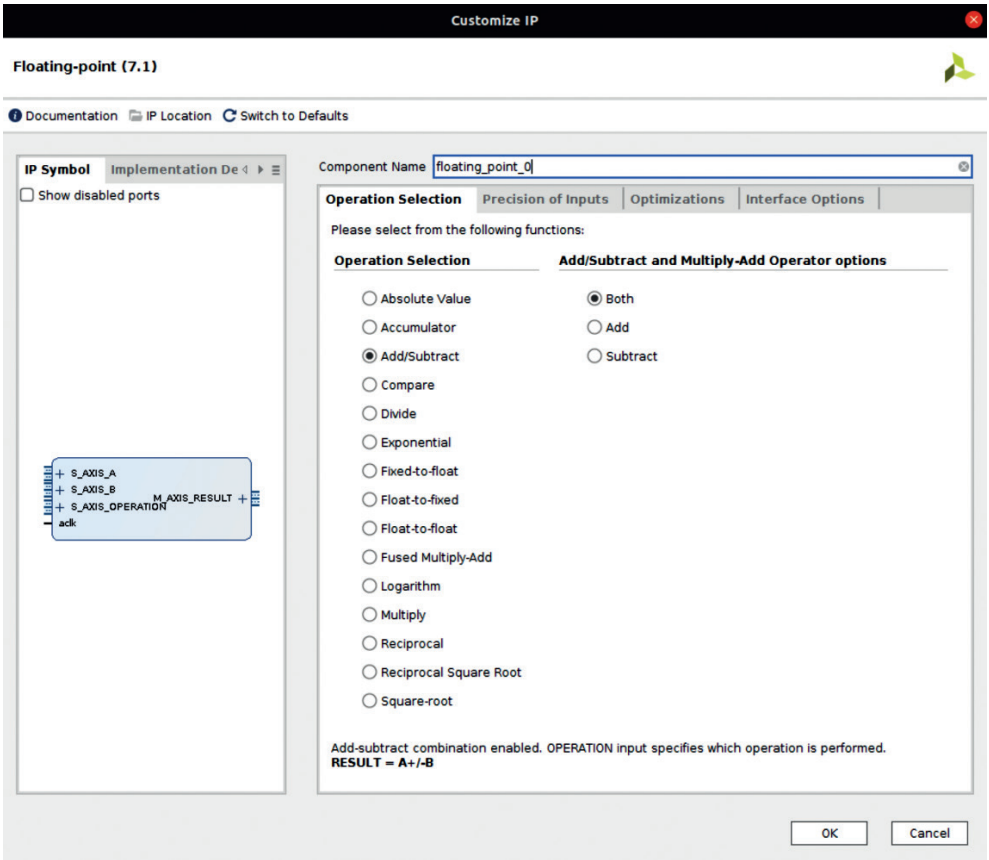


Рис. 6.7. Xilinx IP с плавающей точкой

Ранее каталог IP-блоков Xilinx уже упоминался. Наберите в поиске `floating`, и вы найдете мастер IP-блоков для чисел с плавающей точкой. Вы увидите, что Xilinx предоставляет в Vivado полный набор операторов.

Как и большая часть IP-блоков Xilinx, мастер чисел с плавающей точкой использует интерфейс AXI – в частности, потоковый интерфейс AXI (**AXI Stream**). Давайте рассмотрим его более подробно.

Краткий обзор потокового интерфейса AXI

IP-блоки Xilinx почти повсеместно перешли на использование шины AXI, которая представляет собой набор протоколов, определенных компанией ARM. Существует три варианта AXI; они будут рассмотрены более подробно в главе 7 «Введение в AXI». Но для использования IP-блоков Xilinx с плавающей точкой потребуется кратко рассмотреть потоковый интерфейс AXI[1].

[1] Используемое автором название «потоковый интерфейс AXI» (*AXI streaming interface*) соотвествует официальному термину AXI Stream. – *Прим. конс.*

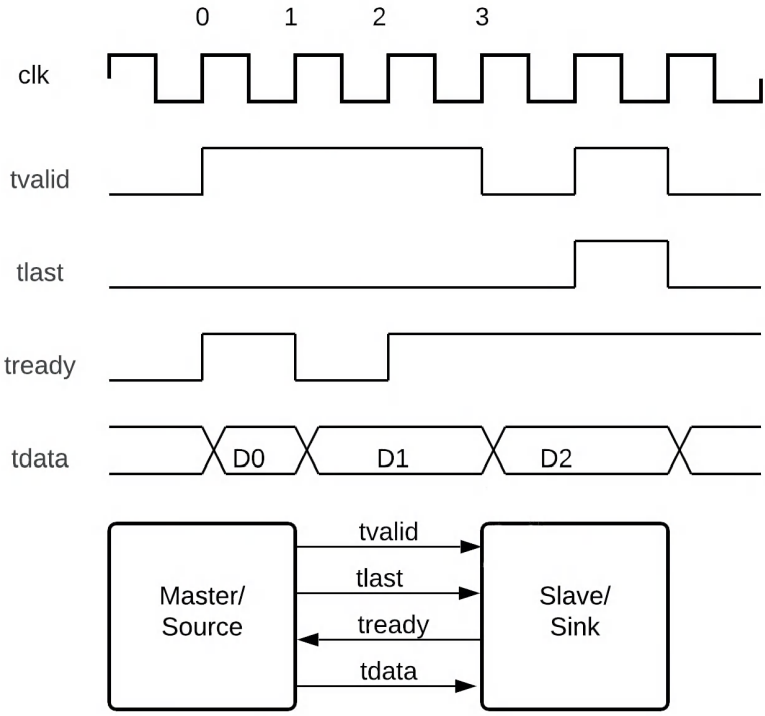


Рис. 6.8. Пример потокового интерфейса AXI

Проект 8. Обновление проекта датчика температуры до конвейерной реализации с плавающей точкой

Составим блок-схему предлагаемого проекта, чтобы определить, что требуется сделать.

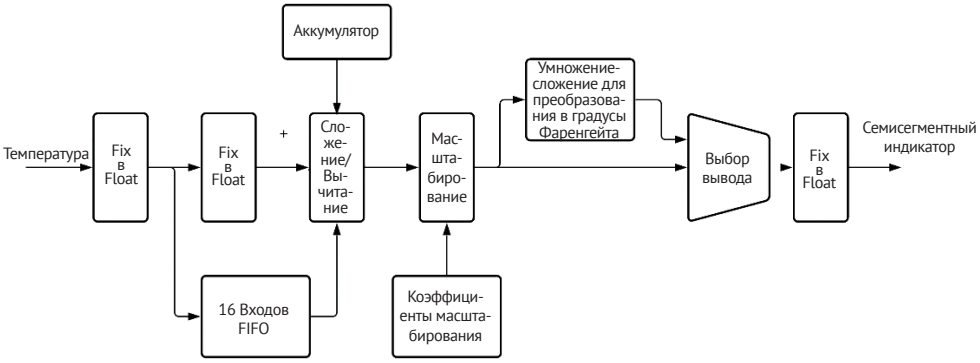


Рис. 6.9. Конвейер преобразования с плавающей точкой

Конвейер выглядит очень похожим на предыдущий конвейер температур. Основные отличия заключаются в том, что теперь числа с фиксированной точкой преобразуются в числа с плавающей точкой на входе и обратно на выходе. Внутренне старый конвейер из 4-5 этапов обрабатывается аналогично. Но теперь каждый этап занимает не один тактовый цикл, поскольку операции с плавающей точкой обрабатываются дольше.

Чтобы приспособить датчик температуры в том числе и для преобразования его показаний в градусы по Фаренгейту, понадобятся различные операции с плавающей точкой, которые можно сгенерировать из каталога IP-блоков Vivado, как будет показано в следующем разделе.

Преобразование чисел из представления с фиксированной точкой в формат с плавающей точкой

Потребуется сделать несколько изменений, чтобы настроить оператор `fix_to_float` для текущего случая.

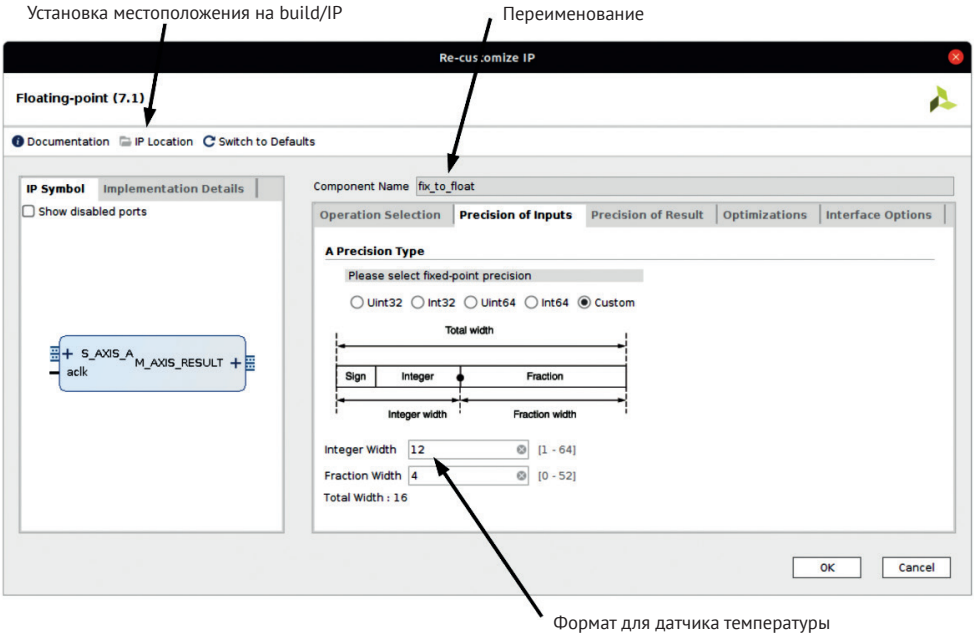


Рис. 6.10. Конфигурация формата fix_to_float

Вспомните, что формат датчика температуры – это 4 бита дробной части и 9 бит целой части. При проектировании для плат Xilinx лучше чтобы потоковый интерфейс был кратен 8, поэтому установим преобразование на 12.4.

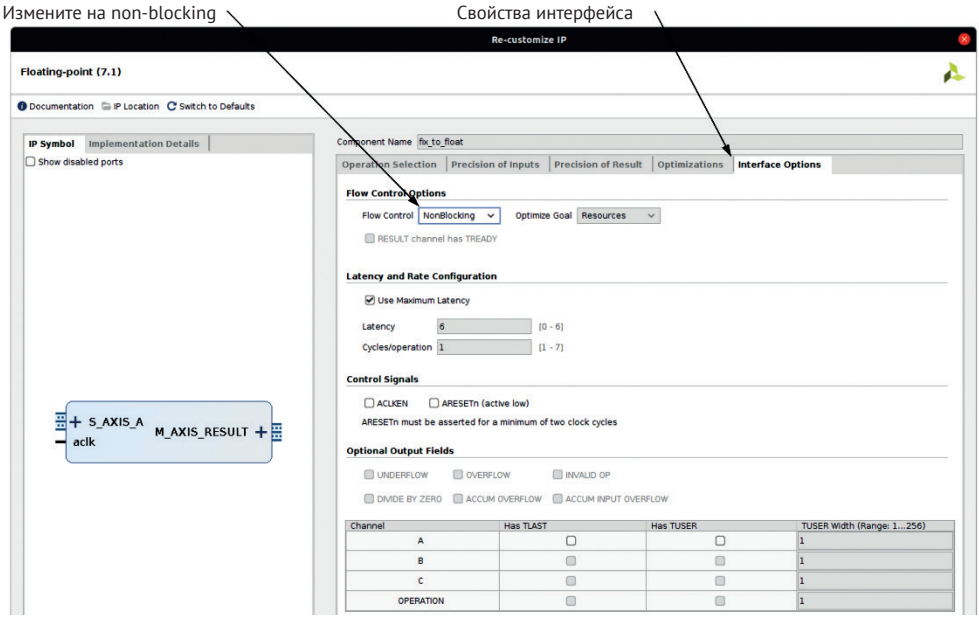


Рис. 6.11. Настройка интерфейса

Требуется изменить управление потоком, чтобы оно было неблокирующим. Это не является строго обязательным для данного проекта, но это приведет к большим затратам ресурсов при конвейеризации проекта для реализации вычислений с плавающей точкой. На этой панели также можно добавить некоторые дополнительные компоненты AXI: сигналы **tlast** и **tuser**. **Tlast** полезен, если нужно передавать большие объемы данных и требуется определить, когда закончится группировка данных. С другой стороны, **tuser** позволяет передавать информацию вместе с данными, чтобы можно было использовать ее в проекте.

Математические операции с плавающей точкой

Если построить конвейер с небольшой дополнительной логикой управления, то можно разделить сложение и вычитание, как это было сделано в случае с фиксированной точкой:

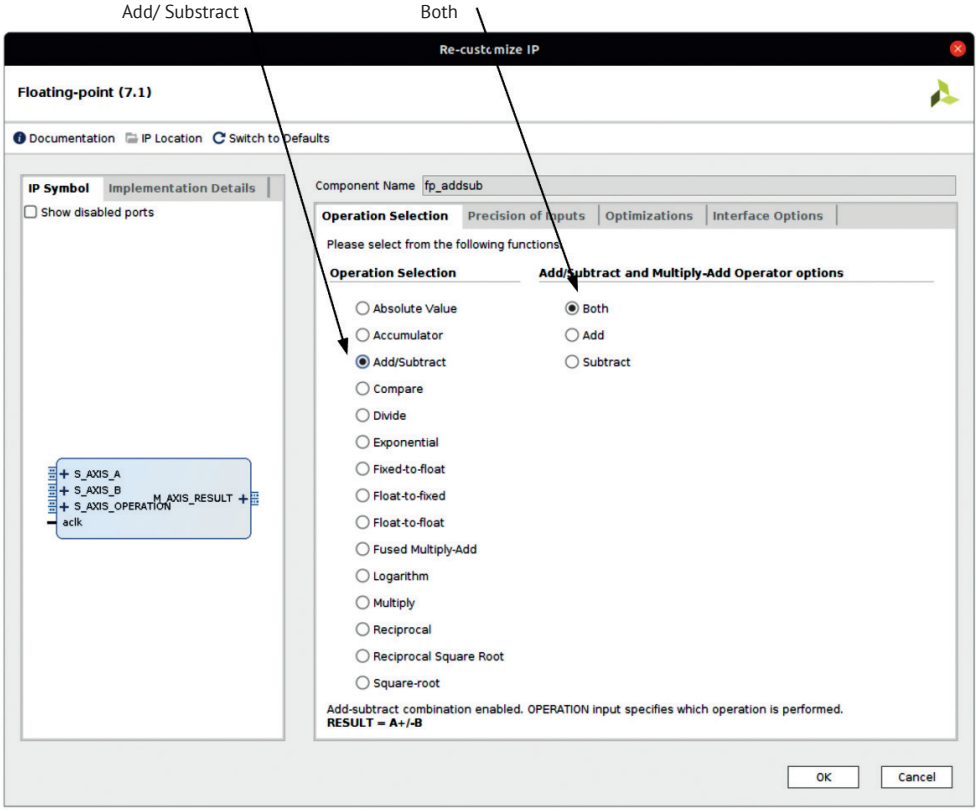


Рис. 6.12. Настройка сложения/вычитания

Убедитесь, что вы настроили интерфейс, как показано на рис. 6.11.

Для проекта еще понадобятся два дополнительных компонента: умножитель и умножение-сложение. Когда будете настраивать операцию умножения-сложения, необходимо убедиться, что выбрана только **Add**:

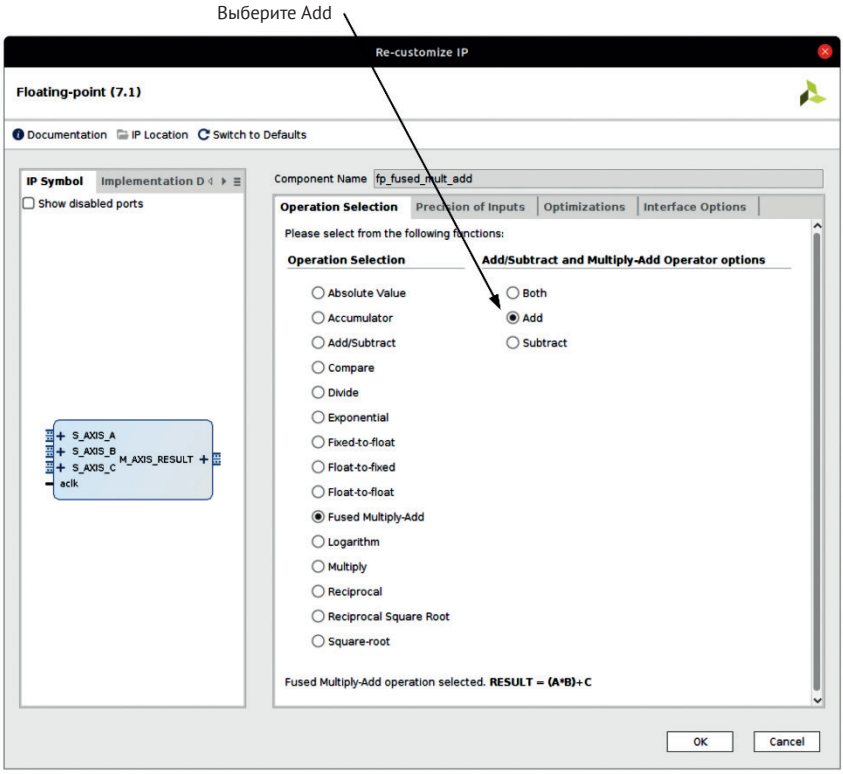


Рис. 6.13. Настройка операции умножения-сложения

И наконец, результат нужно будет преобразовать обратно в число с фиксированной точкой для отображения на семисегментном индикаторе.

Преобразование формата с плавающей точкой в формат с фиксированной точкой

Рассмотрим, что нужно сделать для создания выходных данных с фиксированной точкой.

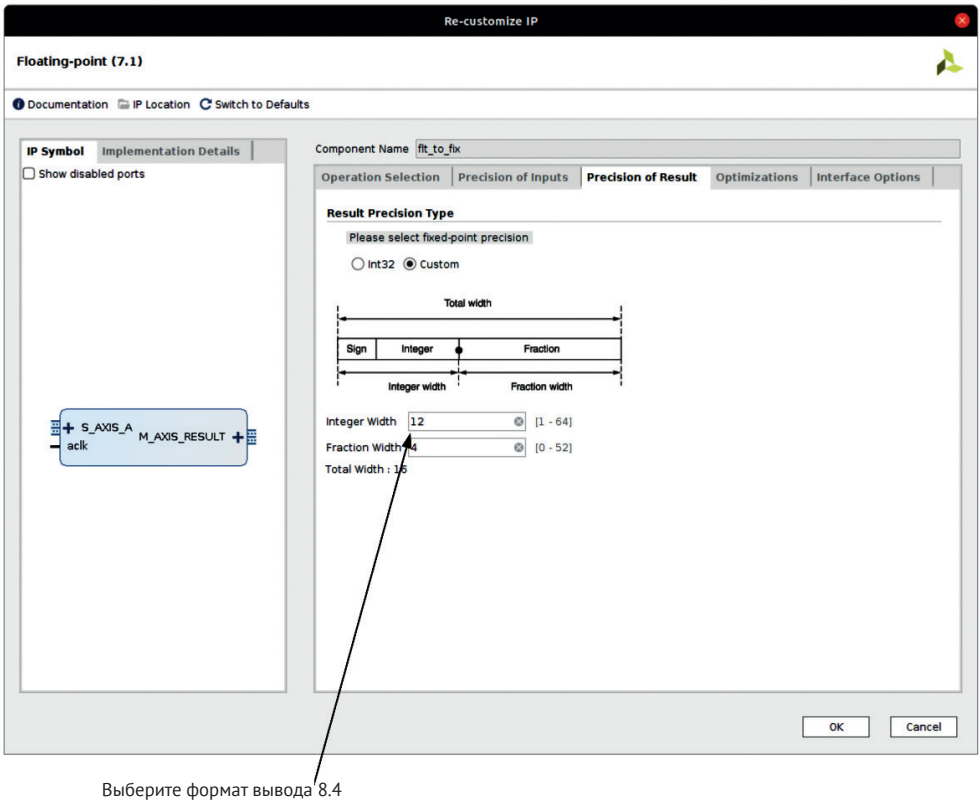


Рис. 6.14. Настройка преобразования формата с плавающей точкой в формат с фиксированной точкой

Таким образом, готовы все компоненты конвейера, чтобы можно было принимать значения с фиксированной точкой, полностью работать с данными как с плавающей точкой, а затем записывать данные как значения с фиксированной точкой.

Давайте посмотрим на результаты моделирования, чтобы увидеть, как выглядит задержка такой схемы.

Моделирование

Если посмотреть на задержки в компонентах, то можно увидеть, что каждая операция с плавающей точкой добавляет довольно много задержек по сравнению с работой с фиксированной точкой. Проанализируем результаты моделирования, чтобы увидеть, как выглядит реальная задержка.

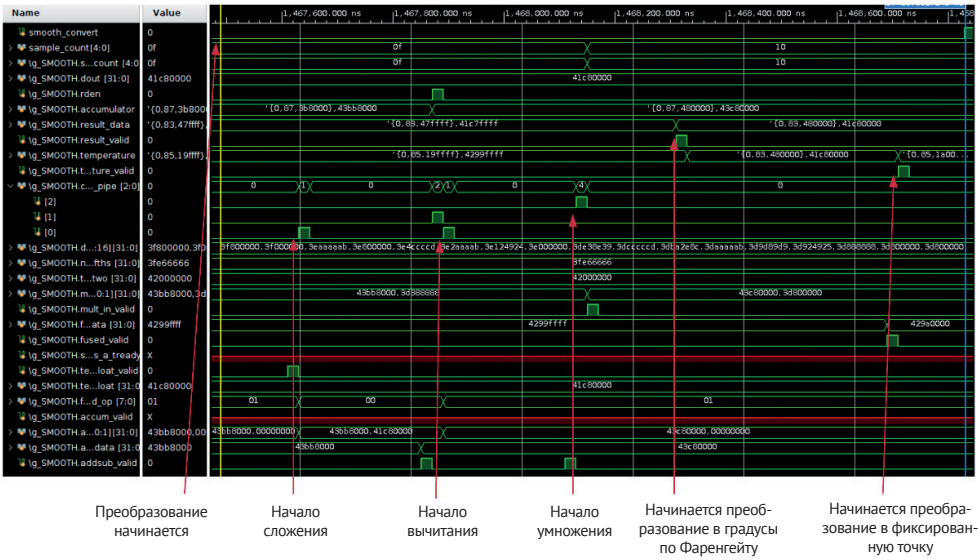


Рис. 6.15. Моделирование обработки показаний температуры в формате с плавающей точкой

На рисунке показано, сколько задержек было добавлено. Теперь задержка в 5 тактов стала равна 50. Но поскольку имеется большой запас по временны́м параметрам, это не является проблемой. Проанализируем использование ресурсов.

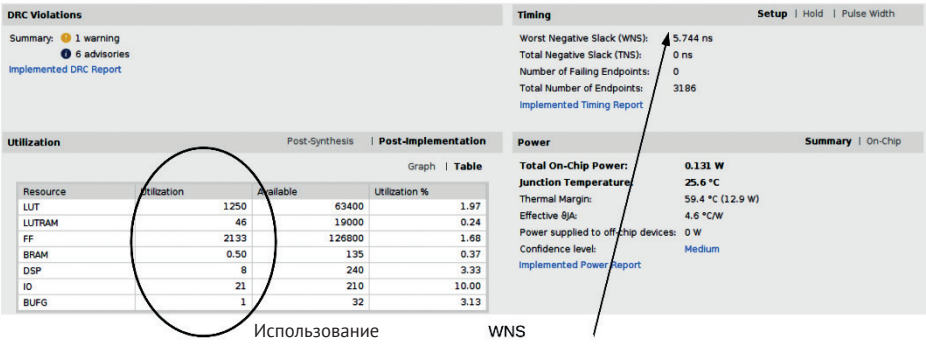


Рис. 6.16. Использование ресурсов FPGA для вывода температуры с плавающей точкой

Сравним использование реализации проекта с фиксированной точкой с реализацией с плавающей точкой.

Ресурс	Фиксированная	Плавающая	Изменение
LUT	433	1250	2.82x
LUTRAM	13	46	3.53x
FF	264	2133	8.07x
BRAM	0	0.5	+0.5
DSP	1	8	8x

Несмотря на то что проект занимает относительно небольшое количество ресурсов, разница достаточно заметна с точки зрения временны́х параметров и занимаемой площади чипа.

Этот проект должен дать вам какое-то представление о некоторых расширенных математических возможностях при использовании плавающей точки, а также о том, как потоковый интерфейс AXI может применяться для соединения нескольких IP-блоков вместе. Возможно, вы заметили, что не понадобилось следить за конвейеризацией, поскольку просто использовались действующие сигналы на входе и выходе каждого IP-блока в качестве управляющих сигналов.

Теперь, когда с плавающей точкой покончено, давайте кратко рассмотрим параллелизм.

Параллельные конструкции

FPGA, как некоторая заготовка, обеспечивает основу, которую можно использовать для разработки различных устройств. Люди используют FPGA в приложениях для обработки сигналов, таких как **программно-определяемые радиосистемы (Software-defined radio, SDR)**, в высокопроизводительных вычислительных приложениях, а в последнее время в **искусственном интеллекте (artificial intelligence, AI)** и **машинном обучении (machine learning, ML)**.

ML, AI и массовый параллелизм

В последние годы ML и AI переживают бум. Автономные автомобили, создание и анализ дипфейков[1], прогнозирование рынка – вот лишь некоторые из тематик, в которых нашли применение FPGA.

Легко понять почему. Плата Artix, на которую ориентируется данная книга, имеет до 240 блоков DSP. Самый большой Virtex Ultrascale+, который производит компания Xilinx, имеет почти 4000 блоков DSP и 9000 логических ячеек. Xilinx заявляет возможность достичь до 38,3 TOP/s (тераопераций в секунду, 10^{12} оп/с) для операций с целыми 8-битными числами (INT8) в микросхеме VU13P[2].

---

[1] Дипфейк (*deepfake*) – способ фальсификации фотографий или видео с помощью наложения изображений. Например, можно заменить лицо любого персонажа из произвольной видеосъемки на лицо известного политика или артиста. В изготовлении качественных дипфейков используются методы искусственного интеллекта, в том числе машинное обучение, соотвественно, их также приходится применять для доказательства фальсификации. – *Прим. ред.*

[2] VU13P – микросхема FPGA из серии Virtex Ultrascale Plus. – *Прим. ред.*

В рамках данной книги не представляется возможным дать полное представление об этом. Но, безусловно, стоит изучить доступные ресурсы для параллельного проектирования.

Параллельное проектирование – небольшой пример

Давайте рассмотрим небольшой пример, демонстрирующий массово-параллельную реализацию. В данном случае требуется создать дерево сумматоров, которое будет выводить сумму 256 входов за 8 тактовых циклов.

Обсудим задержку и пропускную способность. **Задержка (Latency)** – это количество тактов (или времени), которое требуется для получения результата. В представленном примере параллельной конструкции задержка составляет 8 тактовых циклов. Поскольку реализация проекта построена по принципу конвейера, можно получать новый результат каждый тактовый цикл после обработки исходных данных, пока поступают новые данные.

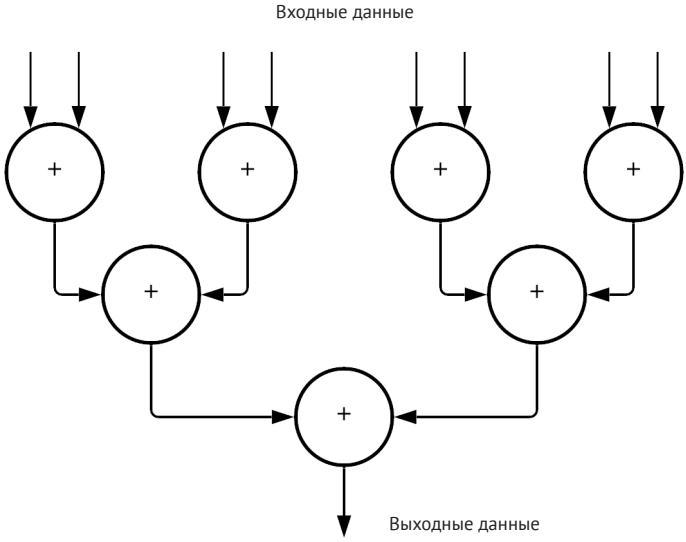


Рис. 6.17. Пример параллельного проектирования

Давайте проанализируем, как это может быть реализовано в `SystemVerilog`:

```
always @(posedge clk) begin
  for (int i = 0; i < 128; i++)
    int_data0[i] <= in_data[i*2+0] + in_data[i*2+1];
  for (int i = 0; i < 64; i++)
    int_data1[i] <= int_data0[i*2+0] + int_data0[i*2+1];
  for (int i = 0; i < 32; i++)
    int_data2[i] <= int_data1[i*2+0] + int_data1[i*2+1];
  for (int i = 0; i < 16; i++)
    int_data3[i] <= int_data2[i*2+0] + int_data2[i*2+1];
  for (int i = 0; i < 8; i++)
    int_data4[i] <= int_data3[i*2+0] + int_data3[i*2+1];
  for (int i = 0; i < 4; i++)
    int_data5[i] <= int_data4[i*2+0] + int_data4[i*2+1];
```

```
 for (int i = 0; i < 2; i++)
    int_data6[i] <= int_data5[i*2+0] + int_data5[i*2+1];
  out_data  <= int_data6[0] + int_data6[1];
  int_valid <= int_valid << 1 | in_valid;
  out_valid <= int_valid[6];
end // always @ (posedge clk)
```

Этот код создаст дерево из 255 сумматоров. Эта операция является как параллельной, в том смысле, что будут обрабатываться все входы одновременно, так и конвейерной, поскольку новые данные могут поступать каждый такт. Как уже упоминалось, через 8 тактовых циклов на выходе конвейера будет доступен первый результат. Каждый последующий цикл будет доступна новая сумма.

Выводы

В этой главе был взят проект датчика температуры и улучшен с использованием математики с фиксированной точкой. Было убрано условие отложенного запуска, чтобы температура выводилась почти сразу и постоянно фильтровалась в течение всего срока работы проекта. Затем были рассмотрены операции с плавающей точкой и проект был преобразован в конвейер, где вычисления выполняются в формате с плавающей точкой. Это дало возможность познакомиться с потоковой передачей AXI, которая будет становиться все более важной по мере изучения этой книги.

В следующей главе изучение интерфейса AXI будет более глубоким, а некоторые из разработанных IP-блоков будут упакованы в формат AXI, чтобы их можно было использовать повторно, а также будут представлены инструменты интеграции IP-блоков (IP integrator) и инструменты проектирования IP-блоков (block design tool).

Вопросы

1. Если используемые числа находятся в большом динамическом диапазоне, что лучше использовать?

   a) Целые числа.
   b) Числа с фиксированной точкой.
   c) Числа с плавающей точкой.
   d) Комплексные числа.

2. Расположите числа от наименее сложного к наиболее сложному.

   a) Числа с фиксированной точкой, целые числа, числа с плавающей точкой.
   b) Целые числа, числа с фиксированной точкой, числа с плавающей точкой.
   c) Числа с плавающей точкой, числа с фиксированной точкой, целые числа.
   d) Целые числа, числа с плавающей точкой, числа с фиксированной точкой.

3. Примером какого типа проектирования является следующий код?

```
always @(posedge clk) begin
   if (stage[0]) out[0] <= fp_out[0];
   if (stage[1]) out[1] <= out[0] + fp_out[1];
   if (stage[2]) out[2] <= out[1] + out[0] + fp_out[2];
end
```

a) Конвейерный.
b) Параллельный.
c) Конечный автомат.

4. Примером какого типа проектирования является следующий код?

```
always @(posedge clk) begin
  for (int i = 0; i < 128; i++) dout[i] <= din[i*2] +
din[i*2+1];
end
```

a) Конвейерный.
b) Параллельный.
c) Конечный автомат.

5. Какой из следующих сигналов активирует потоковый интерфейс AXI?

a) tdata.
b) tvalid.
c) tready.
d) tlast.
e) tuser.
f) taddr.

6. Что даст на выходе умножитель с фиксированной точкой 16.16×8.16?

a) 16.16.
b) 17.16.
c) 32.32.
d) 24.32.

Задание повышенной сложности

В демонстрационном примере используются не все цифры на семисегментном индикаторе. Ранее использовался светодиод для индикации градусов по Цельсию или Фаренгейту. Можете ли вы изменить код так, чтобы он использовал одну или две секции семисегментного индикатора для отображения C/F или °C/°F?

Дополнительное чтение

Обратитесь к следующим ссылкам для получения дополнительной информации о том, что было рассмотрено в этой главе.

- Справочное руководство по Nexys A7: https://reference.digilentinc.com/reference/programmable-logic/nexys-a7/reference-manual.
- Спецификация датчика температуры: https://www.analog.com/media/en/technical-documentation/data-sheets/adt7420.pdf.
- Руководство пользователя Xilinx DSP 48 для компонентов 7-й серии (Artix-7): https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_ DSP48E1.pdf.

Раздел 3

Взаимодействие с внешними компонентами

До сих пор связь с внешним миром учебных проектов в данной книге ограничивалась кнопками, переключателями, светодиодами и семисегментным индикатором. В этом разделе эти элементы будут обновлены, чтобы их можно было использовать многократно, а также добавить несколько более интересных компонентов. К концу этого раздела вы сможете заменить кнопки и переключатели клавиатурой PS/2, а светодиоды и семисегментный индикатор – контроллером **Video Graphics Array (VGA).**

В эту часть книги входят:

Глава 7 «Введение в AXI»;

Глава 8 «Много данных? MIG и DDR2»;

Глава 9 «Лучший способ отображения – VGA»;

Глава 10 «Свести все воедино»;

Глава 11 «Темы повышенной сложности».

Глава 7

Введение в AXI

По мере того как **FPGA** становились все больше и сложнее, разработчики, такие как Xilinx, начали предлагать готовые реализации отдельных модулей (сложнофункциональные блоки) – **интеллектуальную собственность (Intellectual Property, IP**), разработанную и протестированную для ускорения реализации проекта. Эти первые IP-блоки (ядра) часто имели простые интерфейсы, иногда называемые пользовательскими (native) интерфейсами. Xilinx предлагала первые высокопроизводительные компоненты с ядрами PowerPC и собственными ядрами MicroBlaze, каждый из которых имел различные интерфейсы. Когда компания Xilinx приняла процессоры ARM как часть семейства Zynq, она стандартизировала интерфейсы процессоров ARM, используя усовершенствованный расширяемый интерфейс (**Advanced eXtensible Interface, AXI**). Чтобы наилучшим образом использовать Xilinx IP, был уже рассмотрен потоковый интерфейс. Есть еще два широко используемых интерфейса: AXI-Lite и AXI full.

К концу этой главы вы будете хорошо разбираться в разновидностях AXI и в том, когда их следует использовать. Вы будете знать, как создавать собственные IP-блоки с использованием AXI, чтобы упростить интеграцию с другими IP-блоками. В конце главы будет разработан датчик температуры с использованием AXI и IP-интегратора.

В этой главе будут рассмотрены следующие основные темы:

- потоковая передача AXI;
- проект 9 – создание IP-блоков для Vivado с использованием потоковых интерфейсов AXI;
- введение в IP-интегратор;
- интерфейсы AXI4 (AXI full и AXI-Lite);
- разработка IP-блоков – AXI-Lite, AXI full[1] и потоковые интерфейсы.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH7.

---

[1] Интерфейс, называемый автором AXI full, чаще встречается под названием AXI Memory Mapped. – *Прим. конс.*

Потоковая передача AXI

В главе 6 «Математика, параллелизм и конвейерное проектирование» был кратко рассмотрен интерфейс AXI и потоковая передача данных. Потоковая передача AXI используется в основном как простой канал для перемещения данных между двумя точками, как показано на рис. 7.1:

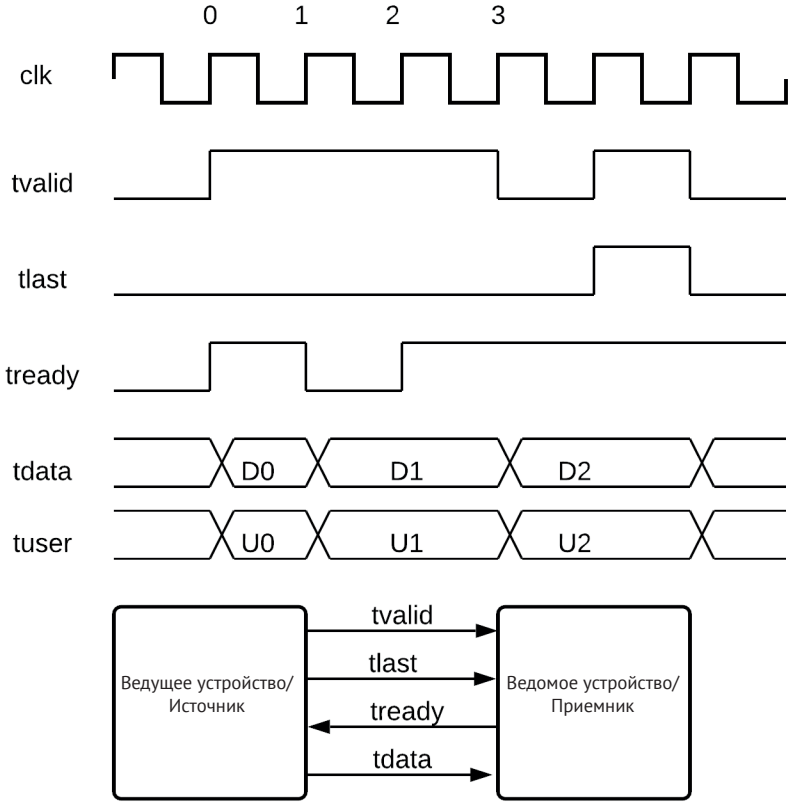


Рис. 7.1. Потоковая передача AXI с необязательным сигналом tuser

Существует необязательный дополнительный сигнал **tuser**, включенный для полноты. Этот сигнал может быть передан вместе с потоком, но понимание того, как этот сигнал будет использоваться, зависит от источника и приемника данных.

Прежде чем приступить к изучению других типов AXI, необходимо представить датчик температуры I2C в виде IP-блоков на основе потоковой передачи AXI.

Проект 9. Создание IP-блоков для Vivado с использованием потоковых интерфейсов AXI

В этом проекте датчик температуры I2C и будет разделен на IP-блоки, которые можно использовать в IP-интеграторе для сборки проекта.

Первоначальный проект выглядел следующим образом:

Рис. 7.2. Оригинальный конвейер для датчика температуры

Если посмотреть на Xilinx IP с плавающей точкой, то преобразование чисел из формата с фиксированной точкой в формат с плавающей точкой, преобразование чисел из формата с плавающей точкой в формат с фиксированной точкой, сложение/вычитание, масштабирование и умножение-сложение – это все IP-блоки с потоковыми интерфейсами. Необходимо рассмотреть интерфейс I2C, который считывает температуру с ADT7420, сам конвейер расчета температуры и интерфейс семисегментного индикатора. Давайте сначала разберемся с семисегментным индикатором.

Потоковый интерфейс для семисегментного индикатора

Первое, что нужно сделать, – это создать каталог для размещения IP-блоков. Это облегчит процесс упаковки. Для этого необходимо создать каталог `CH7/build/IP/seven_segment`. Внутри него есть каталог `hdl`, который содержит выделенный в отдельный блок семисегментный индикатор температурного датчика.

Если придерживаться нескольких правил, создать IP-блок будет проще. Полное руководство можно найти по адресу: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1118-vivado-creating-packaging-custom-ip.pdf.

