Е.И.Литвинов, И.И.Шагурин

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ

С ИСПОЛЬЗОВАНИЕМ ЯЗЫКА VERILOG HDL

НА БАЗЕ FPGA ФИРМЫ «XILINX»

Москва, 2012 г.

ОГЛАВЛЕНИЕ

1. Введение....4
2. Базовые понятия....7
2.1 Что такое PLD, FPGA и ASIC....7
2.2 Принципиальная структура FPGA....10
2.3 Что такое HDL. Распространенные HDL....12
2.4 Понятие уровней абстракции и иерархии в проектировании аппаратуры....14
2.4.1 Уровни абстракции....14
2.4.2 Понятие проекта. Составные части проекта. Иерархия проекта....16
2.5 Маршрут проектирования цифровых схем для реализации на FPGA....18
2.6 Функциональное тестирование....20
2.7 Процесс синтеза....22
2.8 Процесс имплементации и создания конфигурационного файла....24
2.9 Понятия «система на кристалле» и IP-ядро....26
3. Специфика проектирования цифровых устройств для FPGA с использованием средств фирмы Xilinx....27
3.1 Обзор семейств FPGA и средств проектирования фирмы Xilinx....27
3.1.1 Обзор семейств FPGA фирмы Xilinx....27
3.2 Обзор семейства FPGA Spartan 6....30
3.3 Структура логической ячейки FPGA на примере семейства Spartan 6....32
3.4 Дополнительные ресурсы FPGA фирмы Xilinx на примере семейства Spartan 6....35
3.4.1 Блочная память (Block RAM)....35
3.4.2 Аппаратные блоки умножения....35
3.4.3 Контроллер памяти (Memory Controller Block)....36
3.4.4 Контроллер PCI Express....36
3.4.5 Гигабитные приемопередатчики (Gigabit Transceiver)....36
3.4.6 Блоки ввода-вывода....37
3.4.7 Clock Management Tile (CMT)....37
3.5 Понятие глобальных временных ограничений....37
3.6 Генерация IP-блоков....39
3.7 Моделирование проектов. Обзор ISIM....40
3.8 Внутрисхемная отладка проектов. Обзор ChipScope....41
3.9 Обзор микроконтроллеров класса «soft-core»....43
3.9.1 PicoBlaze....43
3.9.2 MicroBlaze....44
4. Проектирование аппаратуры с использование языка Verilog HDL....46
4.1 Понятие модуля. Базовая структура модуля....46
4.1.1 Объявление портов ввода-вывода....47
4.1.2. Объявление внутренних сигналов, параметров или переменных....48
4.1.3. Тело модуля....49
4.2 Типы данных....49
4.3 Реализация комбинационных схем вентильного уровня....50
4.4 Реализация модульной структуры проекта....54
4.5 Конструкции для реализации комбинационных схем уровня RTL....57
4.5.1 Операторы....58
4.5.2 Блок always....60

4.6 Реализация комбинационных схем уровня RTL 63
4.6.1 Модифицированное описание 1-битного компаратора 63
4.6.2 Описание двоичного дешифратора 64
4.6.3 Описание блока АЛУ 65
4.7 Конструкции для реализации последовательных схем уровня RTL 67
4.7.1 Триггер D-типа 68
4.7.2 Регистр 70
4.7.3 Регистровый файл 71
4.8 Реализация последовательных схем 72
4.8.1 Двоичный счетчик 73
4.8.2 Стек 74
4.9 Константы и параметры. Проектирование IP-блоков 78
4.9.1. Константы 78
4.9.2. Параметры 79
4.10 Конструкции для реализации конечных автоматов. Автоматы Мура и Мили 82
4.11. Реализация конечных автоматов 86
5. Функциональное тестирование аппаратуры средствами языка Verilog HDL 89
5.1 Базовая структура тестового окружения 89
5.2 Описание настроек моделирования 91
5.3 Тестовые последовательности и блок initial 92
5.4 Описание генератора тактового сигнала 93
5.5 Вспомогательные конструкции 94
5.5.1 Task и function 94
5.5.2 Циклы 96
5.5.3 Ожидание событий и состояний 97
5.5.4 Системные функции 98
5.6 Реализация законченного тестового окружения 99
6. Лабораторные работы 104
6.1 Лабораторная работа №1 104
6.2 Лабораторная работа №2 118
6.3 Лабораторная работа №3 129
6.4 Лабораторная работа №4 142
6.5 Лабораторная работа №5 155
7. Приложение 167
7.1 Краткое описание лабораторного макета Atlys 167
7.2 Правила оформления отчетов по лабораторным работам 169
7.3 Правила оформления файлов проектов по лабораторным работам 170
8. Рекомендуемая литература 172

1. Введение

Данная книга постарается послужить для вас хорошим учебным пособием в понимании основ FPGA (ПЛИС) и проектирования цифровых устройств. Что же такое FPGA? Где они используются? Как проектируются цифровые устройства? На все эти вопросы мы постараемся ответить в этом небольшом пособии.

Наряду с теоретическим материалом, зачастую являющимся обзорным, в данном пособии представлены также лабораторные работы, содержащие поэтапные руководства к действию при их выполнении. Данный подход позволит вам разобраться в основах проектирования цифровых устройств и закрепить изученный материал на реальных живых примерах.

В качестве платформы для реализации проектов выбрана новейшая разработка компании Xilinx, плата Atlys, являющаяся частью университетской программы для ВУЗов. Данная плата позволит вам получить наглядное представление о работе цифровых устройств и отладке готовых проектов на реально работающей аппаратуре.

Сейчас же предлагается окунуться в небольшой исторический экскурс по зарождению FPGA и их развитию. Это поможет вам лучше понять предмет изучения и начать свое знакомство с индустрией.

Первые FPGA, т.е. интегральные микросхемы, содержащие многократно программируемые массивы логики, появились в 80-х годах и применялись в составе микропроцессорных систем для замены стандартной логики, выполнявшей вспомогательные функции. Микропроцессорная техника в это время доминировала, поскольку для нее был найден ясный инженерный подход к проектированию систем. Разработчики быстро освоили магистрально-модульную структуру аппаратных средств на основе ведущего микропроцессора и

подчиненных ему интерфейсных микросхем. Создание прикладных программ осуществлялось известными методами и средствами, использовался ранее накопленные знания. Затруднения имели место на этапе комплексной отладки аппаратуры и программного обеспечения в реальном масштабе времени, но они были преодолены созданием метода внутрисхемной эмуляции и комплексов инструментальных средств на основе эмуляторов. Появление персональных компьютеров дало дополнительное ускорение процессу обучения специалистов интегрированным методам проектирования аппаратуры и программ.

Основные трудности при развитии идеи ПЛИС (как идеи свободного проектирования и изготовления разработчиком произвольного цифрового устройства) заключались в невозможности использования простого инженерного подхода и необходимости создания новых математических методов синтеза цифровых структур в некотором элементном базисе на основе описания целевой логической функции. Требовалось во взаимосвязи решить следующие задачи:

- определить элементный базис FPGA – достаточно развитый, чтобы реализовать необходимые функции целевых устройств, и достаточно простой, чтобы время расчетов на персональном компьютере или рабочей станции не было чрезмерным;
- разработать математические методы синтеза устройств (в выбранном базисе), декомпозиции, компиляции, межэлементной трассировки, функционального моделирования и временного анализа;
- создать интегрированную систему проектирования цифровых устройств на FPGA.

Вначале появились микросхемы типа PLD, их программирование осуществлялось в кодах через заполнение таблицы истинности. Позже появились микросхемы типа PAL и стали применяться языки программирования

ассемблерного типа, как например PALASM.

В настоящее время ведущими классами микросхем программируемой логики являются FPGA, имеющие очень высокую степень интеграции и быстродействие до 600МГц. В качестве средств описания проектов применяются языки высокого уровня типа HDL (Hardware Description Language). Кроме того основная часть проблем проектирования, описанных выше была решена. Появились эффективные автоматизированные средства для всех этапов проектирования, позволяющие в довольно сжатые сроки реализовывать крайне большие проекты.

Новый импульс развитию FPGA сообщило развитие коммуникационных технологий. Именно здесь, на больших потоках и предельных скоростях обработки информации стали проявляться принципиальные ограничения микропроцессоров, связанные с избыточностью их архитектуры. В свою очередь, микросхемы FPGA позволяют реализовать специализированную структуру, поэтому при одинаковой тактовой частоте реализованное устройство имеет существенное преимущество в быстродействии. С этой особенностью связана и другая быстро развивающаяся область применения FPGA - реализация функций цифровой обработки сигналов.

2. Базовые понятия

В данной главе будут описаны базовые понятия и принципы, характерные для проектирования цифровых устройств для FPGA. После прочтения этой главы, вы будете иметь представление о структуре, методах и средствах проектирования и маршруте проектирования для FPGA.

2.1 Что такое PLD, FPGA и ASIC

На сегодняшний день существует множество различных типов интегральных микросхем. Среди всего многообразия можно выделить несколько наиболее интересных и устоявшихся групп интегральных микросхем, таких как PLD (Programmable Logic Devices), FPGA (Field Programmable Gate Arrays), ASIC (Application Specific Integrated Circuit). Данные группы интегральных микросхем различаются между собой по быстродействию, емкости, определяемой по количеству простейших логических вентилей (элементы, выполняющие такие логические операции как «И» или «ИЛИ»), которое может содержать микросхема, и ряду архитектурных особенностей.

PLD (ПЛУ) являются цифровыми интегральными микросхемами, состоящими из программируемых логических блоков и программируемых соединений между этими блоками. Внутренняя архитектура PLD определена производителем таким образом, что они могут быть перепрограммированы для выполнения самых различных функций. Данные интегральные схемы имеют малую емкость, характеризуются отсутствием встроенных аппаратных блоков, таких как память, процессорные ядра и т.д. Преимуществом данной технологии является то, что при отключении питания микросхема сохраняет всю запрограммированную логику работы, поэтому после включения питания повторное программирование не требуется. В данный момент основным представителем данного семейства

являются CPLD (Complex Programmable Logic Devices), названные так вследствие увеличенной емкости и улучшенной архитектуры.

FPGA (ПЛИС) представляют собой логические устройства, содержащие двумерный массив универсальных логических ячеек и программируемых переключателей. Логические ячейки конфигурируются так, чтобы они выполняли необходимые элементарные функции, а программируемый переключатель осуществляют требуемое межсоединение логических ячеек. Проект, требующий разработки, может быть реализован с помощью определения функций каждой логической ячейки и выборочной установки каждого программируемого переключателя. В архитектуру данных интегральных микросхем включено также огромное многообразие аппаратных блоков, таких как DSP-ядра, блоки памяти большой емкости, процессорные и сопроцессорные ядра, а также различные интерфейсные модули. Данные интегральные микросхемы также могут работать на высоких частотах (до 600 MHz), что позволяет использовать их в самых передовых разработках. FPGA, однако, теряют свою конфигурацию при отключении питания, что приводит к необходимости применения внешней памяти. Возможность конфигурирования данных интегральных микросхем и их свойства позволяют решать инженерам огромное количество различных задач.

ASIC (заказные интегральные микросхемы) в свою очередь содержат сотни миллионов логических вентилей и могут выполнять чрезвычайно сложные функции. Данный тип устройств разрабатывается для использования в составе специальных приложений, т.е. призван максимально эффективно решать ограниченный круг задач. Высокая сложность данного типа интегральных микросхем, привела к делению на следующие основные подтипы: структурированные ASIC, схемы на стандартных элементах (Standard Cell) и полностью заказные интегральные схемы (Full Custom). Данное деление обусловлено сочетанием производительность/время разработки/стоимость, что

позволяет выбрать оптимальные показатели для конкретного технического решения на этапе планирования.

Таким образом, FPGA занимают промежуточное положение между сравнительно простыми CPLD и сложными ASIC. Их функциональность может быть задана достаточно быстро, без применения сложного технологического оборудования. Они могут содержать миллионы логических вентилей и, следовательно, реализовывать чрезвычайно сложные функции, которые изначально могли быть реализованы только с помощью ASIC. Данное положение FPGA позволяет использовать их в ряде важных технических решений, таких как:

- Прототипирование ASIC. Данное решение позволяет оценить работу разрабатываемой микросхемы в реальных условиях, а также в сотни раз сократить время моделирования при ее тестировании;
- Создание конечных изделий для небольших рынков. Вследствие того, что стоимость FPGA существенно меньше стоимости разработки ASIC, для небольших партий изделий данное решение оказывается единственным возможным. Однако данный подход неприемлем для больших рынков, где конечная стоимость единичной продукции к объему произведенных изделий будет существенно ниже, чем стоимость аналогичного решения на FPGA. Также немаловажным фактором является гораздо более высокая производительность ASIC по сравнению с FPGA (при одинаковом уровне технологии производства в рамках одного и того же проекта);
- Создание специфических сопроцессоров для работы в составе системы совместно с ведущим процессором. Данное решение используется как аппаратное ускорение программных алгоритмов, вследствие возможности распараллеливания вычислительной задачи;
- Цифровая обработка сигналов. Современные FPGA содержат аппаратные

DSP-ядра (Digital Signal Processing), что в сочетании с большими объемами внутренней оперативной памяти и возможностью использования дополнительных логических функций дает в ряде случаев преимущества над традиционными подходами, которые предполагают использование специально разработанных DSP-процессоров;

- Физический уровень передачи данных. FPGA используются как связующее звено, выполняющее функцию интерфейса, что в сочетании с аппаратными высокоскоростными приемопередатчиками позволяет реализовывать сетевые и коммуникационные функции в одном устройстве.

В этой книге будет подробно рассматриваться технология FPGA, что позволит вам овладеть основами проектирования цифровых устройств и позволит вам мгновенно наблюдать поведение разработанного устройства.

2.2 Принципиальная структура FPGA

Принципиальная структура FPGA устройства показана на рисунке 2.2.1. Как видно из рисунка 2.2.1, структурно, FPGA состоит из следующих компонент.

- Logic Cell. Логическая ячейка обычно содержит небольшие комбинационные схемы с перестраиваемой конфигурацией и один или несколько триггеров D-типа;
- Macro Cell. Макро ячейка представляет из себя некоторый законченный физический блок, дополняющий базовый функционал FPGA. Т.е. Блок спроектированный и изготовленный на транзисторном уровне. Обычно используют макро ячейки, содержащие блоки памяти, комбинационные умножители, схемы управления временем, периферийные схемы. Продвинутые FPGA устройства могут даже содержать одно или несколько готовых процессорных ядер;
- S. Представляет собой программируемый переключатель.

