Оглавление

Участники

Об авторе

Фрэнк Бруно – опытный инженер-проектировщик 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 FPGA в компании Cruise.

О рецензенте

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.

Предисловие

Готовьтесь повеселиться. Автор этой книги разрабатывает ASIC и FPGA[1] уже 30 лет, и каждый день приносят новые вызовы и волнения, поскольку позволяет подталкивать технологии к разработке новых приложений. За свою карьеру автор разработал ASIC, которые обеспечивали работу военных самолетов; графику, работающую на высококлассных рабочих станциях и обычных ПК; технологию для питания следующего поколения программно-определяемых радиосистем; а также обеспечивал космический интернет 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. Мы предполагаем, что вы ничего не знаете о цифровой логике, и начнем с представления базовых вентилей и их функций, а затем разработаем полноценные системы. Некоторые знания в области программирования или аппаратного обеспечения полезны, но не обязательны. Если вы сможете установить программу, подключить USB-кабель и следовать проектам, вы узнаете много нового.

Что включает в себя эта книга

Глава 1. Введение в архитектуры FPGA и Xilinx Vivado. Объясняет, что такое ASIC и FPGA и как установить Xilinx Vivado и создать небольшой проект.

Глава 2. Комбинационная логика. Рассматривает написание с нуля полного модуля SystemVerilog для выполнения некоторых базовых операций, чтобы продемонстрировать, как использовать комбинационную логику в собственных проектах. Мы также познакомимся с `testbench`[2] и узнаем, как написать один из них 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 матрица вентилей») – разновидность программируемых логических интегральных схем, ПЛИС. Существуют и другие разновидности программируемых схем, однако FPGA, как самая распространенная и универсальная, фактически стала 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 см. главу 1. – *Прим. ред.*

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

Глава 3. Подсчет нажатий на кнопку. Основывается на комбинационной логике из предыдущей главы, добавляя последовательностные элементы – встроенную память данных. Мы узнаем о возможностях Artix-7 и других устройств FPGA для хранения данных и разработаем простой проект для подсчета кликов на кнопку. Мы также рассмотрим использование тактовых генераторов и синхронизации, того немногого, что может полностью разрушить проект, если сделано неправильно.

Глава 4. Давайте построим калькулятор. Рассматривает, как при создании более сложных конструкций неизбежно возникает необходимость отслеживать состояние устройства. В этой главе мы узнаем о конечных автоматах и воспользуемся классическим инженерным устройством – контроллером светофора. А также улучшим калькулятор и покажем, как можно спроектировать делитель, используя конструкции на основе состояний.

Глава 5. Ресурсы FPGA, и как их использовать. Делает шаг назад после быстрого погружения в проектирование FPGA и рассматривает некоторые ресурсы FPGA более подробно. Чтобы использовать эти ресурсы, мы представим некоторые устройства с платы разработчика: микрофон PDM и датчик температуры I2C, подключенные к FPGA, и используем их в проектах.

Глава 6. Математика, параллелизм и конвейерное проектирование. Более подробно рассматривает числа с фиксированной и плавающей точкой. Мы также рассмотрим конвейерное проектирование и параллелизм для повышения производительности.

Глава 7. Введение в AXI[3]. Рассказывает о том, как компания Xilinx приняла стандарт AXI для сопряжения своих IP[4] и разработала инструмент IP integrator для простого графического соединения IP. В этой главе мы рассмотрим AXI, взяв датчик температуры и используя IP integrator для интеграции проекта.

Глава 8. Много данных? MIG и DDR2. Рассматривает, как Artix-7 обеспечивает хороший объем памяти, но что произойдет, если нам понадобится доступ к мегабайтам или гигабайтам оперативной памяти? На плате есть память DDR2, и в преддверии реализации контроллера дисплея мы рассмотрим Xilinx Memory Interface Generator для реализации интерфейса DDR2 и протестируем его в симуляции и на плате.

Глава 9. Лучший способ отображения – VGA. Рассматривает реализацию VGA и простой способ отображения текста. Мы использовали светодиоды и дисплей с семью сегментами для вывода информации в проектах. Но это ограничивает нас в том, что можно отображать. Например, мы не можем отображать захваченные аудиоданные и текст.

Глава 10. Свести все воедино. Посвящена добавлениям входных устройств. Мы уже рассмотрели вывод с помощью VGA, но добавим входы, подключив-

---

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

[4] 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.*

шись к клавиатуре с помощью PS/2. Мы возьмем датчик температуры и микрофон PDM и создадим проект, использующий VGA для отображения этих данных.

Глава 11. Темы повышенной сложности. Рассматривает некоторые концепции SystemVerilog, которые были пропущены в других главах, но которые могут оказаться полезными. Мы рассмотрим более продвинутые методы тестирования и, наконец, разберем отдельные проблемы и способы их предотвращения.

Как получить максимальную пользу от этой книги

Эта книга не предполагает наличия знаний о FPGA, логическом проектировании или программировании. Вам понадобится компьютер с операционной системой Windows или Linux. В первой главе вы получите 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, мы советуем вам набирать код самостоятельно или получить доступ к коду через репозиторий GitHub (ссылка доступна в следующем разделе). Это поможет избежать возможных ошибок, связанных с копированием и вставкой кода.

Скачать файлы примеров кода

Можете загрузить файлы кода примеров для этой книги из своей учетной записи на сайте www.packt.com. Если вы приобрели эту книгу в другом месте, можете посетить сайт поддержки www.packtpub.com/support и зарегистрироваться, чтобы получить файлы по электронной почте непосредственно для вас.

Можете загрузить файлы кода, выполнив следующие действия.

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

После загрузки файла убедитесь, что вы разархивировали или распаковали папку 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 репозитории GitHub.

У нас также есть другие наборы кодов из нашего богатого каталога книг и видео, доступные по адресу https://github.com/PacktPublishing/. Посмотрите их!

Скачать цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями скриншотов/диаграмм, используемых в этой книге. Вы можете скачать его здесь:

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 шрифтом: «файл `logic_ex.xpr`».

Блок кода задается следующим образом:

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

Когда мы хотим обратить ваше 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 вы видите на экране. Например, слова в меню или диалоговых окнах отображаются в тексте следующим образом, например: «В проекте блока кликните правой кнопкой мыши и выберите **Add Module**».

> **Подсказки и важные замечания**
>
> Выглядят вот так.

Свяжитесь с нами

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 нам по адресу customercare@packtpub.com.

Ошибки: хотя мы приложили все усилия, чтобы обеспечить точность нашего материала, ошибки все же случаются. Если вы обнаружили ошибку в этой книге, мы будем благодарны, если вы сообщите нам об этом. Пожалуйста, зайдите на сайт www.packtpub.com/support/errata, выберите вашу книгу, нажмите на ссылку Errata Submission Form и введите информацию.

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 написании книги или в участии 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.

Раздел 1

Введение в FPGA и архитектуры Xilinx

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.

Эта часть книги включает в себя следующие главы:

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

Глава 1

Введение в архитектуры FPGA и Xilinx Vivado

В данной главе мы рассмотрим **Field Programmable Gate Arrays (FPGA)** (программируемые логические интегральные схемы, ПЛИС) и технологию, лежащую в их основе. Эта технология позволяет таким компаниям, как Xilinx, производить перепрограммируемые микросхемы из процесса **Application Specific Integrated Circuit (ASIC).** Затем мы узнаем, как использовать FPGA для 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. Не может быть лучшего времени, чем сейчас, 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 выполнить несколько примеров проектов, которые познакомят вас с разработкой на FPGA, а кульминацией станет проект, способный управлять монитором VGA.

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

Основные темы, которые мы рассмотрим в этой главе:

- что такое ASIC;
- как компания создает 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 плоды своей работы 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 проявите амбициозность 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 в этой главе можно найти в репозитарии 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, сотового телефона, цифровых часов, практически всего, чем вы пользуетесь ежедневно. Это также фундаментальный строительный блок, на основе которого создается FPGA, которую мы будем рассматривать. 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 сотовый 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.