Тактовый сигнал уже назван `clk`, что является одним из способов автоматической идентификации тактовых сигналов. Чтобы создать потоковую шину AXI, которая может быть сгенерирована автоматически, нужно назвать интерфейсные сигналы следующим образом:

- `<interface name>_tdata` (обязательно)
- `<interface name>_tvalid` (обязательно)
- `<interface name>_tready`
- `<interface name>_tstrb`
- `<interface name>_tkeep`
- `<interface name>_tlast`
- `<interface name>_tid`

- `<interface name>_tdest`
- `<interface name>_tuser`

Теперь список портов для IP-блока выглядит следующим образом:

```
module seven_segment
  #(
    parameter NUM_SEGMENTS = 8,
    parameter CLK_PER      = 10,  // Тактовый период в нс
    parameter REFR_RATE    = 1000 // Частота обновления в Гц
    )(
    input wire                          clk,
    input wire                          seven_segment_tvalid,
    input wire [NUM_SEGMENTS*4-1:0]     seven_segment_tdata,
    input wire [NUM_SEGMENTS-1:0]       seven_segment_tuser,
    output logic [NUM_SEGMENTS-1:0]     anode,
    output logic [7:0]                  cathode
     );
```

Сигналы `anode` и `cathode` станут внешними интерфейсами на плате. Потоковый интерфейс AXI названы `seven_segment`, он и `clk` должны быть распознаны без каких-либо специальных действий.

Рассмотрим процесс сборки пользовательского IP-блока.

1. Создание IP-блока.

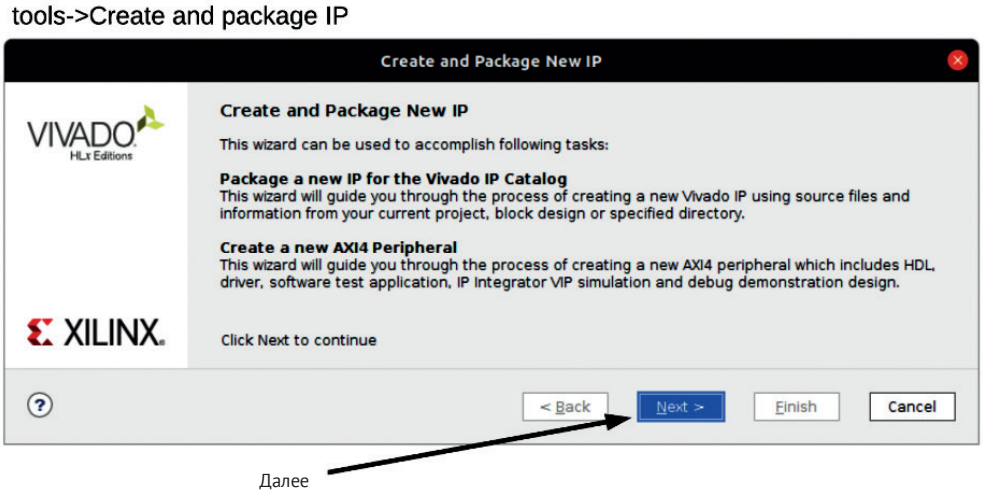


Рис. 7.3. Создание и сборка пользовательского IP-блока

Первое диалоговое окно просто резюмирует то, что будет сделано. Был изменен код, чтобы собрать IP-блок.

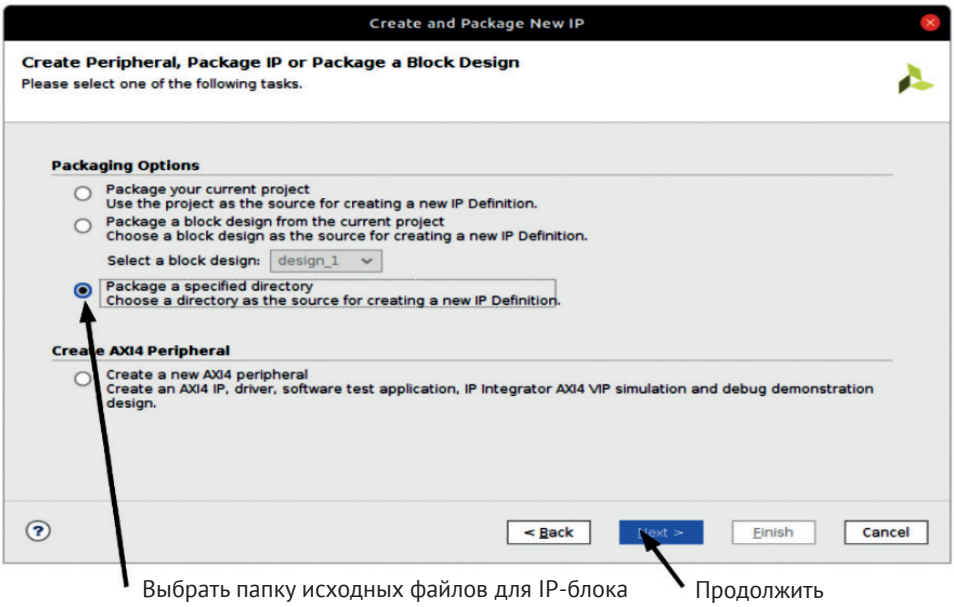


Рис. 7.4. Выбор варианта сборки

2. Убедитесь, что выбран вариант **Package a Specified Directory**, и нажмите **Next**.

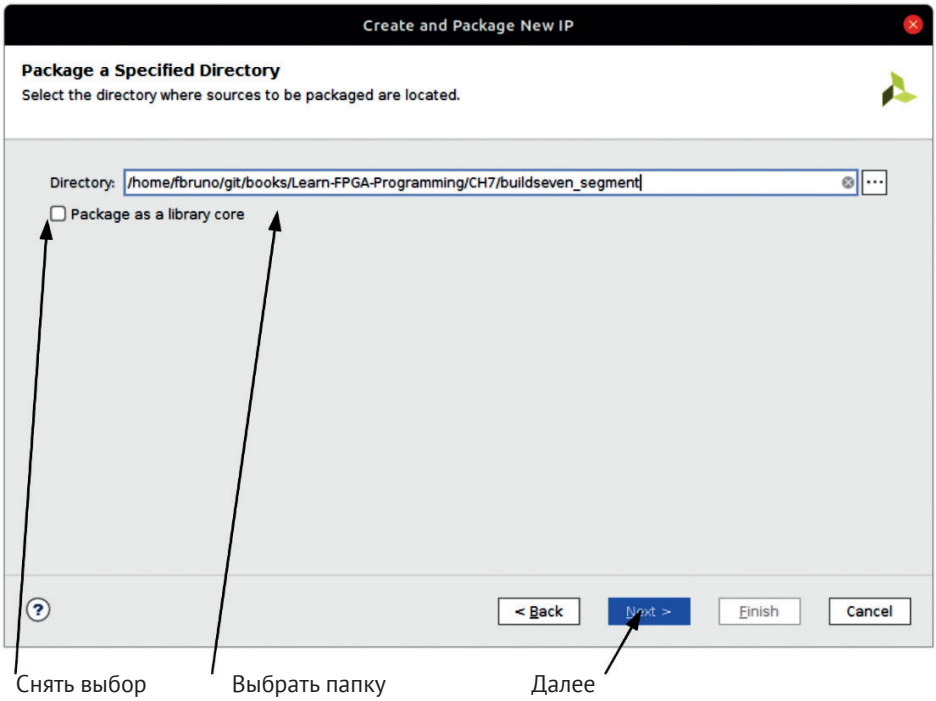


Рис. 7.5. Указание исходного каталога

3. Убедитесь, что снят флажок **Package as a library core** (выполнить сборку как ядро библиотеки). Выберите каталог `seven_segment` в разделе **Create and Package New IP** (создать и выполнить сборку нового IP-блока).

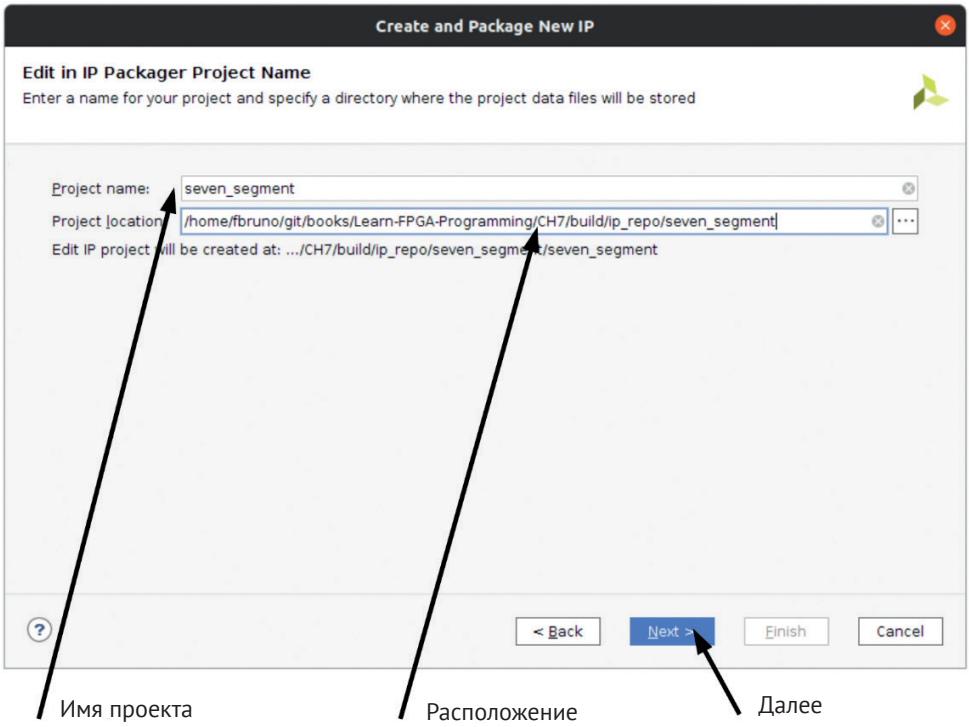


Рис. 7.6. Папка назначения для сгенерированного IP-блока

4. Нужно указать название проекта и место назначения. В данном случае создана папка `ip_repo` в каталоге сборки. Затем выберите **Next**.

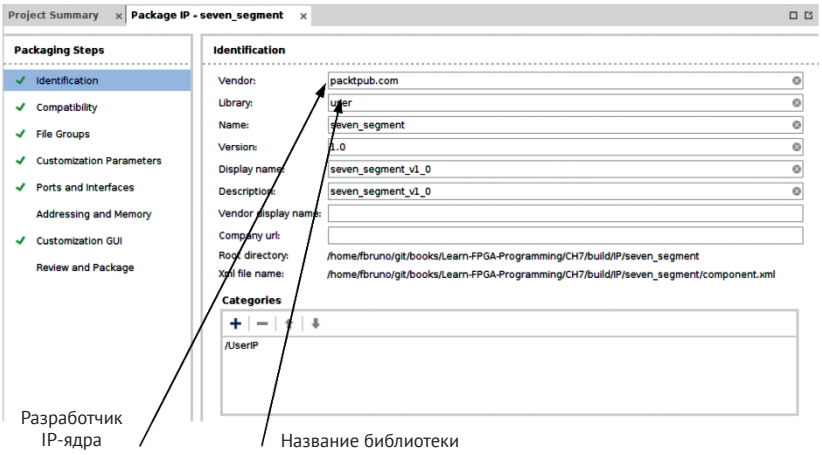


Рис. 7.7. Базовые настройки

5. Можно определить базовые настройки IP-блока, как на приведенном скриншоте. Проверьте названия портов и интерфейсов чтобы они были такими же, как показано на следующем скриншоте.

Порты и Интерфейсы

Name	Interface Mode	Enablement Dependency	Direction	Driver Value	Size Left	Size Right	Size Left Dependency	Size Right Dependency	Type Name
seven_segment	slave								
seven_segment_tdata			in	0	31	0	((NUM_SEGMENTS * 4)		wire
seven_segment_tuser			in	0	7	0	(NUM_SEGMENTS - 1)		wire
seven_segment_tvalid			in						wire
Clock and Reset Signals									
clk	slave								
anode			out		7	0	(NUM_SEGMENTS - 1)		logic
cathode			out		7	0			logic

Рис. 7.8. Проверка портов и интерфейсов IP-блоков

6. IP-упаковщик смог автоматически найти порт тактового сигнала и семисегментный ведомый порт AXI. Два других порта просто выведены как есть.

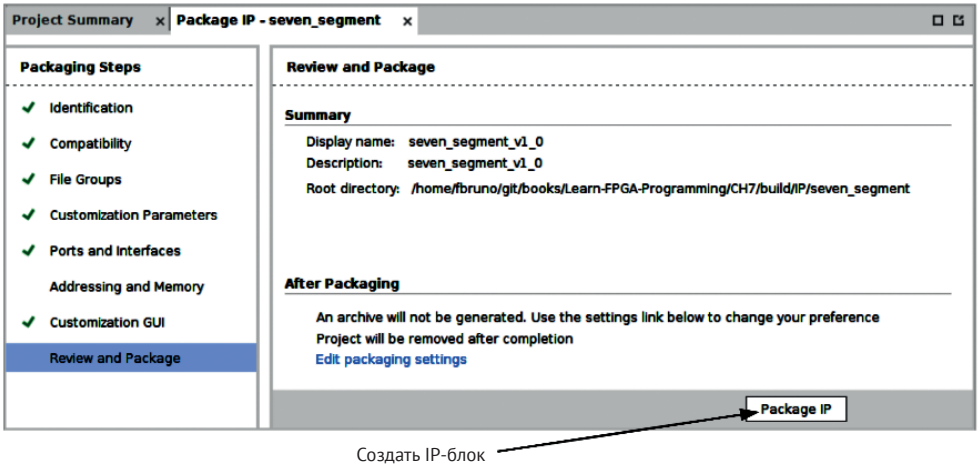


Рис. 7.9. Итоговая страница конфигурации пользовательского IP-блока

7. Проверьте результаты конфигурации IP-блока, а затем нажмите кнопку **Package IP**.

На этом этапе IP-блок готов к использованию. Можно заметить, что был добавлен дополнительный интерфейс `tuser` в этой версии потокового интерфейса. Это сделано потому, что нужен способ передачи десятичной точки для отображения. Поскольку не существует стандарта для использования `tuser`, важно, чтобы ведущий интерфейс знал, как управлять им, а ведомый знал, как интерпретировать его.

Таким же образом упаковано еще два IP-блока для использования: интерфейс `adt7420_i2c` и ядро датчика температуры в формате с плавающей точкой. Причина, по которой создано ядро датчика температуры, заключается в том, что можно использовать инструмент **Block Design, BD (схемный редактор)**, в качестве альтернативы для разработки этой версии проекта. Можно было сохранить старый файл

верхнего уровня и просто размножить два новых ядра. Обычно предпочтительно работать непосредственно в SystemVerilog, но бывают случаи, например, при разработке на FPGA **системы на кристалле** (**System-on-Chip, SoC),** когда необходимо использовать инструмент **схемного редактора** хотя бы для части проекта.

Рассмотрим код ADT7420, который составляет новый IP-блок.

Разработка IP ADT7420

Проанализируем файл верхнего уровня иерархии для IP-блока:

```
module adt7420_i2c
  #(
   parameter  INTERVAL     = 1000000000,
   parameter  CLK_PER      = 10
   )(
   input wire          clk, // частота 100МГц
   // Интерфейс датчика температуры
   inout wire          TMP_SCL,
   inout wire          TMP_SDA,
   inout wire          TMP_INT,
   inout wire          TMP_CT,
   output logic        fix_temp_tvalid,
   output logic [15:0] fix_temp_tdata
   );
```

Имеется тактовый сигнал, шина I2C и соединения с микросхемой датчика температуры.

Этот файл содержит конечный автомат I2C. Проанализируйте его. Особенность файла в том, что выход температуры теперь является потоковым AXI. IP-блок уже собран, но, если вы хотите, можете собрать его снова в качестве упражнения.

Проанализируем ядро проекта, где выполняется усреднение температуры в формате с плавающей точкой, преобразование в градусы по Фаренгейту и вывод на семисегментный индикатор с помощью созданного IP-блока.

Ядро t_temp

Ядро датчика температуры, работающее с числами в формате с плавающей точкой, является самым сложным блоком в проекте.

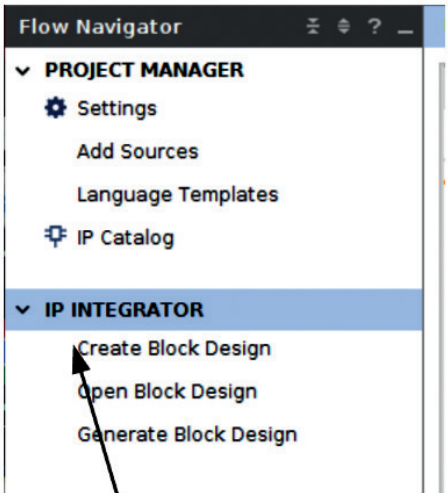
Оно соединяется с двумя другими IP-блоками, а также с частью проекта, работающей в формате с плавающей точкой, которая была создана ранее. Ядро модуля `flt_temp` не изменилось по сравнению с предыдущим проектом, но теперь имеется новое определение интерфейса для потоковых интерфейсов AXI.

Рекомендуется анализировать то, как IP-блок реализован на **языке описания аппаратуры (Hardware Description Language, HDL).** В следующем разделе, когда проект будет собираться в схемном редакторе, рассмотрим, как выглядят IP-блоки изнутри.

IP-интегратор

IP-интегратор предоставляет собой графический пользовательский интерфейс для подключения IP-блоков с помощью схематического подключения с использованием в схемном редакторе. Предыдущие шаги были выполнены для всех трех IP-модулей. В результате IP-модули добавлены в проект, и теперь их можно использовать.

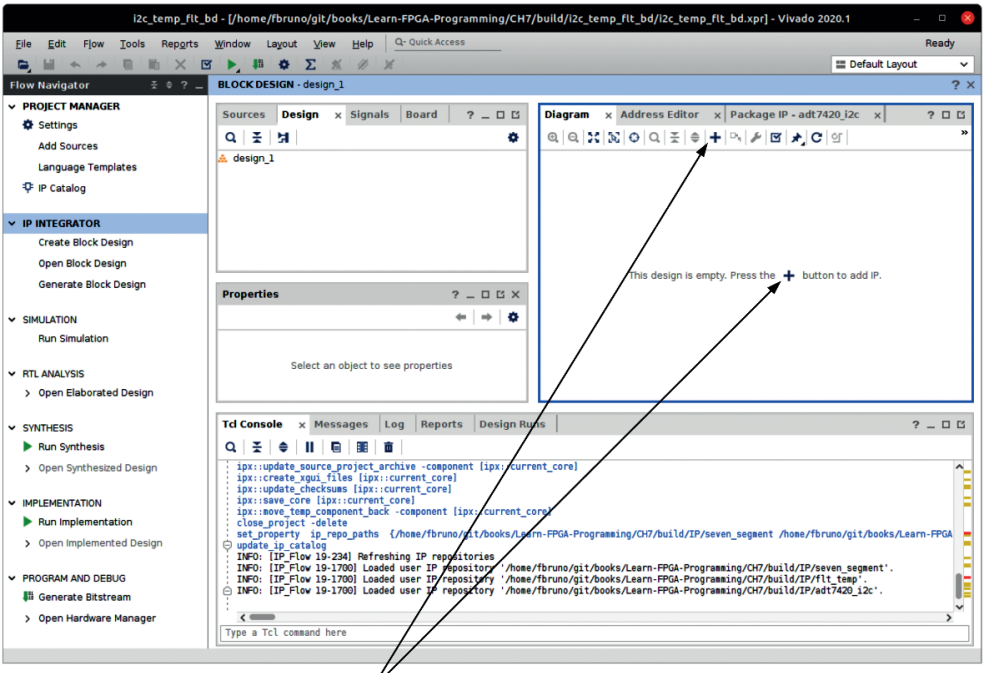
1. Первым шагом будет создание проекта в схемном редакторе и добавление IP-модулей.



Выбрать Create Block Design (Создание блок-схемы)

Рис. 7.10. Создание Block Design

Выбор пункта **Create Block Design** вызовет инструмент, который будет использоваться в этом разделе.



Выбрать добавление IP-блока

Рис.7.11. Добавление IP-блока

2. Когда главное окно схемного редактора открыто, первое, что нужно сделать, – это добавить IP-блок. Начнем с IP-блока `adt7420_i2c` и рассмотрим его.

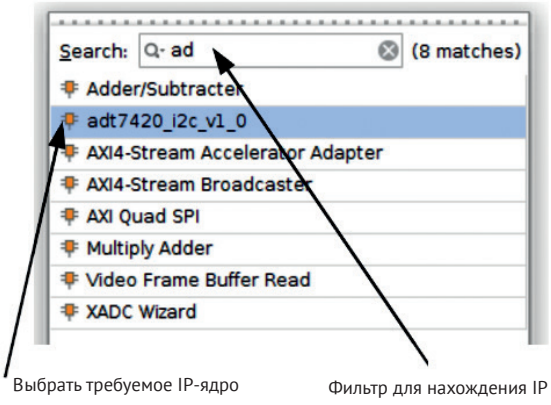


Рис. 7.12. Выбор IP-блока

3. При нажатии символа + появится всплывающее окно для поиска IP-блока. Можно набрать `adt` и найти нужный интерфейс. Кликните дважды, чтобы добавить интерфейс в проект. Теперь исследуем, как выглядит ядро при размещении и какие параметры можно настроить.

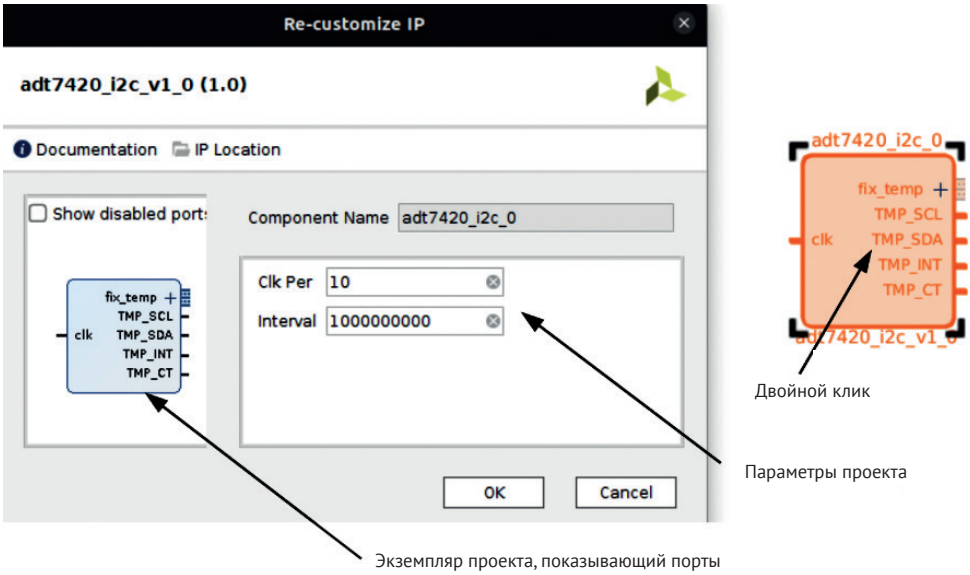


Рис. 7.13. Конфигурация IP-блока

В первоначальном проекте были указаны два параметра для этого ядра. Если их не изменить, то они будут отображаться в конфигурации ядра как есть при двойном щелчке на экземпляре ядра на блок-схеме.

```
module adt7420_i2c
  #(
   parameter  INTERVAL     = 1000000000,
   parameter  CLK_PER      = 10)
```

Поскольку параметры в IP-блоках уже установлены на те, которые нужны для платы Nexys A7, ничего менять не нужно. Проанализируйте экземпляр проекта, и вы увидите список портов:

```
input wire          clk, // частота 100МГц
 // Интерфейс датчика температуры
 inout wire          TMP_SCL,
 inout wire          TMP_SDA,
 inout wire          TMP_INT,
 inout wire          TMP_CT,
 output logic        fix_temp_tvalid,
 output logic [15:0] fix_temp_tdata
```

Поскольку потоковый интерфейс был определен при сборке IP-модуля, он свернут в схему и помечен как `fix_temp`. Порт синхронизации будет подключен к генератору тактового сигнала, а порты `TMP_*` будут подключены к внешним интерфейсам.

1. Продолжим и добавим все IP-блоки, которые понадобятся для проекта.

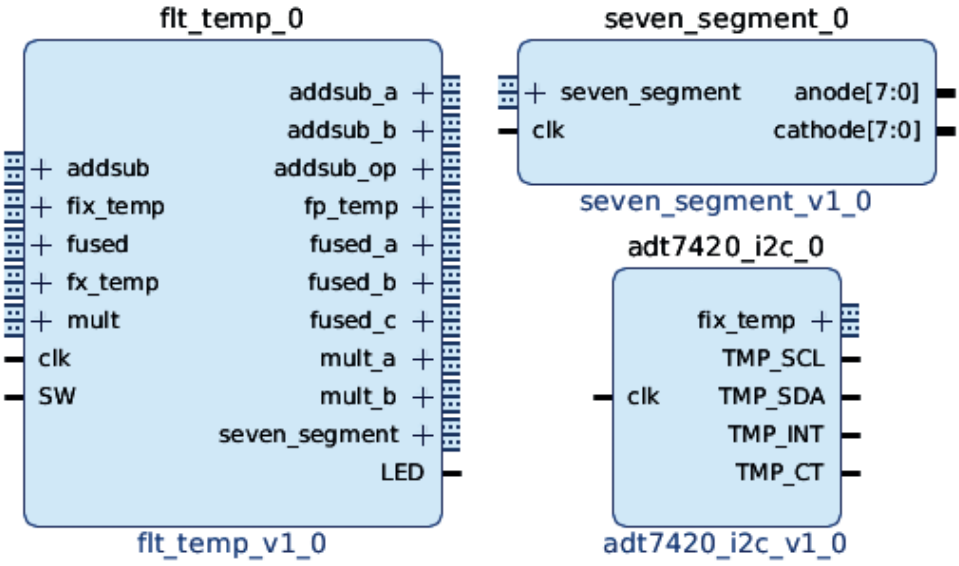


Рис. 7.14. Пользовательские IP-блоки, добавленные в проект

Предстоит добавить довольно много IP-блоков в проект (все ядра с плавающей точкой). Структура проекта показана на рис. 7.2. Но даже без этого названия блоков дают подсказку о том, что нужно добавить. Все IP-ядра с плавающей точкой являются просто конфигурациями IP-блоков с плавающей точкой, поэтому во всплывающем окне добавления IP-блоков необходимо найти `floating point`. Нужно будет выполнить этот шаг пять раз.

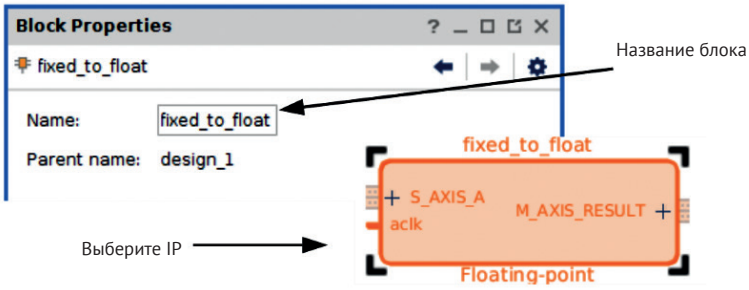


Рис. 7.15. Свойства блока

Название IP-блока по умолчанию – это просто его имя с добавлением числа, увеличивающегося на единицу. Можно переименовать экземпляры модулей, чтобы они были более удобными для пользователя. По окончании настройки выберите IP-блок, и затем в левой части окна схемного редактора будет панель **Block Properties** (свойства блока). Это дает простой доступ к его свойствам, а также возможность изменить имя экземпляра.

2. Продолжим работу с остальными IP-блокам, работающим с числами в формате с плавающей точкой.

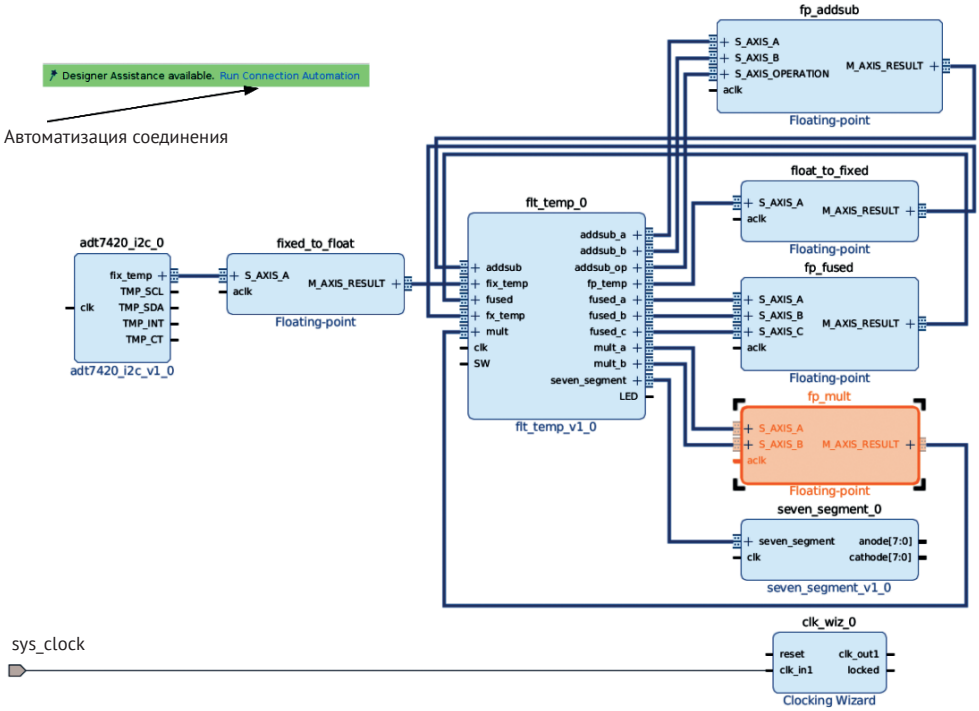


Рис. 7.16. Полная схема проекта

Теперь можно посмотреть на схему проекта. Надо будет подключить потоковые интерфейсы вручную, как это сделано на рис. 7.16. Также можно заметить,

что в какой-то момент при добавлении IP-блока появилось всплывающее сообщение о том, что инструмент схемного редактора распознал, что он может автоматизировать некоторые соединения.

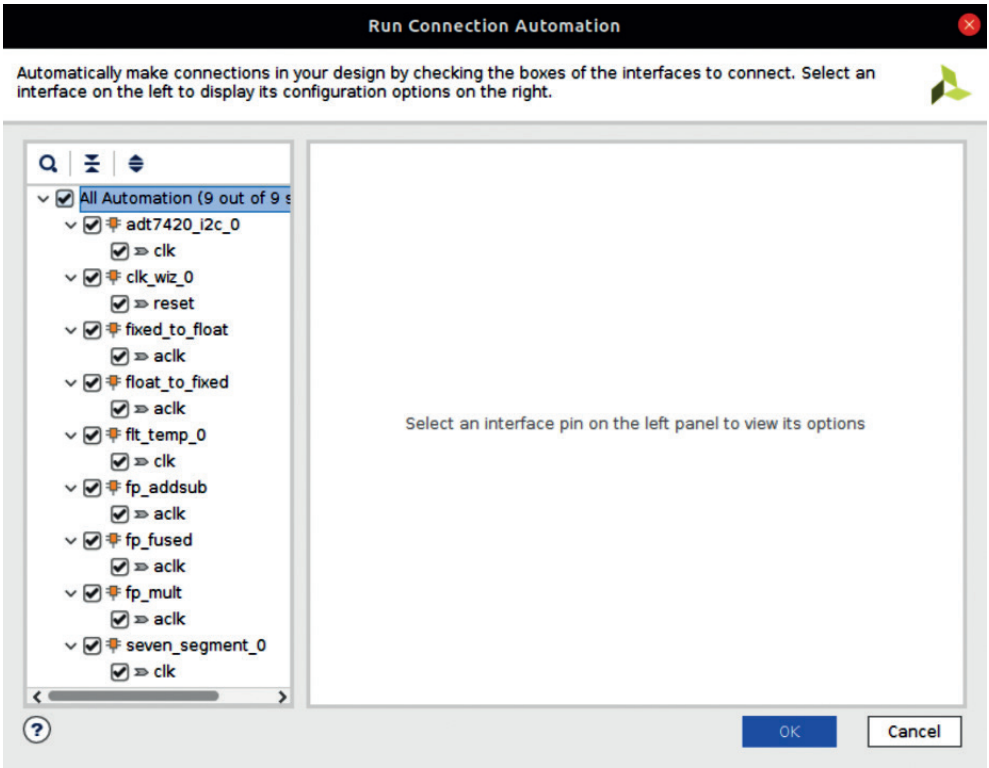


Рис. 7.17. Автоматизация соединений в проекте

3. В данном случае инструмент автоматически подключит генераторы тактового сигнала и сигнал сброса. После этого можно подключить внешние порты. Сделаем это вручную.

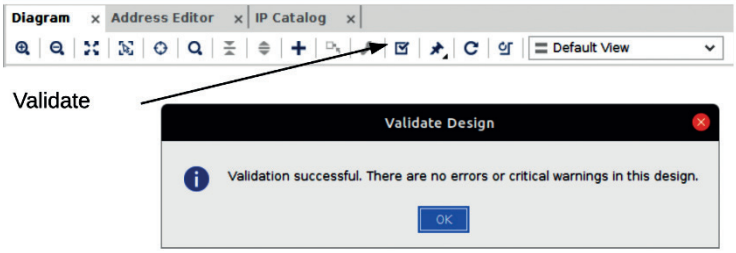


Рис. 7.18. Проверка проекта

Когда вы закончите работу над схемой, нажмите кнопку проверки проекта. Это позволит убедиться в отсутствии ошибок в проекте.

> **Важное замечание**
>
> Если вы создаете что-то с интерфейсом AXI-Lite или с полнофункциональным AXI-интерфейсом, вам следует убедиться, что адресация настроена до сборки. Это будет рассмотрено в следующей главе.

4. В проекте нет ошибок, поэтому нужно сконфигурировать выходные соединения с выводами FPGA, а затем сгенерировать загрузочный файл.

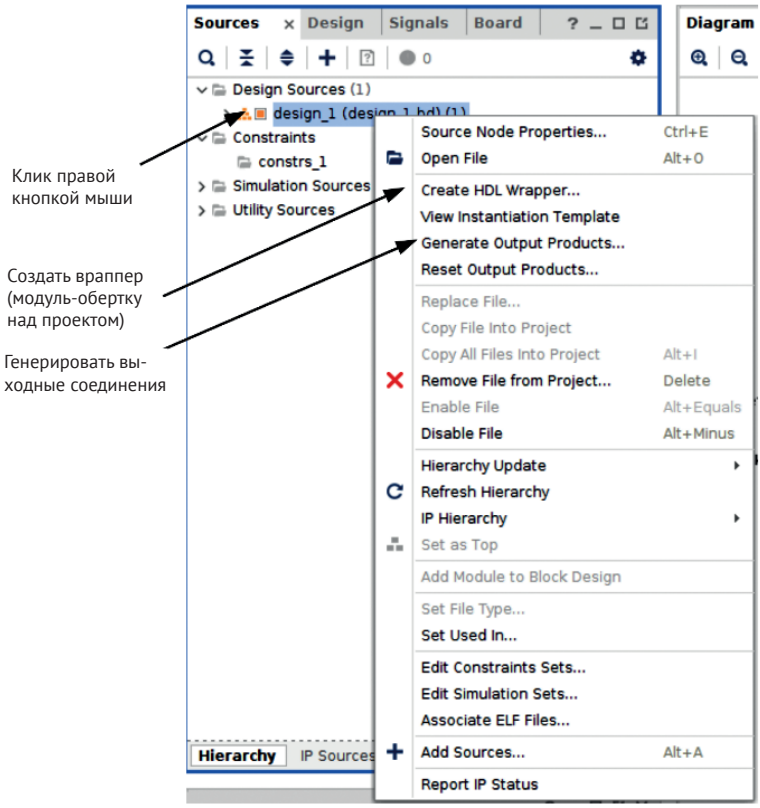


Рис. 7.19. Генерация выходных соединений

5. Один важный момент – создаваемая цифровая схема имеет внутри буферы с тремя состояниями. Чтобы проект функционировал должным образом, необходимо выбрать **Global** (Глобальный) в опциях синтеза. Это настройка, создающая ограничение, которое не позволяет выполнять синтез проекта **Out Of Context (OOC)** (вне контекста), если в проекте используются порты с тремя состояниями.

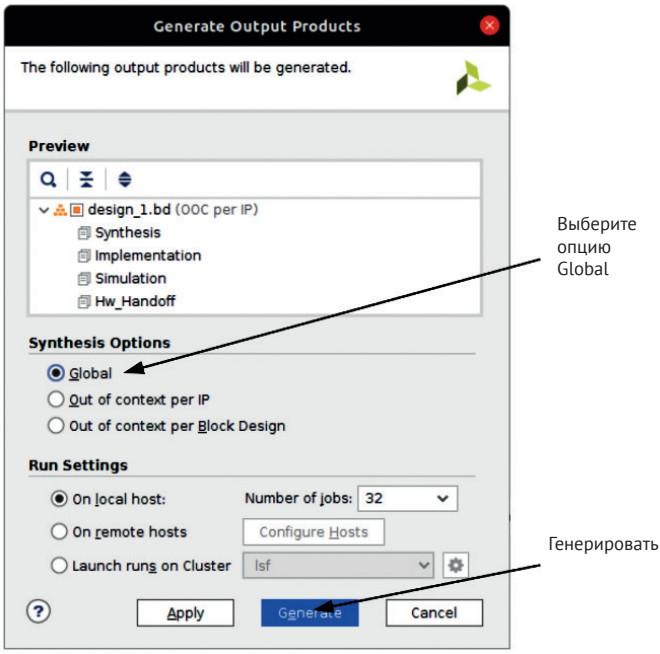


Рис. 7.20. Глобальный синтез

6. Наконец, нужно создать HDL-обертку (враппер, wrapper), которая станет верхним уровнем проекта (рис. 7.18):

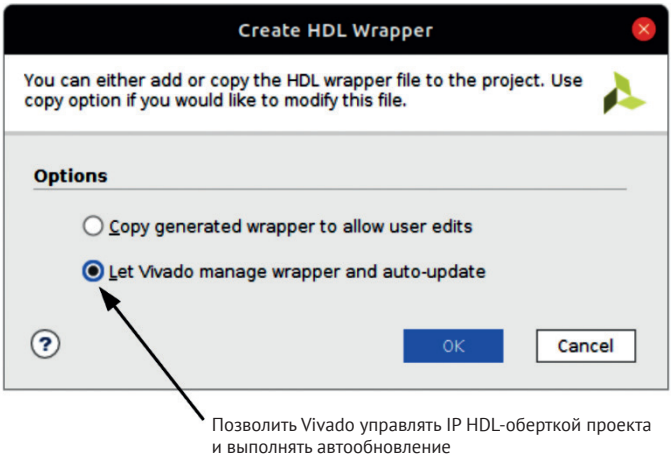


Рис. 7.21. Создание HDL-обертки

Проанализируем получившийся финальный проект. Финальная схема проекта имеет следующий вид:

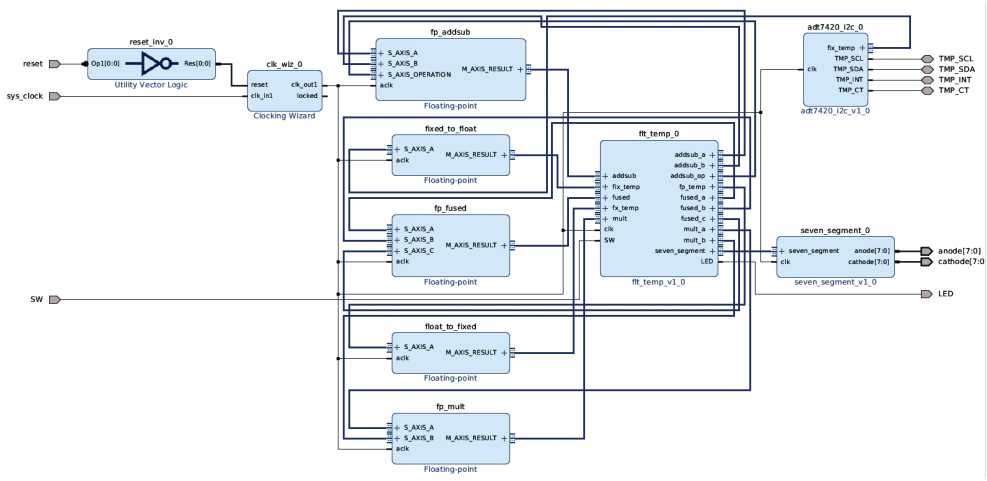


Рис. 7.22. Законченный проект в схемном редакторе Vivado

Функционально получившийся проект эквивалентен тому, который был разработан в главе 6 «Математика, параллелизм и конвейерное проектирование». Это просто еще один способ ускорить разработку, если вы используете большое количество IP-блоков. Он наиболее полезен при использовании процессоров и практически необходим при проектировании систем на кристалле. Можно встроить свои собственные IP-блоки, как это было сделано в данном примере, или упаковать схему, созданную в схемном редакторе, и включить ее в HDL-проект.