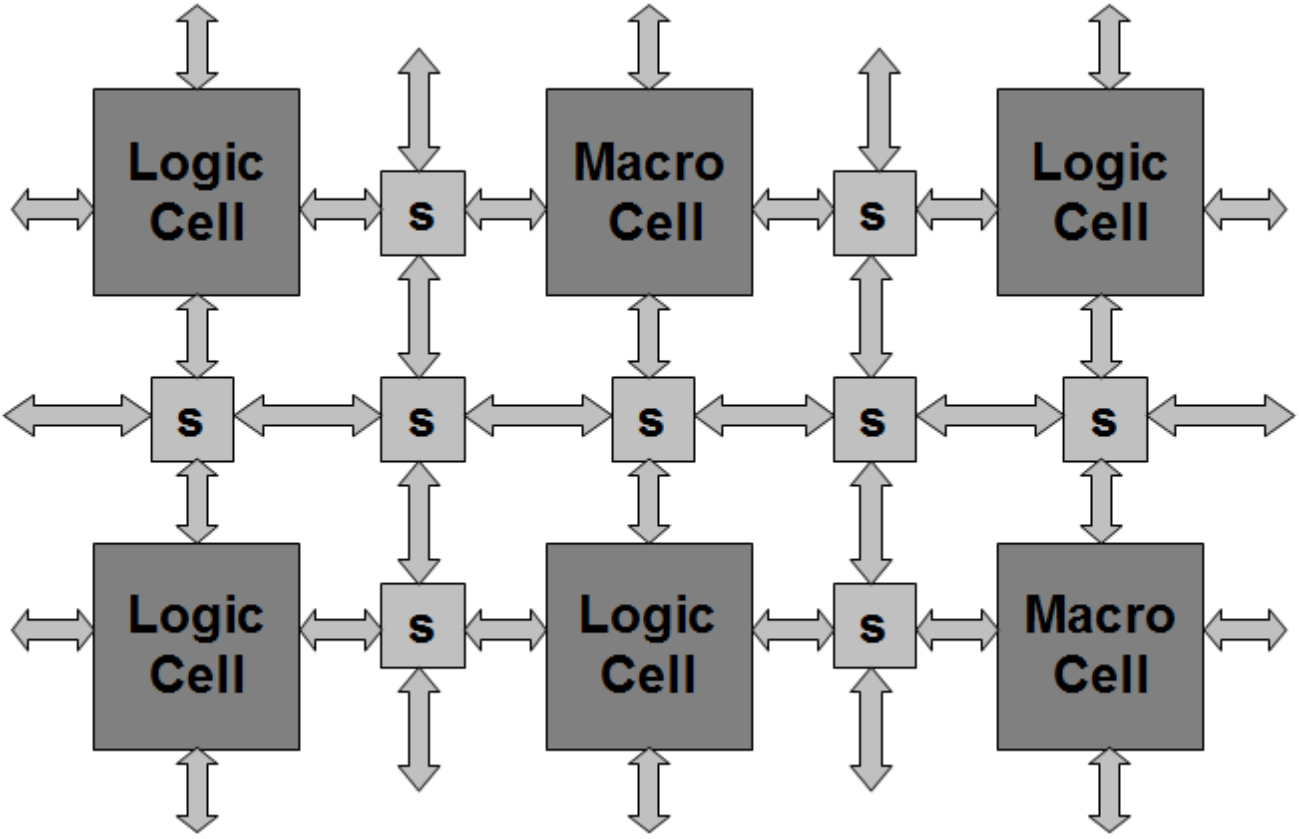


Рисунок 2.2.1 — Принципиальная структура FPGA

Логическая ячейка может быть сконфигурирована таким образом, чтобы выполнять элементарную функцию, а программируемый переключатель может быть выбран, чтобы осуществлять межсоединения среди логических ячеек. Выбранный проект (проект, требующий разработки) может быть реализован с помощью определения функций каждой логической ячейки и выборочной установки каждого программируемого переключателя. Результатом проектирования цифрового устройства для FPGA как раз и является файл, содержащий конфигурацию некоторого количества логических ячеек и межсоединений, реализующих требуемое поведение. На последнем этапе проектирования данный файл используют для конфигурации некоторой целевой FPGA. Поскольку процесс конфигурации выполняют чаще всего в «полевых условиях», нежели чем на заводе, то и устройство получило название: программируемое в полевых условиях (field-programmable).

2.3 Что такое HDL. Распространенные HDL

В настоящее время для разработки сложных цифровых устройств широко используется метод проектирования, основанный на языках описания аппаратных средств или HDL (Hardware Description Language).

HDL – язык описания аппаратуры, представляет собой любой язык для формального описания логики функционирования аппаратуры. С помощью такого языка можно описать поведение и архитектуру цифровой схемы, а также набор тестов для проверки качества проекта. Данный тип языков отличается от обычных языков программирования наличием «синтезируемого подмножества», т.е. наличием конструкций, которые можно формализовать для перевода высокоуровневого описания устройства к вентильному представлению, возможности явного использования времени, а также чертами параллельных языков программирования.

На данный момент разработано значительно число различных HDL. Имеется много исследовательских проектов по созданию новых или улучшению уже устоявшихся языков. Однако существенную ценность в коммерческом процессе проектирования играют только стандартизированные и устоявшиеся языки. В связи с этим можно утверждать, что в настоящее время в индустрии доминируют два языка описания аппаратуры: Verilog HDL и VHDL.

VHDL. Самым старым представителем HDL является VHDL (Very high speed integrated circuits Hardware Description Language), появившийся в 1980 году и получивший стандартизацию в 1987 (IEEE 1076). Данный язык был разработан по заказу Министерства обороны США с целью формального описания логических схем для всех этапов разработки электронных систем.

Первоначально данный язык предназначался для моделирования, но позднее из него было выделено «синтезируемое подмножество». В соответствии с техническим заданием и предпочтениями разработчиков средствами языка VHDL

возможно проектирование на различных уровнях абстракции (подробнее о уровнях абстракции будет рассказано далее). Также, отличительной особенностью VHDL является его большая мощность на поведенческом и функциональном уровнях абстракции. В данный язык заложена возможность иерархического проектирования, максимально реализующая себя в экстремально больших проектах с участием большой группы разработчиков. Представляется возможным выделить следующие три составные части языка: алгоритмическую — основанную на языках Ada и Pascal и придающую языку VHDL свойства языков программирования; проблемно ориентированную — обращающую VHDL в язык описания аппаратуры; и объектно-ориентированную, интенсивно развиваемую в последнее время.

Verilog HDL. Verilog HDL (Verilog Hardware Description Language) был разработан в середине 1980х и, позднее, официально определен как стандарт IEEE 1364. Стандарт пересматривался в 1995 году – Verilog-1995, в 2001 году – Verilog-2001 и в 2005 году – Verilog-2005. В последних версиях языка было добавлено множество полезных расширений.

Язык Verilog HDL позволяет осуществить проектирование, верификацию и реализацию аналоговых, цифровых и смешанных электронных систем на различных уровнях абстракции. Разработчики языка сделали его синтаксис очень похожим на синтаксис языка С, что упрощает его освоение. Verilog HDL имеет препроцессор, очень похожий на препроцессор языка С, и основные управляющие конструкции языка, такие как «if», «while» и т.д. также подобны одноимённым конструкциям языка С.

В этом пособии мы будем использовать язык Verilog HDL. Более того, вместо того, чтобы раскрывать все аспекты Verilog HDL, мы остановимся на ключевых конструкциях языка с помощью рассмотрения ряда примеров. Более детальное представление о Verilog HDL может быть получено с помощью источников,

представленных в списке используемой литературы. Базовые конструкции данного языка будут рассмотрены в отдельной главе далее.

2.4 Понятие уровней абстракции и иерархии в проектировании аппаратуры

Со времени первого появления цифровой микросхемы в 1958 году разработчики наблюдают экспоненциальный рост числа транзисторов на интегральной микросхеме, что приводит к росту проектов и их стремительному усложнению. Для упрощения процесса проектирования, в проектах используются техники, такие как абстракция и иерархия. Эти техники работают по принципу «разделяй и властвуй» и оказывается довольно эффективным в рамках экстремально больших проектов.

2.4.1 Уровни абстракции

Функциональные возможности цифровых микросхем могут быть представлены с помощью HDL на различных уровнях абстракции. Самым низким уровнем абстракции цифровых HDL является уровень транзисторных ключей, который определяет способность описывать схему в виде таблицы транзисторных ключей. Выше него находится уровень вентилей, который описывает схему в виде таблицы соединений простых логических вентилей и функций. Оба описанных уровня абстракции можно отнести к структурному представлению устройства.

Следующий, более сложный уровень HDL, определяется способностью поддерживать логические примитивы. Этот уровень называется функциональным или RTL (Register Transfer Level), т.е. уровнем регистровых передач. Устройство на данном уровне абстракции представляется набором регистров, связанных между собой элементами комбинационной логики.

Высшим уровнем абстракции считается поведенческий, который поддерживается современными версиями HDL и обозначает возможность описывать поведение схемы, используя абстрактные логические структуры,

например циклы и процессы. Этот уровень также подразумевает использование в выражениях алгоритмических элементов, таких как сумматоры и умножители.

В рамках представления аппаратных блоков имеется также системный уровень абстракции, представляющий наборы алгоритмов в рамках крупных блоков и их соединение между собой. К сожалению, данный подход используется только в моделировании поведения системы и пока не пригоден для разработки описания реального устройства.

На рисунке 2.4.1 показано упрощенное представление о функционировании аппаратных блоков на различных уровнях абстракции.

В настоящее время основным способом представления аппаратуры является уровень RTL. Все остальные представления либо являются вспомогательными, либо впоследствии конвертируются в однозначный RTL-код.

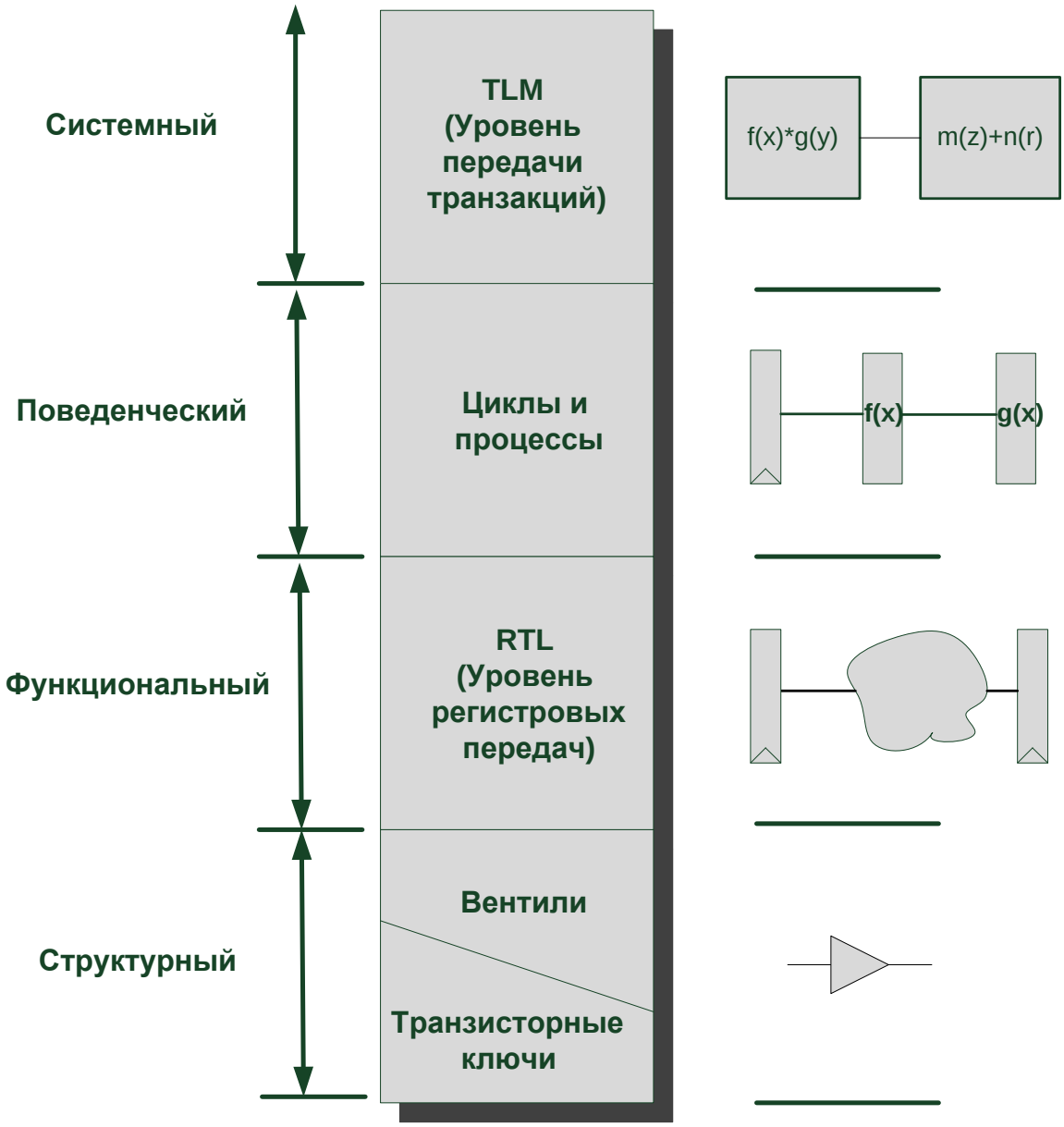


Рисунок 2.4.1 – Представление о функционировании аппаратных блоков на различных уровнях абстракции

2.4.2 Понятие проекта. Составные части проекта. Иерархия проекта.

В начале рассмотрим понятие проекта в проектировании цифровых устройств для FPGA. Проектом является набор файлов, содержащих графическое описание блоков, описание блоков на языках HDL, описания ограничений проекта, и т.д. Глобально, можно сказать, что любой проект содержит файлы исходного

кода, описывающие модули и блоки проекта и ряд вспомогательных файлов.

В рамках современного подхода к проектированию любой проект является иерархическим, т.е. построенным по модульному принципу (можно сравнить с матрешкой). В проекте есть некоторый файл, содержащий модуль верхнего уровня, который содержит некоторую логику работы и подключает модули более низкого уровня, которые в свою очередь осуществляют ту же последовательность действий — т.е. содержат свою внутреннюю логику и подключают модули более низкого уровня и т.д. На рисунке 2.4.2 представлена упрощенная структура проекта с учетом иерархии.