Мы можем 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 утверждает следующее 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/:

«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 миллион 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 мы увидим 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 мы увидим в наших проектах, где иногда будем намеренно

совершать ошибки, стоимость их исправления занимает всего несколько минут времени, потраченных в основном на то, чтобы понять, почему ошибки произошли:

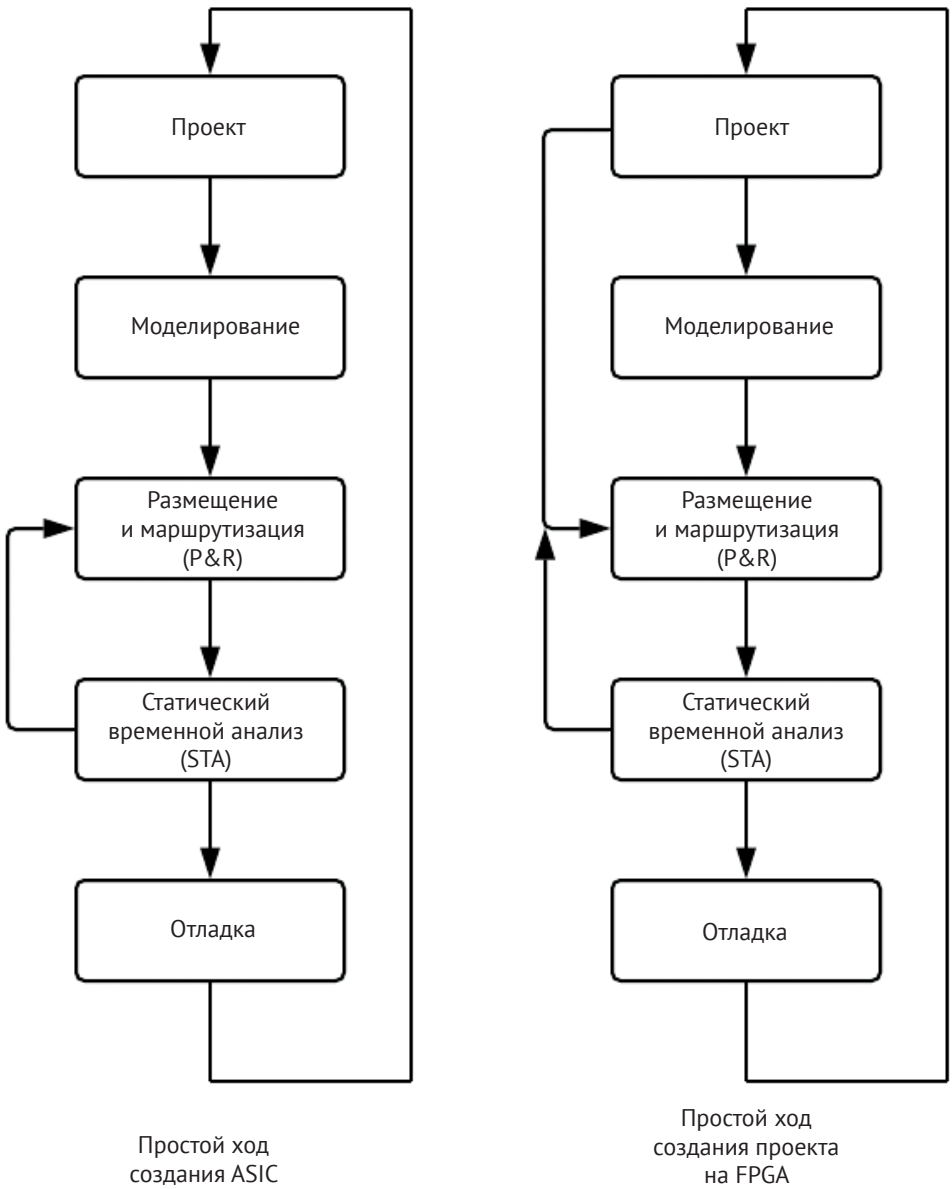


Рис. 1.1. Процессы разработки ASIC и FPGA

Ход создания ASIC и FPGA по сути одинаков. Процесс разработки ASIC, как правило, более линейный, поскольку у вас есть 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 этапам стоит только затраченного времени. В той же ситуации любые изменения в реализованном проекте ASIC требуют определенного количества новых наборов масок, стоимость которых может исчисляться миллионами долларов.

Мы кратко рассмотрели, что такое ASIC и для каких целей могут выбираться ASIC и FPGA. Теперь давайте рассмотрим, как создается FPGA с использованием процесса 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 бинарной логике, другими словами, на включенных или выключенных транзисторах. Эти включенные или выключенные транзисторы можно представить как булевы значения «истина» и «ложь».

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

Базовые логические элементы

Существует четыре основных типа логических элементов. Обычно мы приводим таблицы истинности для этих элементов, чтобы разобраться с их функциональностью. Таблица истинности показывает, что будет на выходе для каждого набора входов схемы. Обратитесь к следующему примеру 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 мы можем использовать оператор присваивания `assign`, чтобы присвоить значение, находящееся в правой части от знака равенства, его левой части. Он используется следующим образом:

```
assign out = in;
```

in может быть другим сигналом, функцией или операцией над несколькими сигналами. out может быть любым допустимым сигналом, объявленным до оператора assign.

Комментарии

SystemVerilog предоставляет два способа комментирования. Первый – это использование двойной косой черты – //. Этот тип комментария действует до конца строки, на которой он расположен. Второй тип – блочный комментарий. Оба варианта показаны ниже:

```
// Everything here is a comment.
/* I can span
Multiple
Lines */
```

Оператор if

SystemVerilog предоставляет возможность проверки условий с помощью оператора if. Основной синтаксис выглядит следующим образом:

```
if (условие) событие
```

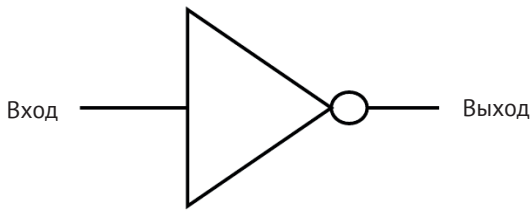
Более подробно оператор if мы рассмотрим в *Главе 2. Комбинационная логика*.

Логическое НЕ (!)

На выходе логического элемента НЕ (NOT) формируется сигнал, обратный сигналу на входе. Функция НЕ в SystemVerilog может быть записана следующим образом:

```
assign out = !in; // логический оператор
```

Соответствующая таблица элемента НЕ истинности выглядит следующим образом:



Графическое представление

Вход	Выход
0	1
1	0

Таблица истинности

Рис. 1.2. Представление логического элемента НЕ (NOT)

Операция НЕ (NOT) – один из самых распространенных операторов, которые мы будем использовать:

```
if (!empty) ...
```

Часто необходимо проверить сигнал перед выполнением операции. Например, если мы используем память, устроенную по принципу **First in First Out (FIFO, «первым пришел – первым ушел»)** для сглаживания нерегулярных

выбросов данных[5] или для пересечения тактовых доменов[6], нам нужно проверить, есть ли данные, прежде чем принимать их из очереди их 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 мы более подробно рассмотрим, как спроектировать FIFO, а также как его использовать.

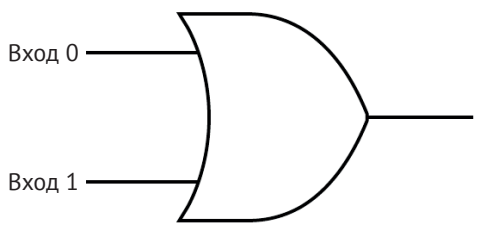
Логическое И (&), побитовое И (&)

Часто нам будет необходимо проверить, являются ли активными одно или несколько условий одновременно. Для этого мы будем использовать логический элемент `И (AND)`.

Функция на языке SystemVerilog может быть записана следующим образом:

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

Соответствующая таблица истинности выглядит следующим образом:



Графическое представление

Вход 0	Вход 1	Выход
0	0	0
0	1	1
1	0	1
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 вы хотите убедиться, что в исходном FIFO есть данные (там не пусто) и что место назначения не переполнено. Мы можем сделать это, 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.

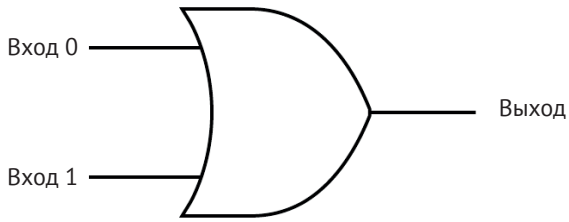
Функция в SystemVerilog может быть записана следующим образом:

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

Соответствующая таблица истинности выглядит следующим образом:

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

[6] 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)

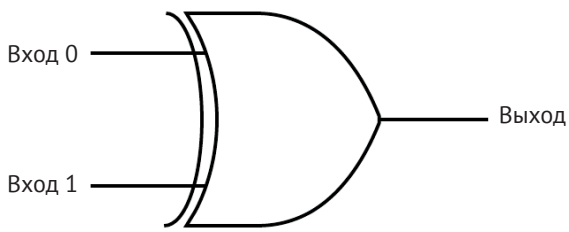
Далее рассмотрим функцию исключающего ИЛИ (`exclusive OR`).