Выполним отладку проекта.

Отладка проекта с помощью IP-интегратора

IP-интегратор упрощает отладку проекта. Просто кликните правой кнопкой мыши на любом соединении или шине и выберите команду **Debug**. Это особенно полезно при работе с шинами AXI, поскольку **интегрированный логический анализатор (Integrated Logic Analyzer, ILA)** понимает структуру шины и протокол транзакций и отображает информацию в осмысленном виде.

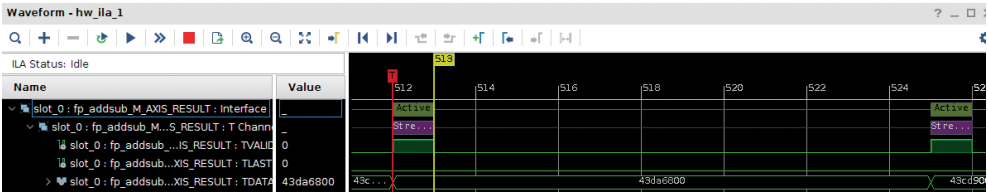


Рис. 7.23. AXI-трансляция ILA

Добавьте отладочное ядро, кликнув правой кнопкой мыши на выводе `fp_addsub`. Сгенерируйте загрузочный файл и установите триггер по переднему фронту `(R) tvalid`. Вы должны отследить несколько транзакций. Попробуйте добавить несколько других ILA и понаблюдайте за транзакциями. Когда датчик температуры работает, можно запускать tvalid-сигналы и наблюдать, как

данные распространяются по конвейеру, работающему с числами в формате с плавающей точкой.

Теперь, когда было рассмотрено использование IP-интегратора для создания и отладки проекта, взглянем на два других типа интерфейсов AXI, которые будут использоваться в оставшейся части книги.

Интерфейсы AXI4 (AXI full и AXI-Lite)

Интерфейс AXI4 – это полнофункциональный процессорный интерфейс, используемый компанией ARM для подключения периферийных устройств к своим процессорам. Компания Xilinx приняла этот интерфейс (облегченный, полнофункциональный и потоковый) для подключения своих аппаратных (hard) и программных (soft) процессоров к другим IP-блокам. Поскольку этот интерфейс является полнофункциональным, его реализация может быть дорогостоящей, и его следует рассматривать только в том случае, если нужен адресный интерфейс с возможностью высокопроизводительной пакетной передачи данных. Интерфейсы AXI full и AXI-Lite состоят из пяти компонентов. Операция чтения состоит из адресного компонента и компонента данных.

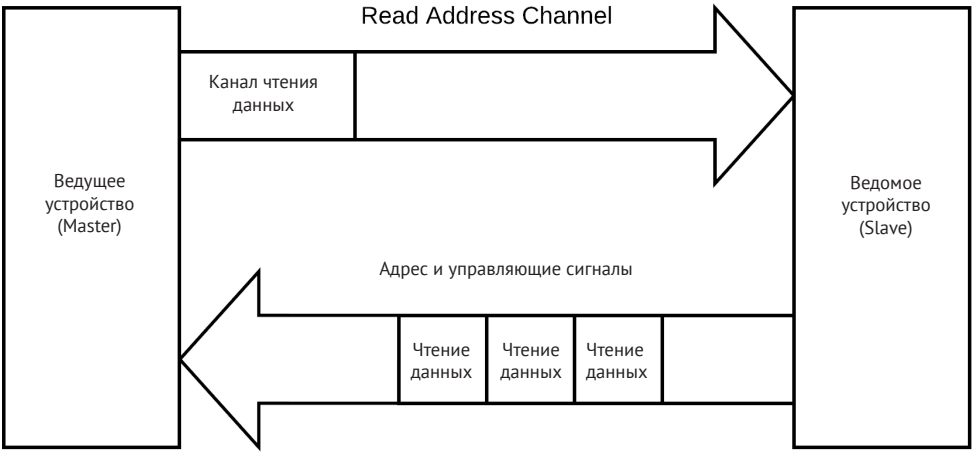


Рис. 7.24. Канал чтения AXI

На рисунке выше схематически показано, как происходит операция чтения в AXI. Адрес и шина управления посылают ведомому устройству сигнал на выполнение чтения. В интерфейсе AXI-Lite это одна операция чтения, в AXI full это может быть пакет данных. Эти типы чтения являются отложенными, т. е., если интерфейс поддерживает такое, может быть выполнено несколько запросов на чтение, прежде чем данные начнут поступать.

Когда интерфейс ведомого устройства готов, он может начать отправлять данные. Если ведущее и ведомое устройства поддерживают эту функцию, ведомое устройство может выполнять чтение не по порядку, используя канал

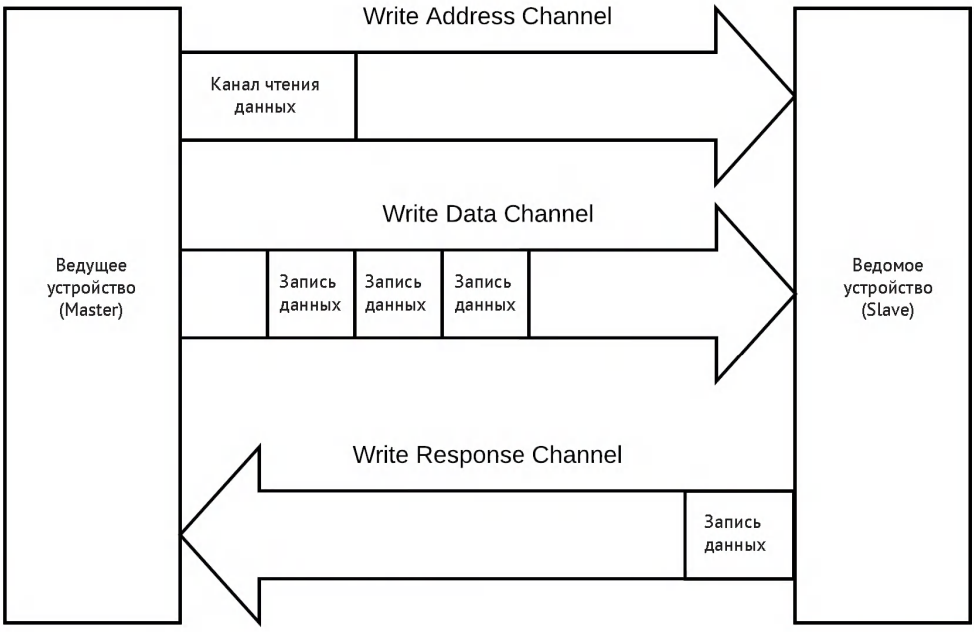


Рис. 7.25. Канал записи AXI

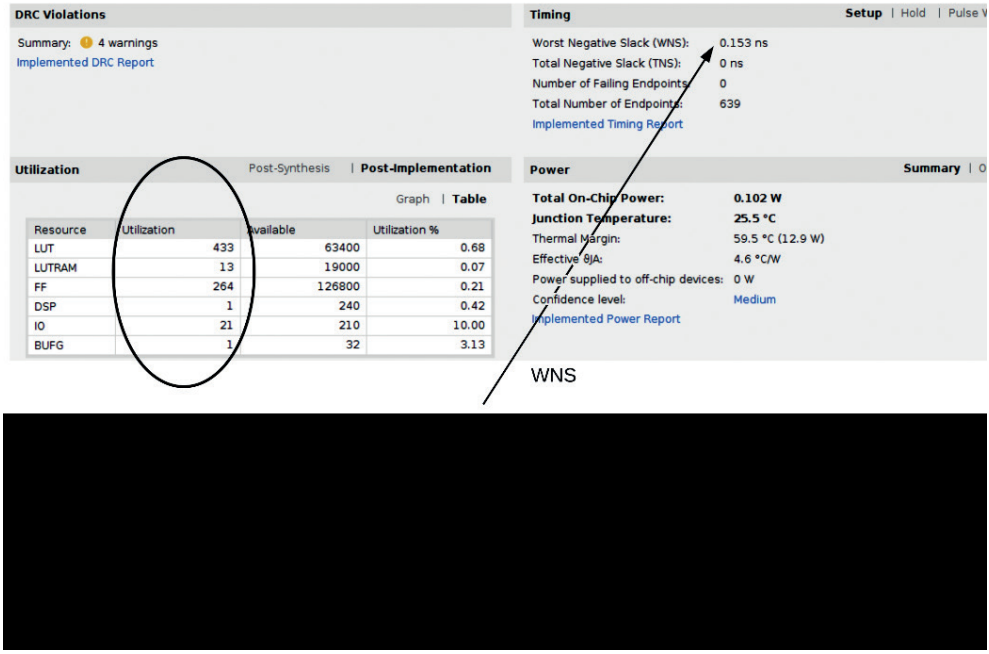


Рис. 7.26. Доступные AXI IP-блоки

Типичный проект состоит из микроконтроллера, такого как ARM Cortex или Xilinx MicroBlaze, или пользовательской логики и подключенных периферийных устройств. Периферийные устройства обычно подключаются к соединительному блоку AXI, реализация которого может оказаться дорогостоящей в зависимости от количества активных ведущих интерфейсов, а также от того, включена ли буферизация и является ли коммутатор полным, или нет.

Рассмотрим альтернативный способ создания IP-блока с нуля.

Разработка IP-блоков – AXI-Lite, AXI full и AXI Stream

В этом разделе будет показано, как можно разработать IP-блок путем создания пакета (package), начав с определения интерфейса.

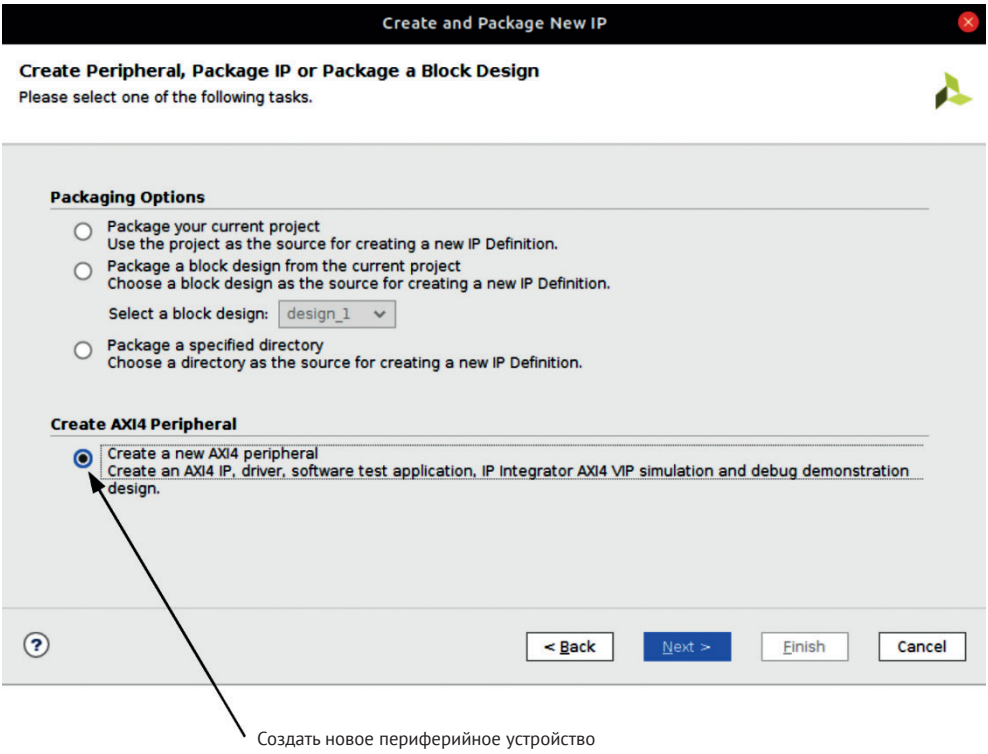


Рис. 7.27. Создание нового периферийного устройства, поддерживающего интерфейс AXI4

Это способ создания IP-блока путем создания сначала HDL-обертки, а затем вставки в нее разработанного IP-блока.

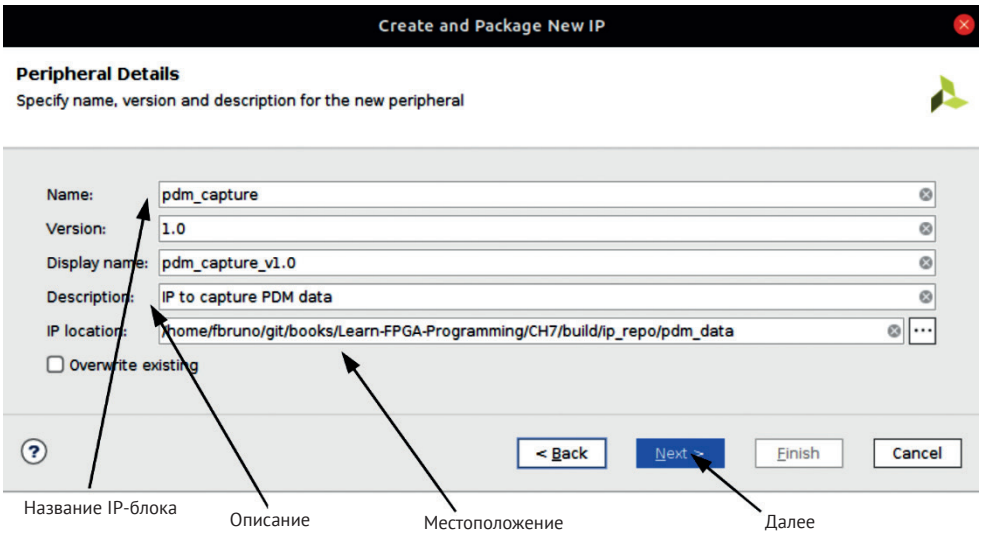


Рис. 7.28. Описание IP-блока

Создается модуль pdm_capture, в котором будет определен регистр для запуска чтения. Затем можно считать данные из того же регистра, чтобы определить, завершено ли чтение. После чего данные могут быть считаны из второго регистра.

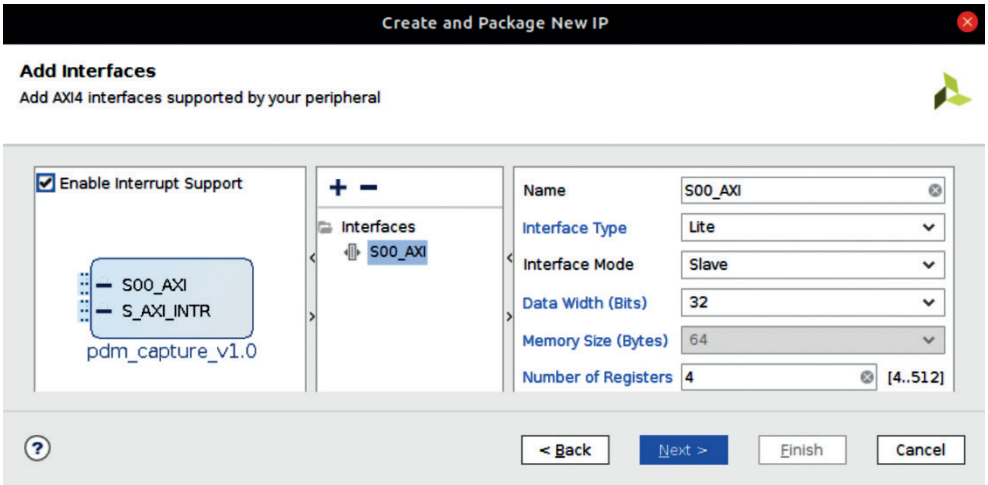


Рис. 7.29. Определение интерфейса по умолчанию

Определение интерфейса по умолчанию идеально подходит для той задачи, которая сейчас реализуется. Вы можете изучить опции и убедиться, что очень легко добавить любой из интерфейсов AXI, которые здесь обсуждались. Если вы исследуете созданную папку IP-блока, то увидите следующие файлы, созданные в папке HDL:

- pdm_capture_v1_0.v: файл верхнего уровня иерархии IP-блока. Здесь будет добавлен интерфейс к микрофону;

- pdm_capture_v1_0_S00_AXI.v: AXI-часть проекта с регистрами.

Вы увидите в обоих сгенерированных модулях места для размещения логики и портов:

```
module pdm_capture_v1_0_S00_AXI #
(
  // Users to add parameters here (Здесь пользователи добавляют параметры)
  Place USER parameters here
  // User parameters ends (Конец секции для добавления параметров пользователя)
  // Не изменяйте параметры после этой линии
```

Посмотрев на список портов, можно увидеть, где разместить порты, необходимые для файла верхнего уровня иерархии:

```
(
  // Users to add ports here (Здесь пользователи добавляют определения портов)
  Place USER ports here
  // User ports ends (Конец секции определения портов пользователем)
  // Не изменяйте порты после этой линии
```

В самом проекте есть комментарии о том, куда добавлять пользовательский код:

```
// Add user logic here (Добавляйте здесь пользовательскую логику)
  Place USER logic here
// User logic ends (Конец пользовательской логики)
```

Существует дополнительный способ добавления IP-блоков в IP-интегратор, который не предполагает явного создания IP-блока, но в настоящее время он требует, чтобы файл верхнего уровня иерархии IP-блока был файлом Verilog, а не SystemVerilog.

Добавление неупакованного IP-блока в IP-интегратор

По адресу https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH7/build/i2c_temp_flt_bd/i2c_temp_flt_bd.xpr находятся два файла, `adt7420_i2c_mod.sv` и `adt7420_i2c_bd.v`. Сейчас `adt7420_i2c_mod.sv` – это копия IP-блока `adt7420_i2c.sv`, переименованная так, чтобы не возникало проблем с конфликтом имен. `adt7420_i2c_bd.v` является Verilog-оберткой.

Добавьте IP-блок, выполнив следующие действия.

1. Откройте https://github.com/PacktPublishing/Learn-FPGA-Programming/CH7/build/i2c_temp_dlt_bd/i2c_temp_dlt_bd.xpr.
2. Кликните правой кнопкой мыши на **Design Sources** и добавьте файлы, как показано на рис. 7.30:

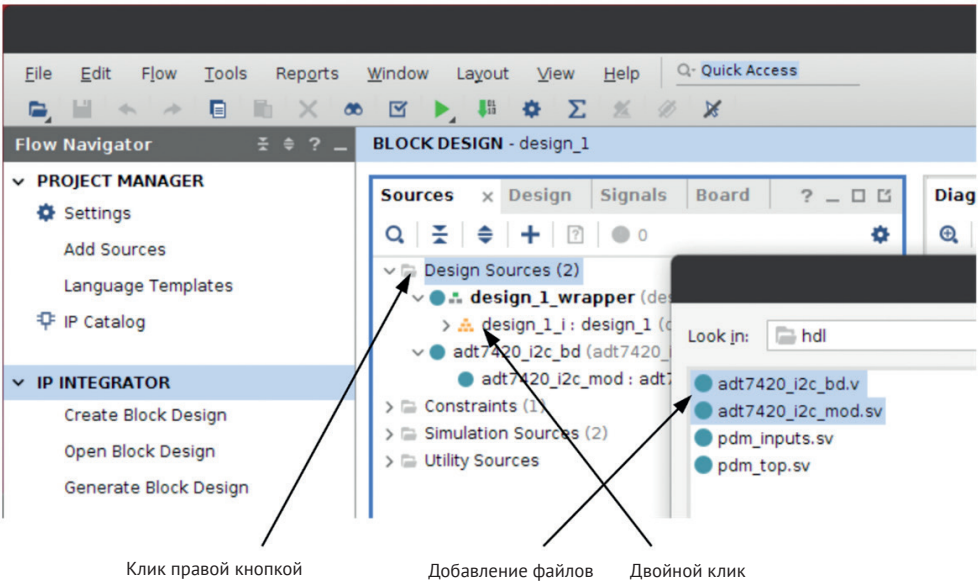


Рис. 7.30. Добавление файлов в проект

3. Откройте IP-интегратор.
4. В схемном редакторе кликните правой кнопкой мыши и выберите действие **Add Module** (добавить модуль).

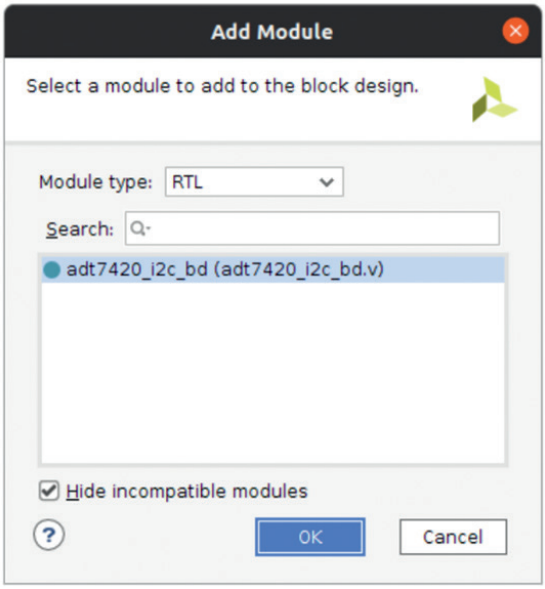


Рис. 7.31. Добавление модуля в IP-интегратор

5. Выберите **OK**.

Теперь можно сравнить HDL-модуль с IP-блоком, который был создан ранее.

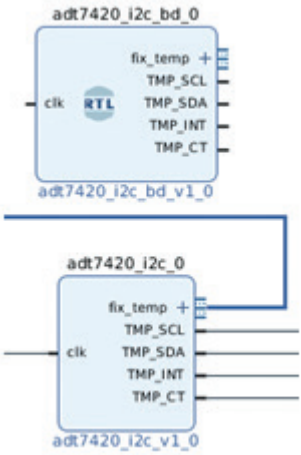


Рис. 7.32. HDL-модуль в сравнении с IP-блоком

HDL-модуль выполняет те же функции, что и IP-блок, созданный ранее в этой главе. Вы можете изменять параметры, а интерфейсы автоматически определяются, если они названы правильно.

На этом изучение IP-интегратора можно пока приостановить и изучить другие возможности FPGA в следующих главах.

Выводы

В этой главе было рассмотрено, как генерировать IP-блоки из существующего файла SystemVerilog и как это использовать для воссоздания проекта датчика температуры с помощью IP-интегратора. Было показано, как можно выполнять отладку проекта с помощью IP-интегратора и как ILA поддерживает AXI, а также как можно упаковать IP-блок с помощью IP-упаковщика для создания обертки с интерфейсами AXI, которую можно использовать для создания проектов.

Был пройден путь от мигания светодиодов в главе 1 «Введение в архитектуры FPGA и Xilinx Vivado» к использованию семисегментного индикатора и для отображения информации в главе 3 «Подсчет нажатий на кнопку». В главе 8 «Много данных? MIG и DDR2» будет рассмотрена разработка контроллера дисплея с использованием интерфейса **Video Graphics Array (VGA),** что даст гораздо больше возможностей для отображения выходных данных от датчика температуры, микрофона и калькулятора.

Вопросы

1. Для чего лучше всего подходят потоковые интерфейсы AXI?
   a) Для потоковых транзакций к нескольким адресам памяти.
   b) Для соединений точка–точка.
   c) Для высокопроизводительных соединений.
   d) b и c.
2. Что такое IP-интегратор?
   a) Простой способ создания блочных конструкций с использованием IP-блоков компании Xilinx или пользовательских IP-блоков.
   b) Контекстно-зависимый редактор для HDL-проектов.
   c) Не очень хорошо помогает в отладке проекта.
3. Если нужно создать IP-блок на основе существующего проекта, то требуется использовать **Create and package new IP** (создать и собрать новый IP). Верно или нет?
4. Вы не можете использовать **Create and package new IP**, чтобы создать обертку проекта с интерфейсами AXI для создания собственных проектов. Верно или нет?
5. Когда следует использовать интерфейс AXI full?
   a) Когда нужен высокопроизводительный интерфейс, способный передавать данные в несколько адресов памяти.
   b) При редкой записи только в один регистр за раз.
   c) Когда нужно переместить много данных между двумя отдельными ядрами, где местом назначения является FIFO-подобный интерфейс.
   d) Постоянно. Они дешевы в реализации и могут выполнять все необходимые функции.

Дополнительное чтение

Для получения дополнительной информации о том, что было рассмотрено в этой главе, обратитесь к следующим ссылкам.

- https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf.
- https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification.

Глава 8

Много данных? MIG и DDR2

До сих пор работа выполнялась над созданием все более функционального устройства, способного собирать информацию, выполнять полезную работу и представлять ее в осмысленном виде. В предыдущих главах собирались аудиоданные и данные о температуре. Было показано, как создавать обертку некоторых интерфейсов, чтобы можно было использовать IP-интегратор. IP-интегратор также позволил легко внедрять в проект выполнение операций с плавающей точкой. Это дало возможность создать несколько относительно сложных проектов, но при этом все еще оставалось ограничение, состоящее в том, что выводить информацию можно было с помощью светодиодов и семисегментного индикатора, что затрудняло визуализацию такой информации, как форма сигнала PDM или даже температуры.

Что касается отображения информации с помощью используемых отладочных плат, то есть еще один вариант: разъем VGA. Чтобы эффективно его использовать, необходим доступ к довольно большому объему памяти. Для отображения 8-битного цвета 640×480 требуется 300 килобайт, а для полноцветного изображения – почти 1 Мб. Конечно, такой объем памяти можно попытаться реализовать на ресурсах FPGA, но в качестве альтернативы можно использовать расположенную на платах внешнюю память **Double Data Rate, 2nd generation (DDR2)** в качестве кадрового буфера и формировать изображение в нем.

К концу этой главы вы познакомитесь с внешней памятью, создадите контроллер DDR2 и выполните его отладку как с помощью моделирования, так и путем прототипирования на плате. Вы освоите использование внешней памяти при проектировании устройств на FPGA.

В этой главе будут рассмотрены следующие основные темы:

- основы организации памяти DDR;
- использование генератора интерфейса памяти Xilinx **Memory Interface Generator (MIG)**;
- краткий обзор других типов памяти.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/ PacktPublishing/Learn-FPGA-Programming/tree/master/CH8.

Проект 10. Подключение внешней памяти

До сих пор в разобранных примерах в этой книге использовалась внутренняя **блочная оперативная память (Block RAM, BRAM)** или распределенная оперативная память. Эти типы памяти очень быстрые. Доступ к BRAM можно получить за один тактовый цикл вплоть до **максимальной частоты (maximum frequency, fmax)** устройства при определенных ограничениях. **Память LUT (Look up table memories, LUTRAM)** немного более гибкая, поскольку может использоваться асинхронно. Оба типа памяти очень удобны для хранения небольших объемов данных, таблиц поиска, быстрой памяти для реализации кеша, а также, если в чипе FPGA достаточно памяти для проектирования, для снижения стоимости и сложности конечного устройства.

Существует множество типов внешней памяти, доступных для использования в проектах. Если посмотреть только на разновидности синхронной **динамической оперативной памяти (Dynamic RAM, DRAM)**, которые все еще используются, то можно увидеть, как менялась их производительность с каждым новым поколением.

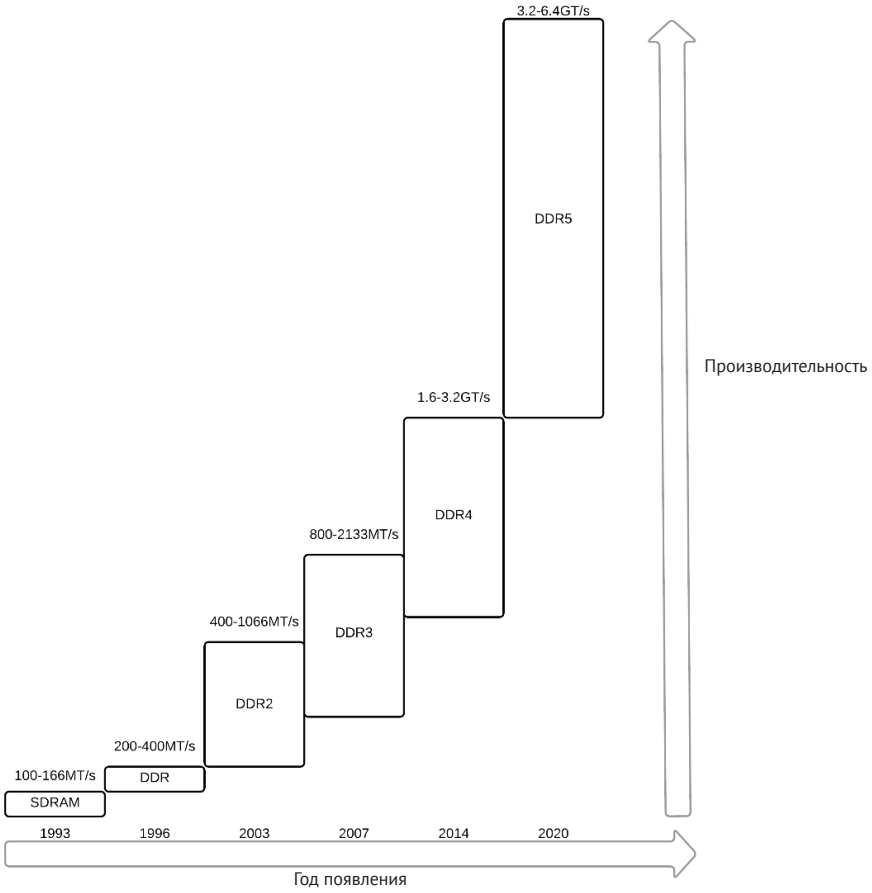


Рис. 8.1. Производительность внешней DRAM (в миллионах (M) и миллиардах (G) тактов в секунду, T/s)

Глядя на представленный график, первый вопрос, который возникает, – какова производительность внутренней памяти BRAM по сравнению с внешней памятью? Чем приходится жертвовать в каждом случае? Производительность одной BRAM в FPGA находится в пределах 300 МТ/с. Как обсуждалось ранее, можно воспользоваться преимуществами параллелизма и получить доступ ко многим BRAM параллельно, хотя для эффективного использования всей этой памяти потребуется довольно сложная параллельная реализация проекта.

Недостатком внешней памяти с каждым последующим поколением является то, что задержка – время, которое требуется от момента запроса данных до их получения – увеличивается. Обращение к SDRAM может занимать от пяти и более тактовых циклов при частоте 100 МГц. Обращение к DDR4 может занимать 80 тактовых циклов при частоте 300 МГц. Логика взаимодействия с памятью также становится очень сложной. Если контроллер DDR2 можно разработать самостоятельно, то контроллеры DDR4 от Xilinx имеют встроенные процессоры для обработки инициализации и периодических операций с памятью. Когда в этой главе будет реализован контроллер памяти, то можно будет проанализировать его временные параметры с помощью моделирования.

Данная глава не охватывает проектирование контроллера памяти. Если вам интересно, то можно поискать контроллеры памяти SDRAM или DDR в сети Интернет. Их много в свободном доступе.

Как уже упоминалось, DRAM предоставляет большой объем памяти для хранения данных. Новые FPGA оснащены дополнительной **памятью с высокой пропускной способностью (High-Bandwidth Memory, HBM)**, которая обеспечивает внутреннюю память DRAM большой емкости, но для большинства FPGA экономически эффективным решением является использование внешней памяти.

Память DDR2

Контроллер памяти Xilinx делает за разработчика очень многое. BRAM/LUTRAM – это **статическая оперативная память (Static RAM, SRAM**). Статическая память занимает слишком много места по сравнению с динамической памятью. Статическая память может занимать четыре транзистора на битовую ячейку. Динамическая память, напротив, в основном представляет собой конденсатор, используемый для удержания заряда, с присоединенным к нему одним транзистором. Она также построена на оптимизированном процессе ASIC для минимизации площади чипа. Это позволяет достичь гораздо более высокой плотности размещения (рис. 8.1).

Динамическая память разделена на **строки, столбцы** и **банки**. Одна строка, содержащая несколько столбцов в банке, может быть активна в любой момент времени. Это достигается путем отправки в память команды активации. Как только строка открывается в определенном банке, столбцы быстро становятся доступными.

В памяти DRAM может быть несколько банков, в каждом из которых одновременно открыты разные строки. Это обеспечивает быстрый доступ к большим блокам данных. Когда нужно переключить строку в банке, открытая строка должна быть предварительно перезаряжена, чтобы закрыть ее перед активацией нового банка или строки.

На DRAM периодически должна подаваться команда обновления. Перед обновлением все открытые банки должны быть предварительно перезаряжены.

Обновление считывает строку и записывает ее обратно, чтобы обновить заряд в конденсаторах, хранящих данные.

Теперь, получив эту информацию, давайте рассмотрим шаги по созданию памяти.

Генерация контроллера DDR2 с помощью Xilinx MIG

Nexys A7 имеет то же расположение пинов, что и Nexys DDR. Вы можете найти готовый проект на сайте https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start?_ga=2.168036321.1345263114.1604794648-84804473.1599434198#additional_resources.

Тем не менее здесь будет рассмотрен процесс создания компонента, чтобы показать настройки и их связь с основной архитектурой DDR2.

1. Для генерации контроллера MIG (Memory Interface Generator, генератор интерфейса памяти) используют каталог IP-блоков.

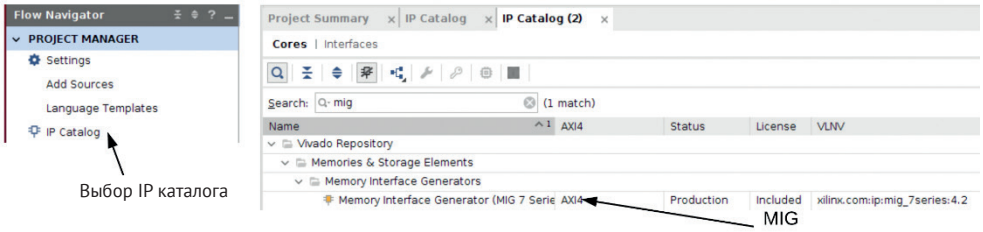


Рис. 8.2. Запуск MIG

2. Выберите **IP Catalog** (IP-каталог) и выполните поиск по слову `MIG`. Затем дважды кликните мышкой, чтобы запустить мастер для настройки MIG.

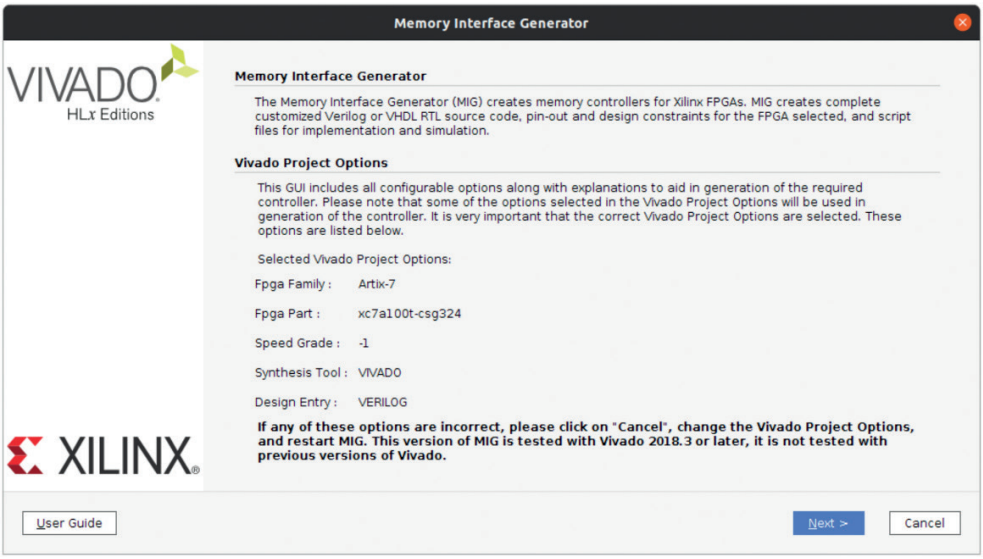


Рис. 8.3. Стартовый экран мастера для настройки MIG

3. Проект уже должен быть настроен для конкретной платы. Выберите тип FPGA для платы, после чего нажмите **Next.**

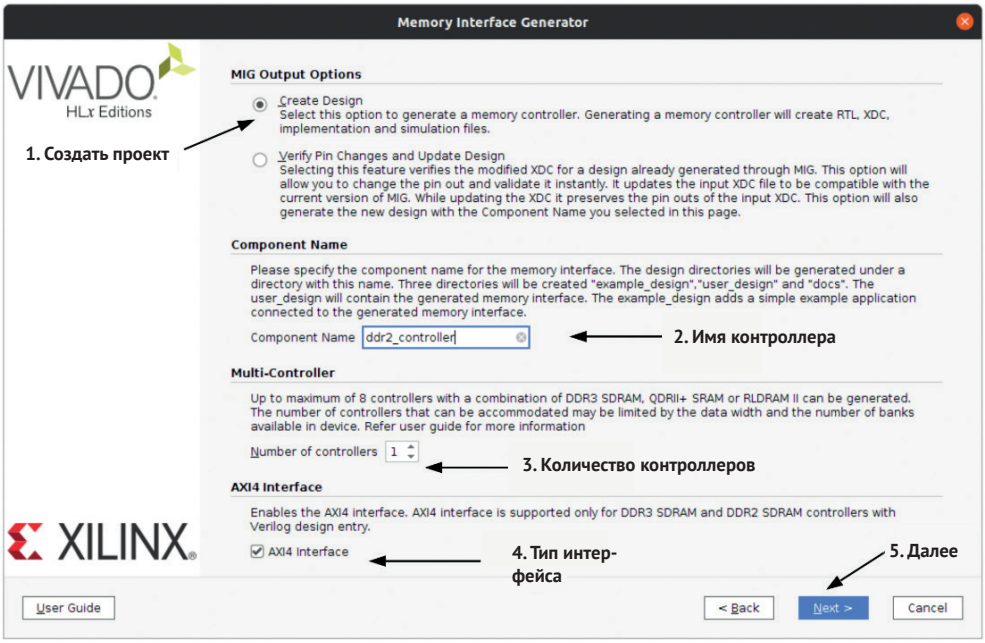


Рис. 8.4. Варианты MIG

4. Теперь нужно выбрать опцию **Create Design** (создать проект), чтобы создать новый проект MIG. После этого следует дать контроллеру более значимое название. На плате нужен только один контроллер. Интерфейс выбран с поддержкой AXI4, а не пользовательского интерфейса. Затем выберите **Next**, чтобы проверить совместимость выводов.