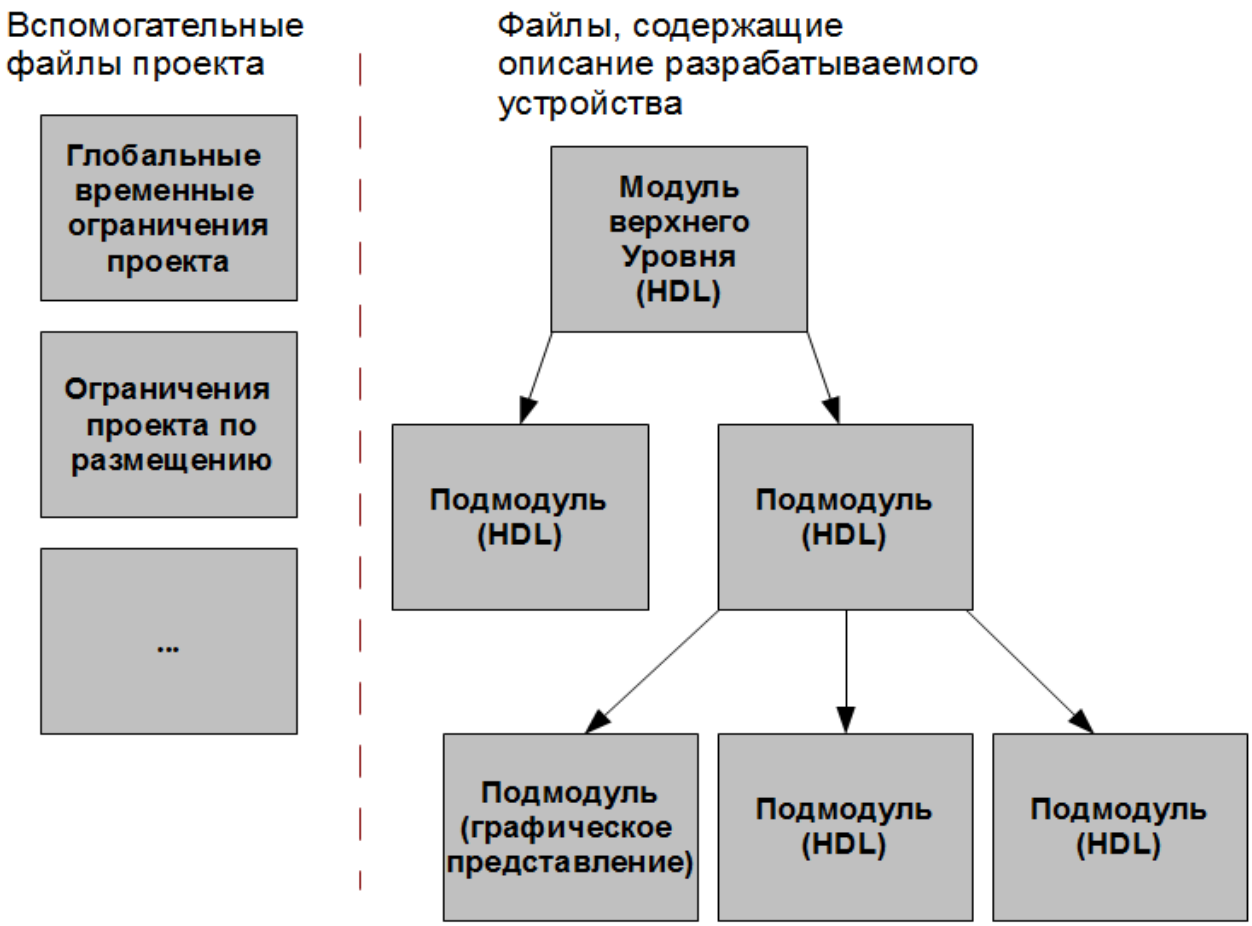


Рисунок 2.4.2 — Упрощенная структура иерархии проекта

Иерархия проекта поддерживается непосредственно конструкциями языков HDL, о чем будет написано в главе, посвященной языку Verilog HDL.

2.5 Маршрут проектирования цифровых схем для реализации на FPGA

Упрощенная схема маршрута проектирования систем, основанных на FPGA, с помощью языков HDL на уровне абстракции RTL показана на рисунке 2.5.1. В левой части маршрута проектирования представлены этапы преобразования текстового HDL описания (RTL code) в конфигурацию устройства уровня Cell-level, которая затем загружается в целевую FPGA (device programming). В правой части представлен процесс валидации (проверки правильности), который использует тестовое окружение (testbench) для проверки, удовлетворяет ли система техническим требованиям (RTL verification) и требованиям производительности (static timing analysis).

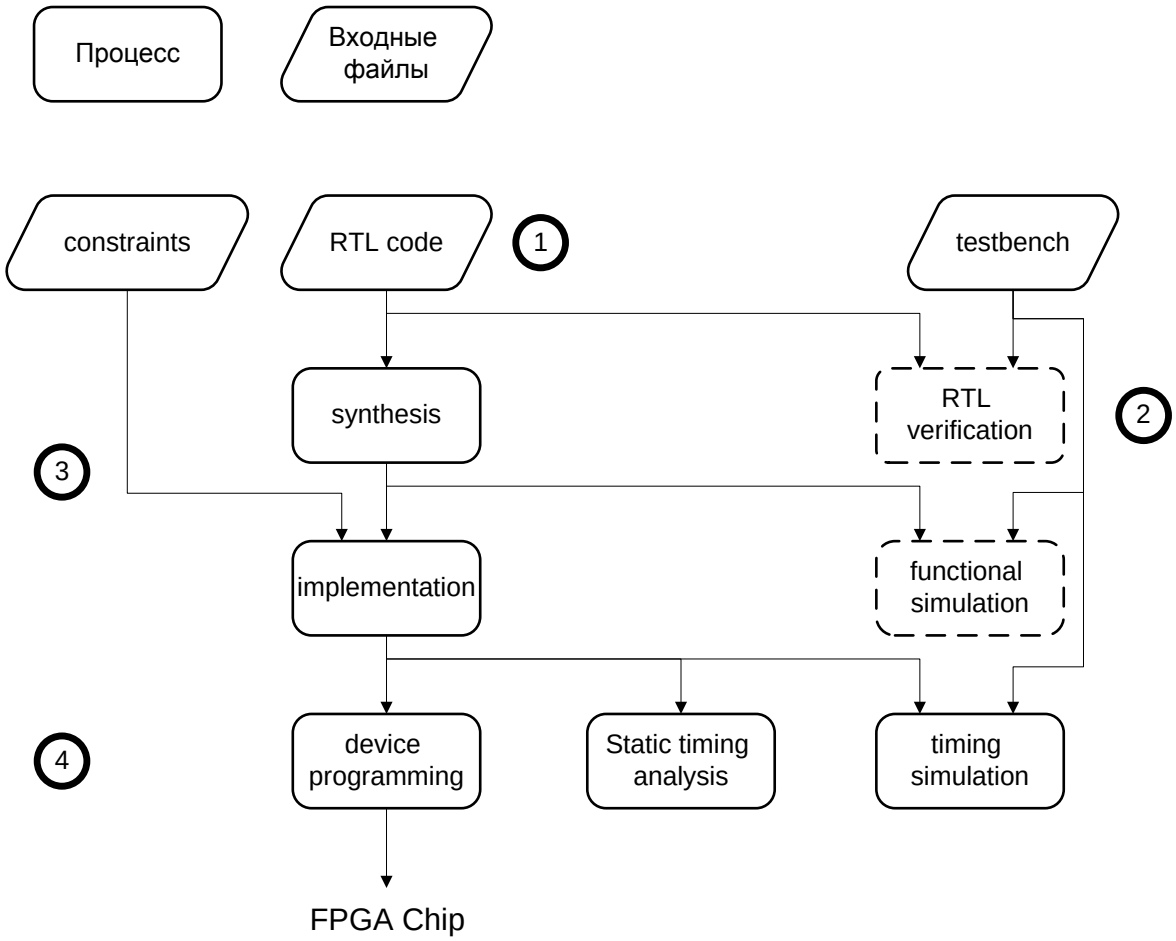


Рисунок 2.5.1 — Упрощенный маршрут проектирования для FPGA

Главные этапы в маршруте проектирования:

1. Проектирование системы и получение HDL файлов. Добавление отдельных файлов ограничения (constraints) для определения ограничений реализации;
2. Создание тестового окружения (testbench) и осуществление моделирования RTL-кода;
3. Выполнение синтеза (synthesis) и имплементации (implementation). Процесс синтеза в основном известен как логический синтез (logic synthesis). На ланной стадии HDL файлы преобразуются в списки соединений (netlist). Процесс имплементации состоит из трех ступеней: трансляции (translate), отображения (map), а также размещения и разводки (place and route). Процесс трансляции заключается в переводе всех списков соединений к единому формату и единому файлу, зависящему от средств САПР. Процесс, который известен как отображение, отображает универсальные вентили в логические ячейки, специфичные для некоторой целевой микросхемы FPGA. Процесс размещения и разводки, служит для получения физического размещения (топологии) разрабатываемого устройства внутри микросхемы FPGA. Этот процесс размещает ячейки в физической области и определяет маршруты соединения различных сигналов. Статический временной анализ (static timing analysis), выполняется в конце процесса реализации. Он определяет временные параметры соединений, такие как максимальную задержку распространения сигнала и максимальную тактовую частоту;
4. Создание и прошивка программного файла. На данном этапе, согласно окончательному представлению проекта после физического размещения, создается файл конфигурации. Данный файл служит для конфигурирования логических ячеек и коммутационных матриц целевой FPGA.

Также, при необходимости, можно выполнить два дополнительных шага: произвести функциональное моделирование (functional simulation), которое может быть выполнено после синтеза, а также временное моделирование (timing

simulation), выполняемое после этапа имплементации. Функциональное моделирование использует полученный на этапе синтеза netlist для замены RTL описания и проверки корректности процесса синтеза. Временное моделирование использует полученное на этапе имплементации конечное представление разрабатываемого устройства наряду с детальными временными данными, также для осуществления проверки на корректность.

2.6 Функциональное тестирование

После создания описания проекта уровня RTL, требуется произвести его функциональное тестирование. т.е. удостовериться, что все заявленные функции в проектной спецификации реализованы и работают корректно для подавляющего большинства входных воздействий (невозможно проверить корректность работы устройства для абсолютно всех воздействий — т.к. количество таких воздействий очень велико).

В настоящее время все чаще термин «функциональное тестирование» заменяют на термин «функциональная верификация», подчеркивая тем самым резкое усложнение этого процесса в связи с не прекращающимся ростом сложности проектов.

Также, часто термин «функциональная верификация» относят к более сложному процессу тестирования в котором применяются современные техники и фреймворки для создания сложных тестовых окружений, в частности основывающихся на языке SystemVerilog, а «функциональным тестированием» считают относительно простую проверку проекта на корректность работы. В этой книге мы будем придерживаться этого же мнения. В связи с этим в этом разделе будут рассмотрены базовые принципы функционального тестирования, а в пятой главе будут рассмотрены конструкции языка Verilog HDL для реализации простейшего тестового окружения.

Прежде всего, для любого тестирования необходимо тестовое окружение — т.е. некоторая надстройка над разработанным проектом. Такая надстройка в своем минимальном исполнении будет включать в себя лишь генератор входных последовательностей, представляющий компонент, подающий на входы тестируемого блока некоторый набор входных сигналов, имеющих смысл для тестируемого блока. При этом анализ результата работы тестируемого блока происходит в ручном режиме с помощью анализа временных диаграмм. Подобный процесс представлен на рисунке 2.6.1.

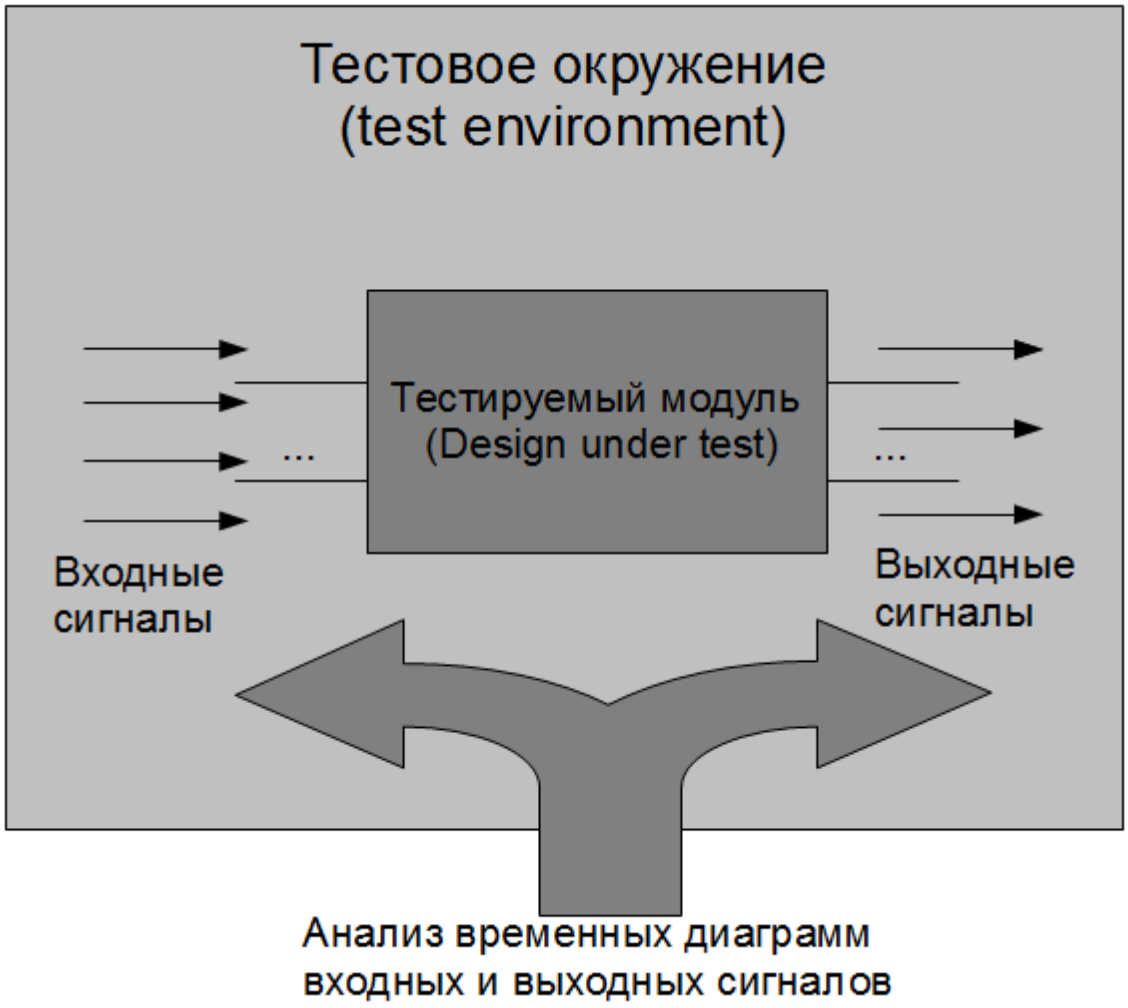


Рисунок 2.6.1 — Процесс функционального тестирования

Более сложные тестовые окружения могут содержать эталонные модели,

средства автоматической проверки корректности разрабатываемого устройства, блоки сбора разнообразных метрик покрытия и т.д.