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 следующим образом:



Графическое представление

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

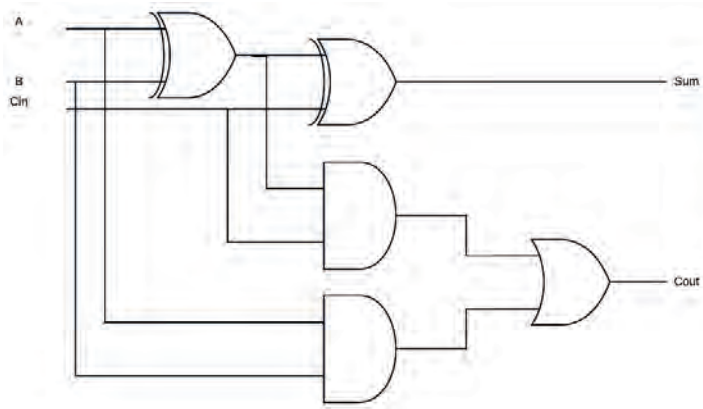
Таблица истинности

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 мы рассмотрим, как строится сумматор с использованием представленных логических элементов.

Более сложные операции

В предыдущих разделах мы рассмотрели основные компоненты, из которых состоит любая цифровая схема. Здесь мы 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: сумма и перенос. Давайте посмотрим на таблицу истинности:



Графическое представление

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

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 мы используем побитовые операторы `И (AND, &)` и `ИЛИ (OR, |)`, поскольку оперируем битами. Из этого простого, но важного примера видно, 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, вам не нужно погружаться на этот уровень детализации, если только вы действительно этого не хотите.

Знакомство с FPGA

Массив логических элементов (вентилей) в терминах ASIC – это множество логических элементов с некоторым количеством конфигурируемых 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 вы платите 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 мы рассмотрим функции из предыдущего раздела на примере сумматора, то увидим одну общую черту: все они могут быть получены с помощью таблицы истинности, которая становится ключевой при разработке FPGA. Мы можем рассматривать эти таблицы истинности как представления функций в **постоянном запоминающем устройстве (Read Only Memory,**

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

Давайте разберем пример основных логических функций. Мы можем воспроизвести любую из них с помощью **LUT (Lookup Table, таблицы поиска)**[7] с двумя входами, которая может выглядеть следующим образом:

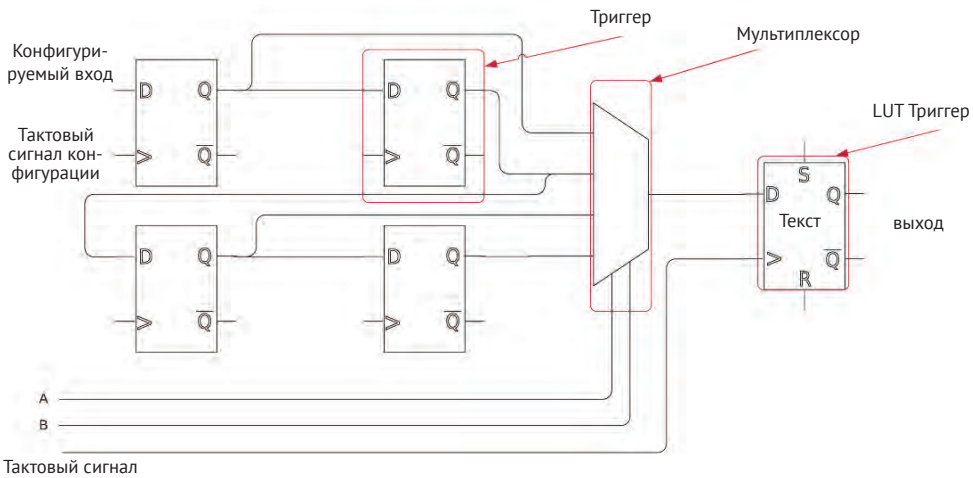


Рис. 1.7. Пример LUT с двумя входами

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. Добавив триггер, сформируется окончательная структура LUT.

Преимущество простоты структуры заключается в возможности многократного повторения этой конструкции. В случае современных FPGA они строятся из множества рядов логических элементов, подобных приведенному, что позволяет спроектировать, 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.

[7] 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 мы считаем 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, на который мы ориентируемся.

В основе FPGA лежит программируемая матрица. Матрица состоит из LUT с соответствующими триггерами, составляющими отдельные slice[8], из которых состоят CLB. Все эти блоки соединены между собой посредством разветвленной сети каналов маршрутизации, что позволяет создавать практически безграничные конфигурации. FPGA также содержат множество других ресурсов, которые мы будем изучать 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 мы будем рассматривать 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 ваших проектов. Компоненты FPGA, которые мы здесь представим, являются общими для устройств Spartan (младшего класса), Kintex (среднего класса) и Virtex (старшего класса) 7-й серии.

Комбинационные логические блоки

ASIC состоят из логических элементах, 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 вы будете использовать те же уравнения булевой логики, что и в ASIC. Мы будем использовать очень похожую схему. 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 этими элементами CLB. Чтобы полностью изучить детали 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.

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

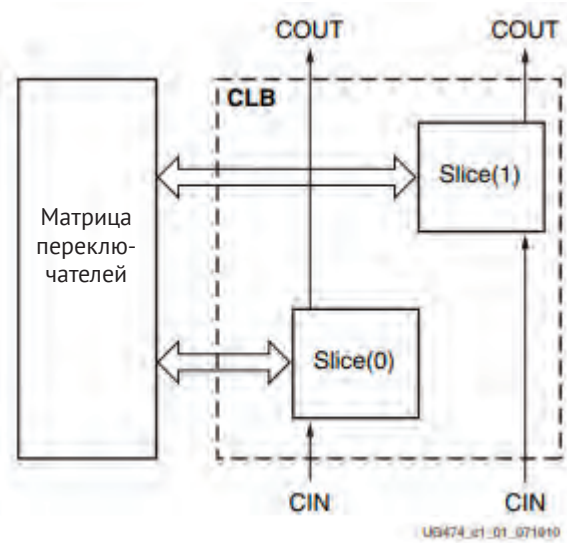


Рис. 1.8. Рисунок 1-1 из «Руководства пользователя CLB FPGA Xilinx UG474 7-й серии» (используется с разрешения)

`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 следующей таблице показано их распределение для двух рассматриваемых в этой книге плат для разработки:

Плата	Устройство	Секции	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 секции или LUT, и принудительно задействовать их функциональные возможности, это выходит за рамки данной книги, а данный метод не имеет широкого применения. Мы будем ориентироваться на использование CLB через синтез в Vivado разработанного нами HDL.

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 платах для разработки:

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 порт 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. Мы рассмотрим FIFO в одной из последующих 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, мы не будем подробно останавливаться на этом в данной книге.

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 вы найдете гораздо больше подробностей.

Устройства ввода/вывода (I/Os)

В основном мы ограничимся рассмотрением устройств ввода/вывода, поддерживаемых двумя целевыми платами для разработки. В целом устройства 7-й серии поддерживают множество интерфейсов от 3.3 В CMOS/TTL до LVDS[9] и интерфейсов памяти. Используемые нами платы будут определять типы устройств ввода/вывода в проектных файлах. Для получения дополнительной информации обо всех поддерживаемых типах можно обратиться к «Руководству пользователя UG471 7 Series FPGA SelectIO resources».

DSP48E1

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

[9] 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.*

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

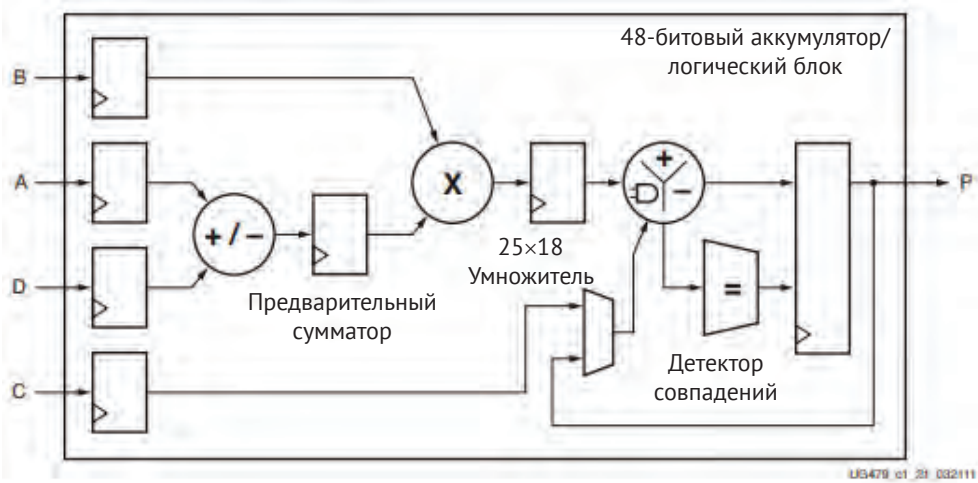


Рис. 1.9. Рис. 1-1 из «Руководства пользователя Xilinx UG479 серии 7 DSP48E1» (используется с разрешения)

Арифметические операции являются одними из самых дорогих в FPGA. В ASIC самой громоздкой и медленной операцией 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 ядра. 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. Мы подробнее рассмотрим блок DSP в последующих главах. «Руководство пользователя UG479 7 Series DSP48E1» (https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf) является хорошим справочником, если вы хотите углубиться в детали.