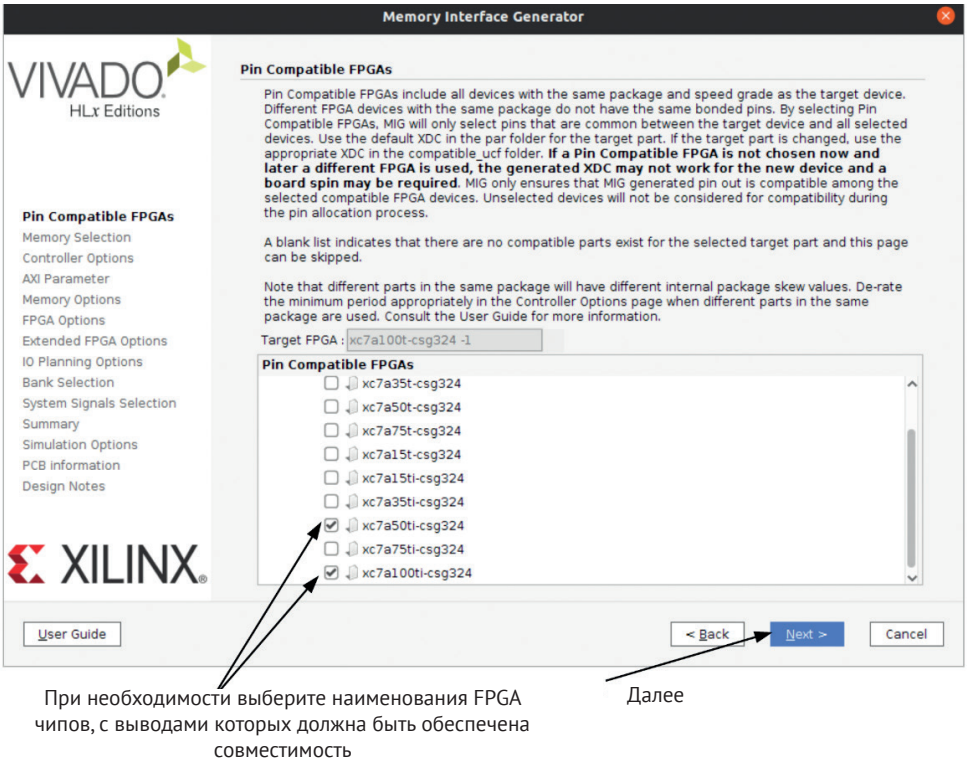


Рис. 8.5. Совместимость выводов

5. Если вы создаете контроллер MIG для своей собственной платы, то, возможно, захотите иметь возможность использовать контроллер с большей или меньшей FPGA. В этом случае можно выбрать несколько совместимых по выводам FPGA, чтобы убедиться, что контроллер MIG будет работать в любом чипе FPGA, который вы в конечном итоге установите на свою плату. Для этого можно выбрать 50-ю и 100-ю серию чипа, но в Digilent уже позаботились о правильной конфигурации для обеих плат. Нажмите **Next**, чтобы выбрать тип памяти:

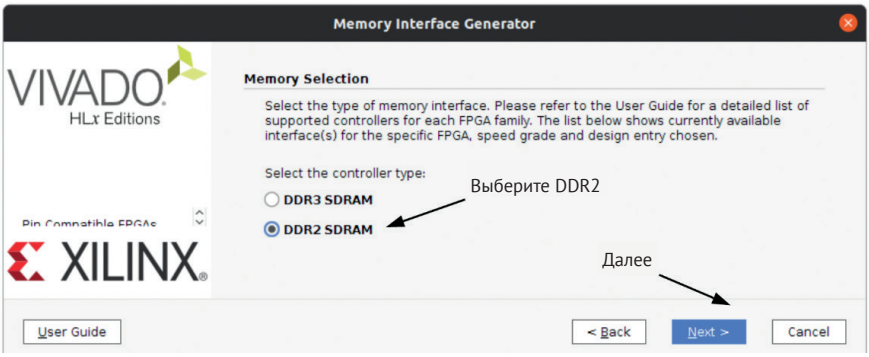


Рис. 8.6. Выбор памяти

6. Artix 7 поддерживает DDR2 и 3 с использованием MIG. На используемой плате есть DDR2, поэтому убедитесь, что выбрали ее. Хотя все DDR имеют похожие имена, они несовместимы, поэтому убедитесь, что вы выбрали правильный тип для своей платы. Нажмите **Next**, чтобы мы могли выбрать параметры контроллера:

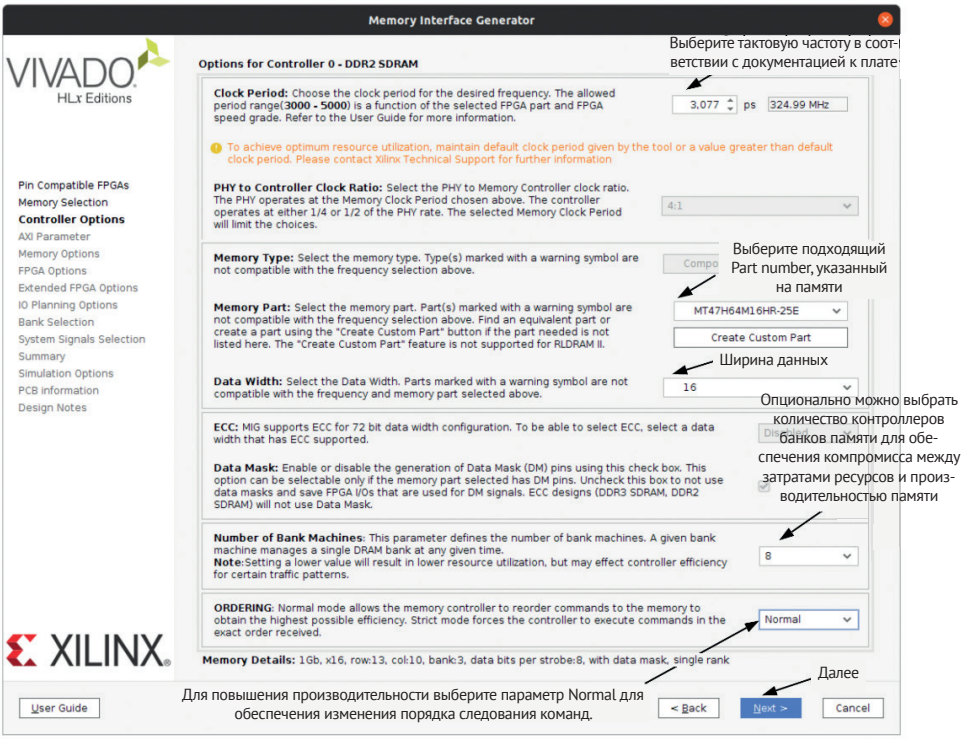


Рис. 8.7. Параметры контроллера

Если посмотреть на раздел 3.1 документации Nexys A7 (ссылка в разделе «Дополнительное чтение»), то там рекомендуются настройки, которые будут использованы в этом разделе. Самое главное – указать правильный Part number. В документации к платам от компании Digilent также рекомендуется использовать немного меньшую тактовую частоту для упрощения реализации проекта. Выберите Data width (ширину данных) равной 16. Есть еще два параметра, которые являются необязательными.

Первый – это количество контроллеров управления банками памяти (bank machines). Помните, что устройство имеет восемь банков. Для повышения производительности можно реализовать несколько контроллеров управления за счет увеличения используемых ресурсов в разрабатываемом проекте.

Второй параметр определяет упорядочивание запросов. Если разрешить изменение порядка следования команд, то это может повысить производительность.

Рассмотрим параметры AXI.

Установка параметров интерфейса AXI

Есть несколько параметров интерфейса AXI, которые можно контролировать.

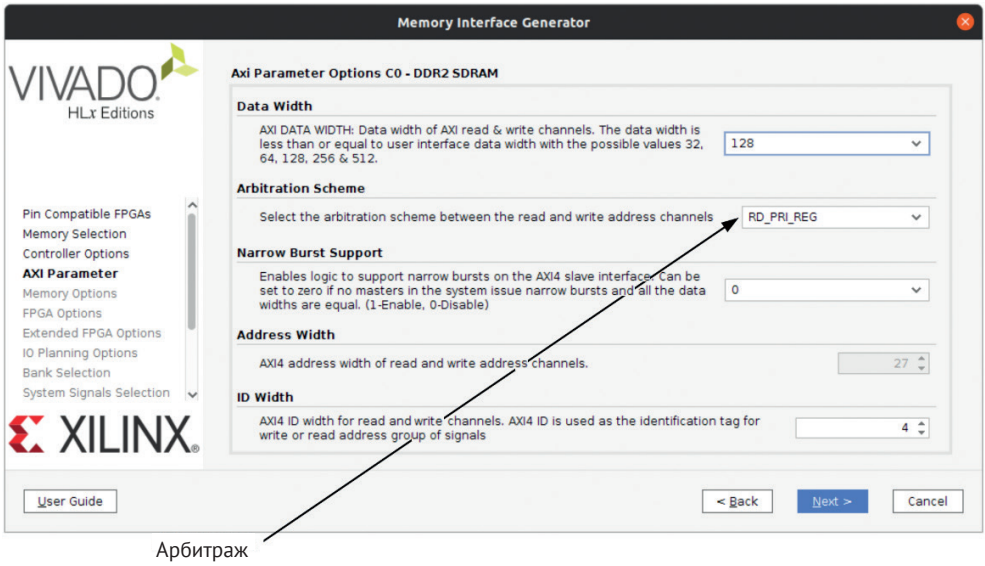


Рис. 8.8. Параметры интерфейса AXI

Главное, что нужно учесть в параметрах интерфейса AXI, – это арбитражная схема. Поскольку основная задача контроллера – управление дисплеем, нужно убедиться, что контроллер дисплея всегда будет иметь доступ к памяти. Поэтому для арбитража следует указать приоритет чтения. В следующем разделе будут заданы параметры памяти.

Настройка параметров памяти

Есть два параметра, которые нужно установить согласно документации Digilent. Они показаны на рис. 8.9. Первый – это способ организации последовательной потоковой передачи данных (burst transmission), который может быть последовательным или с чередованием адресов. Для работы с внешней памятью довольно редко используется тип с чередованием, но такой вариант предусмотрен.

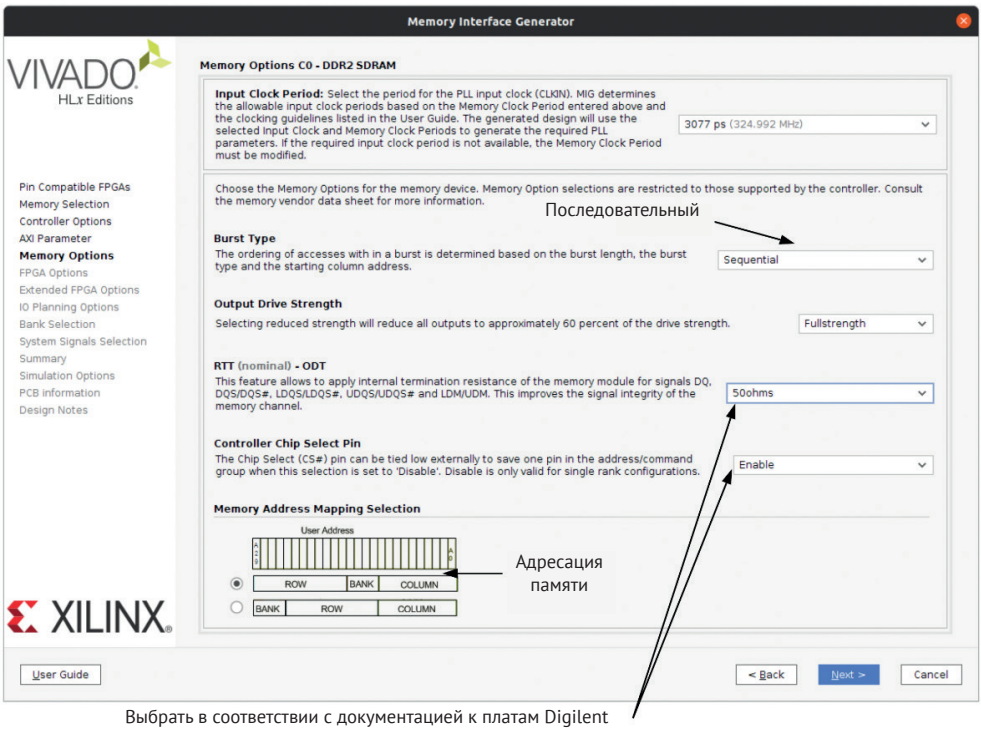


Рис. 8.9. Параметры памяти

Также есть выбор способа адресации памяти. Любой адрес состоит из трех компонентов: банка, строки и столбца. Память может открывать только одну строку в каждом банке, но может иметь одну активную строку для каждого из восьми банков. В идеале нужно предварительно проанализировать схему использования памяти, чтобы максимизировать производительность. Часто это делается путем предварительного анализа разрабатываемой системы[1] или с помощью событийного моделирования. Необходимо выбрать параметры **[ROW, BANK, COLUMN]** ([строка, банк, столбец]), чтобы получить доступ к нескольким банкам при реализации контроллера VGA в следующей главе.

Рассмотрим параметры FPGA для контроллера.

Настройка параметров FPGA

Для FPGA доступен ряд параметров, некоторые из которых нужно изменить для используемой платы.

1. Будут использоваться внутренние тактовые сигналы, поэтому для системного (system clock) и опорного тактового генератора (reference clock) нужно указать значение **No Buffer** (без буфера). Также надо активировать интерфейс отладки, чтобы можно было рассмотреть изнутри, что происходит в FPGA.

[1] Часто разработанной на одном из языков высокоуровневого программирования, например C++. – *Прим. ред.*

Остальные настройки можно оставить по умолчанию.

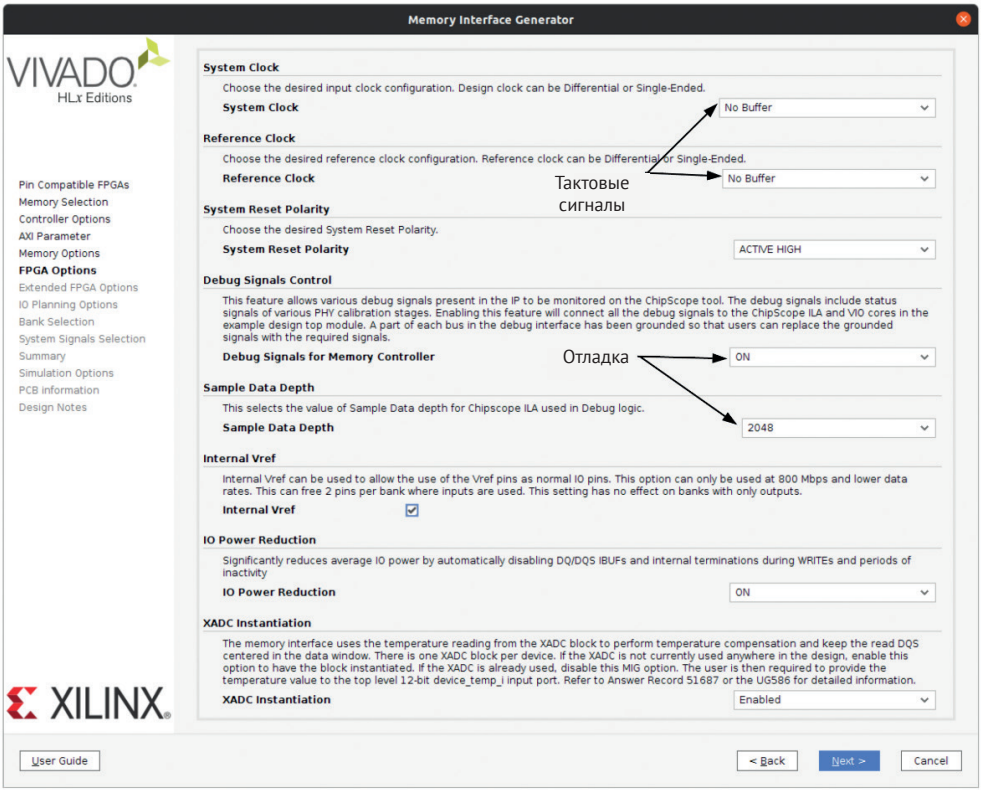


Рис. 8.10. Параметры MIG FPGA

2. На следующем экране можно оставить все по умолчанию.

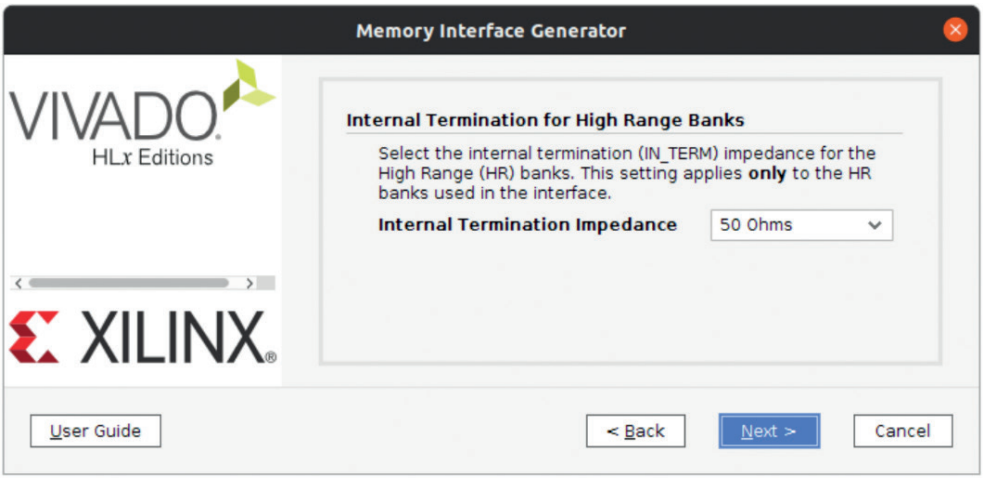


Рис. 8.11. Расширенные параметры FPGA

3. На следующем экране нужно указать соответствие выводов проекта выводам FPGA. Компания Digilent предоставляет файл `.ucf` для ограничений DDR. Он включен в `CH8/build/xdc/mig.ucf`.

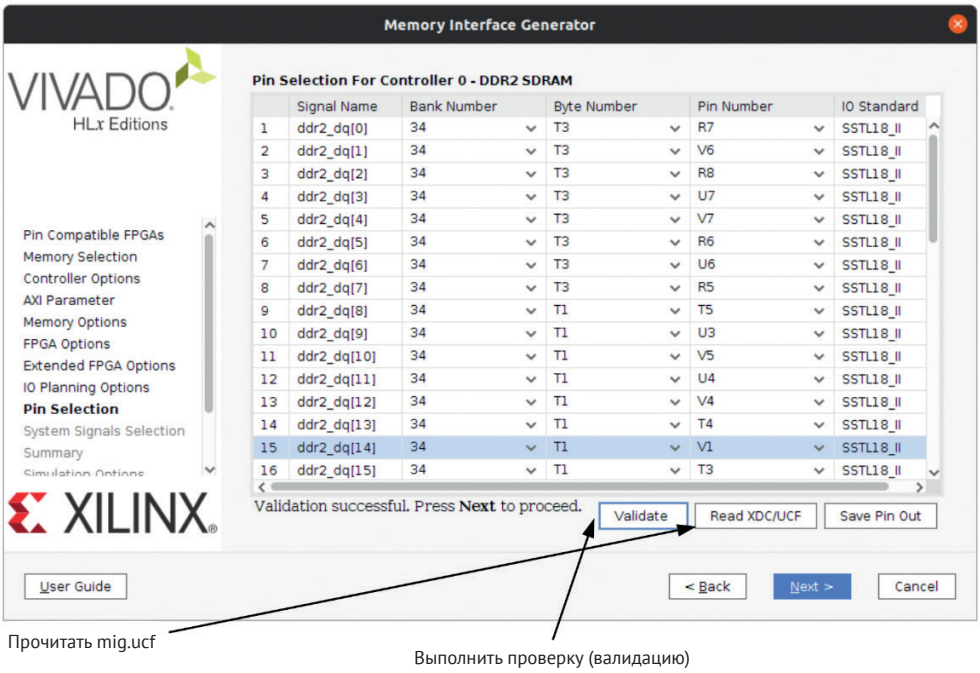


Рис. 8.12. Настройки выводов проекта

4. Сначала выберите **Read XDC/UCF** и прочитайте файл `mig.ucf`. Затем необходимо выбрать **Validate** для подтверждения соответствия выводов требованиям и разблокирования кнопки **Next**. После этого можно перейти к выбору системных сигналов.

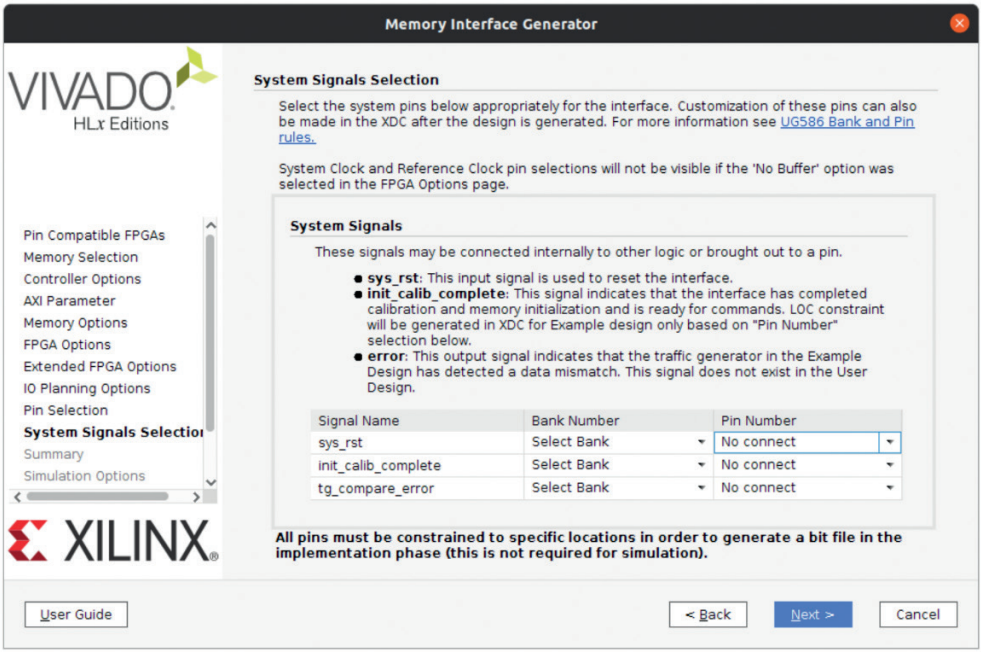


Рис. 8.13. Выбор системных сигналов

5. Для некоторых сигналов уже установлено, что они внутренние. Остальные также будут использоваться как внутренние, поскольку они не выведены на плату. Не меняйте ничего на этом экране. После нажатия на кнопку **Next** будет выведен итоговый экран.

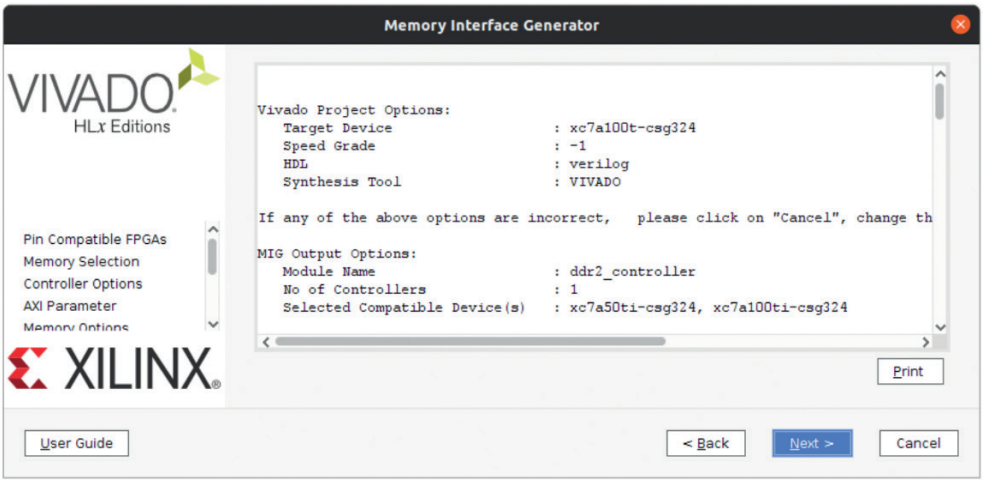


Рис. 8.14. Сводка по MIG

6. На этом экране можно просмотреть сводку контроллера DDR2, который будет сгенерирован. Убедитесь, что она соответствует выбранным параметрам, а затем нажмите кнопку **Next**.

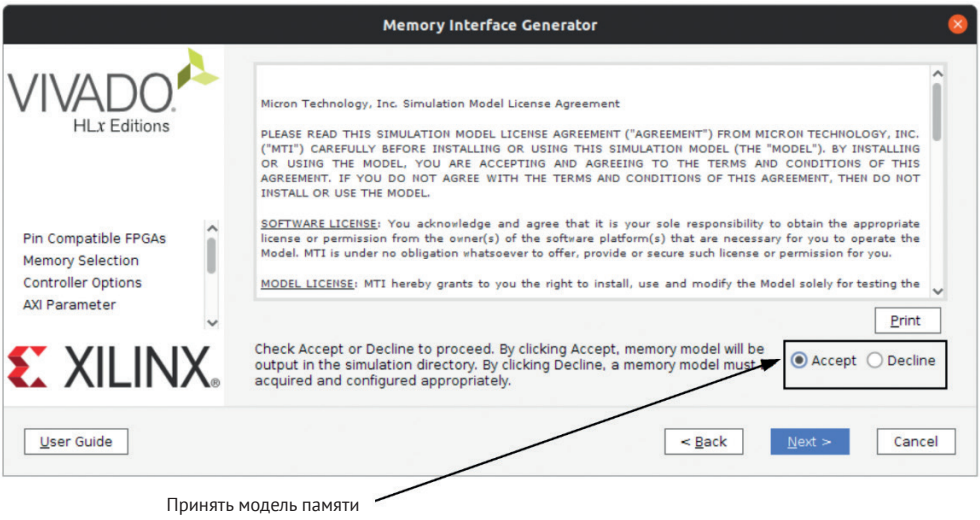


Рис. 8.15. Параметры моделирования памяти

7. Xilinx предоставляет модель памяти DDR от компании Micron. Чтобы ее сгенерировать, требуется принять лицензию. Это целесообразно, поскольку позволит выполнять моделирование на основе реальной модели DDR2. Выберите **Accept** (принять) или **Decline** (отклонить), затем – **Next**.

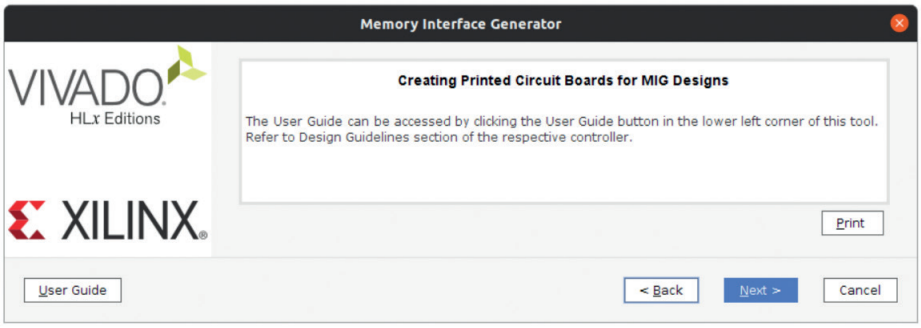


Рис. 8.16. Информация о печатной плате

8. MIG напоминает о возможности ознакомиться с руководством пользователя в случае, если разрабатывается собственная **печатная плата (PCB).** Поскольку компания Digilent уже позаботилась об этом за разработчика, это руководство можно смело игнорировать. Если в будущем потребуется спроектировать собственную плату, Xilinx предоставляет множество ресурсов и контрольных параметров, которым нужно следовать. Изучите руководство или просто нажмите кнопку **Next**.

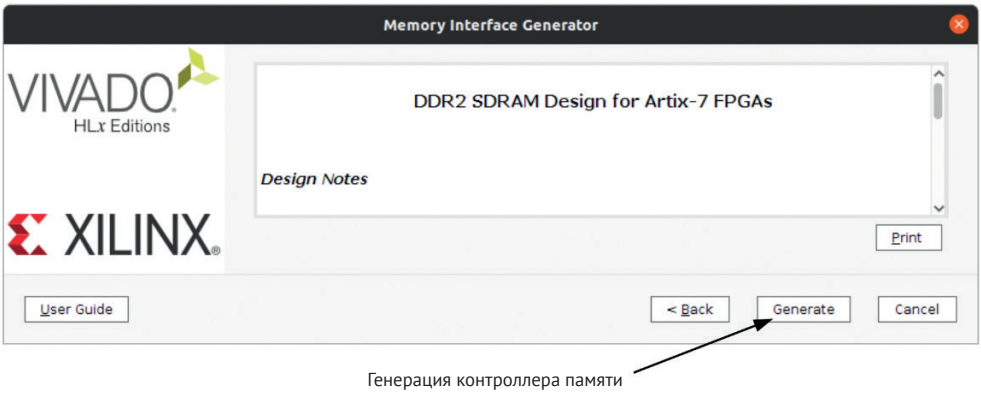


Рис. 8.17. Генерация IP-ядра контроллера памяти

Это последний этап выбора параметров, и теперь можно просто нажать кнопку **Generate** (Сгенерировать). Появится еще одно окно для генерации дополнительных модулей (Out of Contex, вне контекста). Снова нажмите **Generate**.

9. Теперь генерация ядра завершена. Сгенерированное IP-ядро появится в списке исходных компонентов проекта.

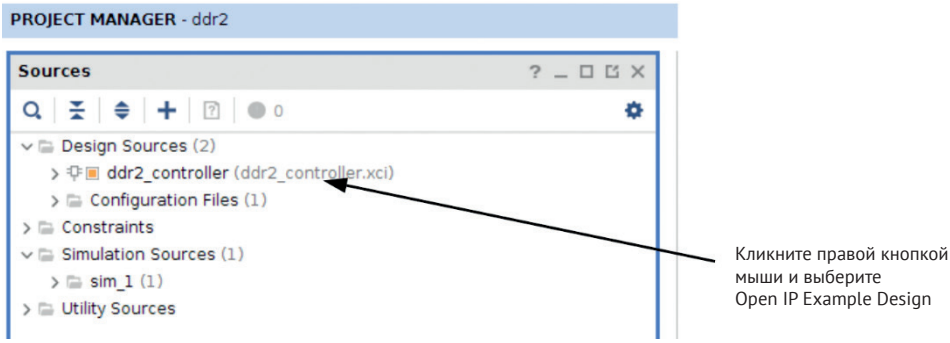


Рис. 8.18. Создание тестового проекта

Полезной особенностью многих ядер Xilinx – в частности, ядра MIG – является то, что можно создать тестовый проект (пример проекта) для его моделирования и иногда реализации. Как уже говорилось ранее, при оптимизации адресации скорее всего потребуется провести моделирование, и тестовый проект может помочь в этом.

10. Чтобы сгенерировать тестовый проект, кликните правой кнопкой мыши на файле `.xci` в Design Sources проекта и выберите **Open IP Example Design**.

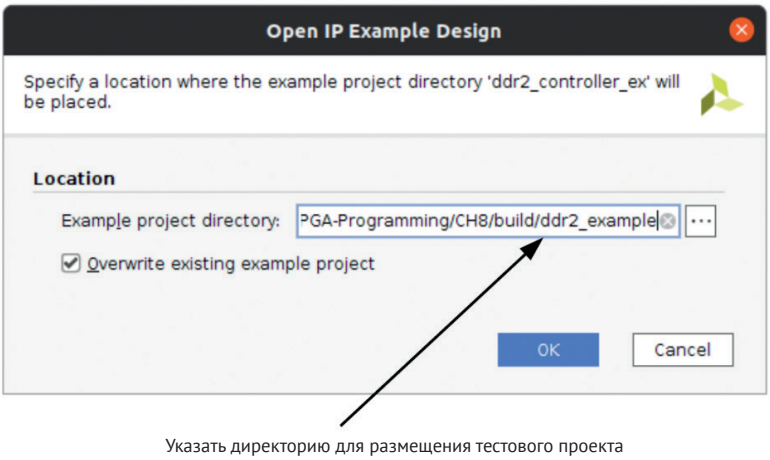


Рис. 8.19. Открытие тестового проекта

11. Убедитесь, что выбрана правильная папка для тестового проекта, и нажмите **OK**.

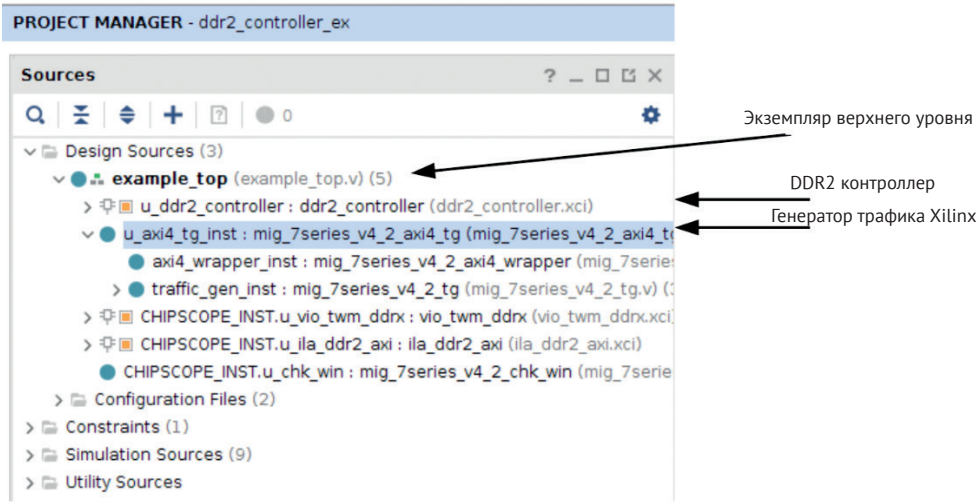


Рис. 8.20. Просмотр тестового проекта

В результате будет загружен тестовый проект, а также простой `testbench`. Нужно будет изменить его файл верхнего уровня иерархии, чтобы запустить его на плате и посмотреть на него с помощью **интегрированного логического анализатора (Integrated Logic Analyzer, ILA)**. Прежде чем что-то менять, необходимо проверить, что созданный `testbench` работает без проблем. Это хорошая инженерная практика. Нет ничего хуже, чем внести изменения и запустить на плате проект, а затем понять, что первоначально сгенерированный проект не работал должным образом.

Для этого выберите **Run Simulation** (Запуск симуляции), как это делалось ранее.

Обратите внимание, что моделирование занимает гораздо больше времени, чем это было сих пор. По временной шкале, на которой работает `testbench`, видно, что для симуляции DDR требуется очень точная синхронизация:

```
`timescale 1ps/100fs
```

Также проанализируйте вывод симуляции, чтобы увидеть, как долго она длится. Фаза инициализации и настройки занимает 100 мкс. Самотестирование занимает примерно половину этого времени:

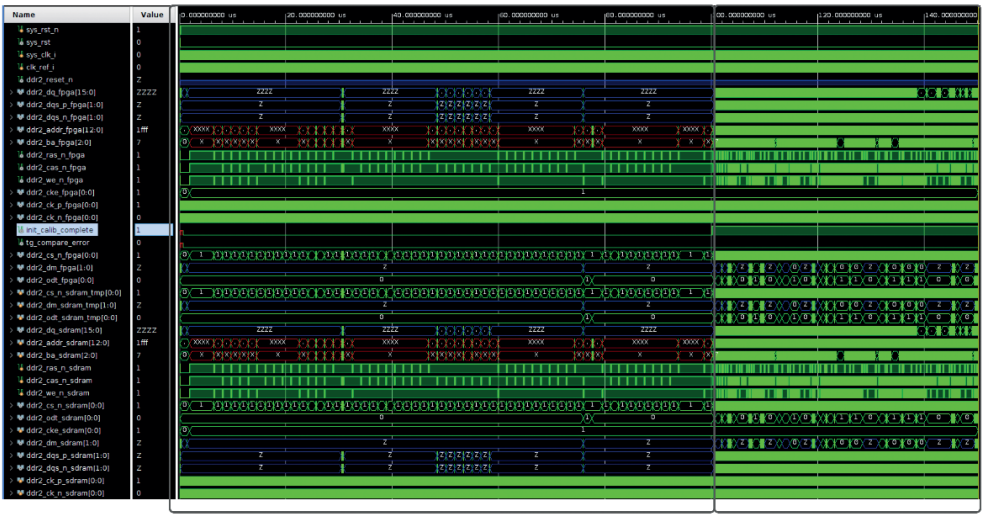


Рис. 8.21. Моделирование MIG

Если этап моделирования завершился успешно, в окне TCL-консоли должно быть выделенное слово `finish` (завершено):

```
548  //*****************************************************************************
549  // Reporting the test case status
550  // Status reporting logic exists both in simulation test bench (sim_tb_top)
551  // and sim.do file for ModelSim. Any update in simulation run time or time out
552  // in this file need to be updated in sim.do file as well.
553  //*****************************************************************************
554  initial
555  begin : Logging
556     fork
557        begin : calibration_done
558           wait (init_calib_complete);
559           $display("Calibration Done");
560           #50000000.0;
561           if (!tg_compare_error) begin
562              $display("TEST PASSED");
563           end
564           else begin
565              $display("TEST FAILED: DATA ERROR");
566           end
567           disable calib_not_done;
568           $finish;
569        end
```

Рис. 8.22. Моделирование успешно завершено

Проверка testbench Xilinx с помощью модуля генератора трафика Xilinx завершена. Продолжим модификацию проекта, чтобы можно было запустить его на плате и проанализировать его работу с помощью ILA.

Модификация проекта для использования на плате

Тестовый проект скопирован, чтобы можно было внести изменения, не затрагивая исходный проект. Если посмотреть на реализованный проект Xilinx, то сигнал тактовой частоты sys_clk_i генерируется на частоте 325 МГц. Первое, что нужно сделать, – это изменить файл верхнего уровня иерархии для платы, что означает, что нужно будет добавить **компонент управления тактовой частотой (Mixed Mode Clock Manager, MMCM)**, взять системный тактовый сигнал с частотой 100 МГц, из него сгенерировать тактовый сигнал с частотой 325 МГц. Также потребуется сгенерировать сигнал clk_ref_i. Если посмотреть на testbench, то можно увидеть, что эта тактовая частота должна быть 200 МГц.

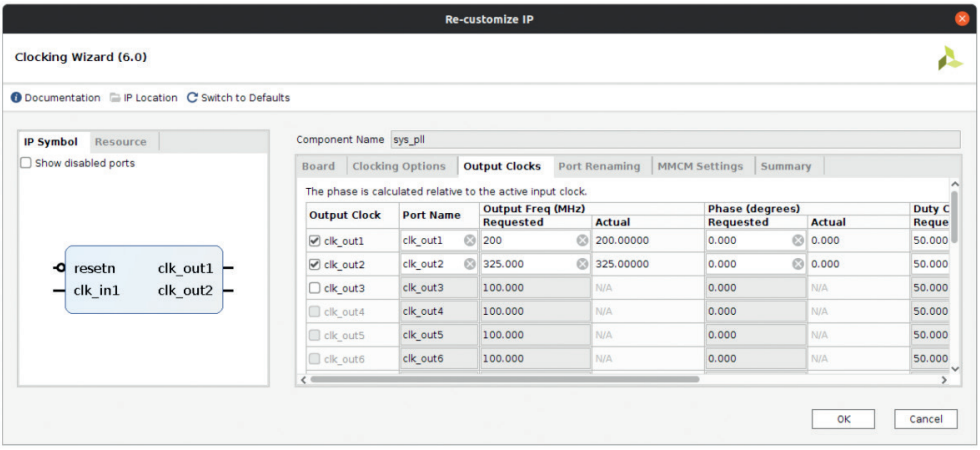


Рис. 8.23. Настройки компонента sys_pll

Обратите внимание, что в определениях выходных тактовых сигналов sys_pll имеет значение порядок сигналов. Если попытаться поменять частоты тактовых сигналов местами, т. е. вместо clk_out1 в 200 МГц и clk_out2 в 325 МГц они станут 325/200 МГц, может оказаться, что тактовые сигналы станут неточными. Иногда необходимо попробовать различные комбинации выходных тактовых сигналов, чтобы приблизиться к желаемому результату.

Далее рассмотрим файл верхнего уровня иерархии. В нем удалены ненужные порты и оставлены только те, что понадобятся.

```
inout [15:0]       ddr2_dq,
```

```
inout [1:0]        ddr2_dqs_n,
inout [1:0]        ddr2_dqs_p,
output [12:0]      ddr2_addr,
output [2:0]       ddr2_ba,
output             ddr2_ras_n,
output             ddr2_cas_n,
output             ddr2_we_n,
output [0:0]       ddr2_ck_p,
output [0:0]       ddr2_ck_n,
output [0:0]       ddr2_cke,
output [0:0]       ddr2_cs_n,
output [1:0]       ddr2_dm,
input              ext_clk,
output             tg_compare_error,
output             init_calib_complete,
input              sys_rst,
output LED
);
wire clk_ref_i;
wire sys_clk_i;
assign LED = sys_rst;
sys_pll u_sys_pll
 (.clk_out1        (clk_ref_i),
  .clk_out2        (sys_clk_i),
  .clk_in1         (ext_clk),
  .resetn          (sys_rst));
```

Удалены два тактовых сигнала, `clk_ref_i` и `sys_clk_i`, и сделаны внутренними. Также добавлен внешний тактовый сигнал, `ext_clk`, чтобы можно было получить 100 МГц от кварцевого генератора и использовать PLL для генерации тактовых сигналов в 200 и 325 МГц, необходимых для DDR2. Также создан экземпляр PLL в файле проекта верхнего уровня иерархии.

Также потребуется модифицировать `testbench` для работы на частоте 100 МГц:

```
parameter CLKIN_PERIOD          = 10000; //3077;
// Период входного тактового сигнала
```

Старый период тактового сигнала в 325 МГц закомментирован и заменен на 100 МГц. Теперь PLL будет генерировать правильные тактовые импульсы, и это можно проверить с помощью моделирования.

Наконец, соберем проект для Nexys A7. К сожалению, его нельзя запустить на плате Basys 3.