Само тестирование может происходить средствами специальных программных пакетов, осуществляющих событийное моделирование — т.е. моделирование на уровне событий, таких как изменение внутренних сигналов устройства, появление фронта тактового сигнала и т.д. К числу таких программных пакетов относятся Synopsys VCS, Mentor Graphics Modelsim, Mentor Graphics Questasim, Xilinx ISIM и т.д. В данной книге при выполнении лабораторных работ, мы будем пользоваться средствами Xilinx ISIM, так как данное средство входит в состав Xilinx WebPack и не требует лицензии для использования.

2.7 Процесс синтеза

В процессе разработки проектировщики описывают поведение аппаратных блоков на технологически-независимых высокоуровневых языках HDL. Процесс синтеза представляет из себя процесс перевода высокоуровневого описания разрабатываемого устройства в оптимизированное представление вентильного уровня (netlist) по некоторому алгоритму, основываясь на ограничениях проекта и выбранной технологической библиотеки. Подобная технологическая библиотека содержит как базовые логические вентили, так и макро-блоки, такие как сумматоры, мультиплексоры, память и триггеры.

На первых этапах развития цифровой электроники, разработчики осуществляли логический синтез своими руками, анализируя проект и переводя разрабатываемые модули и блоки в вентильное представление, опираясь на некоторую технологическую библиотеку и ограничения проекта. На рисунке 2.7.1 представлена блок-схема этапов ручного перевода проекта к оптимизированному представлению в виде набора вентилей.

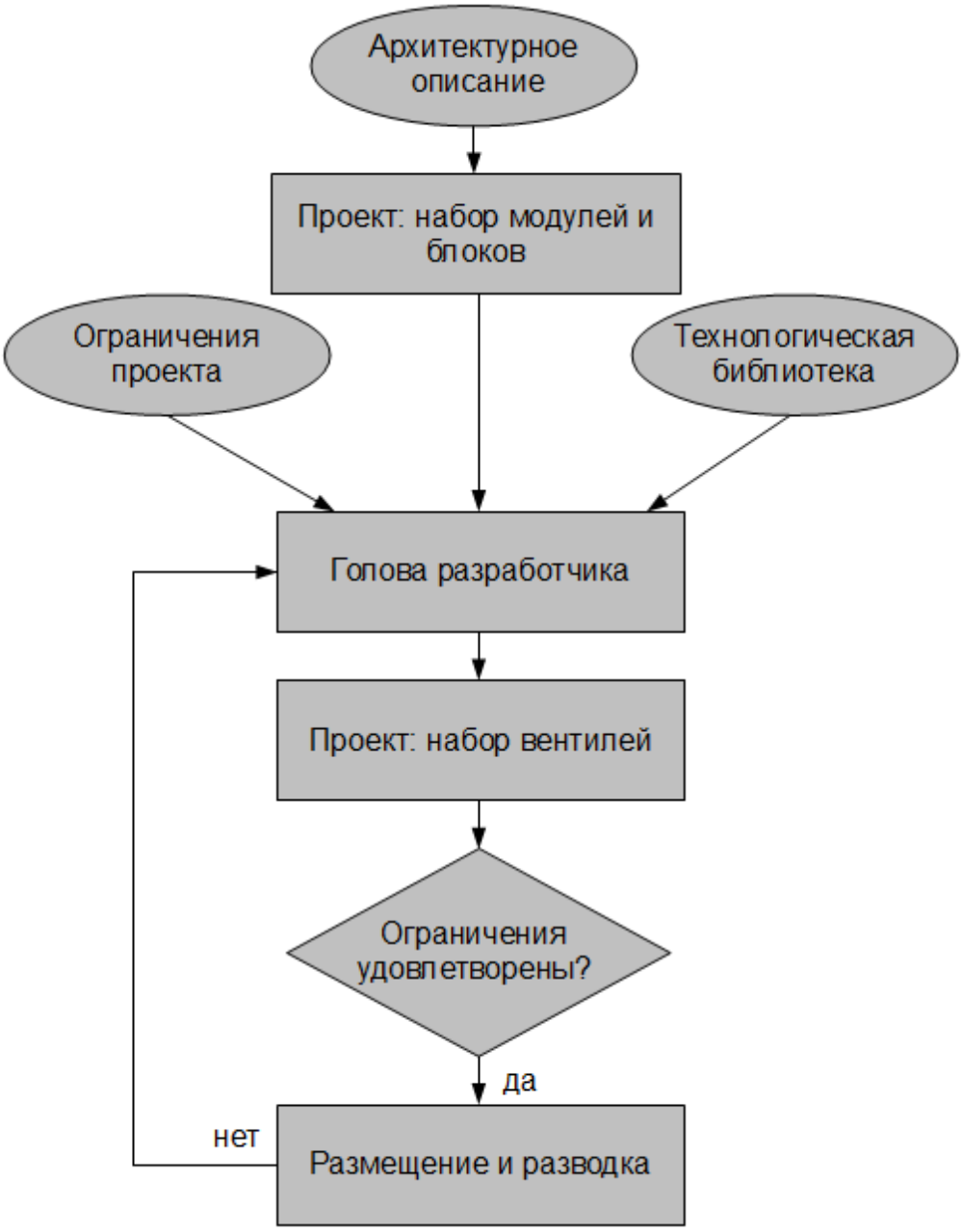


Рисунок 2.7.1 — Процесс логического синтеза без использования средств автоматизированных систем проектирования

В настоящее время процесс логического синтеза является полностью автоматизированным. Это позволяет разработчикам сосредоточить свое внимание на проектировании и использовать в разработке высокоуровневые языки HDL. На рисунке 2.7.2 представлена блок-схема этапов проектирования с использованием автоматизированных систем проектирования.

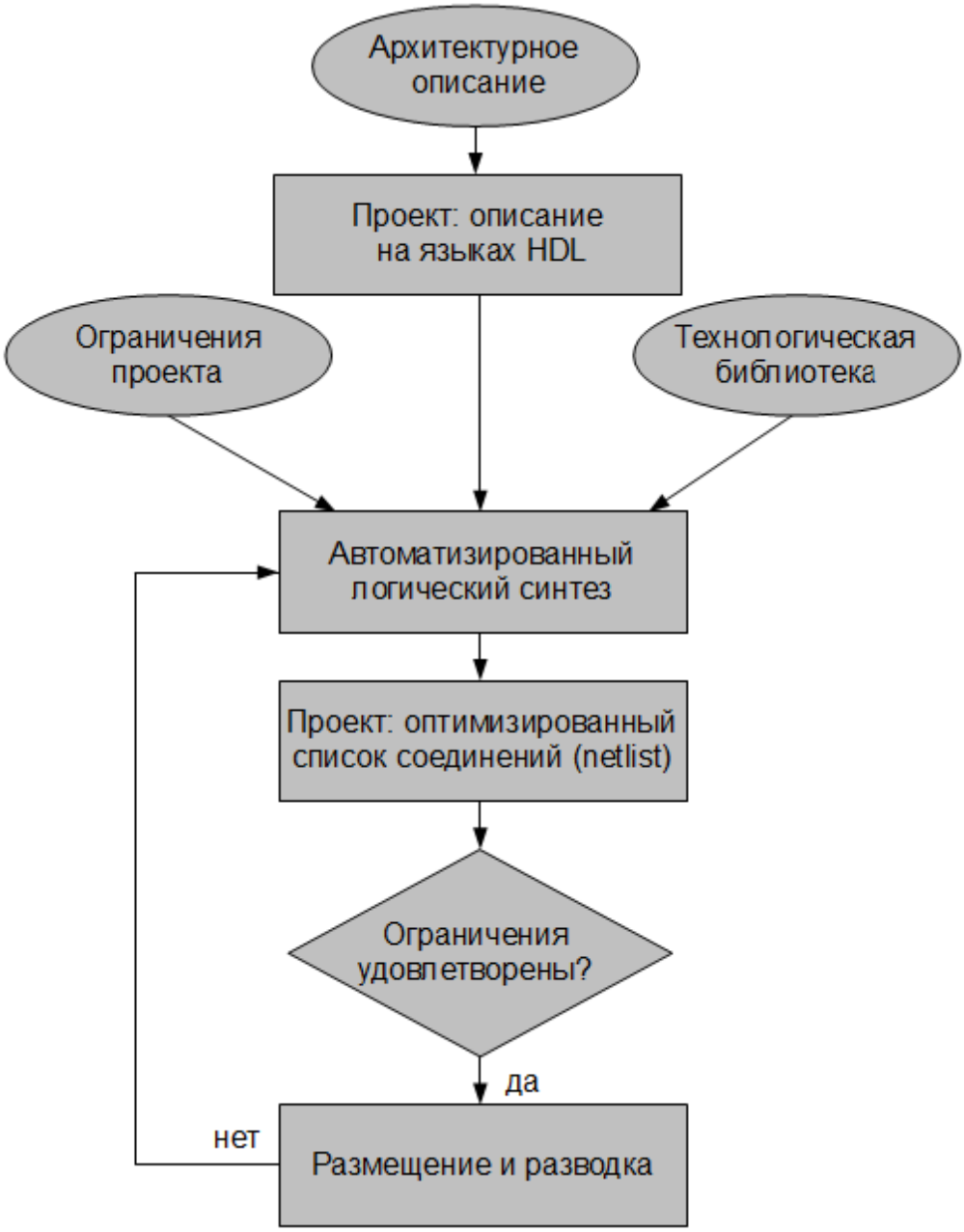


Рисунок 2.7.2 — Процесс логического синтеза с использованием средств автоматизированного проектирования

2.8 Процесс имплементации и создания конфигурационного файла

Процесс имплементации является следующим шагом после процесса синтеза в маршруте проектирования. Для данного этапа можно выделить три характерные ступени:

- Трансляция (Translate);
- Отображение (Map);

- Размещение и разводка (Place and Route).

В связи с тем, что любой проект может состоять не только из множества файлов, содержащих исходный код описания модулей проекта на языках HDL, но и содержать множество файлов, представляющих собой списки соединений (netlist), то требуется некоторый этап, на котором будет осуществлена конвертация всех списков соединений к некоторому единому формату. Этим этапом и является процесс трансляции. При объединении всех списков соединений в единое целое, на этом этапе также происходит слияние с файлами, описывающими ограничения проекта (design constraints). Также следует отметить, что формат и структура результирующего файла, полученного на этом этапе является специфичной от фирмы производителя программного обеспечения.

На стадии отображения, проект отображается на реально доступные ресурсы, характерные для целевого кристалла FPGA. При этом процессе обнаруживаются ошибки по использованию недоступных или отсутствующих для целевого FPGA ресурсов.

Заключительной стадией процесса имплементации проекта является стадия размещения и разводки. На этом этапе средства автоматизированного проектирования размещают проект на физическом уровне. Вся логика проекта размещается на конкретных физических структурах целевой FPGA, кроме этого между логическими блоками прокладывается трассировка. На данном этапе можно выявить ошибки по переполнению целевой FPGA – т.е. выявить потребность в использовании большего количества ресурсов, чем имеется в наличии. Также причиной ошибок могут стать ситуации, при возникновении которых, невозможность корректно соединить некоторый участок логики с общим проектом по причине нехватки соединений и т.д.

По завершению процесса имплементации и отсутствию критических предупреждений и ошибок, проект становится полностью готовым к

заключительному этапу, а именно, к созданию конфигурационного файла для FPGA.

2.9 Понятия «система на кристалле» и IP-ядро

В этом разделе будут освещены некоторые дополнительные понятия в проектировании цифровой техники, имеющие непосредственное отношение к FPGA. Здесь хотелось бы отметить современную методологию проектирования, называемую “система на кристалле” (СнК, System-on-Chip, SoC). Данная методология проектирования предполагает многократное повторное использование законченных и протестированных сложно-функциональных блоков (IP-ядро). При этом по критерию гибкости повторного использования все IP-ядра могут быть разделены на следующие классы:

1) синтезируемые ядра (“soft-core”) – технологически независимые блоки, описание которых осуществляется на высокоуровневых языках описания аппаратуры;

2) ядра класса “firm”, ориентированные на определенную технологию, привязка к которой осуществляется на уровне предварительного разбиения на структурные единицы в выбранном технологическом процессе для достижения требуемых характеристик по быстродействию и площади;

3) “жесткие” ядра (“hard-core”) – привязанные к одному технологическому процессу ядра (готовая топология).

В качестве технологической платформы для реализации цифровых СнК могут выступать программируемые логические интегральные схемы. При этом активно используются IP-ядра класса “soft-core”. В главе, посвященной проектированию цифровой аппаратуры с помощью языка Verilog HDL мы рассмотрим приемы создания таких IP-ядер.

3. Специфика проектирования цифровых устройств для FPGA с использованием средств фирмы Xilinx

Не смотря на то, что существует огромное множество фирм, производящих кристаллы FPGA и программное обеспечение для реализации всех этапов маршрута проектирования этих устройств, в этой книге мы сосредоточимся на продукции фирмы Xilinx, одной из крупнейших компаний в этом секторе рынка наряду с Altera и Lattice. В связи с этим, целью данной главы является некоторое введение в структуру кристаллов и средства проектирования, созданные этой фирмой. Также более детально будет освещена принципиальная структура, внутренние ресурсы и периферия семейства FPGA Spartan 6, являющегося платформой для реализации лабораторных работ, предложенных в данной книге.

3.1 Обзор семейств FPGA и средств проектирования фирмы Xilinx

В этом разделе будут рассмотрены семейства FPGA, выпускаемых фирмой Xilinx. Также будет сделан обзор программных средств от Xilinx, покрывающих весь маршрут проектирования.

3.1.1 Обзор семейств FPGA фирмы Xilinx

Прежде всего хотелось бы представить обзор семейств ПЛИС, имеющихся в арсенале данной фирмы. Каждое из семейств, представленных ниже, имеет свою собственную специфику применения и соответствующие целевые рынки:

- Spartan — являются массовыми FPGA, имеющими довольно низкую стоимость и предназначенные для решения задач не требующих больших вычислительных мощностей. К сожалению, последним представителем данного семейства является Spartan 6 и Xilinx планирует о завершении развития данного семейства. На смену ему придут два новых семейства —

Artix и Kintex;

- Artix — данное семейство ориентированно на массовые рынки недорогой продукции, имеет относительно низку стоимость и самое низкое энергопотребление по сравнению с другими семействами, имеющими одинаковый порядковый номер и выполненными по одинаковой технологической норме. Также данное семейство имеет малые габариты, что делает его в сочетании с другими параметрами очень привлекательным для построения портативных устройств;
- Kintex — кристаллы этого типа имеют большой объем памяти и расширенные ресурсы DSP, что делает это семейство идеальным для построения LTE, светодиодных и 3D цифровых видео дисплеев, устройств отображения для медицины и авиации;
- Virtex — представляют собой самые быстрые и емкие FPGA, содержащие самую богатую периферию и наибольшее количество дополнительных блоков, таких как DSP-ядра, блочная память и т.д. Все это делает данное семейство FPGA сравнимым с кристаллами ASIC;
- CoolRunner — данные семейства представляют собой реализацию архитектуры CPLD, что означает, что эти устройства представляют собой небольшие энергонезависимые кристаллы, служащие для организации небольших, критических к энергопотреблению цифровых схем.