Архитектура ASMBL

Устройства 7-й серии – это четвертое поколение, в котором 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, мы видим, 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 мы будем использовать для будущих проектов:

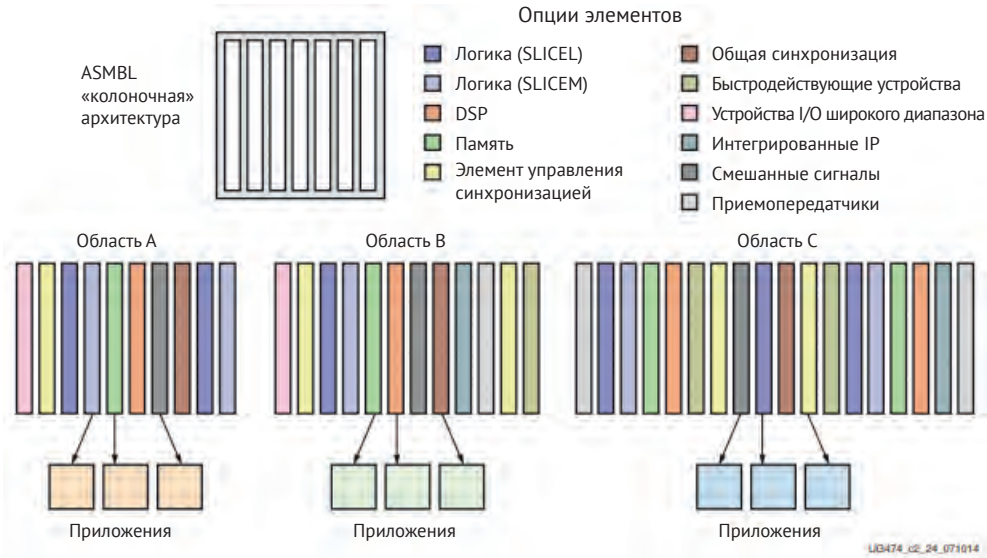


Рис. 1.10. Рисунок 2-1 из «Руководства пользователя CLB FPGA Xilinx UG474 серии 7» (использовано с разрешения)

Теперь, когда мы рассмотрели, из чего состоит Artix-7 и другие элементы 7-й серии, нужно установить инструменты Xilinx, чтобы мы могли приступить к первому проекту.

Знакомство с набором инструментов Vivado и отладочными платами

В этом разделе мы изучим отладочные платы, рекомендуемые для проектов в данной книге, рассмотрим очень простой проект с использованием Vivado для знакомства с инструментами и покажем, как программировать плату и демонстрировать функциональность FPGA.

Оценочные 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 – эта плата рекомендуется для данной книги. На ней есть все устройства, которые мы будем рассматривать в ходе работы далее.

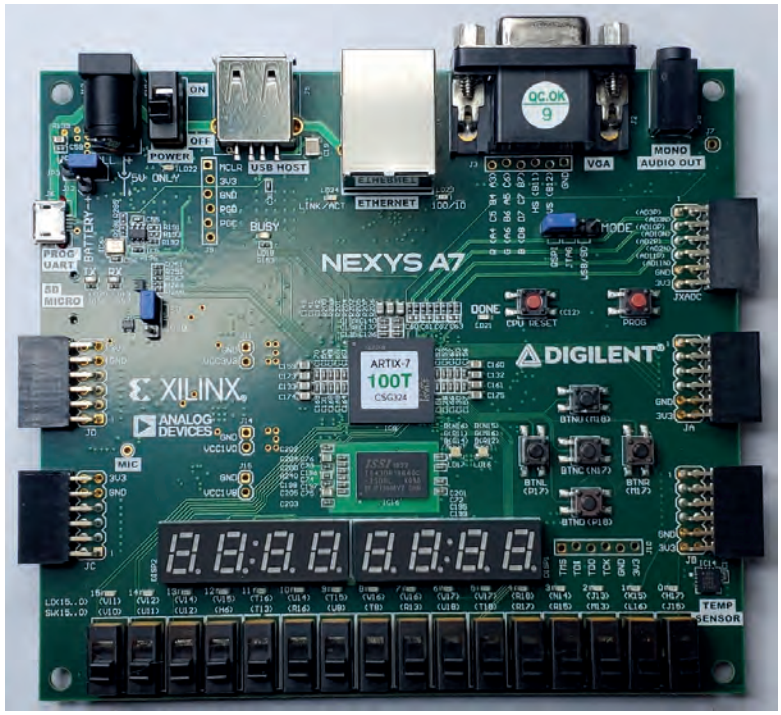


Рис. 1.11. Плата Digilent Nexys A7

Характеристики платы следующие:

- Artix-7 XC7A100T или 50T;
- работа на частоте 450+ МГц;
- 128 МБ DDR2;
- последовательная флеш-память;
- встроенный USB UART для загрузки изображений и отладки ChipScope;
- устройство чтения карт памяти MicroSD;
- 10/100 Ethernet PHY;
- PWM аудиовыход/вход для микрофона;
- датчик температуры;
- 3-осевой акселерометр;
- 16 переключателей;
- 16 светодиодов;
- 5 кнопок;
- два трехцветных светодиода;
- два 4-значных 7-сегментных дисплея;
- поддержка устройств USB;
- пять PMOD[10] (один XADC).

[10] 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 отладку микросхем с помощью ChipScope[11], а дополнительная оперативная память позволит сохранять более широкие шины или увеличить время хранения. Мы обсудим ChipScope в следующей главе.

Basys 3

Альтернативной оценочной платой является Basys 3.

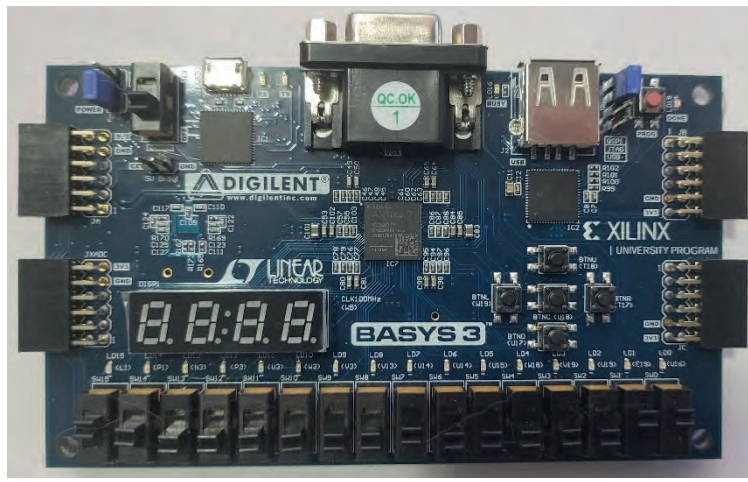


Рис. 1.12. Плата Digilent Basys 3

Эта плата имеет те же кнопки, светодиоды и переключатели, но только вдвое меньше 7-сегментных дисплеев. Мы будем разрабатывать код, который может 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 мы расскажем в следующей главе. На ней также отсутствуют датчик температуры, микрофон и аудио, которые мы будем рассматривать во время работы с последовательными интерфейсами. Но для пре-

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

одоления этих ограничений можно приобрести дополнительные платы PMOD, обладающие этой функциональностью.

Характеристики платы следующие:

- 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 3-цветных светодиода;
- одиночный 4-разрядный семисегментный дисплей;
- поддержка устройств USB;
- четыре PMOD (один двойного назначения, поддерживающий XADC).

Теперь давайте рассмотрим устройство платы Basys 3 :

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 DDR 2, акселерометры и интерфейс для аудио, которые будут рассмотрены в последующих главах. Для всего, кроме DDR2, имеются PMOD. Использовать Nexys A7 вместо Basys предпочтительней.