После загрузки образа на плату можно вызвать ILA. Генератор шаблонов и программа проверки работают постоянно, поэтому можно сразу же включить триггер и посмотреть на изменения сигналов.

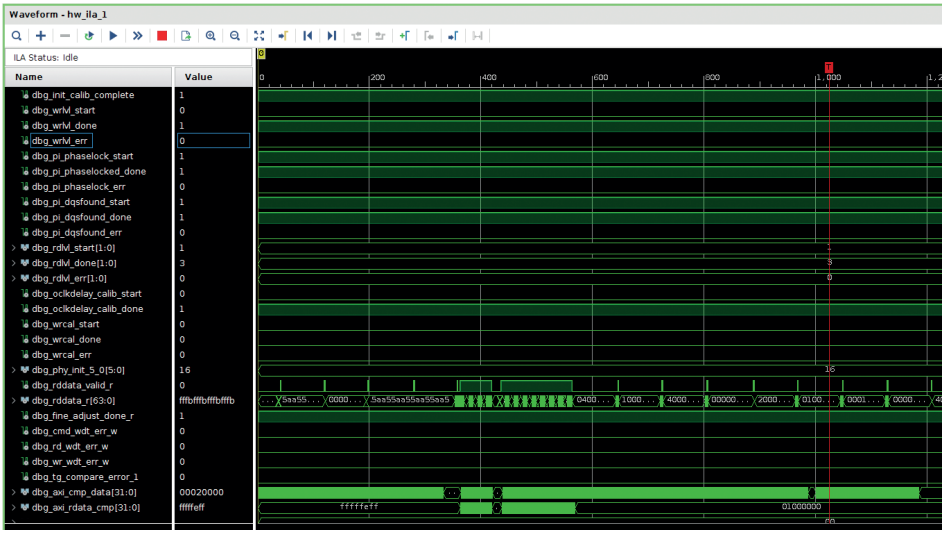


Рис. 8.24. DDR2 ILA

В ILA можно сразу же увидеть активность на внутренних интерфейсах DDR2. Светодиод на плате должен показать отсутствие обнаруженных ошибок, а ILA покажет сгенерированную активность.

Ядро DDR2 в том виде, в котором оно создано, имеет еще одно преимущество. В нем реализован интерфейс **виртуального ввода/вывода (Virtual I/O, VIO)** отладочного логического ядра Vivado. Этот интерфейс можно использовать в проектах для реализации функциональности ввода и вывода из Vivado, чтобы помочь отладке на чипе. В случае интерфейса DDR2 он дает представление о том, что происходит в ядре, а также позволяет изменять конфигурацию проекта на ходу.

Можно вызвать VIO, выбрав вкладку **hw_vios**.

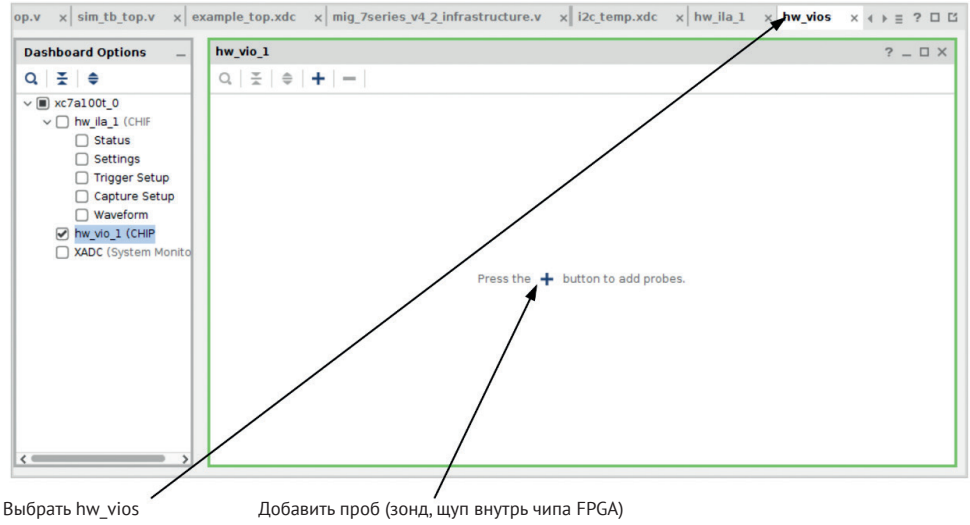


Рис. 8.25. Добавление аппаратных (HW) VIO

Можно нажать кнопку +, чтобы добавить тестовые сигналы. Просто добавьте все сигналы.

hw_vio_1

Name	Value	Activity	Direction	VIO
vio_modify_enable	[B] 0		Output	hw_vio_1
vio_tg_rst	[B] 0		Output	hw_vio_1
vio_tg_simple_data_sel[1:0]	[H] 0		Output	hw_vio_1
wdt_en_w	[B] 0		Output	hw_vio_1
win_active	[B] 0		Input	hw_vio_1
win_byte_select[6:0]	[H] 00		Input	hw_vio_1
win_current_bit[6:0]	[H] 00		Input	hw_vio_1
win_current_byte[3:0]	[H] 0		Input	hw_vio_1
win_start_1	[B] 0		Input	hw_vio_1
vio_addr_mode_value[2:0]	[H] 2		Output	hw_vio_1
vio_dbg_pi_f_inc	[B] 0		Output	hw_vio_1
vio_dbg_po_f_dec	[B] 0		Output	hw_vio_1
vio_dbg_po_f_stg23_sel	[B] 0		Output	hw_vio_1
vio_fixed_instr_value[2:0]	[H] 0		Output	hw_vio_1
po_win_right_ram_out[8:0]	[H] 000		Input	hw_vio_1
dbg_mem_pattern_init_done	[B] 0		Input	hw_vio_1
dbg_pi_counter_read_val[5:0]	[H] 24		Input	hw_vio_1
dbg_po_counter_read_val[8:0]	[H] 097		Input	hw_vio_1
dbg_tg_compare_error	[B] 0		Input	hw_vio_1
dbg_tg_wr_data_counts[47:0]	[H] 0000_0000_00		Input	hw_vio_1
dbg_win_chk[164:0]	[H] 00_0000_0000_		Input	hw_vio_1
vio_pause_traffic	[B] 0		Output	hw_vio_1
vio_sel_mux_rdd[3:0]	[H] 0		Output	hw_vio_1
vio_win_byte_select_dec	[B] 0		Output	hw_vio_1
vio_win_byte_select_inc	[B] 0		Output	hw_vio_1
win_sel_pi_pon	[B] 0		Output	hw_vio_1
win_start	[B] 0		Output	hw_vio_1
pi_win_left_ram_out[5:0]	[H] 00		Input	hw_vio_1
dbg_clear_error	[B] 0		Output	hw_vio_1
dbg_bit[8:0]	[H] 000		Output	hw_vio_1
dbg_dqs[4:0]	[H] 00		Output	hw_vio_1
vio_dbg_po_f_inc	[B] 0		Output	hw_vio_1
vio_fixed_bl_value[9:0]	[H] 000		Output	hw_vio_1
vio_bl_mode_value[1:0]	[H] 2		Output	hw_vio_1
vio_data_mask_gen	[B] 0		Output	hw_vio_1
vio_dbg_sel_po_incdec	[B] 0		Output	hw_vio_1
vio_instr_mode_value[3:0]	[H] 0		Output	hw_vio_1
vio_data_mode_value[3:0]	[H] 7		Output	hw_vio_1
vio_dbg_sel_pi_incdec	[B] 0		Output	hw_vio_1
vio_dbg_pi_f_dec	[B] 0		Output	hw_vio_1
pi_win_right_ram_out[5:0]	[H] 00		Input	hw_vio_1
po_win_left_ram_out[8:0]	[H] 000		Input	hw_vio_1
dbg_tg_rd_data_counts[47:0]	[H] 0000_0000_00		Input	hw_vio_1

Рис. 8.26. Сигналы VIO

На рис. 8.26 показаны сигналы, помеченные как **Output** (Выход) и **Input** (Вход). Направление относится к ядру VIO. Можно контролировать входные сигналы и вносить изменения в выходные сигналы, которые будут отражены в генераторе шаблонов и в программе проверки. На рисунке отмечен порядок внесения изменений в генератор шаблонов. Чтобы изменить конфигурацию, необходимо выполнить следующие действия.

1. Установить `vio_modify_enable` на 1.
2. Установить `vio_addr_mode_value` в 1 (фиксированный адрес, определяемый fixed address), 2 (адрес, определяемый **псевдослучайной двоичной последовательностью (Pseudo Random Binary Sequence, PRBS)** или последовательный адрес).

3. Установить `vio_bl_mode_value` в 1 (fixed bl) или 2 = `PRBS bl`.
4. Установить `vio_data_mode_value` в 1 (fixed bl), `2 = DGEN_ADDR, 3 = DGEN_HAMMER, 4 = DGEN_NEIGHBOR, 5 = DGEN_WALKING1,` и `6 = DGEN_WALKING0, DGEN_ PRBS.`

Итак, в данном примере было рассмотрено использование памяти DDR2, и в следующей главе она будет использоваться для контроллера дисплея. Вкратце разберем другие типы внешней памяти, которые используются в FPGA.

Другие типы внешней памяти

За прошедшие годы появилось множество типов памяти, которые стали использоваться в FPGA. Остановимся кратко на них, поскольку в будущем они могут понадобиться при создании собственных проектов.

Память SRAM с четырехкратной скоростью передачи данных (Quad Data Rate, QDR)

Память SRAM с четырехкратной скоростью передачи данных (QDR) широко используется в сетевых приложениях. Как и в памяти DDR, данные передаются по обоим фронтам тактового сигнала, что обеспечивает высокую производительность. В отличие от DDR, QDR имеет каналы чтения и записи, поэтому команды чтения и записи можно подавать одновременно. Кроме того, в отличие от DDR DRAM, это SRAM, поэтому циклы обновления отсутствуют, а задержка при чтении или записи может составлять всего 13 тактов при частоте 300 МГц.

Возможности QDR намного больше, чем у внутренней памяти FPGA, но намного меньше, чем у DDR. Она также относительно дорогая, и именно поэтому в основном используется в сетевых приложениях.

HyperRAM

HyperRAM – это тип самообновляющейся памяти DRAM, разработанный для приложений **Low Pin Count (LPC)**[1]. По производительности и размерам она похожа на память DDR (не 2+), что делает ее идеальной для некоторых приложений. Для HyperRAM существуют платы PMOD.

SPI RAM

Существует LPC RAM (ОЗУ), использующая **последовательный интерфейс SPI**[2]. Такие RAM имеют возможности, аналогичные DDR (но не 2+), и доволь-

---

[1] LPC (Low Pin Count, «малое количество контактов») – шина, разработанная компанией Intel и предназначенная для подключения устройств, не требующих большой пропускной способности каналов обмена с центральным вычислительным узлом. К таким устройствам относятся, например, загрузочное ПЗУ BIOS и контроллеры традиционных низкопроизводительных интерфейсов передачи данных (LPT и COM-порт, PS/2-порты для подключения мыши и клавиатуры) и т.д. Эта шина пришла на замену шине ISA, с которой совместима программно. – *Прим. ред.*

[2] SPI (Serial Peripheral Interface) – последовательный трехпроводной интерфейс передачи данных в масштабах платы или устройства. SPI широко используется для доступа к внешним устройствам – энергонезависимой памяти или различным датчикам. Например, интерфейсы флеш-карт некоторых распространенных форматов представляют собой разновидности SPI. – *Прим. ред.*

но хорошую производительность при использовании всего восьми выводов. С этими типами памяти также существуют платы PMOD.

Выводы

В этой главе была рассмотрена внешняя память, в частности DDR2, поскольку именно она присутствует на плате на Nexys A7; была рассмотрена генерация ядра контроллера памяти с помощью Xilinx MIG и было показано, как генерировать тестовый проект. Затем был запущен пример проекта на плате и, используя ILA, была показана его работа. Также вкратце были рассмотрены другие типы внешней памяти.

До сих пор все проекты были ограничены светодиодами и семисегментными индикаторами для вывода информации. В следующей главе будет использован контроллер DDR и создан контроллер VGA. Смахните пыль со своего ЭЛТ- или ЖК-дисплея с разъемом VGA, он понадобится для работы над отображением данных датчика температуры, аудиоданных и данных калькулятора на реальном дисплее.

Вопросы

1. Какие из следующих утверждений верны по отношению к внутренней и внешней памяти?

   a) Объем памяти DDR намного меньше, чем BRAM.
   b) При одинаковой ширине данных памяти DDR имеет гораздо более высокую производительность, чем BRAM.
   c) Задержка доступа к данным из BRAM и DDR одинакова.
   d) Всегда следует сначала использовать LUTRAM, прежде чем использовать любой другой тип памяти.

2. Для генерации памяти DDR2 для разрабатываемого проекта был использован:

   a) **Massive IP Goliath (MIG)**;
   b) **Minimally Informative Google (MIG)**;
   c) **Memory Interface Generator (MIG)**.

3. Можно использовать ILA для проверки данных в FPGA и VIO для чтения и записи данных.

   a) Верно.
   b) Неверно.

4. Какую память из MIG среди перечисленных ниже примеров FPGA Artix 7 может использовать?

   a) DDR2.
   b) DDR3.
   c) DDR4.
   d) LPDDR2.
   e) QDR.

5. Можно использовать HyperRAM, SPI RAM и SDRAM, если требуется разработать собственный контроллер памяти.

   a) Верно.
   b) Неверно.

Глава 9

Лучший способ отображения – VGA

До сих пор рассмотренные проекты были ограничены отображением информации с помощью светодиодов, одноцветных или RGB, а также семисегментного индикатора. Этого вполне достаточно чтобы выполнять операции и отображать ограниченную информацию, как было продемонстрировно на примере датчика температуры и калькулятора. Плата Nexys A7 содержит дополнительный выход, который дает практически неограниченные возможности отображения информации, – разъем **Video Graphics Array (VGA).** Разъем VGA на Nexys A7 может работать с разрешением до 1600x1200 с 4096 (2^{12}) цветами. Знание о том, как использовать внешнюю память, которая будет обеспечивать кадровый буфер, позволяет разблокировать возможность использовать VGA.

К концу этой главы будет создан метод отображения данных на мониторе с **электронно-лучевой трубкой (ЭЛТ)** или ЖК-мониторе через разъем VGA. В главе 10 «Свести все воедино» эта методика будет использована для модернизации некоторых проектов, чтобы использовать новый дисплей.

В этой главе будут рассмотрены следующие основные темы на примере проекта «**Основы работы с VGA**»:

- определение регистров;
- генерация синхросигналов для VGA;
- отображение текста.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/CH8.

Чтобы реализовать проект на плате, потребуется монитор с поддержкой VGA и кабель.

Проект 11. Основы работы с VGA

Самые первые профессиональные компьютерные дисплеи были простыми монохромными текстовыми дисплеями. Ранние персональные компьютеры, такие как Apple 2, могли отображать 280×192 пикселей с небольшим количеством цветов. Commodore 64 и IBM/PC могли отображать изображения с разрешением 320×200, опять же с ограниченной цветовой палитрой. Оригинальный IBM VGA был представлен в 1987 году, он позволял использовать более высокие разрешения и стандартизировал разъем, пока цифровые, такие как ЖК-дисплеи, не стали нормой.

Первое, на что нужно обратить внимание, это то, как отрисовывается изображение на экране. Независимо от того, используется ли ЭЛТ или современный ЖК-дисплей, синхронизация по-прежнему поддерживается для обеспечения обратной совместимости. Первоначально выход VGA был предназначен для управления электронной пушкой, чтобы зажечь люминофоры на ЭЛТ. Это означало, что синхронизация охватывала весь экран, плюс время на перемещение пушки с одной стороны экрана на другую или снизу вверх. На рис. 9.1 показаны различные параметры синхронизации и их связь с тем, что отображается на экране.

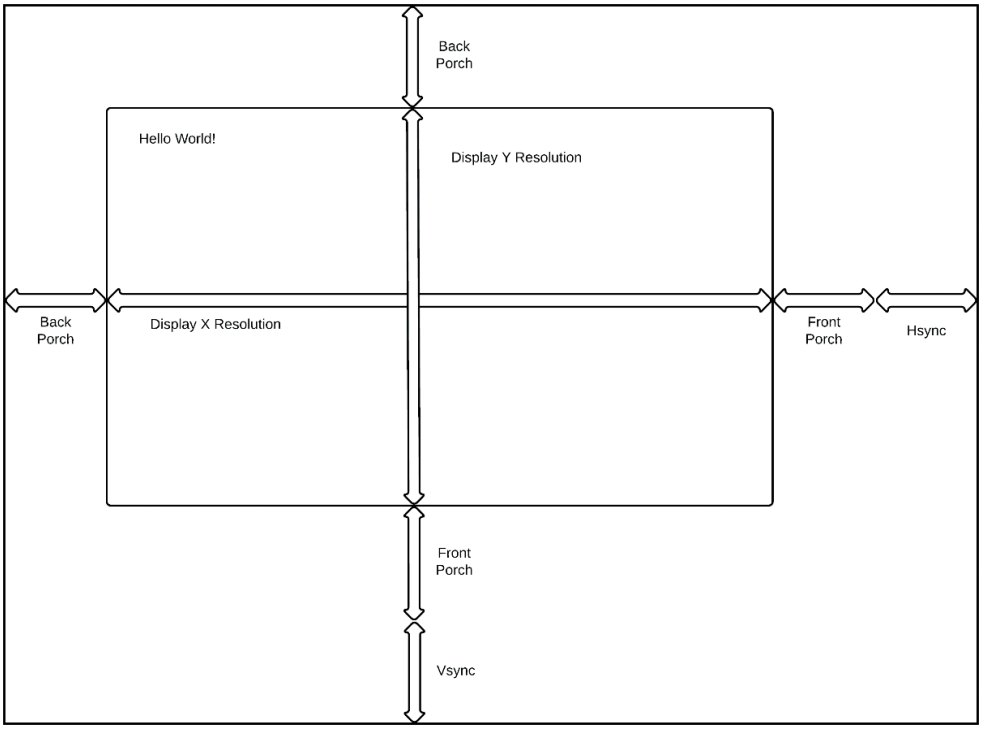


Рис. 9.1. Информация о синхронизации экрана VGA при отображении[1]

[1] Обозначения сигналов на рисунке (из стандартов аналогового телевидения): Front Porch – часть видеосигнала между концом сигнала активной строки и началом синхроимпульса (далее сокращенно *передний порог*); Back porch – часть видеосигнала между задним фронтом строчного синхроимпульса и задним фронтом импульса гашения (далее сокращенно *задний порог*). – *Прим. ред.*

Основные компоненты выходного сигнала, который требуется сгенерировать, следующие:

- **Hsync** – сигнал горизонтальной синхронизации;
- **Vsync** – сигнал вертикальной синхронизации;
- Red[3:0], Green[3:0], Blue[3:0] – значения цветов RGB. В реальном дисплее это 8 бит на цвет (24 **бита на пиксель, bits per pixel, bpp**). Но компания Digilent реализовала не настоящий ЦАП, а схему **цифроаналогового преобразования (ЦАП)** в виде массива резисторов, тем самым ограничив количество бит на пиксель по 4 на каждый цветовой канал RGB (обозначается как 4+4+4, или просто 444[1]), т. е. 12 bpp.

Необходимо, чтобы контроллер VGA был как можно более универсальным. Для этого интерфейс будет создан на основе регистров. Для этого нужно выяснить, какие значения понадобятся для заданного разрешения.

> **Важное замечание**
>
> Приведенный ниже список временных параметров является достаточно полным для дисплея с соотношением сторон 4:3 (старое соотношение сторон для ЭЛТ / телевизора). В зависимости от используемого дисплея, некоторые из них не будут работать. По умолчанию будет использоваться 640×480 @ 60 Гц, поскольку это базовый VGA-дисплей, и он должен поддерживаться любым оборудованием.

Рассмотрим список стандартов **Ассоциации стандартизации видеоэлектроники (Video Electronics Standards Association, VESA**), чтобы получить представление о том, что нужно отображать[2]:

---

[1] Смотрите «Важное замечание» от автора далее на стр. 239. – *Прим. ред.*

[2] Расшифровка сокращений в таблице (см. также вейвформу на рис. 9.2): FP – (Front Porch) и BP (Back Porch) – см. пояснение к рис. 9.1; tHS и tVS – частоты горизонтальной и вертикальной синхронизации; Pol (polarity) – полярность сигнала. Числа в третьей и четвертой колонках таблицы означают объем данных о точках на экране, который необходимо передавать в том или ином режиме. – *Прим. ред.*

Разрешение	Частота обновления	Всего по горизонтали	Всего по вертикали	Тактовая частота	FP	tHS	Pol	BP	FP	tVS	Pol	BP
640x480	60	800	525	25.18	16	96	-	48	10	2	-	33
640x480	72	832	520	31.5	16	40	-	128	9	3	-	28
640x480	75	840	500	31.5	16	64	-	120	1	3	-	16
640x480	85	832	509	36	56	56	-	80	1	3	-	25
800x600	60	1056	628	40	40	128	+	88	1	4	+	23
800x600	72	1040	666	50	56	120	+	64	37	6	+	23
800x600	75	1056	625	49.5	16	80	+	160	1	3	+	21
800x600	85	1048	631	56.25	32	64	+	152	1	3	+	27
1024x768	60	1344	806	65	24	136	-	160	3	6	-	29
1024x768	70	1328	806	75	24	136	-	144	3	6	-	29
1024x768	75	1312	800	78.75	16	96	+	176	1	3	+	28
1024x768	85	1376	808	94.5	48	96	+	208	1	3	+	36
1280x1024	60	1688	1066	108	48	112	+	248	1	3	+	38
1280x1024	75	1688	1066	135	16	144	+	248	1	3	+	38
1280x1024	85	1728	1072	157.5	64	160	+	224	1	3	+	44
1600x1200	60	2160	1250	162	64	192	+	304	1	3	+	46
1920x1200	60	2616	1242	195	96	200	+	400	3	3	+	36

В таблице выше приведены временные параметры для возможных режимов, которые будет поддерживать дисплей. Первое, что следует отметить, – тактовая частота, которая довольно сильно меняется от 25,18 до 195 МГц. Эта проблема решается путем реконфигурации тактовой частоты, которая доступна в мастере синхронизации. Также можно сделать регистры для хранения различных параметров, которые понадобятся, поэтому для регистров будет использоваться интерфейс AXI-Lite.

Теперь давайте возьмем соотвествующие числовые значения из таблицы выше и рассмотрим их применительно к диаграмме синхронизации, чтобы можно было отобразить сигналы, идущие на дисплей.

На рис. 9.2 показано, как работает синхронизация. Временная диаграмма состоит из двух частей. Первая часть – это время кадровой развертки, которое можно рассматривать как частоту сигнала синхронизации `Vsync`, умноженную на количество строк сканирования экрана (scanline). Каждая строка аналогичным образом состоит из `Hsync` и данных.

Для простоты можно считать, что данные хранятся в виде однобитных значений. Как правило, в режимах VGA и VESA используются 8-, 16- или 32-битные цвета. 8-битные значения используются как индексы в палитре 256 цветов из 16 млн цветов. 16-битные цвета обычно представляли собой 565 или 555 бит на каждый цветовой канал RGB, а 32-битный цвет – это 888 бит на канал, способных отображать 16 млн цветов. Для целей проекта будем придерживаться хранения однобитного цвета. Пиксель может быть включен или выключен.

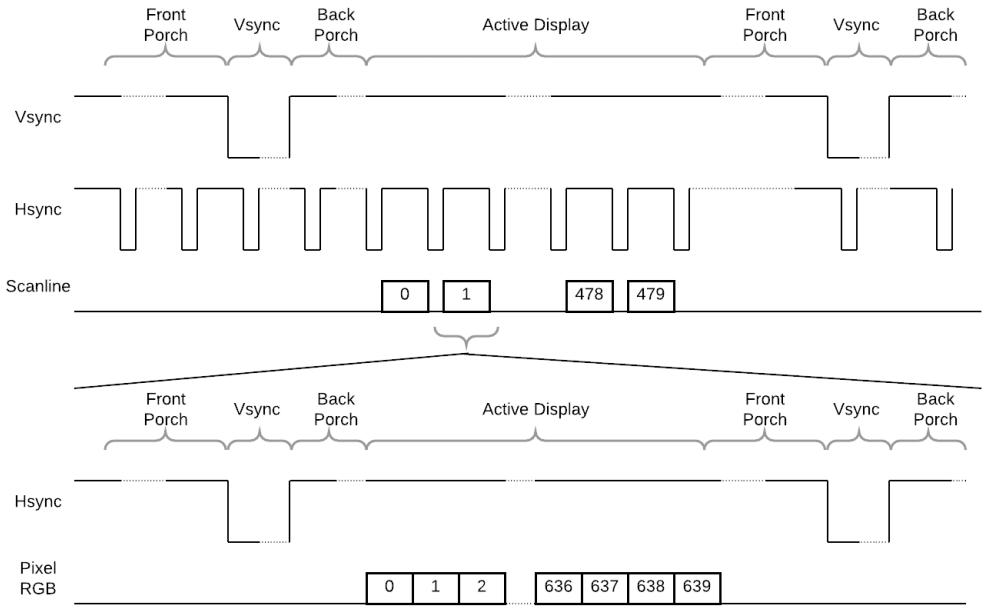


Рис. 9.2. Диаграмма синхронизации видеосигнала

> **Важное замечание**
>
> При работе с цветами они будут обозначаться по количеству битов, используемых для их представления (8, 16, 24 или 32), и по количеству битов на цветовой канал (565, 555 или 888), где каждая цифра представляет собой количество битов, используемых для представления красного, зеленого и синего цветов (RGB).

Определение регистров

Первый шаг, необходимый для разработки VGA-контроллера, – это определение набора регистров, которые можно использовать для решения поставленной задачи. Временны́е параметры известны из рис. 9.1 и соответствующей таблицы. Из этого можно вывести некоторые параметры. Для VGA можно использовать следующий набор:

- **Horizontal display start** (начало горизонтального отображения) – количество горизонтальных пикселей до начала отображения, эквивалентное горизонтальному заднему порогу минус один;
- **Horizontal display width** (ширина горизонтального отображения) – ширина дисплея (например, 640 пикселей);
- **Horizontal sync width** (ширина горизонтальной синхронизации) – ширина импульса `Hsync` (в тактах);
- **Horizontal display total width** (общая ширина горизонтального отображения) – общая ширина отображаемой и неотображаемой частей (колонка 3 в таблице на стр. 242);

- **Vertical display start** (начало вертикального отображения) – количество строк дисплея до начала отображения, равное вертикальному заднему порогу минус один;
- **Vertical display height** (высота вертикального отображения) – высота видимой части дисплея (например, 480 пикселей);
- **Vertical sync width** (ширина вертикальной синхронизации) – ширина импульса `Vsync` в строках сканирования (в тактах);
- **Vertical display total height** (общая высота вертикального отображения) – общая высота отображаемой и неотображаемой частей (колонка 4 в таблице на стр. 242);
- **VGA format** – цветовая глубина пикселя для данного экрана (количество битов на пиксель);
- **Display address** – адрес буфера дисплея, с которого будет производиться чтение. Может быть установлен в любое время, но вступает в силу только в начале следующего кадра для предотвращения разрыва изображения;
- **Horizontal and vertical polarity selections** (выбор горизонтальной и вертикальной полярности) – поскольку различные режимы имеют активную высокую или низкую полярность синхроимпульсов, необходимо обеспечить возможность выбора между ними;
- **Display pitch** (шаг отображения) – нужно знать, сколько страниц отображения нужно считать для данной строки сканирования, а также сколько считать для каждой последующей строки сканирования;
- **Load mode** (режим загрузки) – как правило, в сложных проектах может быть несколько регистров, которые составляют полный набор значений, необходимых для данной функции. Необходимо обеспечить способ одновременного обновления всех этих регистров после завершения обновления.

Эти регистры будут доступны через интерфейс AXI-Lite.

Разработка простого интерфейса AXI-Lite

Со стороны записи интерфейса AXI включает в себя три компонента: шину адреса, шину данных и шину ответа. Адресный интерфейс можно увидеть в определении интерфейса ядра:

```
input wire           reg_awvalid,
output logic         reg_awready,
input wire [11:0]    reg_awaddr,

input wire           reg_wvalid,
output logic         reg_wready,
input wire [31:0]    reg_wdata,
input wire [3:0]     reg_wstrb,

input wire           reg_bready,
output logic         reg_bvalid,
output logic [1:0]   reg_bresp,
```

Ведомое устройство должно уметь работать с шинами адреса и данных независимо друг от друга. В разрабатываемом проекте обе шины будут ра-

ботать одновременно, но возможна ситуация, когда ведущее устройство предоставит адрес или данные раньше. Представим это в виде конечного автомата регистра.

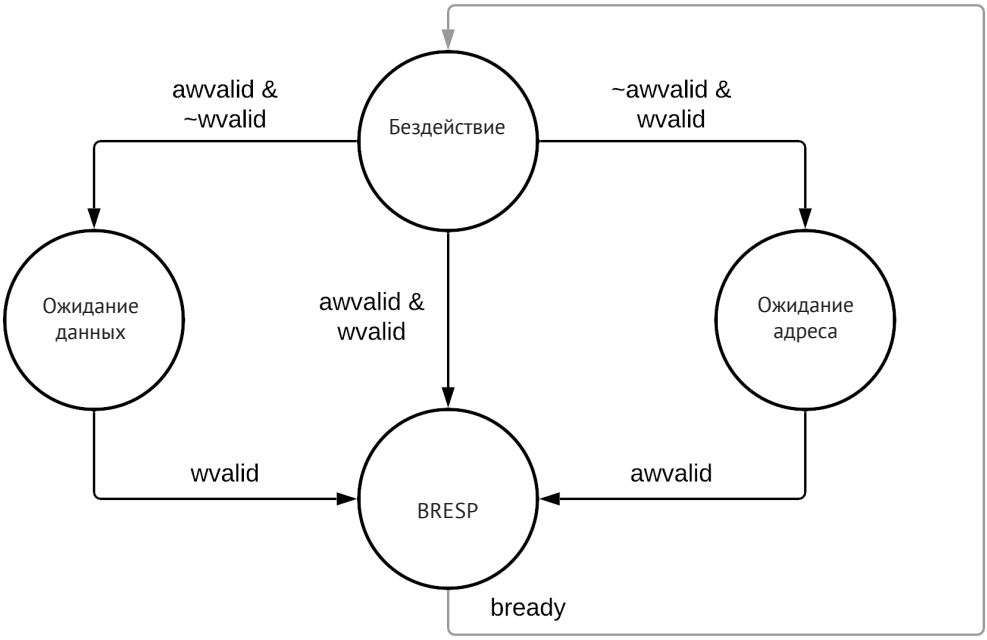


Рис. 9.3. Конечный автомат AXI-Lite

На рис. 9.3 показан базовый конечный автомат. Если `awvalid` и `wvalid` оба установлены в состояние логической единицы, то можно генерировать ответ. Если не хватает одного из сигналов для полной передачи, `awvalid` либо `wvalid`, устройство переходит в состояние ожидания, а затем в состояние `BRESP`.

Наконец, в состоянии `BRESP`, как только поступает сигнал `bready`, генерируется ответ об успехе и происходит переход обратно в состояние бездействия.

Теперь давайте рассмотрим, как происходит генерация сигналов синхронизации.

Генерация сигналов синхронизации для VGA

Для проекта понадобятся два компонента фазовой автоподстройки частоты **ФАПЧ (Phase Locked Loop, PLL)** или компонента управления тактовой частотой (**Mixed Mode Clock Manager, MMCM**). Первый PLL будет дубликатом того, который был создан в главе 8 «Много данных? MIG и DDR2» для генерации тактовых импульсов контроллера памяти DDR2, а также внутренних тактовых импульсов. Будет также создан второй PLL, чтобы иметь возможность изменять параметры сигнала синхронизации. По умолчанию при включении питания проект будет отображать разрешение VGA 640×480 @ 60 Гц. Основным отличием требуемой конфигурации является выбор **Dynamic Reconfig**:

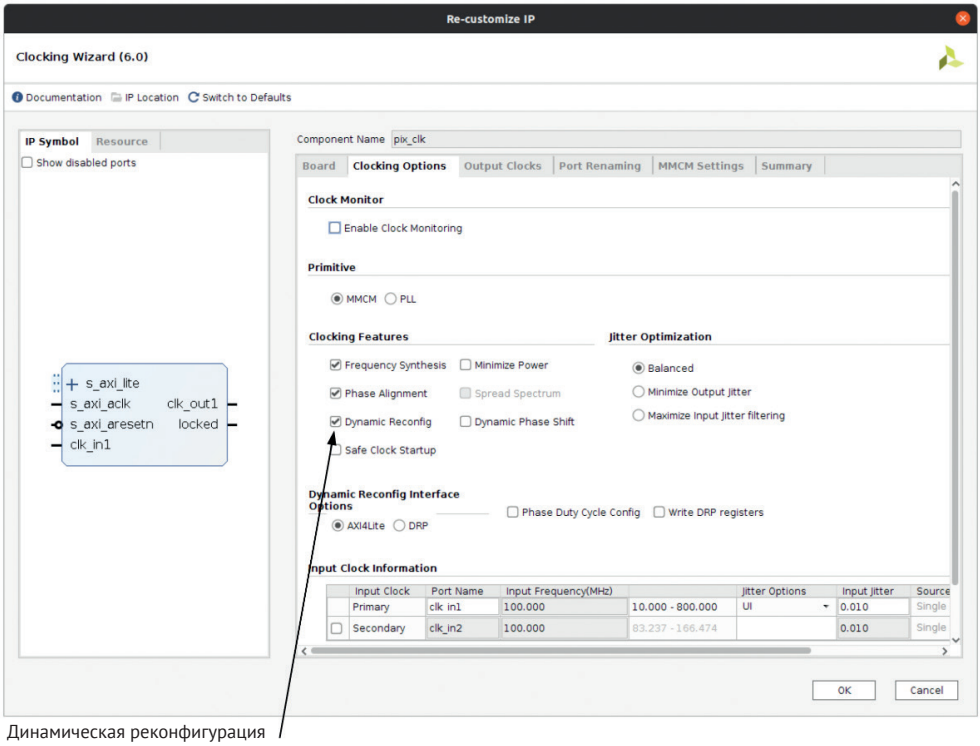


Рис. 9.4. Динамическая реконфигурация

Добавление динамической реконфигурации открывает интерфейс AXI-Lite в мастере синхронизации, который можно использовать для реконфигурации PLL. Регистры, на которых нужно сосредоточиться, можно найти в мастере синхронизации drive 6.0 по ссылке https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf.

Нужно перенастроить только `clk0`. На следующем рисунке показано, как извлечь необходимую информацию из мастера настройки частоты тактовых сигналов. На рис. 9.5 показано, как извлечь эти значения самостоятельно.

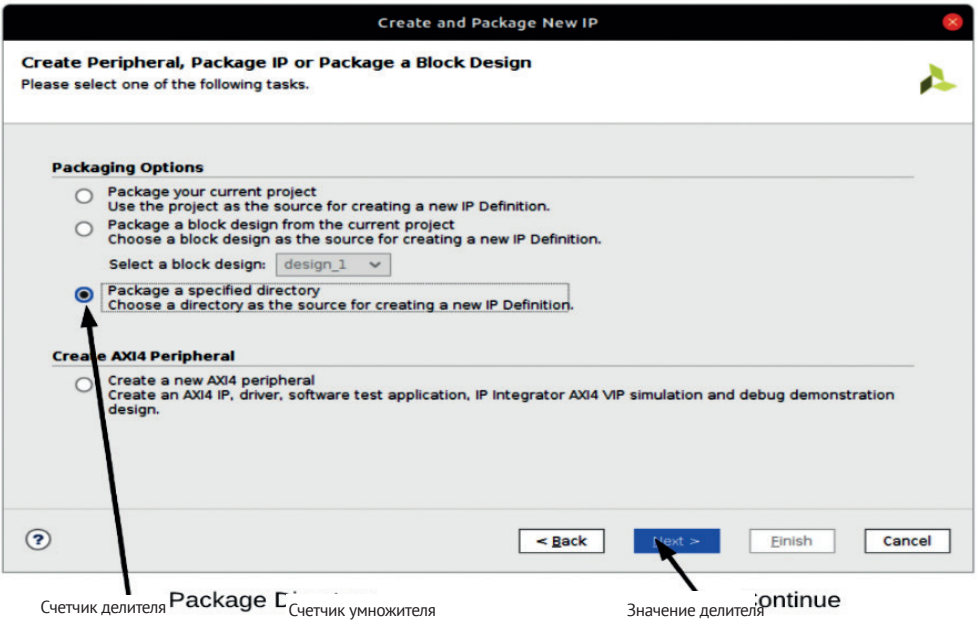


Рис. 9.5. Извлечение параметров реконфигурации

Нужные параметры (основанные на входной тактовой частоте 200 МГц) следующие:

- счетчик делителя (**Divide Counter**): на рис. 9.5 для 25,18 МГц равен 9;
- целая часть множителя обратной связи по тактовой частоте (**Mult Counter**): для 25,18 МГц равна 50;
- дробная часть множителя обратной связи по тактовой частоте (**Mult Counter**): для 25,18 МГц равна 000;
- целая часть значения делителя обратной связи по тактовой частоте: для 25,18 МГц равна 44;
- дробная часть значения делителя обратной связи по тактовой частоте: для 25,18 МГц равна 125.

Значения можно рассчитать, но нужно быть внимательным, чтобы не превысить максимальную частоту PLL. В следующей таблице приведены значения, необходимые для всех частот работы VGA:

Разрешение	Частота VGA	0x200			0x208	
		[7:0]	[15:8]	[25:16]	[7:0]	[17:8]
640x480 60 Hz	25.18	3	21	625	28	625
640x480 72/75 Hz	31.5	4	39	375	31	250
640x480 85 Hz	36	5	49	500	27	500
800x600 60 Hz	40	1	10	000	25	000
800x600 75 Hz	49.5	5	49	500	20	000
800x600 72 Hz	50	1	10	000	20	000
800x600 85 Hz	56.25	1	10	125	18	000
1024x768 60 Hz	65	5	50	375	15	500
1024x768 70 Hz	75	4	40	125	13	375
1024x768 75 Hz	78.75	4	39	375	12	500
1024x768 85 Hz	94.5	5	47	500	10	000
1280x1024 60 Hz	108	1	10	125	9	375
1280x1024 75 Hz	135	1	10	125	7	500
1280x1024 85 Hz	157.5	4	39	375	6	250
1600x1200 60 Hz	162	1	10	125	6	250
1920x1200 60 Hz	195	1	9	750	5	000

Эту таблицу можно использовать для создания кода для загрузки пиксельного PLL. Также необходимо загрузить значения регистров для нужного разрешения. Сначала создадим эту таблицу.

Сперва создадим структуру для хранения таблицы, которая будет использоваться для настройки PLL и контроллера VGA для каждого из 17 поддерживаемых разрешений. Создадим простой конечный автомат, поддерживающий протокол AXI-Lite, который может конфигурировать нужное разрешение, но в будущем вместо него в системе можно будет использовать микроконтроллер.

```
typedef struct packed {
  logic [7:0 ]  divide_count;
  logic [15:8]  mult_integer;
  logic [25:16] mult_fraction;
  logic [7:0]   divide_integer;
  logic [17:0]  divide_fraction;
 logic [11:0]  horiz_display_start;
  logic [11:0]  horiz_display_width;
  logic [11:0]  horiz_sync_width;
  logic [11:0]  horiz_total_width;
  logic [11:0]  vert_display_start;
  logic [11:0]  vert_display_width;
  logic [11:0]  vert_sync_width;
  logic [11:0]  vert_total_width;
  logic hpol;
  logic vpol;
  logic [12:0]  pitch;
 } resolution_t;
```

Структура rezolution инкапсулирует в себе все необходимые параметры. Можно определить переменную и инициализировать ее в начальном блоке, чтобы использовать ее поля в качестве констант.

```
resolution_t resolution[17];
initial begin
  // 640x480 @ 60Hz
  resolution[0].divide_count         = 8'd3;
  resolution[0].mult_integer         = 8'd21;
  resolution[0].mult_fraction        = 10'd625;
  resolution[0].divide_integer       = 8'd28;
  resolution[0].divide_fraction      = 10'd625;
  resolution[0].horiz_display_start  = 12'd15;
  resolution[0].horiz_display_width  = 12'd640;
  resolution[0].horiz_sync_width     = 12'd96;
  resolution[0].horiz_total_width    = 12'd799;
  resolution[0].vert_display_start   = 12'd9;
  resolution[0].vert_display_width   = 12'd480;
  resolution[0].vert_sync_width      = 12'd2;
  resolution[0].vert_total_width     = 12'd524;
  resolution[0].hpol                 = '0;
  resolution[0].vpol                 = '0;
  resolution[0].pitch                = 13'd5;
```

Все 17 режимов определены в коде программы. Здесь показан только первый режим по умолчанию. Теперь можно создать конечный автомат для загрузки VGA и PLL.