Новейшие кристаллы с архитектурой FPGA от Xilinx спроектированы по технологии high-K metal gate и 28нм техпроцессу, что позволяет при минимальном энергопотреблении достичь максимальной производительности. На момент написания данной книги, фирмой были анонсированы семейства FPGA 7-ой серии, а также абсолютно новый продукт под названием ZYNQ 7000, содержащий на одном кристалле двухъядерный процессор CortexA9 и большое поле FPGA, основанное, в зависимости от модели, на семействах Artix или Kintex.

3.1.2 Покрытие маршрута проектирования средствами Xilinx

Для покрытия всего стандартного маршрута проектирования для FPGA и CPLD фирмой Xilinx было разработано два программных пакета: ISE (Integrated Software Environment) и PlanAhead. Оба этих пакета представляют собой законченные среды проектирования с удобным графическим интерфейсом, позволяющим вызывать для различных этапов маршрута проектирования специализированные программы и утилиты. Выбор того или иного пакета является в основном делом вкуса, но все же следует учитывать, что пакет PlanAhead считается более профессиональным и имеет ряд более расширенных опций. В данной книге мы будем использовать среду ISE, поэтому далее, пакет PlanAhead рассматриваться не будет.

На рисунке 3.1.1 представлено покрытие всего маршрута проектирования средствами Xilinx, также на рисунке представлены расширения файлов проекта, созданных на каждом из шагов.

Написание HDL может быть выполнено как во встроенном текстовом редакторе, так и в любом другом, по желанию разработчика. На этапе синтеза могут быть использованы как встроенный синтезатор, XST, так и внешние синтезаторы. В качестве внешних синтезаторов, рекомендуется использовать Synplify или Precision.

Процессы имплементации и создания конфигурации реализованы средствами ISE в виде вызова набора соответствующих утилит.

На протяжении всех этапов маршрута проектирования пользователь может контролировать правильность выполнения шагов, путем анализа разнообразных отчетов.

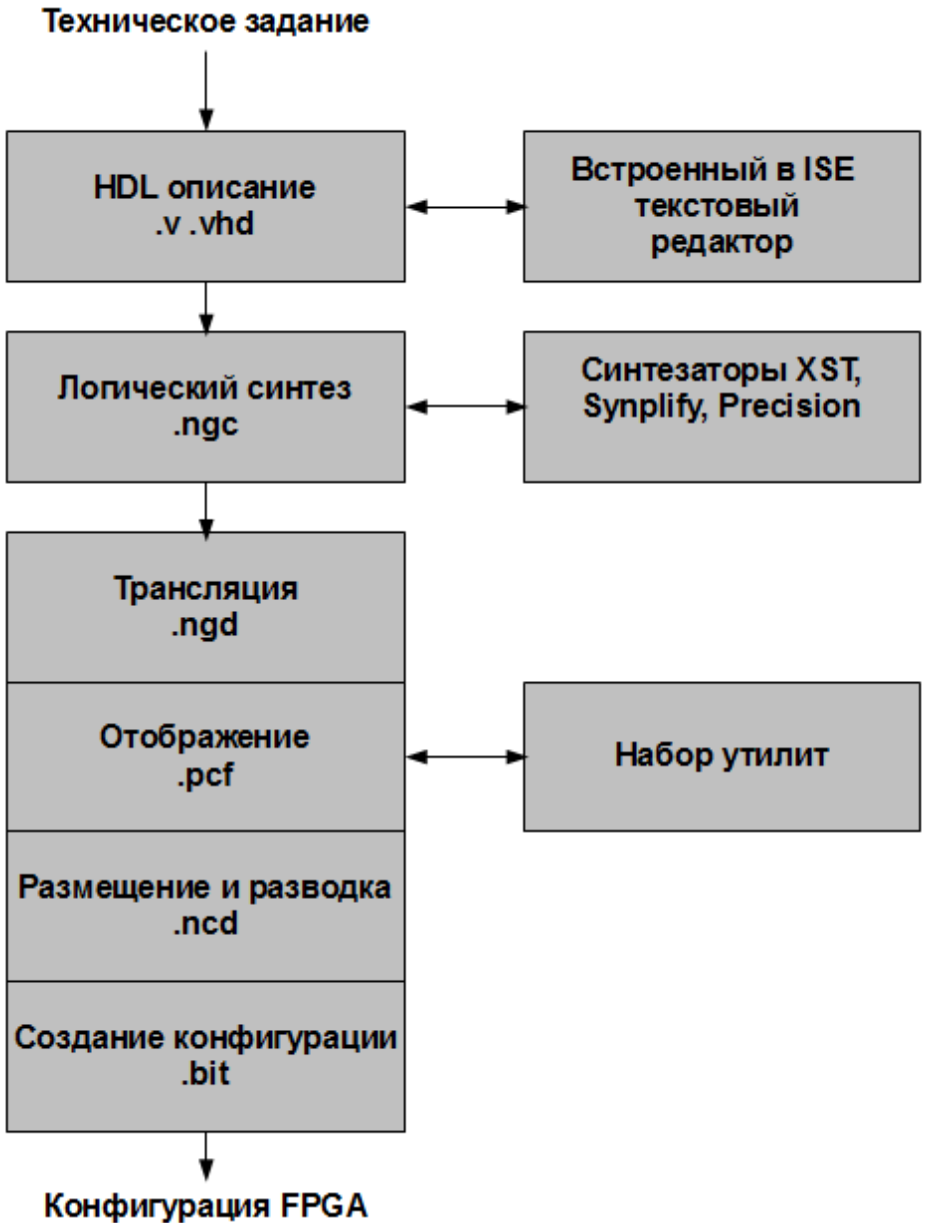


Рисунок 3.1.1 — Покрытие всего маршрута проектирования для FPGA средствами Xilinx

3.2 Обзор семейства FPGA Spartan 6

Семейство Spartan 6 было анонсировано в 2009 году и пришло на замену более старого семейства Spartan 3. Данное семейство включает в себя несколько разновидностей, в обозначении которых у фирмы Xilinx уже сложилась некоторая традиция:

- Spartan 6 LX — набор «логики»;

- Spartan 6 LXT — набор «логики» + скоростные интерфейсы связи, такие как PCIe и т.д.

В таблице 3.2.1 представлена сравнительная характеристика лучших кристаллов семейств Spartan 3 и Spartan 6.

Параметр	Spartan 3 (XC3S5000)	Spartan 6 LX (XC6SLX150)	Spartan 6 LXT (XC6SLX150T)
Технология производства	90нм	45нм	
Архитектурное сходство	Virtex-II Pro	Virtex-6	
Тип SLICE	2 LUT-4 +2 триггера	4 LUT-6 + 8 триггеров	
Количество SLICE	33280	23038	
Количество триггеров	74880	184304	
Тип аппаратных умножителей	18x18бит	DSP48	
Количество аппаратных умножителей	104	180	
Блочная память (18кБит)	104x18кБит = 1872КБит	268x18кБит = 4824 КБит	
Тип блоков тактирования	DCM	DCM + PLL	
Количество блоков тактирования	4	6	
PCIe	-	-	1
GTP	-	-	8
Контроллеры памяти (DDR, DDR2, DDR3)	-	4	
Напряжение питания ядра	1.2V	1.0V, 1.2V	

Таблица 3.2.1 - Сравнительная характеристика лучших кристаллов семейств Spartan 3 и Spartan 6

Анализируя представленную сравнительную характеристику ресурсов, можно отметить, что семейство Spartan 6 выпускается уже по 45-нм технологии, что приводит к существенному уменьшению потребляемой мощности, и имеет существенно больше ресурсов, чем предыдущее поколение. Более того, впервые архитектура Spartan сильно изменилась и стала очень похожа на архитектуру высокопроизводительной линейки Virtex, что приводит к резкому упрощению переносимости проектов.

3.3 Структура логической ячейки FPGA на примере семейства Spartan 6

Как было сказано ранее, в первой главе, структурно, любую FPGA можно представить совокупностью блоков, содержащих логику, переключателей, осуществляющих коммутацию этих блоков и макроблоков, представляющих собой некоторые законченные элементы, жестко реализованные в FPGA на физическом уровне. При этом, в терминологии Xilinx, самые большие блоки, содержащие логику, называются CLB (Configurable Logical Block). Каждый CLB подключен к трассировочной матрице, а также имеет цепь переноса (cin/cout) от соседнего CLB. Любой CLB содержит 2 меньших блока, называемых Slice. При этом существует 3 вида Slice: SliceX, SliceM и SliceL. Виды Slice различаются по предоставляемым возможностям и ресурсам. CLB может содержать один SliceX и один SliceM или SliceL. На рисунке 3.3.1 представлена принципиальная структура FPGA Spartan 6.

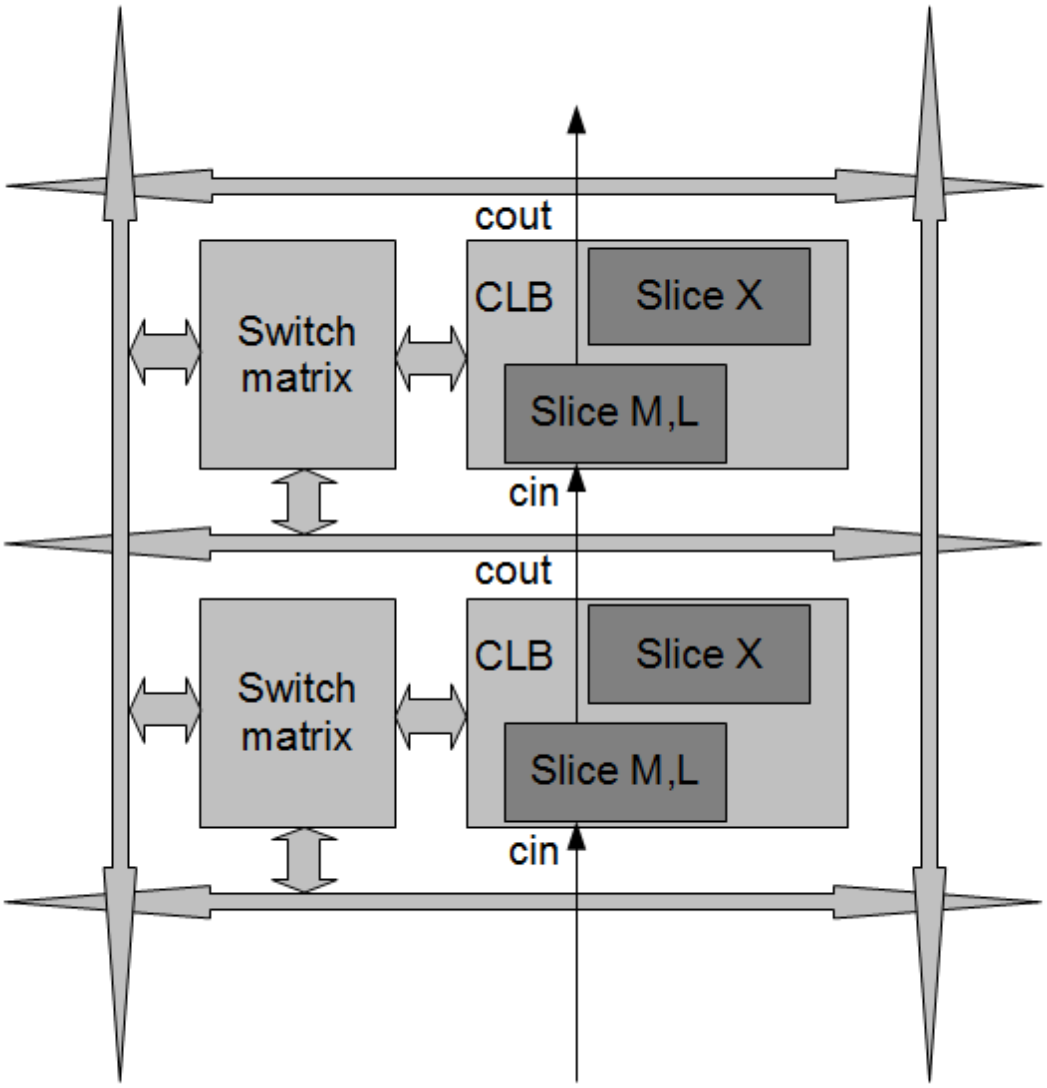


Рисунок 3.3.1 — Принципиальная структура FPGA Spartan 6

Рассмотрим теперь более подробно принципиальную структуру и типы Slice.

- **SLICEM.** Содержит 4 таблицы преобразования LUT-6, каждая из которых имеет 6-ть входов и может использоваться для реализации логических функций, реализации сдвиговых регистров или реализации блока распределенной памяти (distributed RAM), емкостью 64бит. Также этот блок содержит цепи быстрого переноса, расширяемые мультиплексоры и 8 регистров для хранения результата;
- **SLICEL.** Содержит те же ресурсы, что и SLICEM, за исключением сдвиговых регистров и блоков распределенной памяти;

- **SLICEX.** Содержит исключительно ресурсы для реализации логических функций, что уменьшает логическую нагрузку на ячейку, что приводит к уменьшению числа цепей трассировки и увеличению время распространения.

Стоит отметить, что в первых вариантах микросхем FPGA были лишь блоки SliceM типа. В последствии, на основании статистических данных по использованию кристаллов в различных проектах, было решено создать более упрощенные версии ячеек и заполнить ими кристаллы в разных процентных соотношениях. На рисунке 3.3.2 представлена архитектура ячеек Slice всех типов для FPGA Spartan 6.