Мы только что рассмотрели платы, которые мы планируем использовать для этой книги. Теперь нужно ознакомиться с инструментом Xilinx Vivado, который мы будем использовать 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 мы будем использовать для сборки, тестирования, загрузки и отладки проектов. Он может быть запущен как инструмент командной строки или в режиме создания проекта с помощью

графического интерфейса пользователя (GUI). Для наших целей мы будем использовать режим проекта 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 вы сможете 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 мы будем использовать версию 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, вы можете установить либо Vitis, либо Vivado. Мы не будем 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 устройств вам понадобится только 7-я серия.
8. Выберите место для установки или используйте вариант по умолчанию.

Установка может занять некоторое время.

Структура каталогов

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

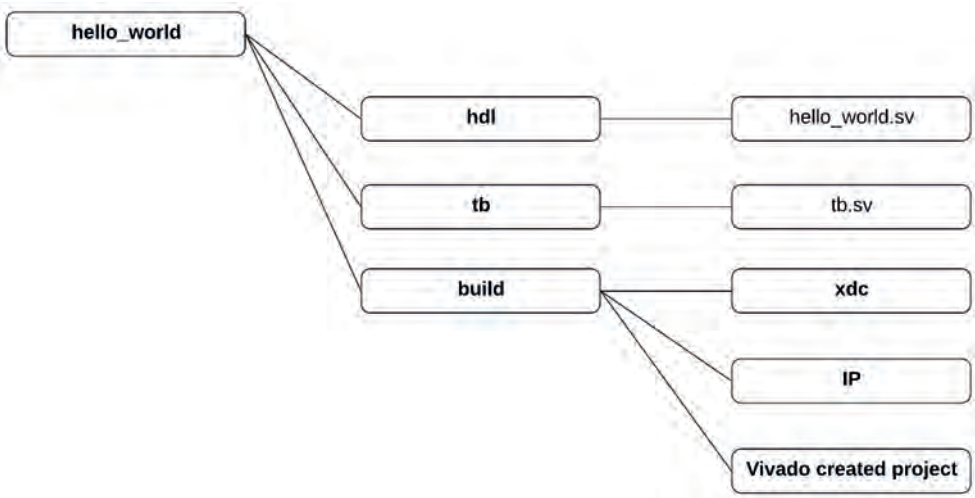


Рис. 1.13. Структура каталогов

Элементы, выделенные жирным шрифтом, являются каталогами. Для нашего первого примера проекта у нас не так много кода. В итоге мы создадим только три файла: исходный код HDL, тестовый файл `testbench` и файл ограничений.

Внутри каталога `hdl` мы создадим простой проект `logic_ex.sv` для запуска в Vivado[12]:

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 ns/10 ps были довольно распространенным стандартом много лет назад, и для того, что мы будем делать, это как раз подойдет. Если вы будете использовать

[12] В коде дальнейших 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.*

высокоскоростные приемопередатчики, то вам потребуются еще меньшие периоды времени, такие как 1 ps/1 fs.