Конечный автомат разделен на две секции: `CFG_WR0-2` загружает **MMCM** с настройками конфигурации тактового сигнала, а `CFG_WR3-5` загружает разрешение для контроллера VGA. Конечный автомат работает следующим образом.

1. Он обнаруживает нажатие кнопки и начинает загрузку параметров MMCM.
2. `CFG_WR0` проверяет, какой из сигналов `valid` активен. В случае, если активен только `wvalid` или `awvalid`, то есть два состояния, `CFG_WR1` и `CFG_WR2`, для ожидания недостающего valid.
3. Как только оба `valid` будут активны, происходит переход к состоянию записи регистров. Должны быть записаны все 24 регистра в MMCM, прежде чем начнется работа с VGA.
4. Часть конечного автомата VGA работает аналогично части MMCM, только в ней записываются параметры VGA. После завершения записи происходит переход в состояние ожидания.

Ядро VGA управляет синхронизацией монитора и выводом изображения.

> **Важное замечание**
>
> В зависимости от типа монитора, возможно, будет нельзя отобразить все разрешения. Некоторые мониторы могут поддерживать больше режимов синхронизации, чем другие. Монитор, на котором происходила апробация данного проекта, мог работать с разрешением 1280×1024 @ 85 Гц, но не выше. Из-за временны́х ограничений не рекомендуется подниматься выше 1280×1024 @ 75 Гц.

Рассмотрим более подробно генератор сигналов синхронизации.

Генератор сигналов синхронизации монитора

Для работы с генерацией сигналов синхронизации понадобятся два счетчика. Первый счетчик, `horiz_count`, будет генерировать сигнал синхронизации и вывод пикселей для каждой строки сканирования. Второй, `vert_ count`, подсчитывает количество строк сканирования, чтобы определить, когда начинать вывод пикселей и генерировать `Vsync`.

```
if (horiz_count >= horiz_total_width) begin
  horiz_count <= '0;
  if (vert_count >= vert_total_width) vert_count <= '0;
  else vert_count <= vert_count + 1'b1;
  scanline <= vert_count - vert_display_start + 2;
  mc_addr  <= scanline * pitch;
  mc_words <= pitch;
end else
  horiz_count <= horiz_count + 1'b1;
```

Представленный код обнуляет сигнал `horiz_count`, когда достигается конец строки сканирования. Это происходит благодаря сравнению больше или равно сигналу `horiz_total_width.` Обновление счетчиков не останавливает и не перезапускает генерацию сигналов синхронизации. Это гарантирует, что счетчики восстановятся, если произойдет выход за пределы диапазона. Аналогичным образом реализован вертикальный счетчик.

Этот блок также генерирует несколько других параметров, необходимых для отображения пикселей. Вычисляется сканируемая строка, с которой в данный момент ведется работа. Нулевая сканируемая строка будет первой отображаемой строкой.

Также генерируется адрес для текущей строки сканирования и шаг чтения, который является количеством 16-байтных слов, считываемых для каждой строки сканирования. Это количество может быть больше или равно необходимому количеству байт.

Когда используются медленные компоненты или необходимо достичь более высокой тактовой частоты, полезно искать возможности для предварительного расчета математических операций, когда это возможно. В коде выше вычисляется адрес, который будет нужен позже.

```
  mc_addr  <= scanline * pitch;
```

Это связано с тем, что в коде, где происходит чтение из памяти, нужно убедиться, что не нарушены правила AXI.

```
vga_hblank  <= ~((horiz_count > horiz_display_start) &
                 (horiz_count <=
           (horiz_display_start + horiz_display_width)));
vga_hsync     <= polarity[1] ^
    ~(horiz_count > (horiz_total_width - horiz_sync_width));
vga_vblank  <= ~((vert_count > vert_display_start) &
                 (vert_count <=
                 (vert_display_start + vert_display_width)));
vga_vsync     <= polarity[0] ^
    ~(vert_count > (vert_total_width - vert_sync_width));
```

Сигналы Hsync и Vsync генерируются, как показано на рис. 9.1, в конце строки сканирования и окна дисплея. Расчет времени генерации происходит путем создания синхросигнала из общей суммы горизонтального или вертикального значений, указаных в регистрах настройки контроллера, из которой вычитается ширина синхросигнала. Также требуется использовать регистры полярности для создания правильной полярности синхронизации. В качестве программируемых инверторов можно использовать логические элементы Исключающее ИЛИ.

Также происходит генерация пустых сигналов. Технически они не нужны, если только не использовать настоящий ЦАП, когда эти сигналы нужны для обнуления пиксельного вывода. В текущем проекте можно использовать их аналогичным образом. Они включены, поскольку при симуляции это может помочь определить, когда ожидается вывод данных.

В этом фрагменте кода также генерируется сигнал переключения mc_req для запроса вывода данных на экран:

```
if (vga_hblank && ~last_hblank && ~vga_vblank)
  mc_req <= ~mc_req;
last_hblank   <= vga_hblank;
```

Мертвое время (dead time) дисплея используется для предварительной выборки следующей строки сканирования. Запрос генерируется по переднему фронту hblank, пока не начался период вертикального гашения.

Теперь, когда дисплей сконфигурирован, нужно отобразить на нем что-нибудь интересное.

Отображение текста

Текстовый символ в его самой старой и простой форме – это растровое изображение. Современные операционные системы могут использовать такие вещи, как TrueType, которые могут масштабироваться при различных разрешениях, сохраняя при этом гладкую форму символа. Но самой старой формой отображения текста было хранение рисунка в памяти и последующее копирование его на экран.

Включим файл под названием text_rom.sv. По сути, это таблица поиска.

```
module text_rom
  (input              clock, // Тактовая частота
   input [7:0]        index, // Индекс символа
   input [2:0]        sub_index, // Y позиция в символе
   output logic [7:0] bitmap_out);
```

Функционально можно представить ПЗУ для хранения текста (text ROM) в виде следующей схемы:

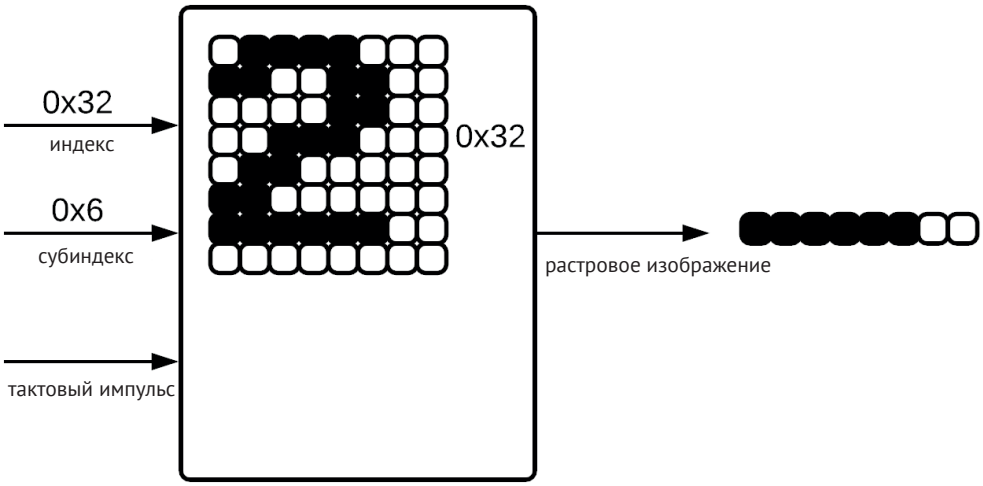


Рис. 9.6. text_rom

Каждый тактовый цикл символ ищется с помощью индекса, а субиндекс ссылается на строку сканирования символа[1]. На рис. 9.6 показан пример, где запрашивается символ 0x32, что соответствует **американскому стандартному коду для обмена информацией (American Standard Code for Information Interchange, ASCII)** для числа 2. Запрашивается шестой символ сканирования символа. В следующем цикле возвращается значение 0xFC, которое представляет собой пиксели шестой строки сканирования числа 2.

> **Важное замечание**
>
> Код ASCII является одним из основных стандартов для кодирования текста. Одна из особенностей ASCII заключается в том, что цифры 0–9 кодируются значениями в диапазоне 0x30–0x39.

В файле `text_rom.sv` содержатся все прописные и строчные символы и цифры ASCII, а также несколько символов заполнения. ASCII обычно представляется 8-битным значением, поэтому существует много места для добавления новых символов[2].

---

[1] Для полноты картины отметим, что автор показал на рисунке символ формата 8×8 пикселей, занимающий 8 строк по высоте; но для VGA такой формат не основной – символы получаются слишком мелкими и сжатыми по высоте (рис. 9.8). Обычный текстовый режим VGA (640×480) предуматривает символы формата 8×16 (на экране помещается 30 строк по 80 символов в строке), но есть и другие форматы – 8×14 и даже 9×8. – *Прим. ред.*

[2] Имеется в виду, что стандарт ASCII описывает только 128 символов (7-битное число). Числом байтового размера (8 разрядов) можно описать вдвое больше символов, отсюда и простор для конструирования новых. В различных устройствах нередко пользуются также тем фактом, что первые 32 символа (с индексами 0x00-0x1F) изначально имели служебное назначение, а в настоящее время не используются или используются частично. Поэтому часто именно на первые 8, 16 или 32 места в ASCII-таблице пользователю предлагается подставлять собственные символы. – *Прим. ред.*

```
always @(posedge clock)
  case ({index, sub_index})
  ...
    // 2
    {8'h32, 3'h0}: bitmap <= 8'h78;
    {8'h32, 3'h1}: bitmap <= 8'hCC;
    {8'h32, 3'h2}: bitmap <= 8'h0C;
    {8'h32, 3'h3}: bitmap <= 8'h38;
    {8'h32, 3'h4}: bitmap <= 8'h60;
    {8'h32, 3'h5}: bitmap <= 8'hC0;
    {8'h32, 3'h6}: bitmap <= 8'hFC;
    {8'h32, 3'h7}: bitmap <= 8'h00;
```

Здесь показано, как выглядит поиск числа 2, или 0x32. Это то же самое, что представлено на рис. 9.6.

Одна особенность способа хранения данных заключается в том, что нужно будет перевернуть выводимые данные. Добавим следующий код:

```
always @* begin
  for (int i = 0; i < 8; i++) begin
    bitmap_out[i] = bitmap[7-i];
  end
end
```

Этот код переворачивает биты. Без него текст будет выглядеть перевернутым. В каких-то случаях это может понадобиться или не понадобиться, поэтому полезно знать, что такой код существует, на случай, если он понадобится или его нужно будет удалить.

Вернемся к VGA верхнего уровня. Добавим строку текста к каждой настройке разрешения, чтобы при установке дисплея можно было напечатать то, какие настройки выбраны.

```
res_text[0] = "zH06 @ 084x046";
```

Обратите внимание, что текст в виде строки записан в обратном порядке. Это потому, что нужно начинать с бита 0 символа 0 и наращивать его до символа 15, бита 7[1].

Запрос памяти

Для дисплея нужно взять сигнал запроса памяти и синхронизировать его с тактовым генератором контроллера памяти. Это также используется для сброса FIFO-буфера пикселей. Обратите внимание, что переключение сигнала запроса происходит в конце строки, что обеспечивает несколько ключевых функций для проекта.

- Поскольку ничего не будет отображаться, можно просто сбросить FIFO.
- Должно быть достаточно времени, чтобы сбросить FIFO и начать получать данные для отображения.

Можно построить конечный автомат для обработки обращений к памяти, как показано в следующем блоке кода:

```
case (mem_cs)
```

[1] Иными словами, первый символ в строке у автора – крайний справа. – *Прим. ред.*

```
MEM_IDLE: begin
  mem_arvalid <= '0;
  if (^mc_req_sync[2:1]) begin
    fifo_rst <= '1;
    mem_cs   <= MEM_W4RSTH;
  end
end
MEM_W4RSTH: begin
  next_addr <= mc_addr + mc_words;
  len_diff  <= 2047 - mc_addr[10:0];
  if (wr_rst_busy) begin
    fifo_rst <= '0;
    mem_cs   <= MEM_W4RSTL;
  end
end
```

Когда происходит синхронизация и обнаруживается передний фронт сигнала запроса, происходит сброс FIFO. FIFO формирует сигнал, информирующий о том, что он занят во время сброса, поэтому во втором состоянии автомат ждет, пока сигнал сброса не установится в высокий уровень, затем происходит сброс и схема переходит в состояние ожидания, до тех пор, пока сигнал сброса снова не установится в логический 0. Во время ожидания вычисляется следующий адрес и количество строк сканирования, которые остались до достижения границы 2K (2048 байт).

> **Важное замечание**
>
> При выполнении пакетной (burst) передачи через AXI нельзя пересекать границу в 2048 байт. Это следует учитывать и прервать передачу при достижении границы по количеству байт.

В следующем блоке кода реализована проверка достижения границы пакетной передачи.

```
MEM_W4RSTL: begin
  if (~wr_rst_busy) begin
    // Сделать запрос с текущего адреса
    mem_araddr  <= mc_addr;
    if (next_addr[31:11] != mc_addr[31:11]) begin
   // проверить, не пересечена ли граница 2K
    mem_arlen <= len_diff;
    if (mem_arready) mem_cs <= MEM_REQ;
    else             mem_cs <= MEM_W4RDY1;
  end else begin
      // Сделать один запрос
      mem_arlen   <= mc_words - 1;
      if (mem_arready) mem_cs <= MEM_IDLE;
      else             mem_cs <= MEM_W4RDY0;
    end // else: !if(next_addr[12])
    // Вычислить параметры для второго запроса
    next_addr   <= mc_addr  + len_diff + 1'b1;
    len_diff    <= mc_words - len_diff;
  end
end // case: MEM_W4RSTH
```

Для обработки возвращаемых данных используется асинхронный модуль Xilinx, `xpm_fifo`, как показано в следующем блоке кода:

```
// Пиксели FIFO
// достаточно большие для одной строки сканирования при 1920x32bpp (480 байт)
xpm_fifo_async
  #(.FIFO_WRITE_DEPTH       (512),
    .WRITE_DATA_WIDTH       (128),
    .READ_MODE              ("fwft"))
u_xpm_fifo_async
  (.rst                    (fifo_rst),
   .wr_clk                 (mem_clk),
   .wr_en                  (mem_rvalid),
   .din                    (mem_rdata),
   .wr_rst_busy            (wr_rst_busy),
   .rd_clk                 (vga_clk),
   .rd_en                  (vga_pop),
   .dout                   (vga_data),
   .empty                  (vga_empty),
   .rd_rst_busy            (rd_rst_busy));
```

Необходимо отметить следующую особенность работы с FIFO – запись происходит на частоте тактового сигнала памяти, а чтение – на частоте тактового сигнала вывода пикселей VGA. В этом проекте не предпринято никаких мер предосторожности, чтобы убедиться, что данные загружены для строки сканирования или для обработки исключений. Это приводит к тому, что чтение из памяти происходит по принципу *«выстрелил и забыл» (fire and forget)*. Имеется конечный автомат, который делает запрос, и данные помещаются обратно в FIFO для считывания.

FIFO сконфигурирована как **first-word fall-through (FWFT)**, что означает, что данные готовы на выходе для немедленного использования[1].

Фрагмент кода, где происходит чтение данных из FIFO и вывод их на экран, следующий:

```
initial scan_cs = SCAN_IDLE;
always @(posedge vga_clk) begin
  vga_pop <= '0;
  case (scan_cs)
    SCAN_IDLE: begin
      if (horiz_count == horiz_display_start) begin
        if (vga_data[0]) vga_rgb <= ~vga_empty;
       else vga_rgb <= '0;
        scan_cs   <= SCAN_OUT;
        pix_count <= '0;
      end
    end
    SCAN_OUT: begin
      pix_count <= pix_count + 1'b1;
      // Прямо сейчас просто сделайте один бит на пиксель
      if (pix_count == 126) begin
        vga_pop <= ~vga_empty;
      end
```

[1] Установка **First-word Fall-Through** (**FWFT**) означает режим чтения с предварительным просмотром данных; позволяет осуществлять предварительный просмотр следующего слова данных, хранящегося в данном элементе памяти, без выполнения операции чтения. – *Прим. ред.*

```
        if (vga_data[pix_count]) vga_rgb <= '1;
        else vga_rgb <= '0;
        if (rd_rst_busy) scan_cs <= SCAN_IDLE;
      end
    endcase // case (scan_cs)
  end
```

Конечный автомат отображения может быть реализован довольно просто. Автомат ждет, пока не будет достигнута первая строка сканирования, после чего, в зависимости от формата пикселей, происходит отображение их на экране. Эта версия кода поддерживает только 1 бит на пиксель.

На этом этапе можно запустить проект на плате и увидеть, что будет сформировано на выходе VGA. Проект инициализирован для работы с разрешением 640×480 при 60 Гц.

Рис. 9.7. Первая попытка вывода на VGA экран

На экране наблюдается не то, что ожидалось. Без начальной очистки памяти на экране отображаются старые данные, полученные при инициализации контроллера памяти.

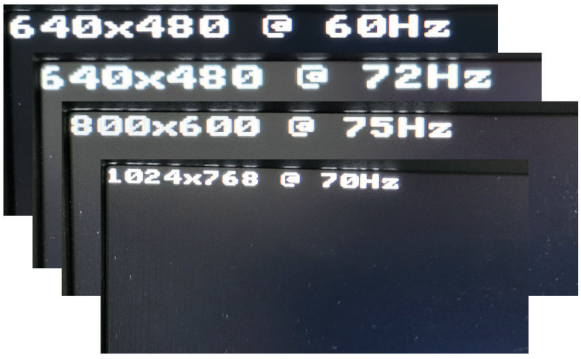


Рис. 9.8. Образцы разрешений

Чтобы исправить эту проблему, необходимо при первом запуске очистить экран и отобразить настройки экрана (в данном случае: 640×480 @ 60 Гц). Это достигается одним состоянием, которое выполняется только при включении питания:

```
CFG_IDLE0: begin
  update_text   <= ~update_text;
  cfg_state     <= CFG_IDLE1;
end
```

Работа над созданием простого, но полезного VGA-контроллера завершена. Этот пример показывает, что создание полезного оборудования не является чем-то недосягаемым. Но, конечно, требуется много работы, чтобы убедиться, что оборудование работает так, как нужно.

Тестирование контроллера VGA

Большая часть тестирования проекта была проведена на плате. Компиляция проекта занимает незначительное время, но время моделирования полного кадра может быть очень велико при использовании симулятора Vivado из-за использования PLL и контроллера памяти. Это хороший способ проверить первые несколько строк дисплея, чтобы убедиться, что они выглядят нормально и что синхронизация работает правильно. Но остальную отладку проекта лучше производить с использованием платы.

Две основные части, которые нужны для моделирования в программном симуляторе, – это генератор тактовых импульсов и загрузка регистров:

```
initial clk = '0;
always begin
 clk = #5 ~clk;
end
...
initial begin
  SW <= 8;
  button_c <= '1;
  repeat (1000) @(posedge clk);
  while (~u_vga.init_calib_complete) @(posedge clk);
  $display("DDR calibration complete");
  while (~u_vga.locked) @(posedge clk);
  button_c <= '1;
  repeat (100) @(posedge clk);
  button_c <= '0;
  repeat (10000) @(posedge clk);
end
```

Более полный `testbench` мог бы содержать задачи сохранения видеокадров, но, учитывая скорость моделирования, лучше работать на плате. Для запуска на плате необходимо учитывать необходимые ограничения.

Проверка ограничений

В VGA нужно обрабатывать довольно много пересечений тактовых доменов. FIFO обрабатывает данные, но есть данные, идущие от интерфейса AXI к тактовому генератору контроллера памяти, а затем к тактовому генератору дис-

плея VGA. Кроме того, в проекте есть тактовый генератор VGA с переменной частотой от программируемого MMCM.

Когда разрабатывается MMCM или PLL, Vivado автоматически создает генерируемые тактовые сигналы на выходе. Поскольку PLL во время работы будет перепрограммироваться, нужно будет переопределить этот параметр для максимальной частоты, которая нужна будет во время работы.

```
create_clock -period 7.41 -name vga_clk -add [get_pins u_clk/
clk_out1]
```

Также надо знать периоды тактовых импульсов для настройки следующих ограничений. Параметр периода (period) можно получить из тактового генератора с помощью команды `get_property.get_clocks.`

```
set vga_clk_period [get_property PERIOD [get_clocks vga_clk]]
set clk200_period  [get_property PERIOD [get_clocks clk_out1_
sys_clk]]
set clkui_period  [get_property PERIOD [get_clocks clk_pll_i]]
```

Немного поэкспериментировав, можно обнаружить, что проект может надежно работать на частоте около 135 МГц, поэтому следует установить эту частоту в качестве частоты тактового генератора на выходе PLL.

Теперь добавим ограничения для входов синхронизаторов. Поскольку это синхронизаторы с переключением одного сигнала, необходимо подать false-path на вход первой стадии триггера синхронизатора.

```
set_false_path -from u_vga_core/load_mode_reg*/C -to */load_
mode_sync_reg[0]/D
set_false_path -from u_vga_core/mc_req_reg*/C -to */mc_req_
sync_reg[0]/D
set_false_path -from update_text_reg/C -to update_text_sync_
reg[0]/D
```

Также необходимо добавить ограничения `max_delay`, чтобы быть уверенными в правильном ограничении регистров между тактовыми доменами и не вынуждать устройство выполнять необоснованные временные требования. Это можно сделать следующим образом:

```
set_max_delay -datapath_only -from */horiz_display_start_reg*
[expr 1.5 * $vga_clk_period]
set_max_delay -datapath_only -from */horiz_display_width_reg*
[expr 1.5 * $vga_clk_period]
...
set_max_delay -datapath_only -from *sw_capt_reg*/C [expr 1.5 *
$clkui_period]
```

Переменная `set_max_delay` позволяет установить промежуток времени от любой точки до любой другой точки. Параметр `-datapath_only` указывает механизму синхронизации не учитывать задержки тактовых сигналов при вычислении задержек.

Таким образом, проект реализован на плате и выполнены требования синхронизации. В следующей главе к проекту будет добавлена клавиатура, а проект VGA будет использоваться в качестве завершающего проекта для отображения данных из предыдущих проектов.

Выводы

В этой главе реализован более совершенный способ отображения данных. Ранее проекты были ограничены физическими выводами: 16 светодиодами, двумя трехцветными светодиодами и семисегментным индикатором. Этого было достаточно для простого тестирования логических функций, для контроллера светофора и простого калькулятора. В данной главе использован ПЗУ для отображения текста, был введен программируемый PLL и использован контроллер DDR2. Теперь все готово к выполнению завершающего проекта.

Следующая глава является завершающей, где все наработки из предыдущих глав будут собраны вместе. Можно использовать VGA для отображения вывода с датчика температуры, калькулятора и микрофона. Также будет реализован интерфейс клавиатуры PS/2, чтобы обеспечить более простой способ управления разработанной системой.

Вопросы

1. Логический элемент XOR (Исключающее ИЛИ) можно использовать как:

   a) способ сложения двух битов;
   b) способ умножения двух битов;
   c) программируемый инвертор.

2. Каково ограничение при генерации контроллера VGA?

   a) 640×480 @ 60 Гц.
   b) 1280×1024 @ 85 Гц.
   c) 1920×1200 @ 60 Гц.
   d) Разрешение, с которым может работать монитор, и частота вывода пикселей, с которой можно надежно удовлетворить всем временны́м требованиям к проекту.

3. Создание контроллера VGA в FPGA нецелесообразно.

   a) Верно.
   b) Неверно.

4. Сколько цветов можно отобразить при глубине цвета 888 или 24 bpp?

   a) 2 цвета.
   б) 16 цветов.
   c) 64K цветов.
   d) True Color (16 млн цветов).

5. Из чего среди перечисленного ниже состоит интерфейс записи AXI-Lite?

   a) Адрес записи.
   b) Данные записи.
   c) Ответ на запись.
   d) Все вышеперечисленное.

6. Из чего среди перечисленного ниже состоит интерфейс чтения AXI-Lite?

   a) Адрес чтения.
   b) Данные чтения.

c) Ответ на чтение.
d) Все вышеперечисленное.
e) (a) и (b)

Задание повышенной сложности

Текущая реализация контроллера VGA отображает только черный и белый цвета. Измените проект, чтобы он отображал два других цвета. Измените проект, чтобы использовать переключатели на плате для выбора этих цветов.

Дополнительное чтение

Для получения дополнительной информации о том, что было рассмотрено в этой главе, обратитесь к следующим ссылкам.

- https://www.xilinx.com/support/documentation/ip_ documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf.
- https://glenwing.github.io/docs/VESA-DMT-1.13.pdf.

Глава 10

Свести все воедино

Сделайте глубокий вдох и подумайте о том, чего вы достигли, дойдя до этого места книги. Вы начали путь с минимальными знаниями о SystemVerilog или вообще без них и не знали, как создавать устройства на FPGA. За время чтения этой книги вы прошли путь от простых логических функций, использующих переключатели для зажигания светодиодов, до вывода текста на VGA-экран.

В этой главе будет изучен интерфейс PS/2, который представляет собой выбранный компанией Digilent способ связи с клавиатурой или мышью. Затем проект VGA из главы 9 «Лучший способ отображения – VGA» будет адаптирован под большее разрешение, чем то, что было раньше. Он будет нужен для вывода кодов сканирования с клавиатуры, чтобы можно было увидеть, как работает клавиатура. Также будет адаптирован модуль датчика температуры для отображения на VGA. В конце работы звук, захваченный микрофоном PDM, будет выведен в виде волны на экран.

К концу этой главы будет создан итоговый проект, представляющий собой интерактивное оборудование, которое отображает скан-коды клавиатуры, температуру в градусах Фаренгейта или Цельсия (выбирается с помощью клавиатуры) и аудиоданные в виде волны.

В этой главе будут рассмотрены следующие основные темы:

- использование клавиатуры – знакомство с интерфейсом USB–PS/2;
- отображение данных с клавиатуры на экране;
- преобразование показаний датчика температуры для отображения на VGA-дисплее;
- преобразование сигналов с микрофона PDM для отображения на VGA-дисплее;
- сведение всего воедино в финальном проекте.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH10.

Чтобы реализовать проект на плате, нужен монитор с поддержкой VGA, кабель и USB-клавиатура.

Важное замечание

Nexys A7 поддерживает USB-клавиатуру, которая может работать в режиме PS/2 BIOS. Это ограничение платы Digilent, поскольку исходный код PIC (Peripheral Interface Controller – контроллер интерфейса для периферийного устройства) для подключения USB к PS/2 является закрытым. Если невозможно найти совместимую клавиатуру или купить, можно проанализировать вывод PS/2 в **интегрированном логическом анализаторе (Integrate Logic Analyzer, ILA).** По ссылке приведен пример клавиатуры, про которую известно, что она точно совместима с платой Nexys A7:

https://www.amazon.ca/gp/product/B07THJFXJN/ref=ppx_yo_dt_b_search_asin_ title?ie=UTF8&psc=1&fpw=alm.

Современные клавиатуры игрового типа, судя по всему, поддерживаются платой далеко не все.

Рассмотрим, как работает интерфейс клавиатуры на платах Digilent.

Изучение интерфейса клавиатуры

Без сомнения, вы, как пользователь, знакомы с компьютерными клавиатурами, но, возможно, не знакомы с тем, как клавиатуры реализованы физически.

Клавиатуры состоят из матрицы переключателей. При нажатии на клавишу происходит замыкание цепи. Контроллер клавиатуры активирует по одной линии за раз и проверяет, какие линии подключены, что позволяет определить уникальную клавишу (при условии, что нажата только одна клавиша). Он также определяет, когда клавиша отпущена.

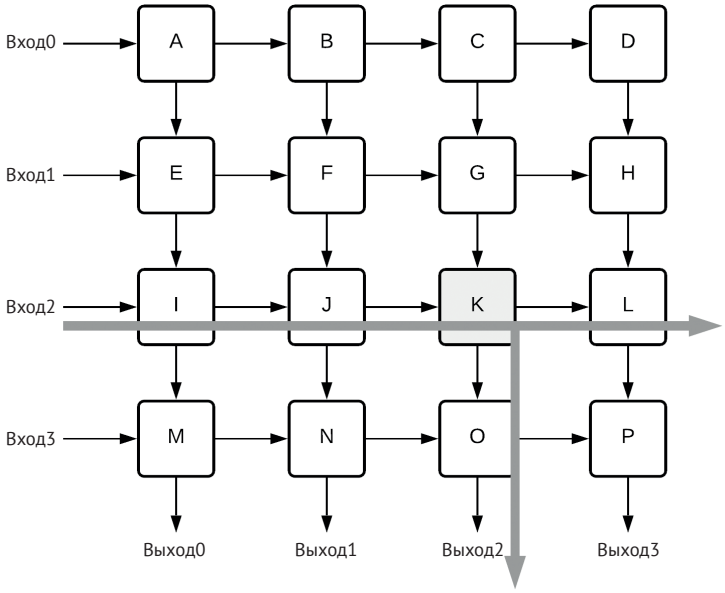


Рис. 10.1. Матрица клавиатуры

Контроллер клавиатуры подает напряжение на каждый вход по очереди. Когда напряжение подано, контроллер анализирует выходы, чтобы определить, нажата ли какая-либо клавиша. На рис. 10.1, когда контроллер сканирует вход 2 и клавиша **K** нажата, на выходе 2 будет сформирован высокий уровень сигнала.

Когда компания IBM представила компьютер PS/2, был введен новый стандарт клавиатуры и мыши. В клавиатуру был помещен матричный декодер, благодаря чему интерфейс клавиатуры упростился до двух проводов. Протокол состоит из 11-битных передач, которые состоят из стартового бита, байта данных, бита паритета, дополняющего количество единиц до нечетного, и стопового бита. Данные передаются от **младшего бита** (**наименее значащего, least significant bit, LSB)** к **старшему** (**наиболее значащему, most significant bit, MSB).** На платах Digilent клавиатура подключается к интерфейсу USB, микроконтроллер PIC выступает в качестве устройства PS/2, а FPGA – в качестве хоста.

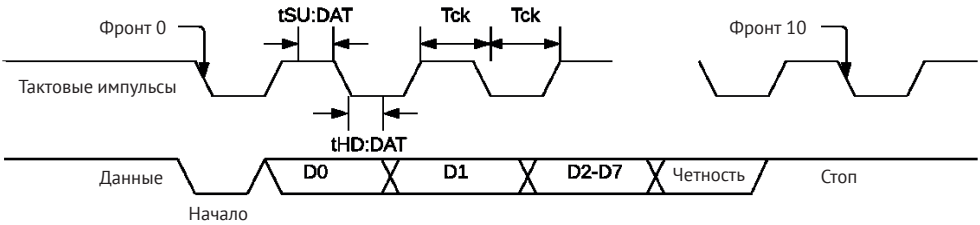


Рис. 10.2. Синхронизация между устройством PS/2 и хостом

На рис. 10.2 показано, как устройство взаимодействует с хостом. Устройство всегда отвечает за генерацию тактовых импульсов для хоста. Проанализируем, как работает этот протокол, изучив конечный автомат PS/2:

```
IDLE: begin
  if (counter_100us != COUNT_100us) begin
    counter_100us <= counter_100us + 1'b1;
    xmit_ready     <= '0;
  end else begin
    xmit_ready     <= '1;
  end
  data_counter  <= '0;
  if (~ps2_clk_clean && ps2_clk_clean_last) begin
    counter_100us <= '0;
    state <= CLK_FALL0;
  end else if (~tx_ready && xmit_ready) begin
    counter_100us <= '0;
    tx_data_out   <= {1'b1, ~^tx_data,tx_data, 1'b0};
    state         <= XMIT0;
  end else if (send_set && xmit_ready) begin
    clr_set       <= '1;
    counter_100us <= '0;
    tx_data_out   <= {1'b1, ~^send_data, send_data, 1'b0};
    state         <= XMIT0;
  end
end
```

В состоянии ожидания автомат следит за нисходящим фронтом `ps2_clock`: `~ps2_clk_clean && ps2_clk_clean_last`, получает внешний запрос на отправку данных или отправляет данные инициализации.

```
CLK_FALL0: begin
  // захват данных
  data_capture <= {ps2_data_clean, data_capture[10:1]};
  data_counter <= data_counter + 1'b1;
  state        <= CLK_FALL1;
end

CLK_FALL1: begin
  // Тактовый сигнал изменился в значение низкого уровня.
  // Ожидание, пока сигнал не изменился в значение высокого уровня.
  if (ps2_clk_clean) state <= CLK_HIGH;
end
CLK_HIGH: begin
  if (data_counter == 11) begin
    counter_100us <= '0;
    done <= '1;
    err           <= ~^data_capture[9:1];
    state         <= IDLE;
    end else if (~ps2_clk_clean) state <= CLK_FALL0;
end
```

Захват данных с устройства происходит в следующих трех состояниях.

1. Данные захватываются при переходе тактового сигнала из состояния низкого уровня в состоянии `CLK_FALL0`.
2. Ожидание высокого уровня тактового сигнала в состоянии `CLK_FALL1`.
3. В `CLK_HIGH`, если получено 11 бит данных, происходит возврат в режим ожидания или ожидание перехода тактового сигнала в нижний уровень и возврат в состояние в `CLK_FALL0`.

С точки зрения FPGA протокол приема очень прост. Данные упаковываются для использования с помощью следующего кода:

```
 initial begin
  out_state = OUT_IDLE;
  rx_data   = '0; rx_user   = '0;  rx_valid  = '0;
end
always @(posedge clk) begin
  rx_valid <= '0;
  case (out_state)
    OUT_IDLE: begin
      if (done && rx_ready) begin
        rx_data                 <= data_capture[8:1];
        rx_user                 <= err; // Флаг ошибки
        rx_valid                <= '1;
        if (~rx_ready) out_state <= OUT_WAIT;
      end
  end
   OUT_WAIT: if (rx_ready) out_state <= OUT_IDLE;
  endcase
  if (reset) out_state <= OUT_IDLE;
end
```

Это создает потоковый интерфейс AXI из модуля `ps2_host`.

Микроконтроллер PIC, выполняющий функции интерфейса USB-PS/2, по сути, является «черным ящиком», содержание которого неизвестно. Это создает проблемы, если клавиатура ведет себя не так, как ожидалось. По мере решения проблемы поиска клавиатуры, работающей с Nexys A7, был разработан полный интерфейс хоста и создана последовательность запуска, похожая на те, которые есть в сети Интернет[1]:

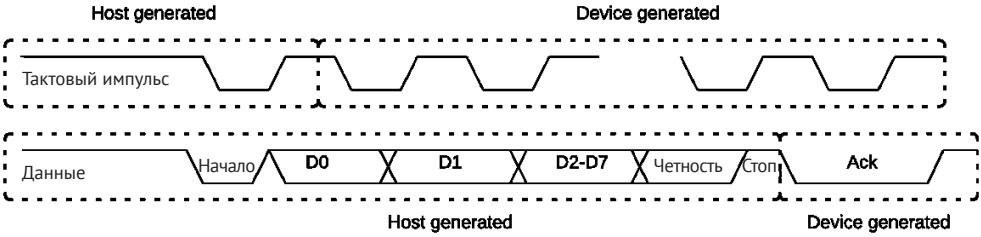


Рис. 10.3. Синхронизация между хостом и устройством PS/2

При использовании мыши PS/2 требуется обратная связь между хостом и устройством. Для клавиатур это не является строго необходимым, хотя установкой подсветки клавиатуры, например клавиш **CapsLock** или **NumLock**, управляет хост, выдавая команды. На рис. 10.3 показано, как происходит обмен данными. Протокол работает в соответствии с частью записи следующего конечного автомата:

```
XMIT0: begin
  // Подать тактовый сигнал на устройство и
  // удерживать его в низком состоянии 100мкс
end
XMIT1: begin
  // вывести данные и освободить тактовый сигнал для устройства
  // Подождать 20мкс
end
XMIT2: begin
  // Каждый отрицательный фронт тактового импульса продвигает данные
end
XMIT3: begin
  // Ожидание подачи тактового сигнала
end
XMIT4: begin
  // Ожидание ACK
end
XMIT5: begin
  // Ожидание, пока данные достигнут высокого уровня
end
XMIT6: begin
  // Ждать, пока придет передний фронт тактового сигнала,
  // и затем перейти в режим ожидания

end
```

[1] С подробностями работы интерфейса PS/2 (на русском языке) можно ознакомиться в справочнике Михаила Гука «Аппаратные средства IBM PC. Энциклопедия» (изд-во «Питер», 2001, 2006). – *Прим. ред.*

Мышь или клавиатура с помощью интерфейса PS/2 могут быть подключены в любое время, но не обе сразу. Разрабатываемый проект поддерживает только клавиатуру. В рамках усилий по отладке была разработана полная процедура инициализации, представленная в предыдущем конечном автомате `start_state`.

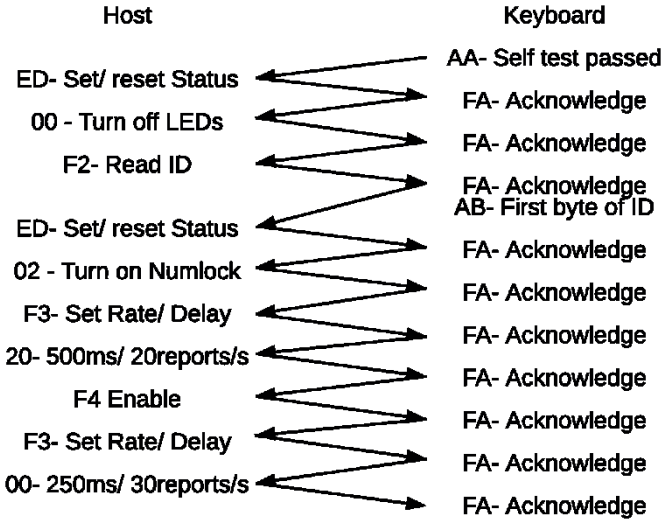


Рис. 10.4. Инициализация интерфейса PS/2

Конечный автомат отправляет и получает последовательность сигналов перед включением устройства. Если у вас возникли проблемы с клавиатурой, которую вы пытаетесь использовать, можно попробовать использовать ILA, который был представлен в главе 3 «Подсчет кликов на кнопку», чтобы определить, правильно ли соблюдена последовательность действий по инициализации клавиатуры.

Во время нормальной работы после инициализации клавиатуры для каждого нажатия клавиши генерируются соотвествующие скан-коды.

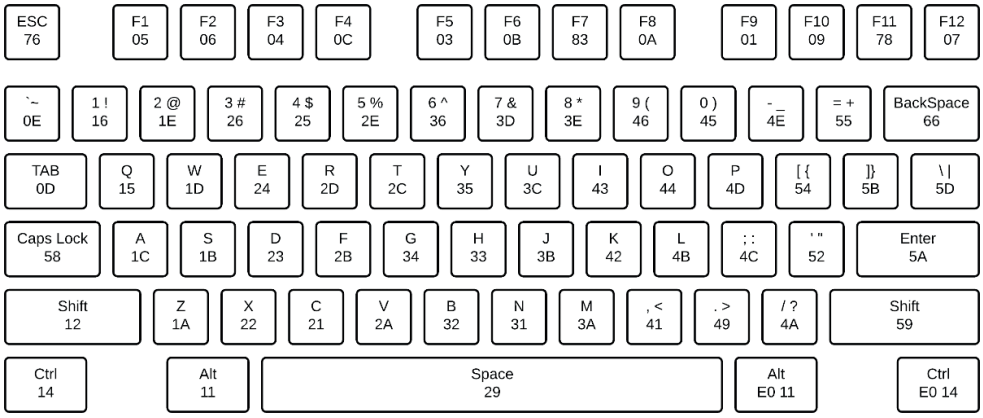


Рис. 10.5. Скан-коды клавиатуры PS/2

На плате Digilent при нажатии клавиши на клавиатуре PIC преобразует протокол USB в PS/2 и генерирует скан-код, представляющий нажатие клавиши для FPGA. На рис. 10.5 показаны скан-коды для большинства клавиш. Если значение клавиши изменяется, перед скан-кодом клавиши передается код модификатора изменения[1]. Если клавиша удерживается нажатой, код клавиши будет посылаться повторно каждые 100 мс. Когда клавиша отпускается, вместе с кодом клавиши отправляется скан-код 0xF0. Также есть некоторые расширенные клавиши, для которых перед скан-кодом посылается код E0. Когда клавиша такого типа отпускается, отправляются значения 0xE0 и 0xF0, чтобы зафиксировать событие поднятия клавиши. FPGA также может взаимодействовать с клавиатурой для установки светодиодов CapsLock или NumLock.