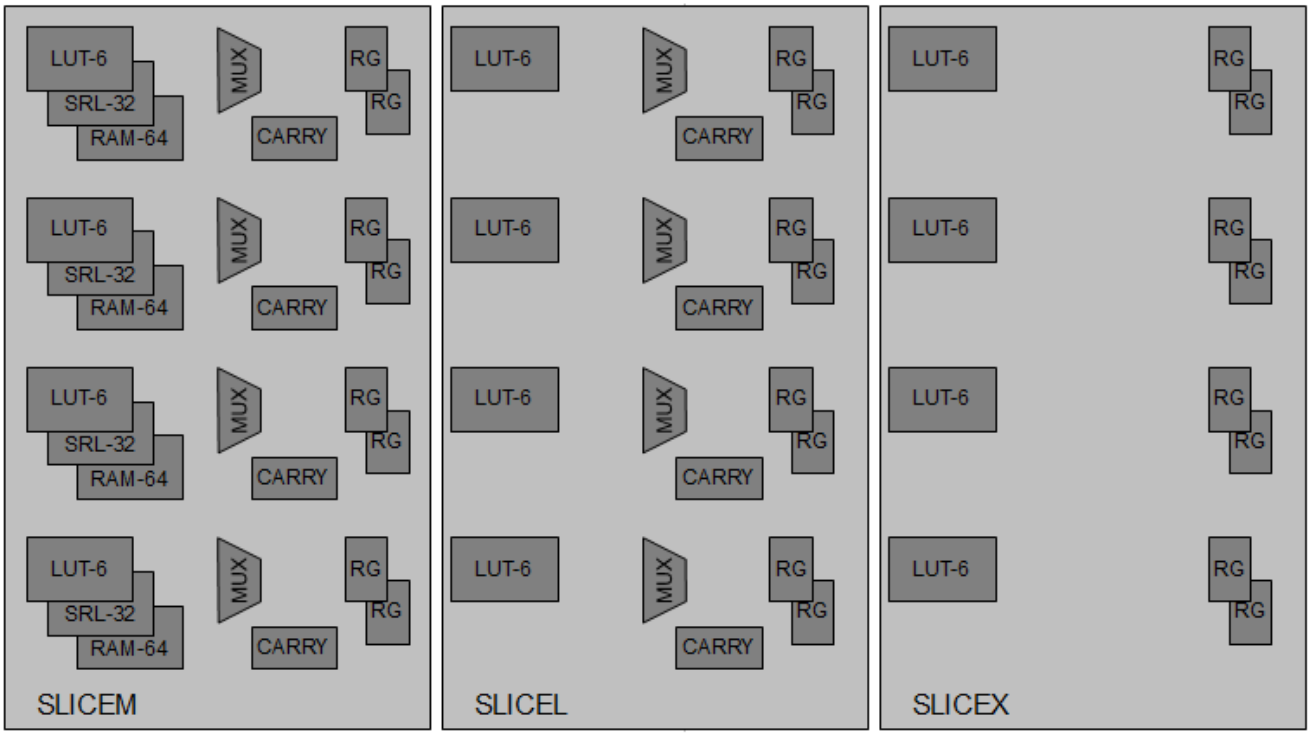


Рисунок 3.3.2 — Архитектура всех типов ячеек Slice FPGA Spartan 6

Как видно из рисунка 3.3.2, любой Slice содержит в своем составе 6-ти входовые LUT. 6-ти входовые LUT могут реализовать любую булеву функцию 6-ти переменных. Мультиплексоры-расширители MUX позволяют объединять выходы таблиц LUT для получения функций 7 и 8 переменных. Это осуществляется

объединением нескольких LUT для построения более сложных логических функций. Кроме этого имеется 8 триггеров, 4 из которых конфигурируются как D-триггеры для хранения результата таблиц LUT, а остальные 4 могут быть сконфигурированы в виде защелок (latch), т.е. триггеров, переключающихся по уровню тактового сигнала. Не смотря на это Xilinx настоятельно не рекомендует использовать latch в проектах для FPGA и применять технику полностью синхронного проекта. Также следует упомянуть, что все триггеры тактируются по переднему фронту тактового сигнала. При необходимости наличия триггеров, тактируемых по заднему фронту, в цепи синхронизации появится инвертор. Кроме этого, в рамках любого проекта можно задать любым триггерам значение по умолчанию. Для более подробной информации по этому вопросу, обращайтесь к документации на среду разработки.

Распределенная память, реализована в виде блоков по 64бит и крайне удобна для реализации небольших буферов или регистровых файлов.

3.4 Дополнительные ресурсы FPGA фирмы Xilinx на примере семейства Spartan 6

В данном разделе представлен обзор всех возможных макро ресурсов семейства Spartan 6, т.е. готовых аппаратных блоков класса «hard-core».

3.4.1 Блочная память (Block RAM)

Блочная память представляет из себя полностью синхронный банк памяти, работающий на частотах до 300 МГц и имеющий объем 18Кбит. Такой банк может быть сконфигурирован в виде однопортовой или двухпортовой памяти. Также банк может быть разделен на два более маленьких банка памяти с емкостью 9Кбит.

3.4.2 Аппаратные блоки умножения

В качестве основного вычислительного ресурса в семействе Spartan 6

выступают блоки DSP-48. Данные блоки представляют собой сверхбыстрые небольшие 18-разрядные сопроцессоры, реализующие каскад операций, таких как: умножение, предварительное суммирование/вычитание и набор логических операций. Структура этих блоков поддерживает их конкатенацию для обработки более широких операндов. Частота работы данных блоков составляет чуть меньше, чем 400Мгц.

3.4.3 Контроллер памяти (Memory Controller Block)

Данные блоки являются некоторым нововведением в семейство Spartan 6 и представляют собой полноценные аппаратные контроллеры для работы со многими типами динамической памяти, широко применяющейся в цифровой технике. Они позволяют с легкостью подключать внешнюю память типа DDR, DDR2, DDR3, LPDDR (Low Power DDR). Задействование данных блоков происходит через утилиту, называемую Core Generator, о которой будет рассказано ниже в этой главе.

3.4.4 Контроллер PCI Express

Еще одним нововведением является наличие в семействе Spartan 6 LXT одного аппаратного контроллера PCI Express v.1.1, который позволяет организовать взаимодействие как с компьютером, так и с другой микросхемой FPGA.

3.4.5 Гигабитные приемопередатчики (Gigabit Transceiver)

Заключительными аппаратными блоками для организации сверхбыстрых коммуникаций, являются специальные гигабитные трансиверы. В FPGA серии Spartan 6 LXT содержится от 2 до 8 таких приемопередатчиков. Каждый такой блок позволяет организовать передачу данных по последовательному дифференциальному интерфейсу со скоростью до 3.125Гбит/с.

3.4.6 Блоки ввода-вывода

Каждый сигнал, выходящий наружу FPGA содержит на выходе микросхемы блок ввода-вывода. Данные блоки обладают богатым набором функционала, а именно: реализуют буферы с 3-м состоянием, позволяют вносить некоторую, программируемую задержку в выходной сигнал (IODELAY), имеют в своем составе сериализаторы/десериализаторы (IOSERDES) с распараллеливанием до 4:1 и каскадированием до 8:1, а также триггеры (IOLOGIC).

3.4.7 Clock Management Tile (CMT)

Данные блоки представляют собой сложные цифро-аналоговые схемы по управлению тактовыми сигналами. С их помощью можно создать тактовый сигнал требуемой частоты на основании некоторого опорного тактового сигнала. Кроме того, можно задать требуемый фазовый сдвиг.

Для семейства Spartan 6, данные блоки состоят из двух DCM, являющимися цифровыми блоками подстройки, а также из одного PLL, представляющего аналоговую часть блока. При этом DCM имеют большую погрешность, но способны синтезировать требуемую частоту в широких пределах, а PLL является более точными схемами, с помощью которых можно выполнять более сложную настройку частоты, а также убирать jitter, т.е. дрожание тактового сигнала.

3.5 Понятие глобальных временных ограничений

При реализации проектов в цифровой технике всегда приходится соблюдать баланс между производительностью и занимаемой площадью. При этом, зачастую, требования по производительности оказываются более предпочтительными.

Требуемая производительность цифрового устройства в рамках некоторого проекта может быть достигнута как выбором и реализации более удачной архитектуры, так и повышением частоты работы модулей. Поскольку частота работы устройства определяется наиболее длинным участком цепи, вносящим

наибольшую задержку, то для увеличения частоты работы проекта требуется максимально сократить значение этой задержки. Это можно осуществить двумя вариантами: путем изменения и оптимизации поведения цифровой схемы или путем повторного размещения и трассировки элементов внутри целевой FPGA.

Как было указано выше, для размещения и разводки элементов внутри кристалла FPGA используются соответствующие средства САПР. Для фирмы Xilinx таким средством является утилита PaR. Данная утилита поддерживает возможность задания пользовательских ожиданий по частоте. Эти ожидания и называются глобальными временными ограничениями.

Существует четыре главных типа глобальных временных ограничений:

- Ограничения на период распространения сигнала (PERIOD). Данное ограничение включает в себя распространение сигнала от одного триггера до другого, дрожание тактового сигнала (jitter) и разброс в тактировании всех триггеров в системе;
- Входное смещение (OFFSET IN). Описывает задержку сигнала от внешнего входа микросхемы до первого триггера на пути следования сигнала;
- Выходное смещение (OFFSET OUT). Описывает задержку сигнала от первого триггера на пути следования сигнала до внешнего выхода микросхемы;
- Задержка от входа к выходу (PAD to PAD). Определяет задержку сигнала от входа микросхемы до его выхода, при этом на пути следования сигналов не должно быть триггеров.

На рисунке 3.5.1 приводится наглядное представление всех типов глобальных временных ограничений.

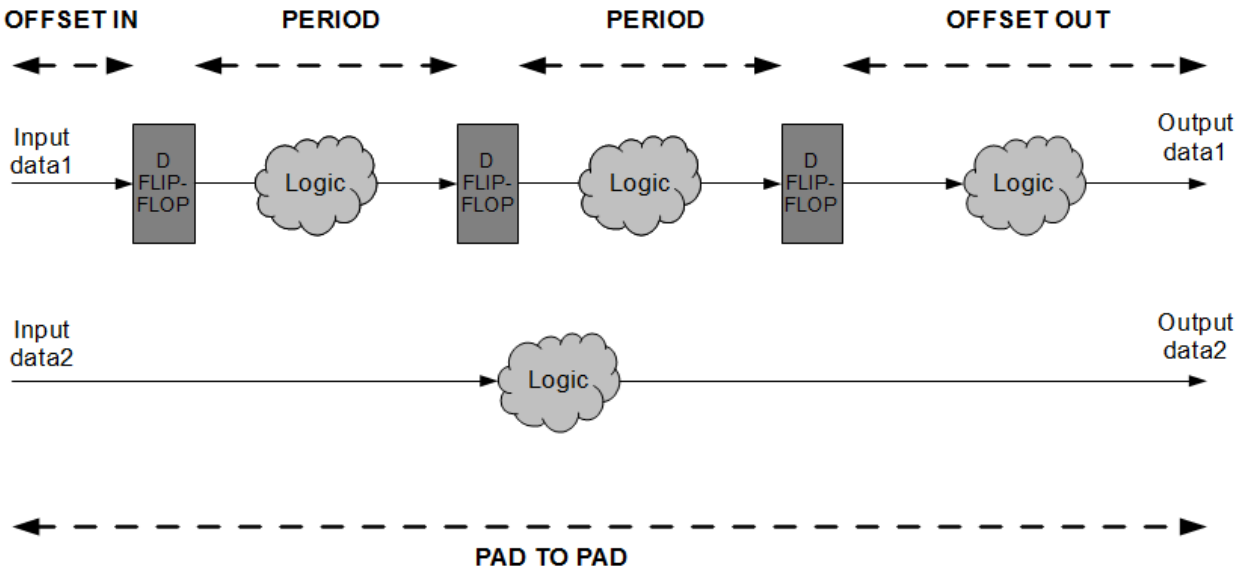


Рисунок 3.5.1 — Представление глобальных временных ограничений

Временные ограничения описываются либо с помощью обычного текстового редактора, либо с использованием специальной утилиты от Xilinx – Timing Constraints Editor.

При наличии временных ограничений средства САПР будут пытаться реализовать их, что, возможно, приведет к увеличению времени, которое будет потрачено на размещение и разводку элементов. Тем не менее, применение данной техники даст существенно лучшие результаты по быстродействию, которое является одной из главных целей при разработке цифровых схем.

3.6 Генерация IP-блоков

При реализации больших проектов для FPGA может возникнуть мысль о использовании законченных блоков, написанных другими разработчиками. Фирма Xilinx обладает своим собственным репозиторием таких модулей, являющихся полностью оттестированными и готовыми к использованию ядрами класса “soft-core”. Подобное ядро представляют собой черный ящик, реализующий требуемые функции. Для многих ядер можно изменить настройки конфигурации, такие как ширину входных и выходных сигналов, методы обработки информации и т.д., что

позволяет получить конкретное решение под конкретную задачу.

Все подобные IP-блоки от Xilinx можно разделить на 2 группы: решения от LogiCore и решения от AllianceCore. При этом решения от LogiCore поддерживаются непосредственно фирмой Xilinx, имеют расширенный набор параметров, полностью задокументированы и зачастую не требуют покупки специальных лицензий. В свою очередь, решения от AllianceCore поддерживаются партнерами Xilinx, требуют покупки дополнительных лицензий и часто не имеют возможности к параметризации.

Для централизованного доступа и конфигурации блоков различных реализаций и типов, Xilinx предоставляет утилиту, называемую Core Generator. Данная утилита имеет удобный графический интерфейс и позволяет просматривать имеющиеся для использования модули. Также эта утилита позволяет использовать дополнительные ресурсы FPGA, реализованные в виде IP-ядер класса «hard-core», таких как: блочная память, CMT, DSP48, контроллеров динамической памяти и т.д.

В заключении, стоит отметить, что при использовании таких блоков, вы получаете ряд преимуществ:

- Существенно экономите время проектирования;
- Получаете гарантированно работающие модули для построения системы;
- Получаете возможность предсказать предельную частоту проекта, основываясь на поставляемой документации к каждому блоку.

Все эти плюсы говорят о существенной роли подобных блоков в проектировании цифровых устройств для FPGA.

3.7 Моделирование проектов. Обзор ISIM

Поведенческое моделирование проектов осуществляется для реализации

процесса функционального тестирования. При этом, в отличии от обычных программ, написанных на языках высокого уровня, требующих для оценки своей работы лишь компиляции, линковки и запуска, для описания цифровых схем на языках HDL требуется реализация событийного моделирования, требуемого для эмуляции параллельно работающей аппаратуры на последовательно работающем компьютере, на котором осуществляется процесс моделирования. Это приводит к появлению специальных программных пакетов, реализующих данную концепцию.

Фирма Xilinx располагает собственным пакетом, для моделирования проектов, которым является ISIM. ISIM является программным пакетом средней сложности, поддерживающим стандарты языков Verilog HDL и VHDL и имеющим довольно неплохие возможности по оценке корректности работы проекта. Программный пакет также содержит расширенный инструментарий для работы с временными диаграммами.

Стоит также отметить, что хотя средства моделирования, поддерживаемые фирмами производителями FPGA зачастую не являются лидерами на рынке, они являются крайне приемлемыми для небольших проектов.