> **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

Мы определяем список портов с одним входом, SW, представляющим собой 2-битное значение, которое мы подключим 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));
```

Здесь мы объявляем модуль верхнего уровня под названием `tb`. Обратите внимание, что модуль `testbench` верхнего уровня не должен иметь никаких портов. Мы также объявляем два логических типа, которые будем подключать к модулю `hello world`.

Здесь мы создаем компонент `logic_ex` и его экземпляр `u_logic_ex`. Существует несколько способов подключения портов. В некомментированном примере мы используем `.*`, что соединит все порты с тем же именем, что и определенный сигнал в шаблонном модуле.

Во втором примере (закомментированном) используется `.<name>` (имя) порта, который вы хотите 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 мы тестируем. 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`, которое устанавливает все биты равными z[13]. 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, и это важный момент. 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. Мы обсудим блокирующие и неблокирующие присвоения, когда будем обсуждать последовательные схемы.

[13] Под состоянием z здесь имеется в виду высокимпедансное 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 вы используете один и тот же знак в нескольких циклах `for`.

Внутри цикла `for` мы выводим текущую настройку переключателей с помощью системной функции `$display`. Поскольку мы хотим вывести только 2 бита, которые инкрементируем без ведущих нулей, то указываем `2%b`. Затем устанавливаем значение `SW` в младшие два бита `i`. Хотя в этом нет необходимости, мы добавляем задержку в 100 ns, используя выражение `#100`.

Мы также используем системную функцию `$stop`, при выполнении которой будет завершено моделирование.

> **Важное замечание**
>
> Мы знаем, что задержка происходит в наносекундах из-за периода времени, который мы определили в `testbench`.

Мы также объявляем блок проверки. В любом хорошем `testbench` блок проверки должен быть самопроверяемым. Это означает, что в конце теста мы должны иметь возможность вывести 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 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 мы хотим, чтобы этот блок реагировал на события в нашем проекте. Для этого используем блок `always`,

чувствительный только к изменениям на входах SW и выходах LED проекта. Это простой случай, когда мы сопоставляем каждый бит LED соответствующим битам SW проходящих через соответствующие логические элементы. Для этого мы используем оператор !==, который является оператором сравнения на не равенство, но при этом учитывает неопределенные состояния (x) в тех случаях, когда в проекте есть ошибки. Более сложные testbench мы рассмотрим в последующих главах.

Также используем 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

Давайте загрузим проект в Vivado.

1. В Windows найдите, где установлен Vivado, и дважды щелкните на иконке Vivado. В Linux процедура выглядит следующим образом[14]:

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

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

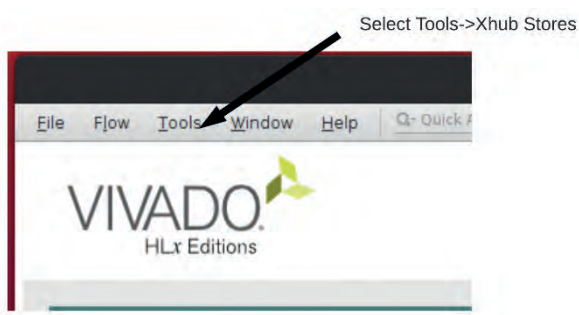


Рис. 1.14. Xhub Stores

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

4. Исталируйте конфигурационный файл платы, используемой в тестовом проекте для тестовых проектов.

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

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

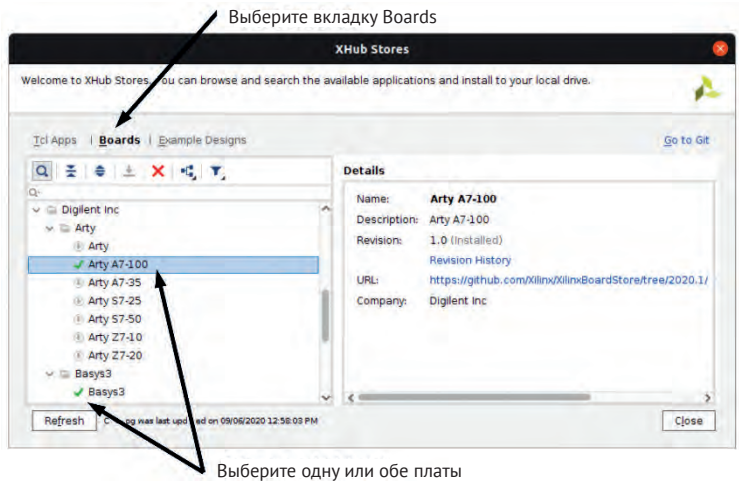


Рис. 1.15. Добавление плат Digilent

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:

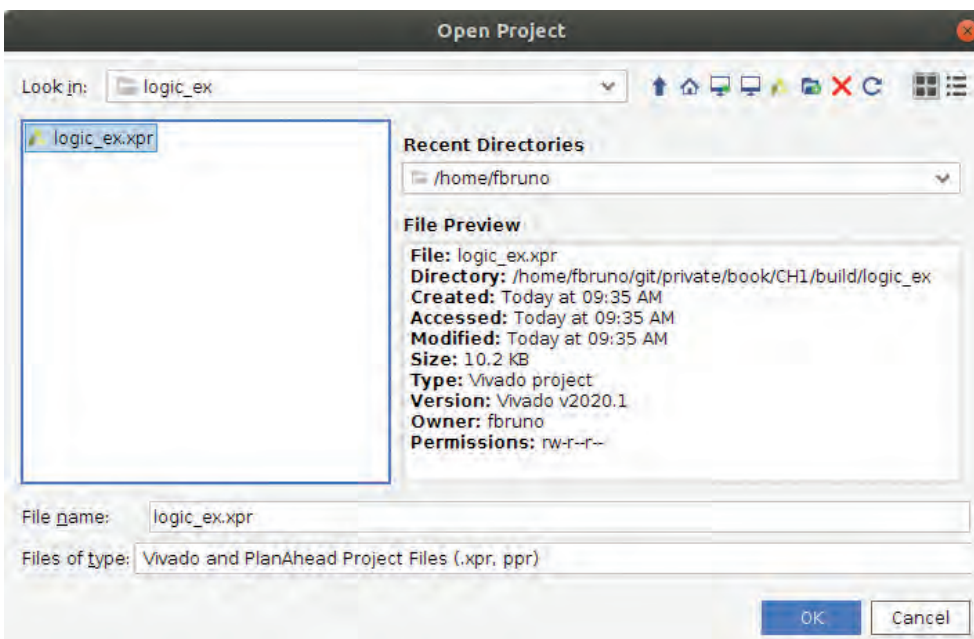


Рис. 1.16. Откройте окно проекта

После открытия вы увидите следующее:

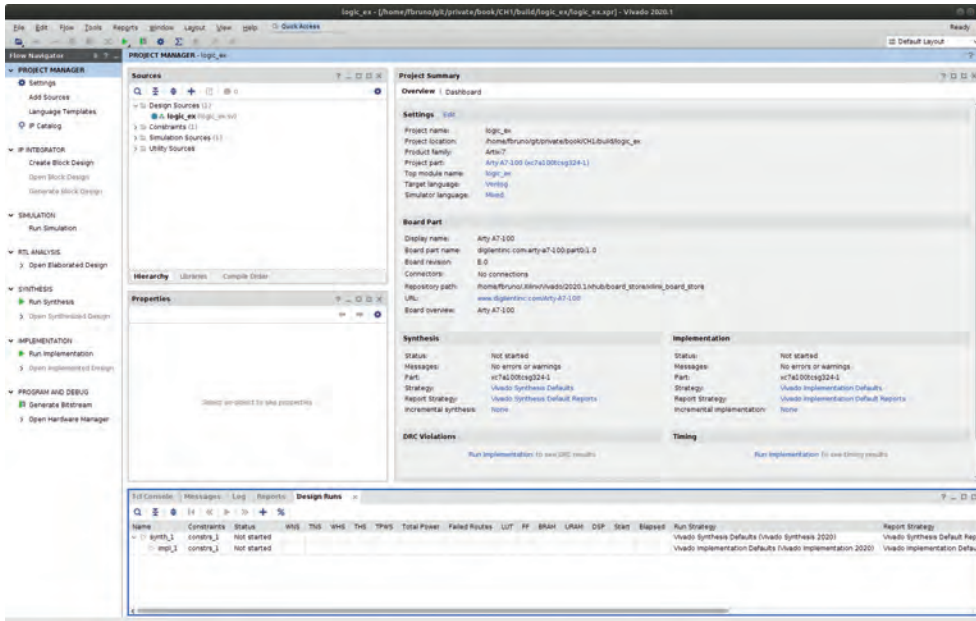


Рис. 1.17. Главный экран Vivado для проекта logic_ex

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

> **Важное замечание**
>
> Чтобы 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.

Давайте 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. Вы можете дважды щелкнуть на любом 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.

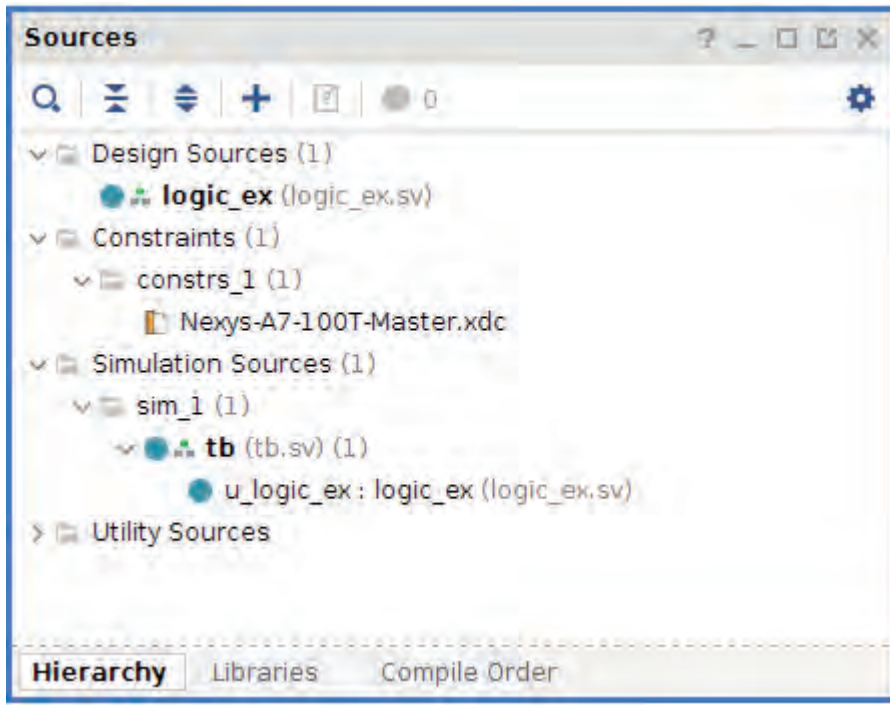


Рис. 1.18. Источники проекта

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

Запуск симуляции

Давайте сначала запустим симулятор Vivado, чтобы проверить корректность проекта.

Для этого нажмите **Run Simulation | Run Behavioral Simulation** в разделе **PROJECT MANAGER.** Вы увидите, 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. Поведенческая симуляция выполняется относительно быстро и точно отображает функции вашего проекта, если код написан правильно. Не рекомендуется запускать симуляцию после синтеза или имплементации, если только вы не отлаживаете сбои платы и не нуждаетесь в точном тестировании реализованной версии проекта, поскольку в таком случае симуляция значительно замедлится.

Запуск поведенческой симуляции является первым шагом в общей последовательности процесса разработки проекта. Окно симуляции занимает весь главный экран Vivado:

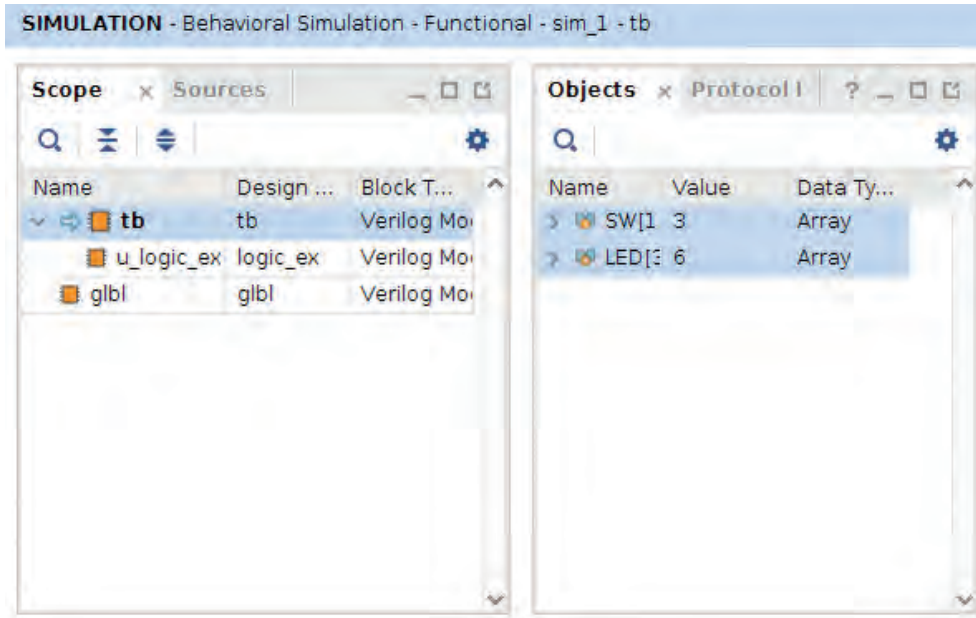


Рис. 1.19. Обзор симуляции

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

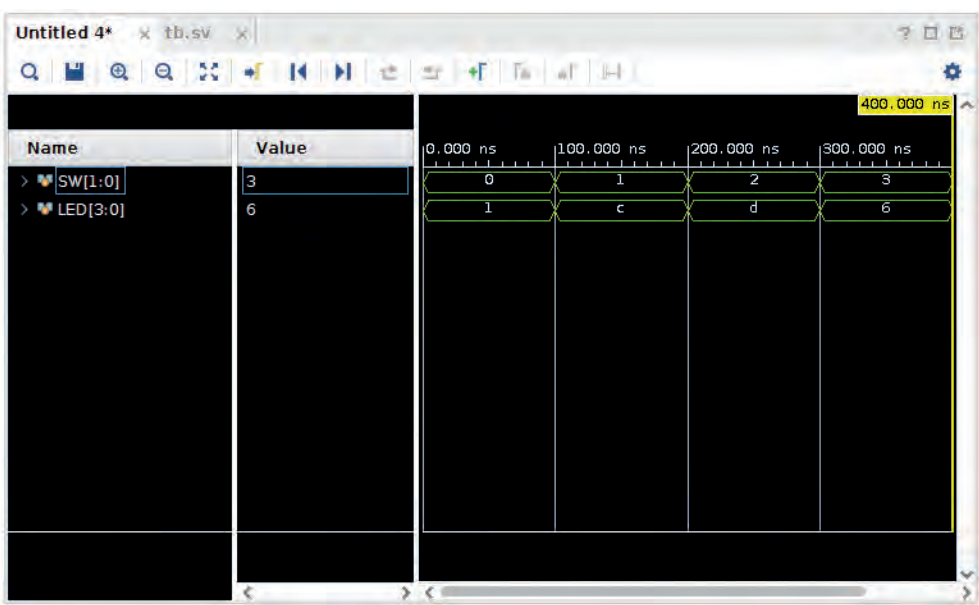


Рис. 1.20. Вейвформы

Вейвформы позволяют проанализировать сигналы в проекте и их поведение в процессе симуляции. Это наиболее часто используемая функция симулятора при отладке проекта. На рисунке сигнал SW инкрементируется с помощью цикла `for` в `testbench`. Соответственно, мы видим и 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 обратите внимание, что каждое изменение сигналов соответствует временному сдвигу на 100 ns. Это связано с командой установки задержки модельного времени `#100`, которую мы используем для движения по шкале времени, и с настройкой периода времени (в timescale)..

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

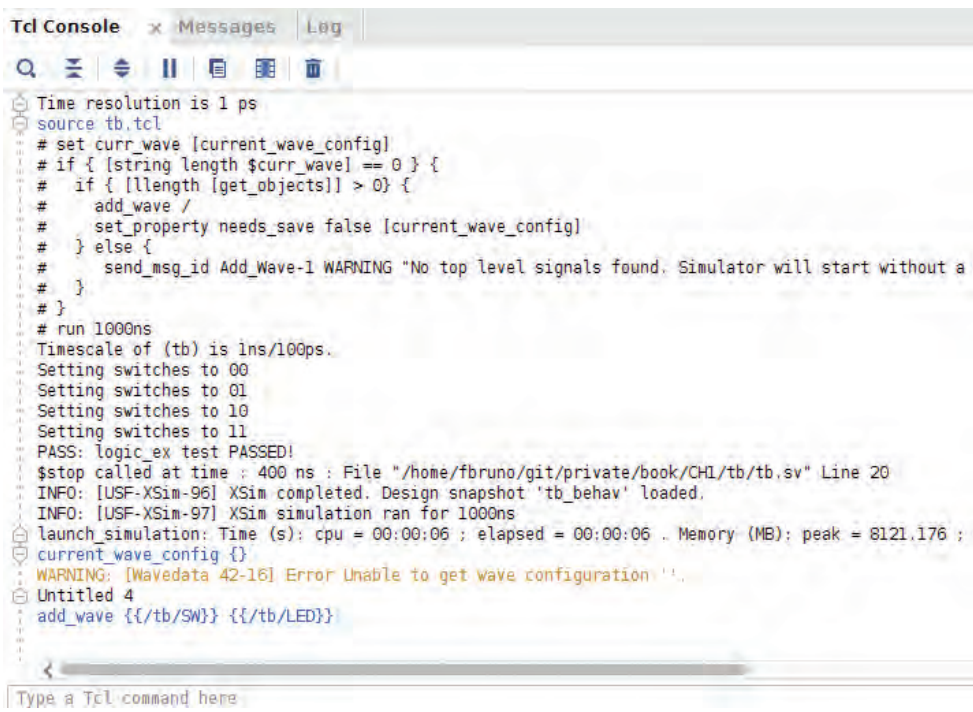


Рис. 1.21. Консоль TCL

Консоль `TCL`[15] отображает все сообщения из функций `$display` или конструкций `assertion`. В данном случае мы видим вывод функции `$printtimescale(tb)` как `1 ns/ 100 ps`. Мы также видим значения, на которые установлены переключатели, и можем видеть те же значения во временных диаграммах. Наконец, мы видим **`PASS: logic_ex test PASSED!`**, что является результатом тестирования. Поэкспериментируйте с `testbench`. Меняйте операторы или переставляйте их, чтобы убедиться, что тест провалится, если вы это сделаете. Такие упражнения дадут вам уверенность в том, что `testbench` работает правильно.

[15] TCL (Tool Command Language) – встроенный интерпретируемый язык программирования. В консоль tcl выводятся служебные сообщения и 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 работе. Здесь простой случай, поэтому на самом деле сейчас это невозможно, но убедитесь, что вы тестируете непредвиденные ситуации, чтобы удостовериться, что ваш 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 у нас есть уверенность в том, что проект работает так, как задумано, пришло время собрать его и проверить на плате.

Сначала давайте посмотрим на 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 конфигурационные напряжения:

```
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 мы устанавливаем `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. Мы используем `TCL`-команду `get_port`, 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:** размещается проект на ресурсах чипа FPGA, 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 характеристик проекта, или если вы проектируете под специализированную плату и вам нужно выполнить планирование выводов. В нашем случае мы можем нажать непосредственно на **Generate Bitstream** и позволить программе выполнить все шаги автоматически. Разрешите системе использовать настройки по умолчанию. После завершения откройте результаты:

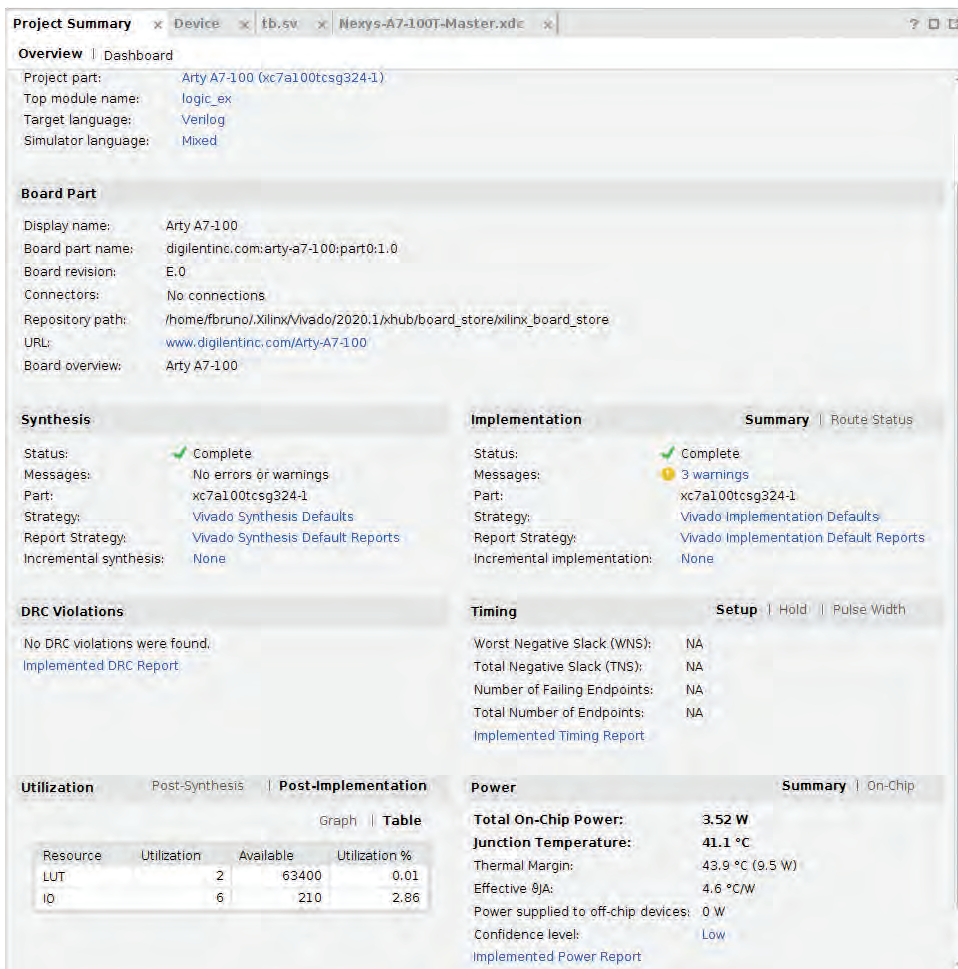


Рис. 1.22. Результаты проекта

Здесь можно увидеть результаты трассировки. Задействовано 2 LUT и 6 устройств ввода/вывода (SW + LED). Синхронизация отсутствуют, 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 используется устройство:

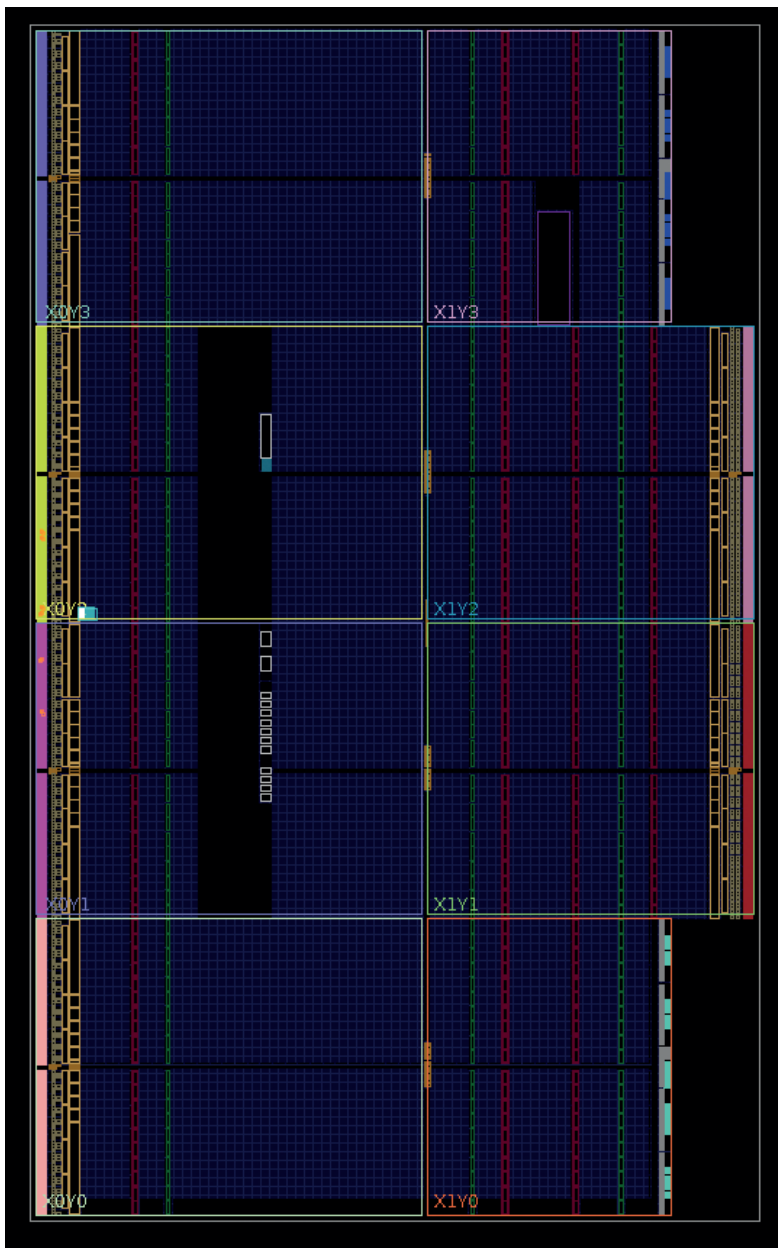


Рис. 1.23. Визуализация устройства

На рисунке ресурсов платы можно увидеть маленькую белую точку в середине левой стороны. Она обозначает место размещения используемых LUT.

Программирование платы

Вы добрались до конца главы, и теперь пришло время увидеть плату в действии.

1. Убедитесь, что она подключена к сети и включена.
2. Теперь кликните на **Open hardware manager**, последнюю опцию в **Flow Navigator**. В главном окне откроется вид менеджера оборудования.
3. Кликните на **Open target | Autoconnect**.
4. Теперь 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, вы увидите следующую картину:

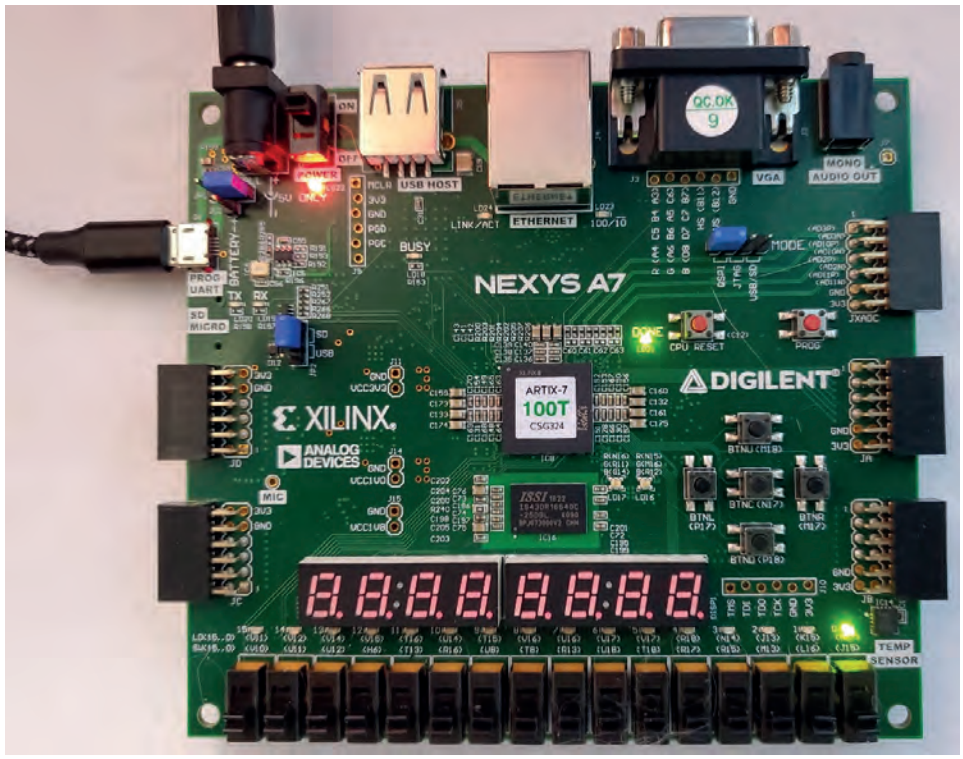


Рис. 1.24. Окончательный вид платы

5. Установите переключатели в состояния 00, 01, 10, 11, где 0 – положение вниз, а 1 – положение вверх. Соответствуют ли свечение сигналов результатам симуляции? Соответствует ли это тому, как должна работать плата по вашему мнению? Видите ли вы иногда мерцание при переключении переключателей? На последний вопрос мы ответим в главе 3 «Подсчет кликов на кнопку».

Поздравляем! Вы завершили свой первый проект на плате FPGA. 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

В этой главе мы познакомились с основами ASIC и FPGA, узнали, как они создаются и в каких ситуациях какие из них имеют больший смысл с точки зрения денежных затрат. Мы научились использовать плату FPGA и программировать ее. Это подготавливает нас к остальной части книги, где мы будем использовать эту плату и наши 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 вы можете использовать FPGA?
   a) Вы создаете прототип приложения, которое в конечном итоге может стать ASIC.
   b) У вас будут очень маленькие объемы производства.
   c) Вам нужно что-то, на чем вы сможете легко изменить алгоритмы в будущем.
   г) Все вышеперечисленное.
2. Когда вы будете использовать ASIC?
   a) Вы разрабатываете очень специализированное приложение, которое должно быть создано в небольшом количестве, и бюджет ограничен.
   b) Вас попросили разработать калькулятор, который будет выпускаться серийно и для которого требуется специализированный процессор.
   c) Вам нужно что-то чрезвычайно маломощное, и стоимость не имеет значения.
   d) Вы разрабатываете спутник для получения изображений земной поверхности и хотите иметь возможность обновлять алгоритмы в течение всего срока службы спутника.
   e) a и b.
3. В этой главе мы рассмотрели полный сумматор. Полусумматор – это схема, которая может складывать два входа без учета 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 проекты 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 тестирование его:

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

Подсказка: вы можете заглянуть в последующие главы книги, чтобы посмотреть дополнительную информацию, или можете произвести быстрый поиск в сети Интернет.

Дополнительное чтение

Для получения дополнительной информации обратитесь к следующим ссылкам.

- Конфигурируемый логический блок FPGA серии 7: 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.