Рассмотрим проект, с помощью которого можно провести проверку возможностей клавиатуры.

Проект 12. Работа с клавиатурой

Выше было рассмотрено, как выглядит интерфейс PS/2. Теперь создадим простой интерфейс, чтобы можно было проверить свои знания, прежде чем переходить к интеграции всего проекта. Первый шаг заключается в том, что нужно избавиться от дребезга сигналов PS/2[2]. Соберем вместе схему для устранения дребезга и `testbench`, чтобы выполнить проверку. Проект находится по адресу:

https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH10/build/debounce/debounce.xpr.

Эта версия кода будет выступать в качестве многократно используемого ядра. Цель тестирования – убедиться, что состояние меняется только после того, как пройдет количество циклов, равное числу CYCLES. Такая схема позволяет избавиться от дребезга.

Следующий код описывает интерфейс модуля антидребезга:

```
module debounce
  #(parameter   CYCLES = 16)
  (input wire   clk,
   input wire   sig_in,
   output logic sig_out);
```

Фактически избавление от дребезга выполняется путем проверки того, сохраняется ли одно и то же состояние в течение периода CYCLES. Обратите внимание, что сигнал `sig_in` тактируется дважды, чтобы убедиться, что в схеме нет проблем с метастабильностью.

---

[1] Модификатор изменения (т. е. клавиша, меняющая назначение обычных клавиш) – это клавиши **Alt**, **Ctrl** или **Shift**. – *Прим. ред.*

[2] Здесь имеется в виду не механический дребезг клавиш, а возникновение дребезга из-за большой разницы частот медленной шины PS/2 и быстрого системного тактового сигнала clk. Нестабильное срабатывание может возникать на фронтах сигналов шины, превышающих по длительности тактовые сигналы. – *Прим. ред.*

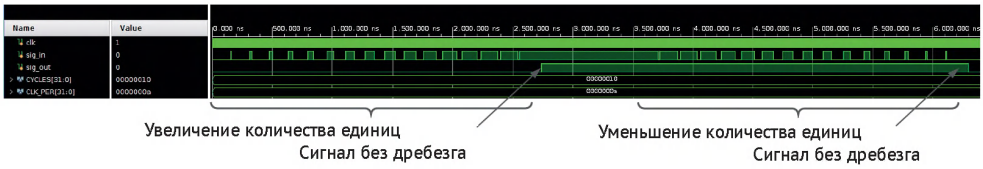


Рис. 10.6. Моделирование защиты от дребезга

После того как защита от дребезга отлажена, можно перейти к анализу кода PS/2. Обработчик сигналов с клавиатуры построен так, чтобы использовать потоковую передачу AXI для более легкой интеграции в другие проекты. Интерфейс к ядру обработчика сигналов клавиатуры выглядит следующим образом:

```
module ps2_host
  #(parameter          CLK_PER = 10,
    parameter          CYCLES  = 16)
  (input wire         clk,
   inout              ps2_clk,
   inout              ps2_data,
   // Передача данных на клавиатуру из FPGA
   input wire         tx_valid,
   input wire [7:0]   tx_data,
  output logic       tx_ready,
   // Данные от устройства на FPGA
   output logic [7:0]  rx_data,
   output logic        rx_user, // Индикатор ошибки
   output logic        rx_valid,
   input wire          rx_ready
   );
```

Имеется два сигнала с тремя состояниями, `ps2_clk` и `ps2_data`, и интерфейс передачи, который пока не разработан. Этот интерфейс может быть использован для установки **CapsLock**, частоты повтора или других параметров, которые может принимать клавиатура. Есть вторая шина, по которой передаются данные, полученные от клавиатуры. Также есть сигнал, поступающий от пользователя, который можно использовать для сообщения об обнаружении ошибки четности, если она возникнет.

```
// Очищение входных сигналов
debounce
  #(.CYCLES   (CYCLES))
u_debounce[2]
  (.clk (clk),
   .sig_in   ({ps2_clk,       ps2_data}),
   .sig_out  ({ps2_clk_clean, ps2_data_clean})
   );
```

Добавим массив экземпляров модулей для создания двух схем защиты от дребезга на линиях данных `ps2`.

```
// Сигналы разрешения (enable) формируют 0 на линиях тактирования и данных
assign ps2_clk  = ps2_clk_en  ? '0 : 'z;
assign ps2_data = ps2_data_en ? '0 : 'z;
```

Три состояния на линиях данных необходимы в любом случае. Три состояния на линии тактирования необходимы для реализации выходного буфера. Обратите внимание, что при объявлении `enable` подается сигнал низкого уровня. При подаче единицы на вход `enable` выход переводится в третье состояние. Это означает, что благодаря подтягивающему резистору на выходе формируется состояние логической единицы.

Разработав интерфейс клавиатуры, проведем его моделирование.

Моделирование работы интерфейса PS/2

Разработаем для конечного автомата клавиатуры PS/2 простой testbench. Проект находится по адресу: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH10/build/ps2/ps2.xpr.

Проект представляет собой testbench, который можно использовать для проверки правильности приема скан-кодов с клавиатуры. Основным компонентом конечного автомата является задача (task) обработки кода клавиши send_key. Эта задача принимает скан-код и преобразует его в интерфейс PS/2:

```
task send_key;
  input [7:0] keycode;
  input error;
  begin
  // Сгенерируйте временную диаграмму PS/2, чтобы послать код клавиши, и
  // попробуйте отправить ошибочный код
  end
endtask // send_key
```

Конструкция task языка SystemVerilog используется для задания серии синхронизированных событий. В данном случае генерируется поток данных PS/2 для хоста.

Также добавим еще одну задачу для обработки принимаемых с клавиатуры данных:

```
task rx_key;
  input [7:0] exp_data;
  begin
    // Ожидание фронта сигнала
    edge_count = '0;
    // Ожидание первого заднего фронта сигнала, а затем переднего фронта сигнала
    @(negedge ps2_clk);
    @(posedge ps2_clk);
    while (edge_count < 10) begin
      repeat (100) @(posedge clk);
      ps2_clk0 = '1;
      repeat (100) @(posedge clk);
      if (edge_count == 10) ps2_data0 = '1;
      data_capt[edge_count++] <= ps2_data;
     ps2_clk0 = '0;
    end
    repeat (100) @(posedge clk);
    ps2_data0 = '1;
    repeat (100) @(posedge clk);
    ps2_clk0 = '1;
    repeat (100) @(posedge clk);
    ps2_data0 = '0;
    ps2_clk0 = '0;
    repeat (100) @(posedge clk);
    $display("Captured data: %h", data_capt[8:1]);
    if (data_capt[8:1] != exp_data) begin
      $error("Data miscompared! Expected %h != Received %h",
             exp_data, data_capt[8:1]);
    end
  end
endtask // rx_key
```

Эта задача или аналогичный блок в testbench необходимы, поскольку между testbench (устройством) и хостом происходит обмен подтверждающими сигналами и устройство отвечает за генерацию тактового сигнала. Следует отметить, что задача rx_key не поддерживает корректную синхронизацию. Этот путь выбран, поскольку хост обнаруживает только фронты сигнала, и это позволяет ускорить моделирование. В целом хорошей практикой является соответствие модели тому, как выглядят реальные сигналы, так как это может помочь выявить неявные проблемы, которые в противном случае можно не заметить.

Необходимо также реализовать функцию самопроверки. Для этого введем конструкцию, которая допускает параллельную работу, – функцию fork...join. На рис. 10.7 концептуально показано, что достигается, если запустить симуляцию работы клавиатуры в одном процессе, а логику проверки скан-кодов – в другом:

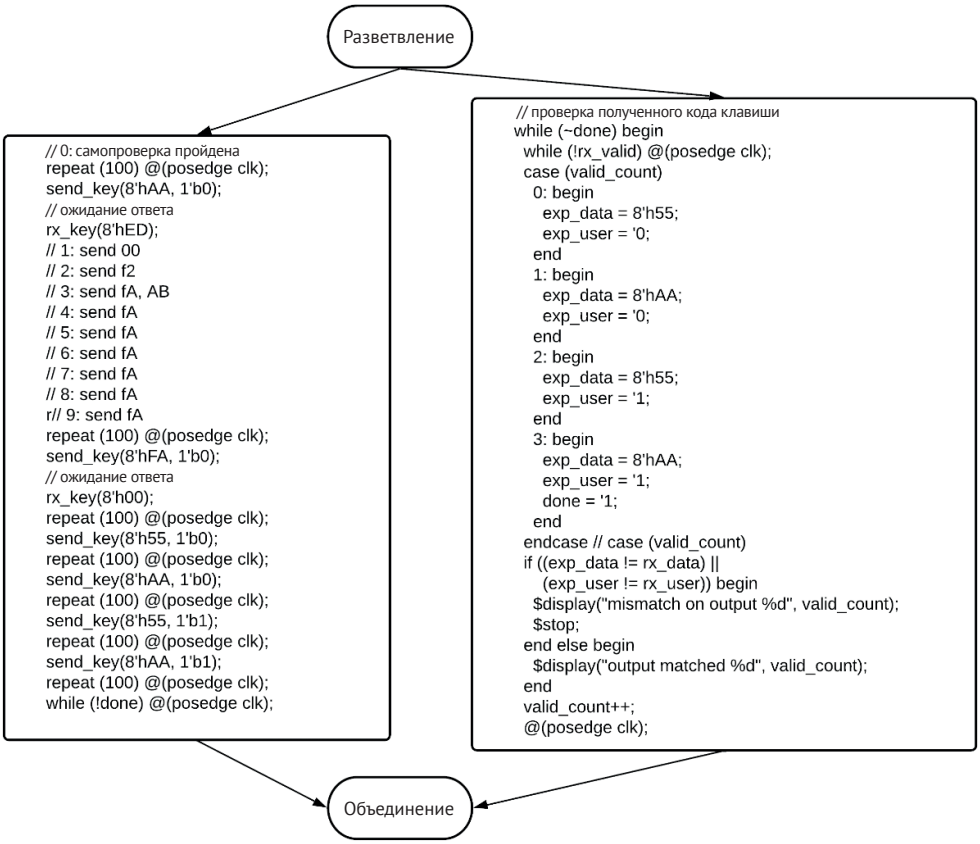
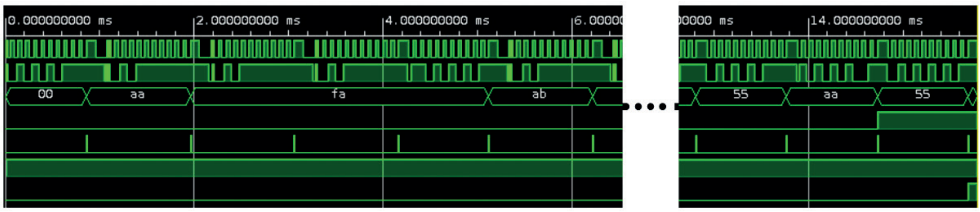


Рис. 10.7. Концептуальное использование разветвления и соединения в testbench

Левый и правый блоки после разветвления представляют собой блоки begin...end. Поскольку они находятся внутри ключевых слов fork...join, два блока begin...end будут выполняться параллельно друг другу. Левая часть генерирует тестовые сигналы и ответы, а правая проверяет последние четыре отправленные комбинации клавиш. Следующая вейвформа демонстрирует результаты запуска testbench в симуляторе:



Последовательность для инициализации

Тестовая последовательность

Рис. 10.8. Тестовая последовательность PS/2

На рис. 10.8 показана вейвформа, полученная в результате моделирования `testbench` для PS/2. Последовательность инициализации представляет собой последовательность, генерируемую компьютером при загрузке. Тестовая последовательность содержит два набора корректных данных и два набора ошибочных данных.

Теперь есть новый способ получения данных в FPGA с помощью клавиатуры, который будет использоваться в финальном проекте.

Проект 13. Сводим все воедино

Стоит приостановиться на минуту и подумать о том, какой путь вы прошли за время работы над книгой. Вначале вы нажимали переключатели и зажигали несколько светодиодов; собрали несколько простых схем, таких как калькулятор и контроллер светофора. Вы собирали и преобразовывали информацию с датчиков температуры, записывали аудиоданные и выводили данные на VGA-монитор.

Теперь взглянем еще раз на эти проекты, чтобы собрать некоторые из них и объединить в финальном проекте. Основой будет VGA, созданный в главе 9 «Лучший способ отображения – VGA». Это позволит легко отображать текст или графику. В предыдущем разделе было проведено моделирование PS/2. Но не была еще проведена апробация клавиатуры в действии. Для большинства клавиш каждое нажатие на них генерирует как минимум 3 байта: 1 байт при нажатии клавиши (keydown) и 2 байта при отпускании (keyup). Реализуем способ отображения этого на экране. Наконец, можно посмотреть на аудиоданные. Можно проанализировать данные в ILA, но было бы неплохо просмотреть форму сигнала на экране.

Этот проект находится по ссылке https://github.com/PacktPublishing/ Learn-FPGA-Programming/blob/master/CH10/build/final_project/final_project.xpr.

Отображение кодов клавиш PS/2 на экране VGA

Блок интерфейса хоста PS/2 обеспечивает удобный способ получения информации через потоковый интерфейс. Этот интерфейс передает за один раз по одному байту, которые поступают с клавиатуры. Рассмотрим, как можно захватить информацию и отобразить ее на экране.

Для отображения информации о режиме VGA создан 16-байтовый (128-битный) массив для хранения информации. Это хорошо подходит для интерфейса DDR2, поэтому для символов PS/2 будут использоваться аналогичные массивы.

Поскольку каждый байт из PS/2 занимает два байта в массиве символов, можно определить хранилище следующим образом:

```
typedef struct packed
               {logic [7:0] data;
                logic       error;
               } ps2_t;
localparam PS2_DEPTH = 8;
ps2_t                        ps2_data_capt;
logic [PS2_DEPTH*2-1:0][7:0] ps2_data_store;
logic                        ps2_toggle;
(* async_reg = "TRUE" *) logic [2:0] ps2_sync;

logic                        update_ps2;
logic                        clear_ps2;
```

Запустим интерфейс PS/2 с использованием тактового сигнала с частотой 200 МГц. Проект реализован независимым от тактовой частоты, благодаря чему можно задать скорость работы, указав период тактовой частоты. Интерфейс от PS/2 к VGA будет асинхронным, поэтому нужно учитывать пересечение тактовых доменов.

Поскольку для выбора между градусами Фаренгейта и Цельсия тоже будет использоваться клавиатура, нужно будет определить, когда нажата клавиша **C** или **F**, и отслеживать ее состояние.

```
// переключение синхронизации и захват данных
always @(posedge clk200) begin
  if (ps2_rx_valid) begin
    ps2_toggle    <= ~ps2_toggle;
    ps2_data_capt <= '{data: ps2_rx_data, error: ps2_rx_err};
    case (ps2_rx_data)
      8'h2B: ftemp <= '1; // F = фаренгейт
      8'h21: ftemp <= '0; // C = цельсий
   endcase
  end
end
```

Если вспомнить, как работает потоковый интерфейс AXI, то вместе с данными передается сигнал `valid`. Когда `ps2_rx_valid` принимает значение логической единицы, необходимо переключать сигнал, захватываемый на линии `ui_clk`. Данные хранятся в структуре вместе с сигналом. Задача состоит в том, чтобы детектировать скан-коды для F и C, 0x2B и 0x21 соответственно. Сигнал `ftemp` определяет, какой формат температуры, в градусах Фаренгейт или Цельсия, используется в текущий момент.

В домене `ui_clk` происходит детектирование фронтов в синхронизированном сигнале переключения. Создадим сдвиговый регистр, как показано на рис. 10.9. PS/2 скан-коды получают шестнадцатеричное число, преобразованное в символы ASCII, и помещаются в позицию 0, после чего каждая позиция символа перемещается по конвейеру.

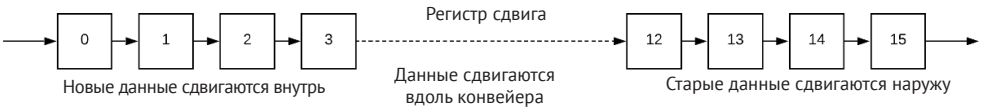


Рис. 10.9. Регистр сдвига

Регистры сдвига настолько широко используются в проектировании, что slice в Xilinx FPGA имеют специальные режимы для более оптимального использования их в качестве регистров сдвига.

В коде финального проекта это реализовано следующим образом:

```
if (^ps2_sync[2:1]) begin
  update_ps2 <= '1;
  for (int i = PS2_DEPTH-1; i >= 0; i--) begin
    if (i == 0) begin
      for (int j = 1; j >= 0; j--) begin
        // Преобразование полубайта в символ
      end
    end else begin
      ps2_data_store[i*2+:2] <= ps2_data_store[(i-1)*2+:2];
    end
  end // for (int i = 0; i < PS2_DEPTH; i++)
end
```

Внешний цикл работает с самого верхнего символа, копируя следующий более низкий символ, пока не достигнет символа 0. При этом каждый полубайт преобразовывается в символ ASCII.

Как только регистр загружен, генерируется сигнал конечному автомату, который генерирует текст для обновления экрана.

```
end else if (update_ps2) begin // if (^update_text_sync[2:1])
  // Вывод потока символов с интерфейса PS2 начинается в строке 8
  y_offset      <= 8 * real_pitch;
  clear_ps2     <= '1;
  char_index    <= ps2_data_store[0];
  capt_text     <= ps2_data_store;
  s_ddr_awvalid <= '0;
  s_ddr_wvalid  <= '0;
  text_sm       <= TEXT_WRITE0;
```

Используется тот же интерфейс, что и для VGA, с такими изменениями, чтобы можно было передавать сигнал char_index, а также сигнал capt_text, который будет использоваться. Расширим этот интерфейс для передачи данных с датчика температуры и аудиоданных. Когда эта программа будет запущена в финальном проекте, будет происходить вывод скан-кодов на дисплее, как показано на рис. 10.10:

Рис. 10.10. Отображение скан-кодов при нажатии клавиш

Каждое нажатие клавиши продвигает один или несколько байтов в регистре сдвига слева направо. Первый байт, B2, который на самом деле равен 0x2B в шестнадцатеричном формате, является кодом нажатия клавиши **F**, f0 – кодом отпускания, а следующее B2 – код клавиши **F** (отпущенной). Эти три

байта представляют нажатие и отпускание клавиши **F**. Далее следует строка значений AF, что соответствует шестнадцатеричному 0xFA, поскольку она заменена полубайтами, которые, если проанализировать процедуру инициализации, представляют собой подтверждение клавиатуры.

С помощью этого кода на экран выводится 8 байт данных PS/2. Теперь рассмотрим добавление датчика температуры.

Отображение показаний датчика температуры

Ранее в главе 6 «Математика, параллелизм и конвейерное проектирование» был разработан модуль датчика температуры с плавающей точкой. Теперь нужно взять данные, которые выводились бы на семисегментный индикатор, преобразовать их в ASCII и вывести на VGA-дисплей. Для этого создана обертка `i2c_wrapper`. Она включает модуль `i2c_temp_flt`, который был разработан ранее, и создает выходную строку из 16 символов, которые были определены ранее.

Напомним, что `temp_valid` и `encoded` являются выходами модуля датчика температуры. Значение в `encoded` – это десятичное представление с десятичной точкой на позиции 4. Также есть возможность выбрать формат представления температуры в формате градусов Фаренгейта и Цельсия, поэтому к выходу необходимо добавлять символ *F* или *C*, чтобы отличить режим, в котором выводится температура.

Final_project.sv

```
always @(posedge clk) begin
  if (temp_valid) begin
    update_temp             <= ~update_temp;
    capt_temp               <= "     F 0000.0000";
    capt_temp[9]            <= 8'h0C; // Degree symbol
    if (ftemp) capt_temp[10] <= "F";
    else       capt_temp[10] <= "C";
    for (int i = 7; i >= 0; i--) begin
      if (i > 3) begin
        capt_temp[7-i] <= 8'h30+encoded[i];
      end else begin
        capt_temp[8-i] <= 8'h30+encoded[i];
       end
     end
   end
end // always @ (posedge clk)
```

https://github.com/PacktPublishing/Learn-FPGA-Programming/CH10/hdl/text_ rom.sv

Ключевым элементом схемы является сигнал переключения `update_temp`, поскольку схема работает в тактовом домене 200 МГц и нужен точный способ сигнализации функции отображения о доступности новых данных. Для этого происходит определение формата `capt_temp` и переопределение F/C на основе сигнала `ftemp`.

Поскольку ASCII-коды для символов 0-9 – это `0x30-0x39`, цикл добавляет пробелы к цифрам около десятичной точки, после чего добавляется целочисленное значение к 0x30, чтобы получить ASCII представление для отображения числовых значений.

Для отображения строки можно использовать ту же функцию в конечном автомате для вывода текста:

```
end else if (update_temp_capt) begin
  // Вывод температуры происходит в строке 16
  y_offset         <= 16 * real_pitch;
  update_temp_capt <= '0;
  char_index       <= capt_temp[0];
  capt_text        <= capt_temp;
  s_ddr_awvalid    <= '0;
  s_ddr_wvalid     <= '0;
  text_sm          <= TEXT_WRITE0;
```

Когда будет захвачен синхронизированный сигнал обновления экрана, и при этом не будет выполняться работа над обновлением другого текста, получившаяся строка будет выведена на экран. Можно сделать еще дополнительное усовершенствование системы вывода, добавив пользовательский символ для представления символа градусов.

Добавление пользовательского символа в ПЗУ для хранения текста

Дополнительный символ градуса будет храниться в ПЗУ, что реализовано следующей строкой кода:

```
capt_temp[9] <= 8'h0C; // Символ градуса
```

Было выбрано пустое место по адресу 0x0C в ПЗУ для хранения текста и создано представление для символа градуса. На рис. 10.11 показано, как строится символ:

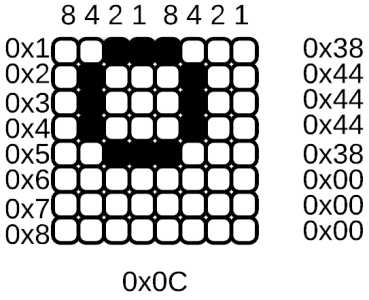


Рис. 10.11. Представление символа градуса

На каждый символ приходится восемь строк сканирования. Каждая строка сканирования представляет биты для отображения. Чтобы вычислить байты, необходимо сложить включенные пиксели в каждой тетраде. В результате получена следующая таблица поиска для символа градуса:

```
// Символ градуса
{8'h0C, 3'h0}: bitmap <= 8'h38;
{8'h0C, 3'h1}: bitmap <= 8'h44;
{8'h0C, 3'h2}: bitmap <= 8'h44;
{8'h0C, 3'h3}: bitmap <= 8'h44;
{8'h0C, 3'h4}: bitmap <= 8'h38;
{8'h0C, 3'h5}: bitmap <= 8'h00;
```

```
{8'h0C, 3'h6}: bitmap <= 8'h00;
{8'h0C, 3'h7}: bitmap <= 8'h00;
```

Теперь, когда работа с данными датчика температуры закончена, рассмотрим, как можно отобразить аудиоданные.

Отображение аудиоданных

Последняя секция отображения фактически представляет форму волны сигнала в виде необработанного графика. Есть несколько вариантов отображения такой информации, но в конечном итоге задача состоит в том, чтобы показать амплитуду волны.

Обычно вейвформы такого типа отображаются на экране слева направо, как показано на рис. 10.12:

Рис. 10.12. Типичное представление синусоидальной волны

Такой тип реализации не может быть легко и эффективно выполнен на аппаратном уровне. Для целей финального проекта можно отображать выходной сигнал вертикально. Вспомним модуль `pdm_input`. Он захватывает 7-битный отсчет аудио, который может быть представлен как точка на 128-битном сегменте строки сканирования.

Поскольку код проекта может быть многократно переиспользован путем указания нужной в конкретной ситуации тактовой частоты, модуль `pdm_inputs` можно использовать без изменений. Потребуется только добавить некоторую внешнюю логику для буферизации данных. Так как данные будут отображаться вертикально, поле вывода ограничено менее чем 480 линиями сканирования. Ограничим область отображения 256 строками сканирования. Поскольку вывод данных и захват отсчетов будут происходить параллельно, для этого понадобится простая двухпортовая оперативная память (один порт для чтения и один порт для записи).

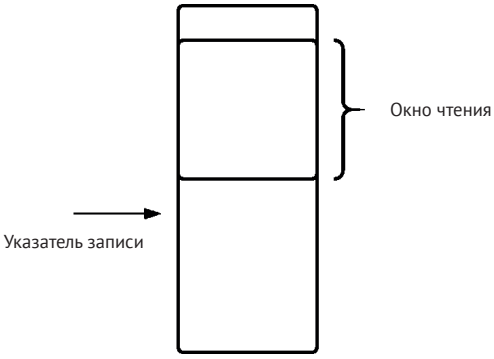


Рис. 10.13. Буферизация аудиоданных для отображения

Создадим буфер для хранения 1024 отсчетов, хотя 512 было бы более чем достаточно. Не стоит ограничиваться только 256 отсчетами, потому что существует возможность перезаписи данных до их считывания. Это обеспечит последовательное построение 256 отсчетов:

```
always @(posedge clk200) begin
  if (amplitude_valid) begin
    amplitude_store[amp_wr] <= amplitude;
    amp_wr                  <= amp_wr + 1'b1;
  end
  amp_data <= amplitude_store[amp_rd];
end
```

Хранилище организовано так же, как и в главе 5 «Ресурсы FPGA, и как их использовать», с той лишь разницей, что указатель `amp_wr` увеличивается при каждой выборке.

Также надо создать сигнал, который будет обновлять дисплей при каждом `VSync`. Это требует модификации модуля VGA для генерации сигнала переключения при каждом `VSync`. Было решено поместить логику в модуль `vga_core`, так как полярность синхронизации VGA меняется для разных разрешений экрана.

Для передачи данных в конечный автомат для вывода текста принято решение использовать FIFO. Это позволяет легко пересекать тактовые домены, и передавать вертикальное расположение, а также сегмент линии сканирования для отображения.

Создадим конечный автомат `wave_sm`, который выполняет генерацию строки сканирования:

```
case (wave_sm)
  WAVE_IDLE: begin
    if (^vga_sync_toggle_sync[2:1]) begin
      // получить данные амплитуды из адресов в памяти выше указателя записи
      // на 256 элементов
      amp_rd   <= amp_wr - 256;
      rd_count <= '0;
      wave_sm  <= WAVE_READ0;
    end
  end
  WAVE_READ0: begin
    // обращение к оперативной памяти в этом цикле разрешено
    amp_rd   <= amp_rd + 1'b1;
    rd_count <= rd_count + 1'b1;
    wave_sm  <= WAVE_READ1;
  end
  WAVE_READ1: begin
    // обращение к оперативной памяти в этом цикле разрешено
    amp_rd           <= amp_rd + 1'b1;
    rd_count         <= rd_count + 1'b1;
    pdm_push         <= '1;
    pdm_din.address  <= 31 + rd_count;
    pdm_din.data     <= 1'b1 << amp_data;
    if (rd_count[8]) wave_sm <= WAVE_IDLE;
  end
endcase // case (wave_sm)
```

Когда vga_sync переключается, указатель чтения сдвигается на 256 семплов вверх, после чего происходит последовательное считывание и вывод 256 линий сканирования на экран. Но запись происходит по одной линии сканирования за раз:

```
end else if (!pdm_empty) begin
  pdm_pop          <= '1;
  char_y           <= '1; // Принудительная запись одной строки
  update_temp_capt <= '0;
  s_ddr_awvalid    <= '1;
 s_ddr_awaddr     <= pdm_dout.address * real_pitch;
  s_ddr_wvalid     <= '1;
  s_ddr_wdata      <= pdm_dout.data;
  text_sm          <= TEXT_WRITE2;
```

Чтобы так происходило, счетчик char_y установлен на значение 7. Благодаря этому происходит запись только одной линии сканирования при каждом извлечении данных из FIFO. При этом также происходит запись, поскольку нет необходимости перебирать несколько строк сканирования.

На этом этапе можно собрать проект и проверить его работу на плате. После программирования платы на экране должно появиться изображение, как показано на рис. 10.14:

Рис. 10.14. Первоначальный вывод

Можно воспроизвести какой-нибудь звук или запустить генератор тона, чтобы получить более интересный результат, как показано на рис. 10.15.

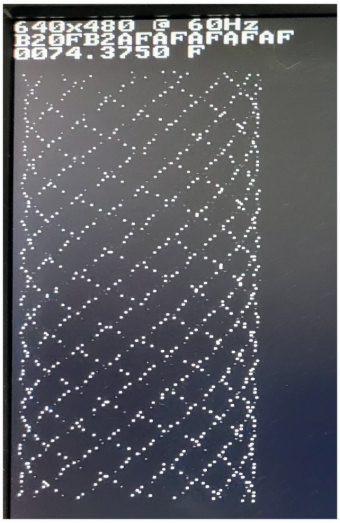


Рис. 10.15. Вывод финального проекта

Этот скриншот экрана был сделан для финального проекта при захвате тонального сигнала из приложения генератора тона для сотового телефона. Здесь показаны все представленные компоненты:

- разрешение экрана;
- скан-коды;
- показания температуры в градусах Фаренгейта;
- визуализация захваченного звука.

Главный проект книги выполнен. Были собраны вместе несколько многократно используемых компонентов и создано из них полезное приложение. Вместо управления несколькими светодиодами был создан интерфейс для дисплея и добавлены к нему выводимый текст и графика.

Выводы

В этой главе был изучен интерфейс клавиатуры PS/2 и создан интерфейсный модуль, который может записывать и получать данные с клавиатуры. После того как PS/2 была готова к использованию, мы взяли некоторые части из последних нескольких глав: VGA-интерфейс для отображения данных, модуль датчика температуры для вывода числовых значений и интерфейс PDM, чтобы можно было выводить что-то более ориентированное на графику. Теперь вы прошли весь путь от базовых логических элементов до разработки кода, способного отображать текст и графику на экране. Можно пойти гораздо дальше, если вести разработку на чистом SystemVerilog, но следующим хорошим шагом было бы изучение софт-процессора (ядра процессора, описанного на HDL).

В этой книге есть еще дополнительная глава, где содержатся некоторые более сложные конструкции, которые могут помочь при проектировании и моделировании, после чего вы будете готовы к решению собственных задач по разработке различных цифровых систем.

Вопросы

1. Клавиатуры PS/2 используют двухпроводной интерфейс, состоящий из:

   a) нажатия/отпускания клавиш;
   b) тактового генератора / данных;
   c) ввода/вывода данных.

2. Скан-код генерируется всякий раз, когда клавиша:

   a) нажата;
   b) отпущена;
   c) удерживается нажатой;
   d) все вышеперечисленное.

3. Для отображения скан-кодов на VGA были использованы:

   a) преобразователь шестнадцатеричного кода в ASCII;
   b) регистр сдвига;
   c) BCD-кодер;
   d) (a) и (b).

4. Как был изменен конечный автомат для вывода текста для отображения аудиоданных?

   a) Он принимает 128 бит графических данных и записывает их в правильный адрес конкретной строки сканирования с помощью `text_sm`.
   b) Графика отображается на символы, и происходит повторное использование переменной состояния `text_sm`.
   c) Был создан новый конечный автомат для отображения графики.

5. Чтобы запустить обновление звука, происходит:

   a) обновление при каждом захвате отсчета;
   b) обновление каждую секунду;
   c) обновление при каждой вертикальной синхронизации;
   d) обновление, когда больше никакая другая работа не выполняется.

Задание повышенной сложности

Обычно аудиоданные отображаются горизонтально. Как изменить код, чтобы создать горизонтальное отображение? Это сложная задача, и ее решение может занять некоторое время. Вот несколько подсказок, как это можно сделать.

- Можно очистить область вывода, а затем просто построить точки, которые нужно установить.
- Можно буферизовать 128 бит каждой строки сканирования и использовать существующий интерфейс FIFO для отображения данных.

Существует множество других способов выполнить данную задачу. Можно выбрать тот, который больше всего нравится.

Дополнительное чтение

Для получения дополнительной информации о том, что было рассмотрено в этой главе, обратитесь к следующей ссылке:

- https://www.avrfreaks.net/sites/default/files/PS2%20Keyboard.pdf.

Глава 11

Темы повышенной сложности

В ходе изучения книги у вас была возможность попробовать свои силы в нескольких различных проектах. Чтобы можно было быстро приступить к работе, синтаксис был несколько ограничен. В этой главе вы узнаете о нескольких новых конструкциях, которые могут пригодиться при синтезе и верификации. Также будет рассказано о некоторых вещах, которых следует остерегаться при проектировании на HDL.

К концу этой главы вы познакомитесь почти со всеми полезными конструкциями SystemVerilog для проектирования и тестирования на FPGA.

В этой главе будут рассмотрены следующие основные темы:

- изучение более продвинутых конструкций SystemVerilog;
- изучение некоторых более продвинутых конструкций для верификации;
- проблемы, возникающие при проектировании, и как их избежать.

Технические требования

Технические требования для этой главы такие же, как и для главы 1 «Введение в FPGA и Xilinx Vivado».

Чтобы выполнять примеры и проекты в этой главе, используйте код из репозитария GitHub по ссылке:

https://github.com/PacktPublishing/Learn-FPGA-Programming/tree/master/CH11.

Изучение более продвинутых конструкций SystemVerilog

В предыдущих разделах при создании проектов использовалось множество базовых конструкций SystemVerilog. Этого синтаксиса достаточно для создания любого проекта. Но есть и другие конструкции, которые могут быть полезны, поэтому они представлены с примерами их использования. Самая полезная конструкция – это интерфейс.

Взаимодействие компонентов с использованием конструкции под названием «интерфейс»

Интерфейсы в SystemVerilog можно рассматривать как модули, соединяющие другие модули. В своей простейшей форме интерфейс представляет со-

бой шину из проводов, очень похожую по своей организации на структуру. Но, в отличие от структуры, направление каждого отдельного сигнала является независимым, что означает, что в интерфейсе могут быть определены как входы, так и выходы.

Проект, демонстрирующий реализацию `ps2_host` с использованием интерфейса расположен по ссылке: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/ps2_host/ps2_host.xpr.

Интерфейсы также имеют дополнительное преимущество: они могут инкапсулировать функции (functions), задачи (tasks) и утверждения (assertions), связанные с сигналами данного интерфейса. Это повышает возможность повторного использования кода и облегчает его разработку. Большинство проектов, которые были показаны в этой книге, имели всего несколько уровней вложенности модулей. В больших проектах сигналы могут распространяться намного глубже. При внедрении **интерфейса** добавление, удаление или изменение размера сигнала становится таким же простым, как изменение определения интерфейса.

Рассмотрим модификацию модуля `ps2_host` для использования интерфейса. Вспомните, что в первоначальной версии в главе 10 «Свести все воедино» интерфейс выглядел так, как показано в следующем коде:

```
// Передача данных на клавиатуру из FPGA
input wire          tx_valid,
input wire [7:0]    tx_data,
output logic        tx_ready,
// Данные от устройства на FPGA
output logic [7:0]  rx_data,
output logic        rx_user, // Индикатор ошибки
output logic        rx_valid,
input wire          rx_ready
```

Этот фрагмент кода – хороший пример для реализации интерфейса. Хорошей идеей является поддержание постоянного соглашения о наименованиях. Рекомендуется сохранять интерфейс в формате `<имя_интерфейса>.intf` и сохранять каждый интерфейс в отдельном файле `.sv`.

Интерфейсы, как и модули, могут содержать параметры и список портов. В `ps2_intf` это не понадобится, но с целью демонстрации такой возможности реализована инкапсуляция функций.

```
interface ps2_intf;
  // Интерфейсы по аналогии с модулями могут содержать списки параметров
  // Интерфейсы по аналогии с модулями могут содержать входы и выходы
  logic          tx_valid;
  logic [7:0]    tx_data;
  logic          tx_ready;
  logic [7:0]    rx_data;
  logic          rx_user;
  logic          rx_valid;
  logic          rx_ready;
```

Первая часть интерфейса определяет сигналы внутри самого интерфейса. Вторая часть содержит элементы **`modport`**. Эти элементы позволяют определять направление сигналов. Если `modport` не используется, сигнал считается двунаправленным:

```
modport master
   (output tx_valid,
    output tx_data,
    input  tx_ready,
    input  rx_data,
    input  rx_user,
    input  rx_valid,
    output rx_ready);
 modport slave
   (input  tx_valid,
    input  tx_data,
    output tx_ready,
    output rx_data,
    output rx_user,
    output rx_valid,
  input  rx_ready,
   import parity_gen,
   import parity_check);
```

Здесь следует обратить внимание на ключевые слова `import` для `modport` в режиме `slave`. Это позволяет использовать функции внутри интерфейса, когда он находится в режиме slave; т. е. таким образом реализуется ограничение видимости функции или внутренних сигналов только в определенном режиме работы интерфейса. Это позволяет ассоциированным функциям, таким как функции проверки четности в следующем коде, использоваться модулем при создании его экземпляра.

```
function parity_gen(input [7:0] din);
  begin
    return ~^din;
  end
endfunction // parity_gen
function parity_check(input [8:0] din);
  begin
    return ~^din;
  end
endfunction // parity_check
```

При инкапсуляции функции внутрь интерфейса реализуется все необходимое для генерации команды PS/2 или проверки входящего скан-кода. Это помогает создавать универсальный код, который может быть использован многократно.

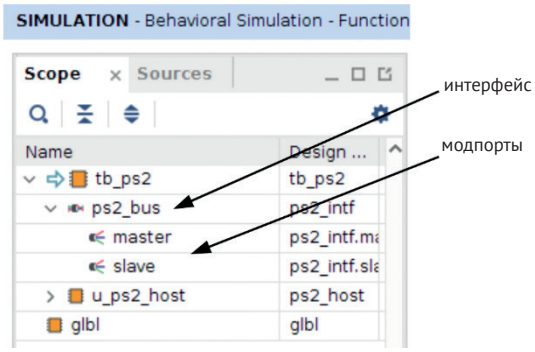


Рис. 11.1. Сигналы интерфейса в симуляторе Vivado

На рис. 11.1 показано, что интерфейс отображается отдельно в списке сигналов, почти как модуль. Также можно отметить, что элементы `modport` отображаются при раскрытии списка. Графики сигналов на вейвформе выглядят также, как и у любых других сигналов.

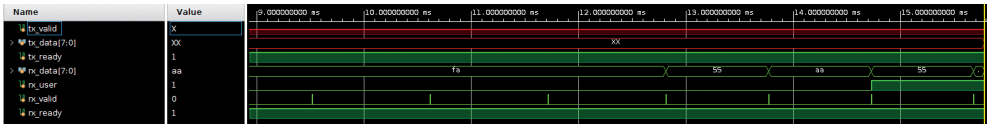


Рис. 11.2. Вейвформа, иллюстрирующая работу интерфейса

`Testbench`, включенный в этот проект, будет пройден при запуске. `Testbench` к этому проекту будет подробно рассмотрен при обсуждении реализации очереди. Интерфейсы можно использовать по своему усмотрению. Некоторые находят их очень эффективными, другие не любят их использовать. При правильном использовании они могут оказаться очень полезными.

При этом следует учитывать, что в настоящее время в Vivado существуют некоторые ограничения для интерфейсов. Модуль верхнего уровня не может использовать интерфейсы в качестве портов. Субблоки в проектах, созданных в схемном редакторе **(Block Design, BD)**, также не могут использовать интерфейсы. Если реализовывать соединения внутри проекта с помощью SystemVerilog, то все будет в порядке.

Рассмотрим более подробно еще один элемент языка SystemVerilog под названием «структура».