3.8 Внутрисхемная отладка проектов. Обзор ChipScope

Наряду с поведенческим моделированием, в индустрии, для отладки проектов применяют техники по внутрисхемной отладке. Данный метод отладки представляет собой проверку корректности работы созданного проекта на реально работающем кристалле FPGA.

В своем стандартном исполнении, данная методика предполагает вывод проверяемых сигналов на внешние выводы микросхемы, к которым подключаются средства для анализа сигналов, такие как: осциллографы, логические анализаторы и т.д. На рисунке 3.8.1 представлена структурная схема, описывающая данный подход. При этом возникает ряд серьезных проблем:

- Нужного количества свободных внешних выводов может не оказаться;
- Трассировка требуемых сигналов к выводам FPGA может привести к абсолютно другому расположению проекта внутри микросхемы, что может привести к исчезновению проблемы или появлению новой;
- Данный подход имеет существенные ограничения по анализу внутренней работы FPGA.

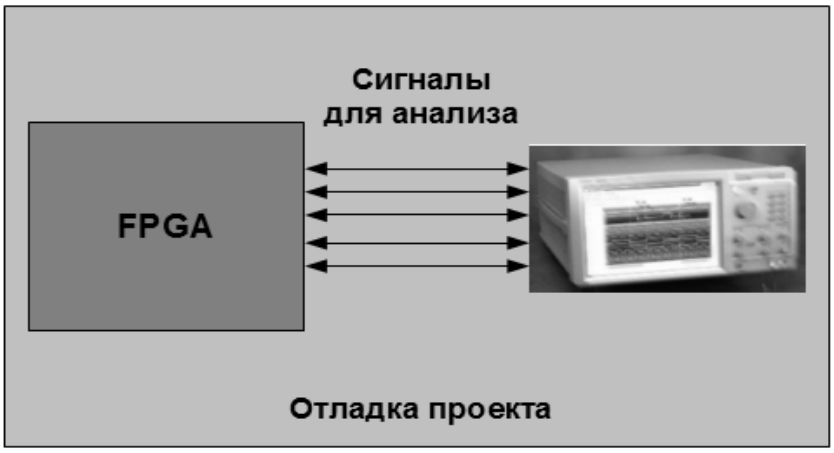


Рисунок 3.8.1 — Структурная схема обычной внутрисхемной отладки

Для решения этого ряда проблем, фирмой Xilinx был создан несколько отличный подход, основывающийся на реализации в FPGA аппаратных блоков, облегчающих процесс отладки. Весь процесс основан на записи активности требуемых для анализа сигналов в течение некоторого промежутка времени и пересылки этих данных по стандартному порту для отладки и конфигурации. Этот порт называется JTAG и имеется в любом кристалле FPGA. Кроме этого, для облегчения восприятия полученной информации, реализован программный пакет, позволяющий наблюдать активность сигналов в виде временных диаграмм. На рисунке 3.8.2 представлена структурная схема реализации внутрисхемной отладки от Xilinx.

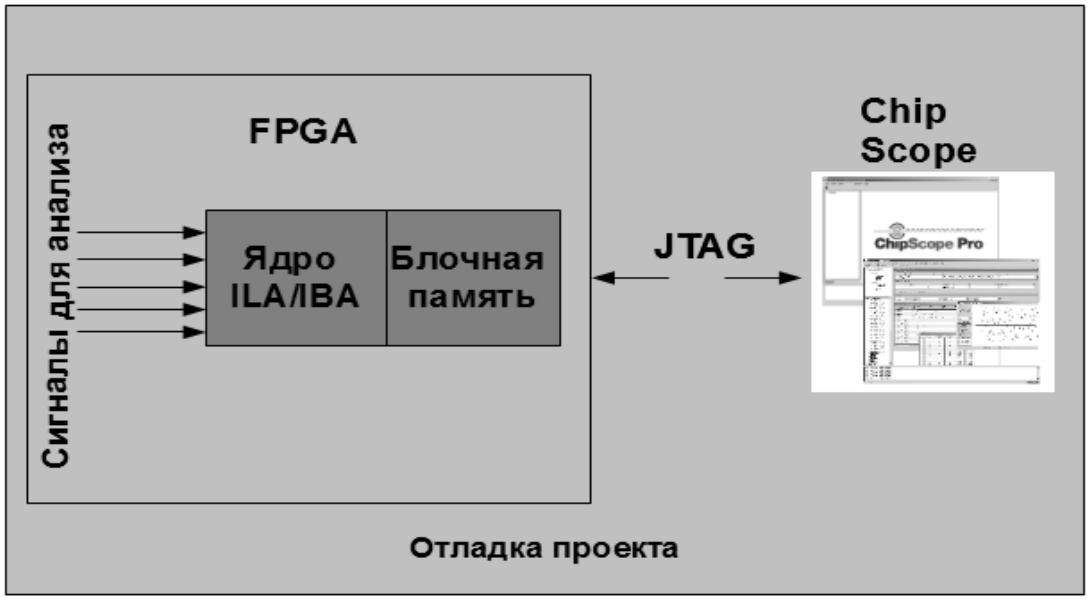


Рисунок 3.8.2 — Структурная схема реализации внутрисхемной отладки от Xilinx

Использование данного подхода позволяет убрать все ограничения и проблемы, рассмотренные ранее. Кроме этого, данный подход позволяет отказаться от дополнительных средств анализа выходных сигналов.

3.9 Обзор микроконтроллеров класса «soft-core»

Кроме всего многообразия поддерживаемых Xilinx средств для проектирования и отладки, облегчающих все этапы маршрута проектирования, фирмой Xilinx также поддерживаются два типа микроконтроллеров, реализованных в виде закрытых HDL описаний, а также описаний на уровне логических примитивов FPGA.

3.9.1 PicoBlaze

PicoBlaze (KCPSM3) представляет собой крошечный 8-ми разрядный микроконтроллер, доступный для использования в семействах FPGA Spartan 3 и Spartan 6, служащий для замены сложных конечных автоматов более простой программной реализацией. При этом данный микроконтроллер использует один единственный внешний банк блочной памяти, рассчитанный не более чем на 1024

инструкции.

При реализации на FPGA, данный блок занимает всего лишь порядка 100 LUT. Тем не менее, не смотря на свои размеры, данный блок является полноценным контроллером, имеющим расширенный набор команд и способный обрабатывать внешние прерывания. PicoBlaze имеет в своем составе 2-х ступенчатый конвейер, позволяющий выполнить абсолютно любую инструкцию за 2 такта. Кроме этого данный блок имеет свой собственный компилятор и соответствующий язык ассемблер. Структурная схема микропроцессора PicoBlaze представлена на рисунке 3.9.1.

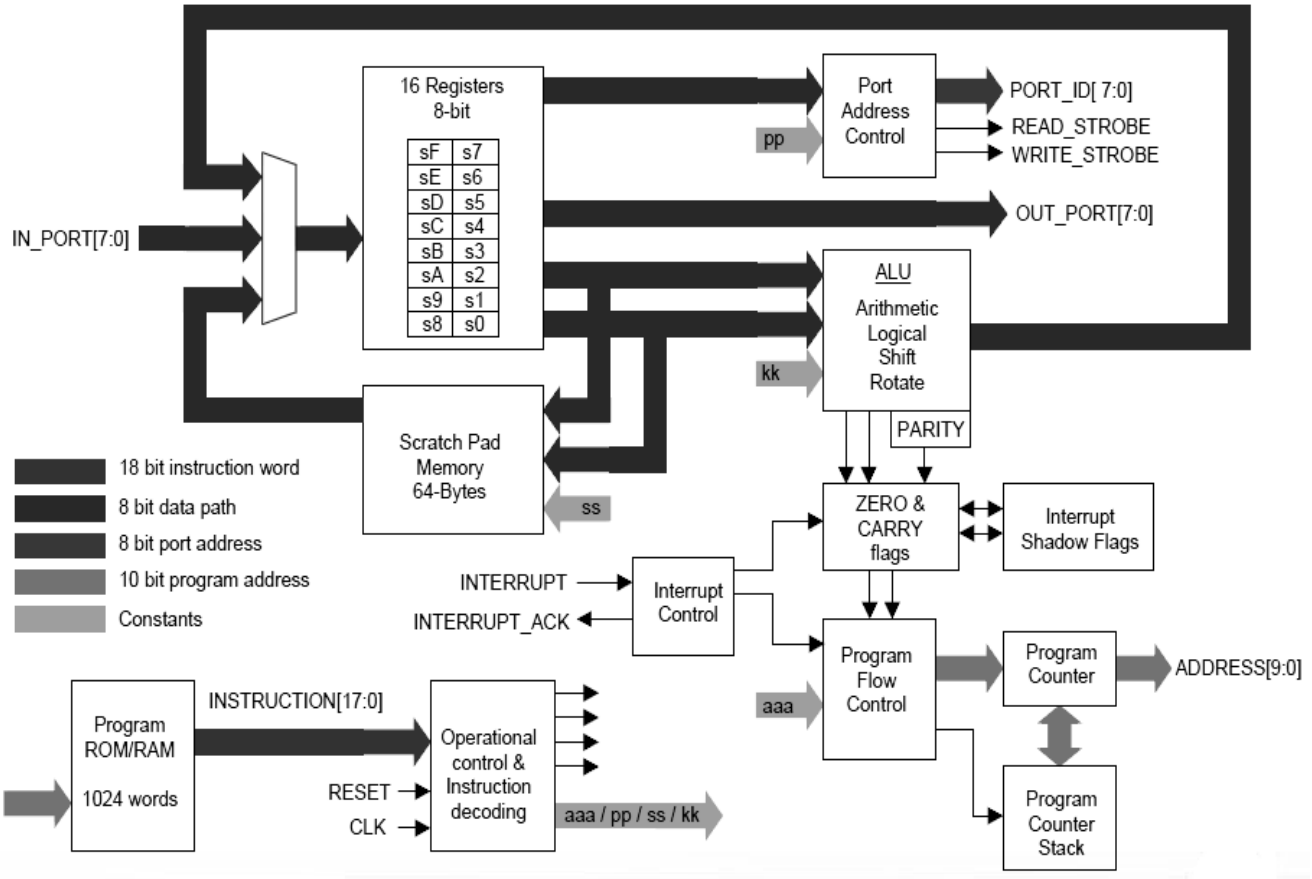


Рисунок 3.9.1 — Структурная блок-схема микропроцессора PicoBlaze

3.9.2 MicroBlaze

MicroBlaze представляет собой высокопроизводительный конфигурируемый 32-х разрядный процессор, основанный на RISC архитектуре. Процессор обладает

конвейером глубиной от 3-х до 5-ти тактов, опциональным кэшем, блоком MMU для управления виртуализацией памяти и т.д. Набор настроек позволяет оптимизировать процессор по занимаемым ресурсам и производительности в зависимости от требуемой задачи. Также имеется простой интерфейс подключения внешних IP-блоков и сопроцессоров, позволяющих аппаратно реализовывать критические для производительности функции. Кроме этого, в связи с тесным сотрудничеством между компаниями Xilinx и ARM, данный микроконтроллер обладает поддержкой высокопроизводительной шины AXI, применяемой в контроллерах с архитектурой ARM, что облегчает использование данного контроллера и его интеграцию с блоками, разработанными сторонними производителями.

Без блока управления памятью (MMU) на MicroBlaze могут работать операционные системы с упрощенной защитой и виртуальной памятью, такие как uClinux или FreeRTOS. С блоком управления памятью возможна работа операционных систем, требующих аппаратной поддержки страничной организации и защиты памяти, таких как Linux.

4. Проектирование аппаратуры с использование языка Verilog HDL

Прежде всего хотелось бы отметить, что обсуждение языка Verilog HDL в данной книге носит скорее вводный характер и не претендует на раскрытие всех конструкций языка. Вместо этого, разделы, описывающие Verilog HDL должны дать вам некоторое понимание в непосредственном процессе проектирования и помочь вам ощутить на простых примерах разницу между написанием программ на языках высокого уровня и описанием работы цифровой аппаратуры.

Не смотря на это, обсуждение языка Verilog HDL предполагает у читателя базовые знания и понимание принципов написания программ на языках высокого уровня. Такие вещи как комментарии, идентификаторы, чувствительность языка к регистру и т.д. не будут подробно освещаться. Мы просто отметим, что язык Verilog HDL чувствителен к регистру, имеет структуру комментариев и ограничения по построению идентификаторов, аналогичные языку C.

За более подробной информацией по Verilog HDL обращайтесь к списку рекомендуемой литературы, представленному в конце книги и особенно к стандарту языка.

4.1 Понятие модуля. Базовая структура модуля

В языке Verilog HDL главным строительным элементом описания поведения проектируемого устройства является модуль. Модуль представляет собой законченный блок, описывающий некоторую часть проекта. Любой модуль на языке Verilog HDL строится по шаблону, представленному на рисунке 4.1.1. Он состоит из трех частей: объявление портов ввода-вывода, объявление внутренних сигналов и тело модуля, описывающее логику работы.

```
// ****************************************
// Базовая структура модуля на Verilog HDL
// ****************************************

module simple_module
                // Описание портов ввода-вывода
                (
                    input wire [3:0] input_port,
                    ...
                    output wire output_port
                );

                // Описание локальных сигналов
                reg [7:0] signal1, signal2;
                wire [3:0] signal3;
                ...

                // Описание тела модуля
                assign signal3 = input_port;
                ...

endmodule : simple_module
```

Рисунок 4.1.1 — Базовая структура модуля на языке Verilog HDL

4.1.1 Объявление портов ввода-вывода.

Объявление портов ввода-вывода определяет режимы, типы данных и имена портов ввода-вывода модуля. Упрощенный синтаксис данного объявления представлен на рисунке 4.1.2.

```
// ********************************
// Объявление портов ввода-вывода
// ********************************

module module_name
            (
                [mod] [data_type] [capacity] [port_name],
                [mod] [data_type] [capacity] [port_name],
                ...
                [mod] [data_type] [capacity] [port_name]
            );
```

Рисунок 4.1.2 — Объявление портов ввода-вывода

- [mod] — может быть **input**, **output** или **inout**, что соответственно определяет

входной, выходной или двунаправленный порт;

- [data_type] — тип может быть **wire** или **reg**. Если это **wire**, то он может быть опущен;
- [capacity] – разрядность сигнала или переменной. Описывается в виде [x:y], где x – целое число, описывающее старший разряд, а y – целое число, описывающее младший разряд;
- [port_name] — название порта ввода-вывода.

4.1.2. Объявление внутренних сигналов, параметров или переменных.

Данная часть модуля служит для объявления сигналов, параметров или переменных, используемых в модуле. Внутренние сигналы могут, например, быть представлены соединением проводов (**wire**) между блоками схемы.