Использование структур

На протяжении всей книги во многих примерах кода используются структуры. Они обеспечивают удобный способ упаковки данных и упрощают анализ внутренней структуры проекта. В главе 9 «Лучший способ отображения – VGA» были использованы структуры для хранения информации о разрешении для VGA, как показано в примере кода ниже:

```
typedef struct packed {

  logic [7:0 ]  divide_count;

  logic [15:8]  mult_integer;

  logic [25:16] mult_fraction;

  ...
  logic hpol;
  logic vpol;
  logic [12:0]  pitch;

} resolution_t;
```

Определение (`typedef`) структуры, как в представленном коде, фактически создает новый пользовательский тип данных. Можно создавать упакованные и неупакованные массивы структур или использовать их так же, как и любой другой тип данных. Примером этого является создание таблицы разрешений экрана:

```
resolution_t resolution[18];
```

Для структуры существует несколько способов присвоения. Первый способ – это присвоение значения отдельному полю структуры, который использовался в главе 9 «Лучший способ отображения – VGA».

```
// 25.18 Mhz 640x480 @ 60Hz
resolution[0].divide_count         = 8'd9;
resolution[0].mult_integer         = 8'd50;
resolution[0].mult_fraction        = 10'd000;
...
resolution[0].hpol                 = '0;
resolution[0].vpol                 = '0;
resolution[0].pitch                = 13'd5*16; // 5 строк на экране с глубиной
                                               // цвета 1 bpp (bits per pixel)
```

Другой способ – присвоение по имени поля структуры.

```
// 25.18 Mhz 640x480 @ 60Hz
resolution[0] = '{default:       '0,
                  divide_count:  8'd9,
                  mult_integer:  8'd50,
                  mult_fraction: 10'd000,
                  ...
                  Pitch:         13'd5*16}; // 5 строк на экране с глубиной
                                            // цвета 1 bpp (bits per pixel)
```

При таком способе присвоения можно использовать ключевое слово `default`, чтобы все поля, для которых значения не были указаны явно, проинициализировались значениями по умолчанию. Два представленных фрагмента кода эквивалентны.

Метки блоков

Любой блок `begin...end` может быть помечен. Это продемонстрировано в примере проекта, расположенном по ссылке: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/labels/labels.xpr.

Рекомендуется таким образом помечать операторы `generate`, поскольку это будет требованием в будущих версиях SystemVerilog и может улучшить читабельность программного кода. Как будет показано далее, метки блоков также могут быть полезны при использовании оператора `disable`. В следующем примере кода показано то, как метки блоков могут помочь отловить ошибки, возникающие при разработке программы:

```
always_ff @(posedge clk) begin
  // некорректная метка (не совпадает с меткой возле окончания блока end)
  if (subtraction) begin : l_addition_op
    dout <= in0 - in1;
 end : l_subtraction_op
  // повторное использование метки
  if (addition) begin : l_addition_op
    dout <= in0 + in1;
  end : l_addition_op
end
```

Расстановка меток также может помочь отследить, в каком блоке находится тот или иной фрагмент кода. Метки нельзя использовать повторно. Кроме того, если начальная и конечная метки не совпадают, это приведет к выводу сообщения об ошибке. Запуск приведенного выше кода в симуляторе Vivado приведет к выдаче следующих сообщений в TCL-консоль:

```
ERROR: [VRFC 10-3516] mismatch in closing label 'l_subtraction_
op'; expected 'l_addition_op' [/home/fbruno/git/books/Learn-
FPGA-Programming/CH11/hdl/labels.sv:12]

ERROR: [VRFC 10-2934] 'l_addition_op' is already declared [/
home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/labels.
sv:16]

ERROR: [VRFC 10-3516] mismatch in closing label 'l_addition_
op'; expected '<unnamed>' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/labels.sv:16]

ERROR: [VRFC 10-2865] module 'labels' ignored due to previous
errors [/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/
labels.sv:1]
```

Расстановка меток необязательна, но может принести пользу, а также является хорошим стилем программирования. Далее рассмотрим как устроены циклы в SystemVerilog.

Цикл for

На протяжении всей книги использовался цикл `for`. В каждом случае переменные цикла были определены внутри цикла `for`, что является хорошим стилем программирования. Циклы `for` также позволяют использовать несколько переменных цикла, хотя для завершения цикла допускается только одна проверка условия выхода из цикла. Пример использования цикла `for`:

```
for (int i = 0, j = 0; i *j < 256; i++, j+=8) begin
```

Этот пример подходит для синтеза и реализации.

Цикл do...while

В этой книге есть несколько примеров использования цикла `while`, в частности при тестировании PS/2 в главе 10 «Свести все воедино». Циклы `do...while` и `while` синтезируемы.

Пример реализации функции `last_ones`, использующей цикл `while`, приведен ниже:

```
always_comb begin
  done = '0;
  i = 0;
  while (!done) begin
    if (vector[i] || (i==15)) done = '1;
    else i += 1;
  end
  last_ones = i;
end
```

Функциональность приведенного программного кода может быть реализована и с помощью цикла `do...while`. Цикл `do...while` особенно полезен, когда требуется гарантировать, что цикл выполнится хотя бы один раз.

```
always_comb begin
  done = '0;
  i = 0;
  do
    if (vector[i] || (i==15)) done = '1;
    else i += 1;
  while (!done);
  last_ones = i;
end
```

Оба приведенных фрагмента кода нужны для определения наименьшей установленной позиции бита. Как уже говорилось о цикле `for`, пока есть возможность развернуть цикл, он синтезируем.

До сих пор рассматривались примеры простых циклов, но что делать, если для реализации более сложной операции требуется использовать вложенные циклы?

Выход из цикла с помощью оператора disable

Оператор `break` в цикле `for` использовался для выхода из цикла при обнаружении установленного бита. А что делать, если в программе есть вложенные циклы? Оператор `disable` позволяет отключить именованный блок. Он является синтезируемым, так же как и оператор `break`:

```
always_comb begin
  first_ones = '0;
  for (int i = 0; i < 4; i++) begin : outer_loop
    for (int j = 0; j < 8; j++) begin : inner_loop
      if (din[i][j]) begin
        first_ones = 6'(i*8 + j);
        disable outer_loop;
      end
    end : inner_loop
  end : outer_loop
end // always_comb begin
```

В приведенном коде в двумерном массиве `din` производится поиск первой обнаруженной единицы. После того как единица найдена, ее положение в массиве `6'(i*8 + j)` фиксируется, и поиск останавливается.

Помимо `break` и `disable`, SystemVerilog также поддерживает `continue`.

Пропуск фрагментов кода с помощью оператора continue

В главе 4 «Разработка калькулятора» был разработан детектор ведущей единицы. Код детектора можно реализовать с использованием оператора `continue`:

```
always_comb begin
  LED = '0;
  for (int i = $low(SW); i <= $high(SW); i++) begin
    if (~SW[i]) continue;
    LED = i + 1;
  end
end
```

В качестве альтернативы разрыву цикла можно пропускать какую-то часть кода в цикле, если она встретилась. Представленный пример кода цикла показывает, как можно пропустить все нули в векторе, фиксируя только позицию последней обнаруженной единицы.

Использование констант

Константы позволяют сообщить средствам синтеза что конкретная переменная не должна изменяться во время выполнения кода:

```
// переменная не должна изменяться во время выполнения кода
const int bus_width = 8;
```

Если `bus_width` будет использоваться в левой части операции присваивания, программа выдаст ошибку.

В этом разделе рассмотрено несколько языковых конструкций для проектирования на SystemVerilog. Возможно, вы найдете некоторые из них достойными для дальнейшего изучения и использования, но они, конечно, необязательны, поэтому используйте или игнорируйте их по своему усмотрению.

Теперь рассмотрим некоторые языковые конструкции для улучшения моделирования проектов на SystemVerilog.

Некоторые продвинутые конструкции языка SystemVerilog для верификации

Моделирование проектов, которое проводилось до сих пор, было довольно простым, даже когда использовалась самопроверка. Есть одна языковая конструкция, которая может быть очень полезной при верификации. Очередь (queue) проста для использования и понимания.

Знакомство с очередями SystemVerilog

Часто в проекте необходимо сгенерировать входные данные, которые дадут ожидаемый выход через некоторое время. Примерами этого являются модули синтаксического анализа, модули обработки данных и, как было показано в главе 9 «Лучший способ отображения – VGA», интерфейс PS/2.

Во время модификации модуля `ps2_host` было решено модернизировать `testbench` для него с использованием очередей. Для этого потребовалось создать структуру для определения того, что требуется хранить в очереди:

```
typedef struct packed
  {
   logic [7:0] data;
   logic       parity;
  } ps2_rx_data_t;
```

Эта структура будет хранить ожидаемые данные, поскольку данные для тестирования генерируются в `ps2_host`.

Очередь определяется следующим образом:

```
ps2_rx_data_t ps2_rx_data[$];
```

Это очень похоже на распакованный массив, за исключением того, что размер определен как [$], что определяет его как очередь, длиной которой можно манипулировать при моделировании. Можно получить доступ к очереди, помещая в нее данные в начало (голову очереди) (push_front) или в конец (хвост очереди) (push_back) и извлекая данные из конца очереди (pop_back) или из начала очереди (pop_front). Эти функции, а также оператор `size()` являются наиболее полезными при моделировании. Существуют и другие функции для добавления или удаления из очереди, которые тоже стоит изучить.

На следующем рисунке концептуально представлена очередь. Как правило, данные проталкивают (push) в одну сторону и извлекают с другой стороны (pop). Выбор направления является произвольным. Можно проталкивать или извлекать данные с обеих сторон, что может пригодиться, если нужно проверить значение на одной стороне и, возможно, записать его обратно в то же место:

Рис. 11.3. Структура очереди SystemVerilog

Очередь может быть полезной в `testbench` для хранения проверочных данных, которые ожидается увидеть на выходе схемы. Чтобы сделать это для `testbench ps2`, в задачу `send_key` добавлено следующее улучшение: поскольку заранее известно, что отправляется в интерфейс PS/2, можно сохранить в очереди ожидаемый результат.

```
task send_key;
  input [7:0] keycode;
  input error;
  ps2_rx_data_t local_data;
  begin
    local_data.data = keycode;
    local_data.parity = error;
    ps2_rx_data.push_front(local_data);
```

Когда вызывается задача `send_key`, создается структура, которая описывает ожидаемые данные и помещается в очередь. Функция проверки заменена на приведенную выше очередь.

```
while (~done) begin
  while (!rx_valid) @(posedge clk);
  popped_data = ps2_rx_data.pop_back();
  exp_data    = popped_data.data;
 exp_user    = popped_data.parity;
  if ((exp_data != rx_data) ||
      (exp_user != rx_user)) begin
    $display("mismatch on output %d", valid_count);
    $stop;
  end else begin
    $display("output matched %d", valid_count);
  end
  valid_count++;
```

```
    @(posedge clk);
    if (valid_count == 16) done = '1;
  end
```

При появлении сигнала `rx_valid` очередь освобождается, и происходит сравнение выходных данных из модуля с данными из очереди. Это хороший способ выявления ошибок в проекте. В этом `testbench` происходит проверка определенного количества ожидаемых значений на выходе. В других случаях можно использовать функцию `size` для определения наличия данных в очереди (длины очереди):

```
if (popped_data.size() != 0)
```

Далее будут рассмотрены некоторые способы улучшить отображение данных с помощью системной функции `$display`.

Продвинутое использование системной функции $display

Системная функция `$display` используется для вывода в консоль TCL в процессе моделирования. Эта системная функция родом из Verilog и поддерживает несколько форматов отображения основных типов данных:

- %h, %H – Шестнадцатеричное значение;
- %d, %D – Десятичное значение;
- %b, %B – Двоичное значение;
- %m, %M – Иерархическое имя;
- %s, %S – Строка;
- %t, %T – Время;
- %f, %F – Вещественное число в десятичном формате;
- %e, %E – Вещественное число в экспоненциальном формате.

Если использовать их как есть, `display` будет подгонять данные под вывод. Например:

```
int a, b;
$display("a=%h  b=%h", a, b);

// пример вывода данных в шестнадцатеричном формате

A=00000001 b=0000FFFF
```

SystemVerilog позволяет использовать данную функцию в несколько усовершенствованном виде. Можно использовать `%0h` для полного удаления ведущих нулей или `%(число)h` для ограничения вывода определенным количеством цифр. Кроме того, вместо `%h` можно использовать `%x`:

```
int a, b;
$display("a=%0x  b=%4x", a, b);

// пример вывода данных в шестнадцатеричном формате в усовершенствованном виде

A=1 b=FFFF
```

`%p` позволяет выводить в консоль содержимое структуры в форматированном виде. Модифицируем файл `tb_ps2.sv`, чтобы использовать %p для вывода в консоль передаваемых значений:

```
$display("output matched %d: %p", valid_count, popped_data);
output matched          12: '{data:85,parity:1'b0}
output matched          13: '{data:170,parity:1'b0}
output matched          14: '{data:85,parity:1'b1}
output matched          15: '{data:170,parity:1'b1}
```

Используя `.name`, можно выводить имя переменной перечислимого типа.

```
enum bit {TRUE = 1'b1, FALSE = 1'b0} my_bool;
$display("The state of my_bool is %s", mybool.name);
```

В SystemVerilog также добавлена системная функция `$sformats`, которая похожа на `$display`, но она возвращает строку, которую можно передать в `$display` или в файл логов.

Следует также хотя бы поверхностно познакомиться с утверждениями в SystemVerilog. Утверждения могут занять отдельную книгу, поскольку верификация – это отдельная комплексная тема.

Утверждения

Утверждения (assertions) – это способ добавить в код самопроверку. Утверждения обычно игнорируются при синтезе, их можно хранить в отдельных файлах и привязывать к модулям проекта. Утверждения не рассматриваются подробно. Для более глубокого знакомства с данной темой рекомендуется изучить материалы по ссылкам, приведенным в разделе «Дополнительное чтение». Рассмотрим несколько других дополнений для отображения информации. Они ведут себя так же, как `$display`, но к ним привязаны уровни важности ошибок (severity levels). Они выглядят следующим образом:

- `$info`
- `$warning`
- `$error`
- `$fatal`

Эти уровни важности ошибок позволяют легче фильтровать сообщения при моделировании проекта. Например, можно решить замаскировать сообщения с помощью системной функции `$info` или даже `$warning` во время длительных запусков моделирования, когда проект выполняется слишком медленно. Есть еще одно интересное применение для системных функций `$error` или `$fatal`, представлено далее.

Использование $error или $fatal при синтезе проекта

Часто разработчики создают переиспользуемые модули, которые могут работать только при определенных комбинациях параметров. Можно использовать `$error` или `$fatal` для проверки этих условий и инициировать прерывание синтеза в случае их возникновения. При использовании этих системных функций для таких целей проверка условия их срабатывания должна быть статической, т. е. не изменяться динамически, а быть чем-то вроде тестирования настройки параметров модуля. Например, если вернуться к главе 3 «Подсчет нажатий на кнопку» и дешифратору семисегментного индикатора, то, возможно, может возникнуть необходимость ограничить его четырьмя или восемью сегментами:

```
module seven_segment #(parameter NUM_SEGMENTS = 8, ...
initial begin
  if (NUM_SEGMENTS != 4 || NUM_SEGMENTS != 8)
    $fatal("Number of segments must be set to 4 or 8");
end
```

В показанном фрагменте кода, если количество сегментов не равно 4 или 8, Vivado не выполнит синтез проекта.

Наконец, давайте рассмотрим некоторые проблемы и особенные ситуации, на которые следует обратить внимание при разработке.

Другие проблемы, и как их избежать

Поскольку книга подходит к концу, есть еще несколько моментов, на которые следует обращать внимание, а также на то, как их можно обнаружить или избежать.

Выведение однобитных проводов

С момента появления Verilog всегда было разрешено использовать сигнал без его определения. Это может произойти с портами модуля при создании его экземпляра. По ссылке находится пример такого проекта: https://github.com/PacktPublishing/Learn-FPGA-Programming/blob/master/CH11/build/inferred_wire/inferred_wire.xpr.

Создан модуль сумматора переменной ширины и три его экземпляра:

```
adder #(4) u_add0 (.in0(SW[3:0]),  .in1(SW[7:4]),
                   .out(add0_out));
adder #(4) u_add1 (.in0(SW[11:8]), .in1(SW[15:12]),
                   .out(add1_out));
adder #(5) u_add2 (.in0(add0_out), .in1(add1_out),
                   .out(LED[5:0]));
```

В приведенном примере нет `testbench`, но если попытаться провести моделирование, то будут сгенерированы следующие предупреждения:

```
WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'out' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:9]

WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'out' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:10]

WARNING: [VRFC 10-3091] actual bit length 1 differs from formal
bit length 5 for port 'in0' [/home/fbruno/git/books/Learn-FPGA-
Programming/CH11/hdl/inferred_wire.sv:12]
```

Из сообщений следует, что выходные порты имеют размерность 1 бит, в то время как нужно 5 бит. Чтобы избежать этих проблем, следует раскомментировать первую и последнюю строки файла. Директива компилятора `` `default_nettype `` позволяет задать, что будет происходить с выходными сигналами. Указав `none`, можно сообщить средствам синтеза и моделирования, что если не установлен `nettype` (тип цепи) сигнала, то это будет ошибкой.

Теперь при запуске процесса компиляции будет выведено следующее сообщение об ошибке:

```
ERROR: [VRFC 10-2989] 'add0_out' is not declared [/home/fbruno/
git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:10]

ERROR: [VRFC 10-2989] 'add1_out' is not declared [/home/fbruno/
git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:11]
ERROR: [VRFC 10-2989] 'add0_out' is not declared [/home/fbruno/
git/books/Learn-FPGA-Programming/CH11/hdl/inferred_wire.sv:13]
```

Лучше всего всегда в начале файла устанавливать директивой компилятора значение `default_nettype в none, а в конце в значение wire. Последнее полезно делать для случая, когда используются устаревшие IP-блоки, у которых могут быть выводимые сигналы, которые нет возможности изменить. Следует всегда отслеживать возможные проблемы с шириной шин.

Несоответствие ширины шин

Поскольку это всего лишь предупреждение, несоотвествие ширины шины можно легко пропустить, но в процессе разработки проекта за этим нужно внимательно следить.

Ранее в главе 2 «Комбинационная логика» уже упоминались защелки. Рассмотрим, как избежать их непреднамеренного появления.

Повышение или понижение приоритетности сообщений Vivado

Vivado отображает множество сообщений в процессе проектирования. Как обсуждалось ранее, защелки следует считать ошибкой, если они возникают.

Проект, созданный для иллюстрации возникновения защелок, находится по ссылке: https://github.com/PacktPublishing/ Learn-FPGA-Programming/blob/master/CH11/build/latch_error/latch_error.xpr.

Без tCl.pre в проекте, который был настроен, как показано на рис. 11.4, было бы выведено предупреждение, приведенное ниже. Но при этом в результате синтеза был бы сгенерирован загрузочный файл. Если же это будет пропущено в критически важном месте, например в конечном автомате, то в какой-то момент проект будет работать некорректно:

```
WARNING: [Synth 8-327] inferring latch for variable 'LED_reg'
[/home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/latch_
error.sv:9]
```

Можно изменить приоритет любого сообщения в потоке всех сообщений, создав tcl-файл, который считывается Vivado перед синтезом:

```
set_msg_config -id {[Synth 8-327]} -new_severity ERROR
```

Чтобы указать использовать этот файл перед синтезом, требуется изменить опцию в настройках синтеза:

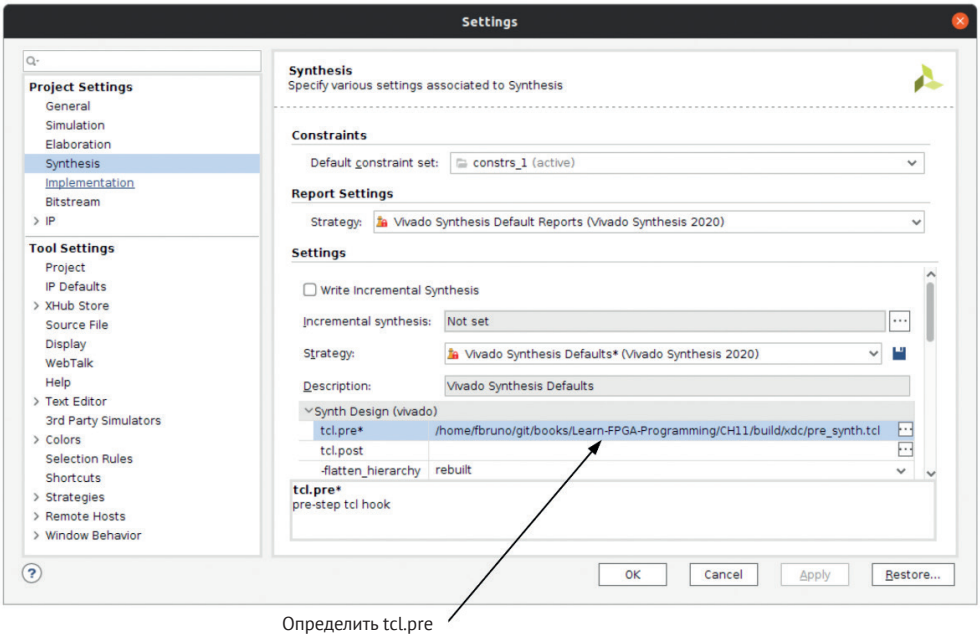


Рис. 11.4. Настройка tcl.pre

Теперь, если попытаться сгенерировать загрузочный файл, будет выведено следующее сообщение, а синтез прервется:

```
ERROR: [Synth 8-327] inferring latch for variable 'LED_reg' [/
home/fbruno/git/books/Learn-FPGA-Programming/CH11/hdl/latch_
error.sv:9]
```

Эта же методология может быть использована для повышения или понижения уровня приоритета любого сообщения.

В следующем подразделе рассмотрены **timing closure**[1] (соответствие проекта временны́м требованиям).

Обработка timing closure

Одна из самых больших проблем, с которой сталкиваются все разработчики, – это соблюдение временны́х требований к проекту. Есть множество типов проблем, которые могут возникать. Первый тип проблем – это отсутствие обработки пересечения тактовых доменов. Чтобы продемонстрировать эту проблему, из финального проекта удалено одно из ограничений:

[1] Timing closure (букв. закрытие по времени) – процесс корректировки логической схемы с целью приведения в соответствие ее временных характеристик. Неоднократно упоминавшаяся на страницах книги проблема пересечения тактовых доменов – одна из составляющих timing closure. – *Прим. ред.*

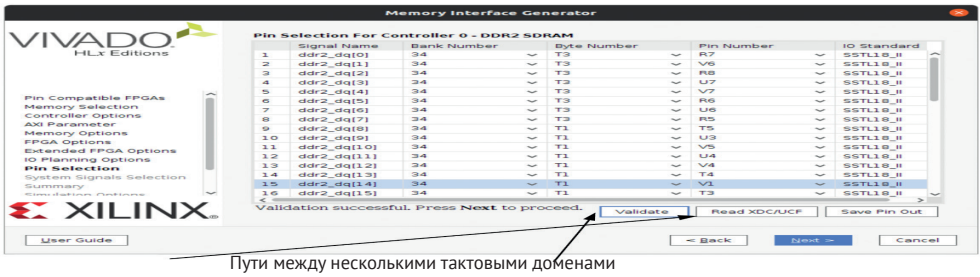


Рис. 11.5. Проблема пересечения тактовых доменов

Проблемы синхронизации с нарушениями межтактовых путей следует исправлять в первую очередь. Средство синтеза часто не будет продолжать оптимизацию путей, если временны́е требования не могут быть соблюдены, поэтому наличие путей между несколькими тактовыми доменами может оказаться ложной тревогой на данном этапе. Исследуем пути прохождения сигналов на следующем рисунке:

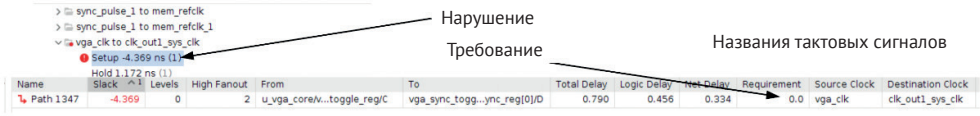


Рис. 11.6. Отчет о нарушении временны́х требований

Посмотрев на отчет, можно заметить несколько странностей. Первое – это вообще наличие тактовых доменов. Известно, что сигнал проходит между двумя тактовыми доменами. Второе – это требование к временны́м соотношениям. Когда требование равно 0 или занимает очень малую часть тактового периода, следует понимать, что это проблема тактового домена. В данном случае синхронизация происходит корректно, просто убрано одно из ограничений.

Второй тип сбоя – это просто недостаточное количество времени в тактовом периоде для выполнения операции, как было задумано. Это может быть вызвано одним из следующих факторов:

- **размещением** – можно попытаться использовать ограничения `pblock`, чтобы исправить размещение. Это не всегда удается и выходит за рамки данной книги;
- **трассировкой** – если пути прохождения сигналов слишком плотно размещены, они могут испытывать перегрузки. Попытка перепроектировать некоторые пути для облегчения синхронизации может помочь;
- **слишком много комбинационных логических элементов в одном пути** – эту проблему можно решить, добавив конвейеризацию или разделив длинные пути, если это возможно;
- **отсутствием конвейеризации DSP** – это относится к предыдущему пункту о слишком большом количестве комбинационной логики на пути распространения сигнала. Если требуется реализовать сложную операцию, которая требует больше ресурсов, чем может реализовать DSP без использования дополнительных ресурсов, может потребоваться добавить этапы конвейера.

Если не получается исправить ситуацию с синхронизацией, последний вариант – это по возможности снизить требования к тактовой частоте.

В следующем разделе будет более подробно рассмотрено, как бороться с проблемой слишком громоздких комбинационных схем.

Конвейеризация

Как было показано, слишком много комбинационной логики на пути прохождения сигнала может приводить к проблемам с удовлетворением временны́х требований. Это может быть решено несколькими способами:

- использованием агрессивной оптимизации (глубокой оптимизации проекта с основным упором на улучшение временны́х параметров схемы);
- конвейеризацией логики;
- конвейеризацией элементов DSP.

Откройте https://github.com/PacktPublishing/Learn-FPGA- Programming/blob/master/CH11/build/pipeline/pipeline.xpr.

В этом проекте реализован умножитель 32×64 бита, имеющий 96-битный выход.

```
always @(posedge clk) begin
  for (int i = 0; i <= PIPELINE; i++) begin
    if (i == 0) result[0] <= mult_a * mult_b;
    else        result[i] <= result[i-1];
  end
  if (button_l) result_rotate <= result[PIPELINE];
  else result_rotate <= {result_rotate[79:0],
                         result_rotate[95:80]};
end
```

Основой проекта является умножитель. Параметр `PIPELINE` (конвейер), определен, чтобы включить конвейеризацию умножителя в случае, если не удастся выполнить временны́е требования. Принцип работы `PIPELINE` заключается в добавлении дополнительных регистров после умножителя, как показано на рис. 11.7.

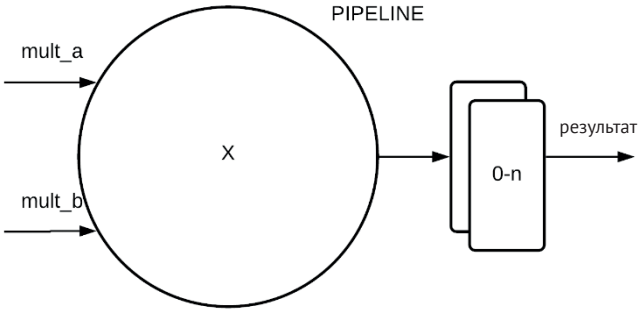


Рис. 11.7. Конвейер до выполнения операции `retiming`

Проверим, может ли операция умножения выполняться за один цикл. Установим частоту тактового сигнала на 200 МГц, чтобы испытать проект. Это приводит к ошибке выполнения временны́х требований.

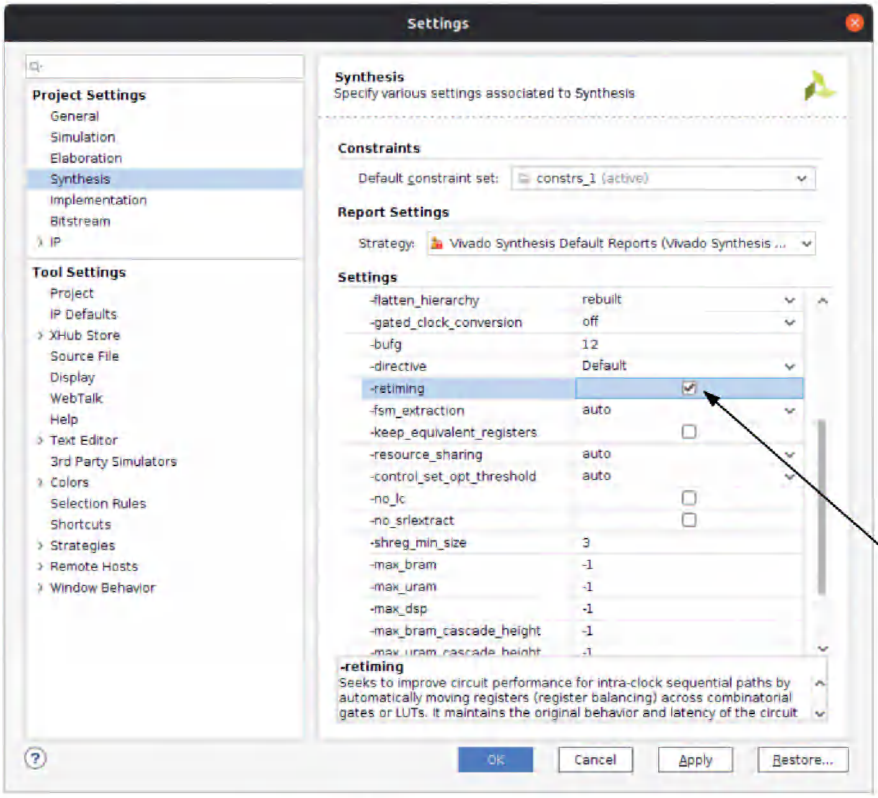
Timing **Setup** | Hold | Pulse Width

Worst Negative Slack (WNS):	-2.454 ns
Total Negative Slack (TNS):	-257.8 ns
Number of Failing Endpoints:	125
Total Number of Endpoints:	535

Implemented Timing Report

Рис. 11.8. Ошибка проверки временны́х требований 32×64-битного умножителя

В качестве первого шага можно попробовать отрегулировать параметры синтеза и реализации проекта.



Разрешить оптимизацию временны́х параметров схемы (retiming)

Рис. 11.9. Настройки стадии синтеза проекта

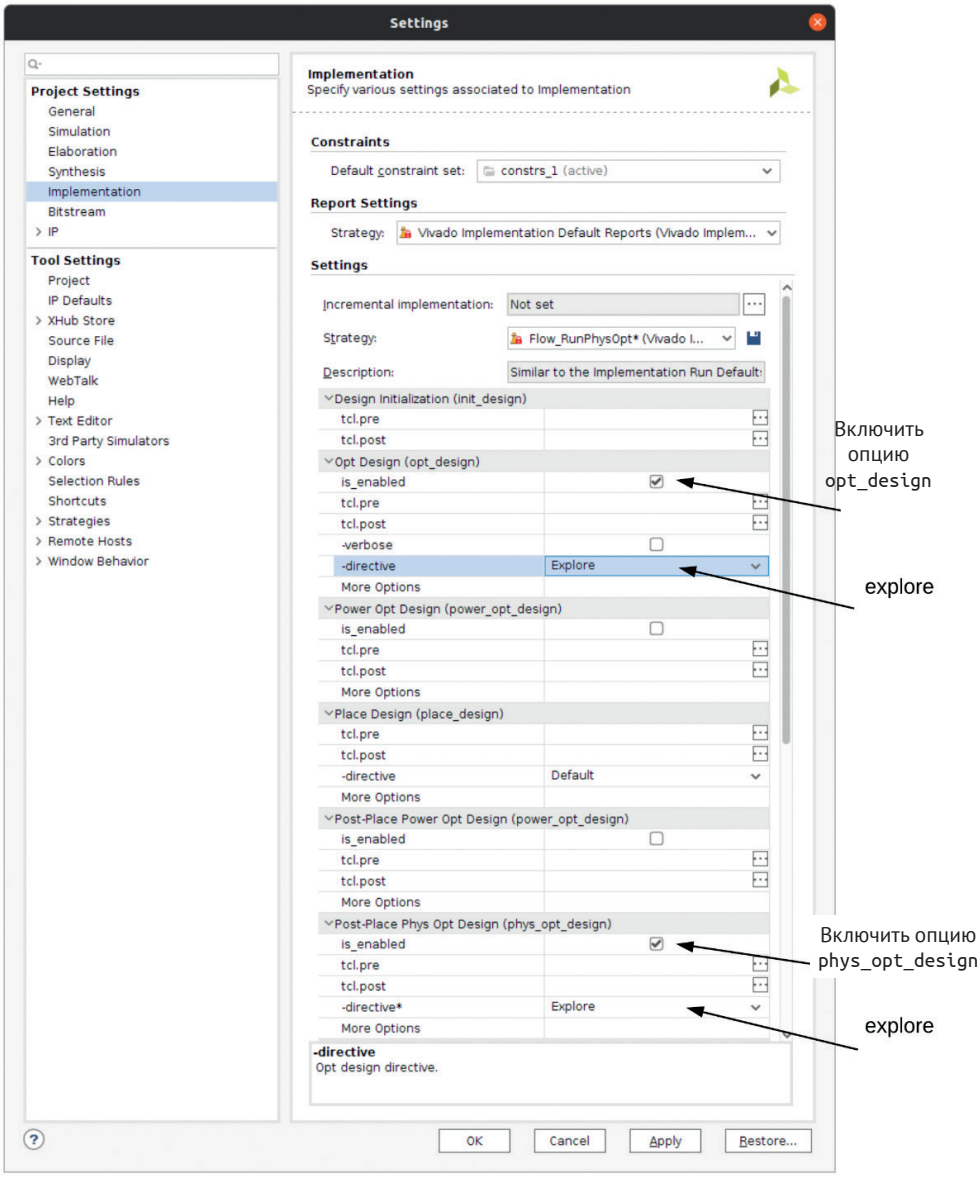


Рис. 11.10. Настройки стадии реализации проекта

Установив эти параметры реализации проекта, можно попытаться еще раз собрать проект в надежде удовлетворить временны́м требованиям. Нарушений временны́х параметров стало немного меньше.

Timing	Setup | Hold | Pulse Width
Worst Negative Slack (WNS):	-2.376 ns
Total Negative Slack (TNS):	-250.84 ns
Number of Failing Endpoints:	125
Total Number of Endpoints:	535
Implemented Timing Report	

Рис. 11.11. Отчет о временны́х параметрах схемы после синтеза с расширенными настройками

Этого явно недостаточно, чтобы существенно изменить результаты, но это позволило немного улучшить характеристики схемы. Теперь можно попробовать добавить конвейеризацию. Проект построен таким образом, что можно вставить конвейеризацию после математической операции. Операция `retiming` помещает дополнительные регистры между комбинационными блоками так, чтобы разделить на части самые длинные пути прохождения сигналов в комбинационной логике. Дополнительные регистры могут быть помещены в комбинационную логику, элементы DSP и BRAM. Попытка установить `PIPELINE = 1` на вкладке **General settings** (общие настройки) приведет к приблизительно следующему результату:

Timing	Setup | Hold | Pulse Width
Worst Negative Slack (WNS):	-1.139 ns
Total Negative Slack (TNS):	-101.029 ns
Number of Failing Endpoints:	97
Total Number of Endpoints:	631
Implemented Timing Report	

Рис. 11.12. Отчет о временны́х параметрах схемы после синтеза при PIPELINE=1

Эта настройка приводит к тому, что дополнительные регистры вставляются для разделения некоторых внешних временных путей, оставляя блок DSP нетронутым. В этом можно убедиться, посмотрев на схему в Vivado и найдя сгенерированные регистры.

Если установить опцию `PIPELINE = 2` в общих настройках и выполнить повторный запуск сборки проекта, то операция `retiming` даст положительный результат:

Timing	Setup | Hold | Pulse Width
Worst Negative Slack (WNS):	0.444 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	617
Implemented Timing Report	

Рис. 11.13. Отчет о временны́х параметрах схемы после синтеза при PIPELINE=2

Положительный запас времени (slack time) достигнут за счет того, что механизм `retiming` вставляет дополнительный набор регистров в умножитель. Концептуально это выглядит так, как показано на рис. 11.14.

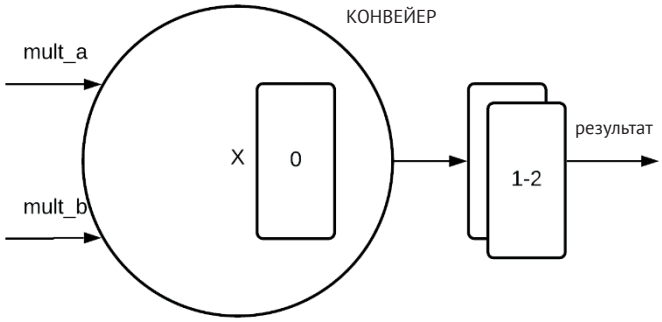


Рис. 11.14. Концептуальная схема умножителя PIPELINE=2

Таким образом, были показаны некоторые идеи, как можно решать проблемы удовлетворения требуемым временны́м параметрам схемы. Иногда необходимо немного вникнуть в то, как выглядит проект на уровне аппаратуры, и переработать код. Например, умножитель 32×64 можно разделить на два умножителя 32×32 и сумматор. Вы можете попытаться реализовать его в качестве упражнения.

Выводы

В этой главе были рассмотрены некоторые более продвинутые и менее используемые конструкции языка SystemVerilog. Был представлен такой элемент языка, как интерфейс, который позволяет лучше использовать инкапсуляцию и повторное использование кода. Были изучены некоторые более продвинутые циклы, структуры и метки.

Также были разобраны некоторые более продвинутые конструкции, используемые для верификации. Они полезны в больших и сложных проектах.

Еще были рассмотрены некоторые проблемы, которые могут возникать при разработке, и то, как их избежать. Было показано, как можно попытаться улучшить временны́е параметры схемы.

Изучение книги закончено, и теперь вы можете самостоятельно решать некоторые задачи. Как было упомянуто в начале, существует множество сообществ, таких как, например, Mister Project, где могут быть задействованы люди со знаниями FPGA. Есть также проекты, которые можно попытаться выполнить самостоятельно, чтобы получить работу. Что бы вы ни выбрали, хочется надеяться, это принесет вам удовольствие и пользу.

Вопросы

1. Интерфейсы полезны для:

   a) объединения сигналов, соответствующих друг другу;
   b) инкапсуляции функций, задач и утверждений, связанных с интерфейсом;
   c) изменения модуля, глубоко встроенного в другие модули;
   d) все вышеперечисленное.

2. Над структурой может быть выполнена операция присвоения с помощью:

   a) компонента;
   b) имени;
   c) интерфейса;
   d) (a) и (b).

3. Метки блоков позволяют легче сопоставлять начало и конец операторных скобок `begin...end`.

   a) Верно.
   b) Неверно.

4. Если требуется выйти из цикла, для этого можно использовать:

   a) `break` в любом цикле;
   b) `disable` на любой метке цикла;
   c) `break` на внешнем цикле или `disable` на любой метке цикла.

5. `Continue` можно использовать для пропуска оставшейся части цикла.

   a) Верно.
   b) Неверно.

6. Очереди полезны для:

   a) создания настраиваемого FIFO для использования при верификации;
   b) создания настраиваемого FIFO для использования при проектировании и верификации;
   c) ничего.

7. Что делает следующий фрагмент кода?

```
initial begin
  if (NUM_SEGMENTS != 4 || NUM_SEGMENTS != 8)
    $fatal("This design only supports 4 or 8 segments");
  end
end
```

   a) Вызывает fatal error (фатальную ошибку) при моделировании, гласящую, что проект может поддерживать только 4 или 8 сегментов.
   b) Вызывает fatal error (фатальную ошибку) при синтезе, гласящую, что проект может поддерживать только 4 или 8 сегментов.
   c) Все вышеперечисленное.

8. При проектировании следует остерегаться следующих вещей:

   a) случайного создания однобитных шин;
   b) несоответствия ширины шин;
   c) появления защелок;
   d) проблемы пересечения тактовых доменов;
   e) всего вышеперечисленного.

9. В данной главе было показано, что можно использовать операцию `retiming` для реализации умножителя 32×64. В этом разделе было упомянуто, что большой умножитель можно реализовать как два умножителя 32×32 и сумматор. Реализуйте такой умножитель на SystemVerilog. Дополнительный вопрос: что изменится в реализации умножителя, если используется умножение с учетом знака?

Дополнительное чтение

Для получения дополнительной информации о том, что было рассмотрено в этой главе, обратитесь к дополнительным материалам по следующей ссылке:

- http://staging.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-assertions-tutorial/.