Упрощенный синтаксис раздела объявления представлен на рисунке 4.1.3.

```
// ******************************
// Объявление внутренних сигналов,
// параметров или переменных
// ******************************

    [data_type] [capacity] [data_name], [data_name], ...;
    [data_type] [capacity] [data_name];
```

Рисунок 4.1.3 — Объявление внутренних сигналов, параметров или переменных модуля

- [data_type] – тип сигнала, параметра или переменной. Может принимать такие значения как **wire**, **reg**, **integer** и т.д.;
- [capacity] – разрядность сигнала, переменной или параметра. Может принимать такие значения как [7:0], [63:0], [2:28] и т. д. В случае использования типа переменной (например **integer**) данное поле опускается, т. к. каждый тип переменных имеет предопределенную размерность (в

случае **integer** это 32 бита). Исключением является тип переменной **reg**, который может иметь любую разрядность;

- [data_name] – название сигнала, параметра или переменной.

Если объявление сигнала было опущено, то оно предполагается неявным соединением. Тип по умолчанию – **wire**. Для ясности кода лучше всегда избегать неявного объявления.

4.1.3. Тело модуля

Тело модуля содержит описание его поведения. В отличии от программ на языке С, в которых операторы выполняются последовательно, тело модуля, описанное на языке Verilog HDL, может быть представлено набором блоков. Эти блоки функционируют параллельно и выполняются одновременно.

Конструкции языка для описания тела модуля будут рассмотрены в следующих разделах этой главы.

4.2 Типы данных

Разнообразие и особенности типов данных являются ключевыми моментами при описании цифровых устройств. Стоит отметить, что в отличие от языков высокого уровня, в большинстве типов данных языка Verilog HDL используются четыре базовых значения:

- **0.** Логический 0 или true;
- **1.** Логическая 1 или false;
- **z.** Высокоимпедансное состояние или разрыв цепи;
- **x.** Неопределенное состояние или неинициализированное значение.

Кроме того, условно, можно разделить типы данных языка на 2 группы: переменные и сигналы. Сигналы моделируют межсоединения между цифровыми

схемами. К сигналам относятся такие типы данных, как: **wire**, **wand**. Переменные представляют собой как обычные переменные, используемые в языках программирования, так и абстрактные хранилища информации при поведенческом моделировании. К переменным относятся следующие типы: **reg**, **integer**, **real**, **realtime** и т. д. При этом не следует путать тип переменной **reg** с физическим регистром (именно поэтому в языке приемнике SystemVerilog, данный тип данных переименовали в logic).

Также следует отметить способы описания целых чисел. Verilog HDL поддерживает несколько форматов написания целых чисел. Упрощенный синтаксис объявления целого числа приведен на рисунке 4.2.1.

```
// ********************************
// Объявление целого числа
// ********************************

    [sign] [size] ' [base], [value]
```

Рисунок 4.2.1 - Упрощенный синтаксис объявления целого числа

- [sign] – знак;
- [size] – количество бит в числе;
- [base] – основание числа. Возможно использовать: b/B (двоичное), o/O (восьмиричное), h/H (шестнадцатиричное), d/D (десятичное);
- [value] – значение числа с учетом выбранного основания.

Также возможно задавать целые числа привычным способом с использованием обычных чисел с десятичным основанием (241, -5 и т.д.). При этом необходимо помнить, что такое число будет расширено минимум до 32-х бит.

4.3 Реализация комбинационных схем вентильного уровня

Любой модуль представляет собой описание логики работы некоторого блока цифрового устройства. В самом простейшем случае, логика работы некоторого

блока состоит только из комбинационных схем — т.е. схем, представляющих собой физическую реализацию некоторой логической функции.

В качестве примера рассмотрим однобитный компаратор, определяющий равенство двух входных сигналов, input1 и input2, и одним выходом, equal. Сигнал equal устанавливается, когда input1 и input2 равны. Таблица истинности этой схемы представлена на рисунке 4.3.1.

Входные сигналы		Выходной сигнал
input1	input2	equal
0	0	1
0	1	0
1	0	0
1	1	1

Рисунок 4.3.1 — Таблица истинности однобитного компаратора

Предполагается, что для реализации этого примера мы хотим использовать базовые логические вентили, такие как: **not**, **and**, **or** и **xor**. Логическое выражение, описывающее подобный однобитный компаратор представлено на рисунке 4.3.2.

$$equal = input1 \cdot input2 + \neg input1 \cdot \neg input2$$

Рисунок 4.3.2 — Логическое выражение, описывающее 1-битный компаратор

На рисунке 4.3.3 представлен один из возможных вариантов описания поведения данного блока на языке Verilog HDL.

```
// ****************************************
// Пример реализации однобитного компаратора
// ****************************************

// Описание модуля
module simple_comparer
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output equal
                );

                // Описание локальных сигналов
                wire local1, local2;

                // Тело модуля
                assign equal = local1 | local2;

                assign local1 = input1 & input2;

                assign local2 = ~input1 & ~input2;

endmodule : simple_comparer
```

Рисунок 4.3.3 — Вариант описания 1-битного компаратора на языке Verilog HDL

Для успешного описания схем на языках HDL требуется мыслить терминами, относящимися к аппаратуре. Реализация данного примера состоит из трех частей, описанных в предыдущем разделе. В первой части, описываются порты ввода-вывода этой схемы, которыми являются input1, input2 и equal. Во второй части модуля объявляются внутренние соединяющие сигналы, которыми являются local1 и local2. И, наконец, тело модуля описывает внутреннюю структуру схемы, содержащую три непрерывных присваивания (**assign**). Каждое такое присваивание представляет собой часть схемы, которая выполняет определенную простую логическую операцию.

Графическое представление этого модуля представлено на рисунке 4.3.4. Три непрерывных присваивания составляют три блока схемы. Соединения между

этими блоками указаны с помощью сигналов и имен портов.

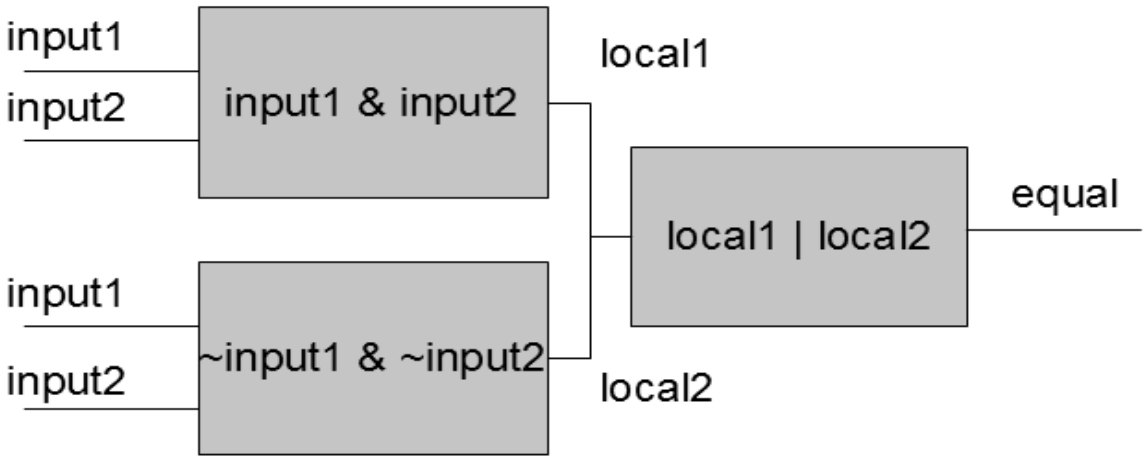


Рисунок 4.3.4 — Графическое представление 1-битного компаратора

Далее стоит сосредоточиться на описании поведения модуля — т.е. на теле модуля. Есть несколько способов описания поведения модуля. В связи с этим в описании используются несколько приемов, таких как:

- непрерывное присваивание (**assign**);
- создание экземпляра модуля (включение подмодуля);
- блок **always.**

Описание блоков с помощью непрерывного присваивания полезно для простых комбинационных схем. Любое непрерывное присваивание может быть представлено в виде блока. Сигнал, находящийся в левой части выражения является выходным, сигналы, находящиеся в правой части выражения описывают некоторую логику работы с использованием логических операторов, таких как **|** (или), **&** (и), **~** (не) и т.д.

Понятие модуля является схожим с понятием класса в объектно-ориентированных языках программирования, таких как C++. При этом модуль, как и класс представляет собой некоторый шаблон для создания экземпляров данного модуля. Т.е. само по себе описание модуля (за исключением модуля верхнего

уровня) не представляет из себя никакой реальной части разрабатываемого устройства. Для использования модуля требуется создать его экземпляр внутри другого модуля и осуществить коммутацию его входных и выходных сигналов. При этом, можно сравнить описание модуля с чертежом здания, а созданный экземпляр модуля с конкретной постройкой, выполненной по этому чертежу. Данная техника позволяет нам создавать устройство, состоящее из множества маленьких частей, путем объединения предварительно разработанных модулей в общую систему. Данная техника будет рассмотрена далее в этой главе.

Блок **always** представляет собой контейнер для более абстрактных процедурных присваиваний. Данная техника используется для описания более высокоуровневых конструкций. Блок **always** будет рассмотрен в последующих разделах.

4.4 Реализация модульной структуры проекта

Модульная структура проекта поддерживается непосредственно конструкциями самих языков HDL. Создадим в качестве примера модульный проект для блока из предыдущего раздела, разбив его на три модуля.

Описание этих трех подмодулей представлено на рисунках 4.4.1 — 4.4.3.

```
// *****************
// Первый подмодуль
// *****************

// Описание модуля
module first_submodule
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output sub_output
                );

                assign sub_output = input1 & input2;

endmodule : first_submodule
```

Рисунок 4.4.1 — Реализация первого подмодуля

```
// *****************
// Второй подмодуль
// *****************

// Описание модуля
module second_submodule
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output sub_output
                );

                assign sub_output = ~input1 & ~input2;

endmodule : second_submodule
```

Рисунок 4.4.2 — Реализация второго подмодуля

```
// *****************
// Третий подмодуль
// *****************

// Описание модуля
module third_submodule
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output sub_output
                );

                assign sub_output = input1 | input2;

endmodule : third_submodule
```

Рисунок 4.4.3 — Реализация третьего подмодуля

После создания всех трех модулей требуется описать модуль верхнего уровня, который будет являться наивысшим в иерархии проекта и создавать экземпляры описанных нами подмодулей. Модуль верхнего уровня представлен на рисунке 4.4.4.

Как видно из рисунка 4.4.4 для создания экземпляров модулей используются новые конструкции языка Verilog HDL. Для начала, нужно отметить, что любое создание экземпляра начинается с имени модуля, экземпляр которого вы хотите создать. Далее идет произвольное название экземпляра модуля, за которым следует подключение его портов ввода-вывода. В данной книге мы будем использовать подключение портов ввода-вывода через «.», называющееся подключением по имени. В действительности, существует также подключение по порядку следования сигналов в описании модуля, но при использовании этой техники теряется наглядность кода, кроме того становится гораздо легче сделать ошибку в описании, т.к. В процессе разработки порядок следования сигналов в модуле может измениться.

```
// ***********************
// Модуль верхнего уровня
// ***********************

// Описание модуля
module simple_comparer
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output equal
                );

                // Описание локальных сигналов
                wire sub_output1, sub_output2;

                // Создание экземпляра подмодуля
                // и его подключение
                first_module module1(
                    .input1(input1),
                    .input2(input2),
                    .sub_output(sub_output1)
                );
                // Создание экземпляра подмодуля
                // и его подключение
                second_module module2(
                    .input1(input1),
                    .input2(input2),
                    .sub_output(sub_output2)
                );
                // Создание экземпляра подмодуля
                // и его подключение
                third_module module3(
                    .input1(sub_output1),
                    .input2(sub_output2),
                    .sub_output(equal)
                );

endmodule : simple_comparer
```

Рисунок 4.4.4 — Реализация модуля верхнего уровня

4.5 Конструкции для реализации комбинационных схем уровня RTL

До этого момента мы описывали поведение простых комбинационных схем, используя при этом простые однобитовые операторы. В данном разделе мы переключимся на более высокоуровневое, RTL, описание для разработки

компонент среднего размера, таких как мультиплексоры, сдвигатели, умножители и т.д.

4.5.1 Операторы

Язык Verilog HDL состоит из множества операторов. В дополнении к поразрядным операторам, рассмотренным ранее, существуют также операторы, позволяющие осуществлять арифметические операции, операции сдвига, относительные операции и т.д.:

- **Арифметические операторы**. Существует 6 арифметических операторов: +, -, *, /, %, **. Операторы + и - в процессе логического синтеза трансформируются в сумматоры и вычитатели, построенные на логических ячейках FPGA. Операция умножения в разработке аппаратного обеспечения не является тривиальной операцией, поэтому успешность логического синтеза данной конструкции зависит от среды проектирования и ресурсов FPGA. Оставшиеся операторы как правило не могут быть синтезированы автоматически;

- **Операторы сдвига**. Существует четыре операторов сдвига : >>, <<, >>>, <<< . Первые два представляют логический сдвиг вправо и влево, последние два представляют арифметический сдвиг вправо и влево. Если оба операнда в операторе сдвига являются сигналами, a << b , то оператор является циклическим сдвиговым устройством (barrel shifter), которое является довольно сложным устройством. Если же количество сдвигов фиксировано, a << 2, то такая конструкция в процессе синтеза будет представлять из себя схему разводки;

- **Относительные и тождественные операторы.** Данная группа операторов аналогична операторам языка C: >, < , <= , >=, ==, !=. Эти операторы

сравнивают два операнда и возвращают булевский результат, который может быть false (0) или true (1). Данная группа операторов, после прохождения процесса логического синтеза, будет представлять из себя компараторы;

- **Операторы объединения и дублирования**. Оператор объединения (или конкатенации), **{}**, объединяет несколько сигналов в один. Например, конструкция {2'b11, 2'b00} объединяет два 2-х разрядных сигнала в один 4-х разрядный. При синтезе данная структура будет заменена простой схемой разводки. Оператор объединения, N{ }, дублирует один и тот же сигнал несколько раз. Константа, N, определяет число дублей. Например, {2{2'b00}} возвращает 4'b0000;

- **Условные операторы**. Условные операторы, **?:** и конструкция **if-then-else**, полностью аналогичны одноименным конструкциям языка C. В процессе логического синтеза данные операторы представляются 2-в-1 мультиплексорами. Условные операторы также могут быть вложенными, что приводит к появлению каскада приоритетных мультиплексоров;

- **Оператор case**. Данный оператор также аналогичен оператору **case** в языке C. Отметим, что в отличии от простых мультиплексоров 2-в-1, в процессе логического синтеза данный оператор представляется большим мультиплексором с описанным числом выборов. Данный оператор особенно удобен при описании множественного выбора, т.к. в случае описание такой схемы средствами обычных условных операторов, получается длинная приоритетная схема, что выливается в появление большой задержки. Данная ситуация представлена на рисунке 4.5.1.

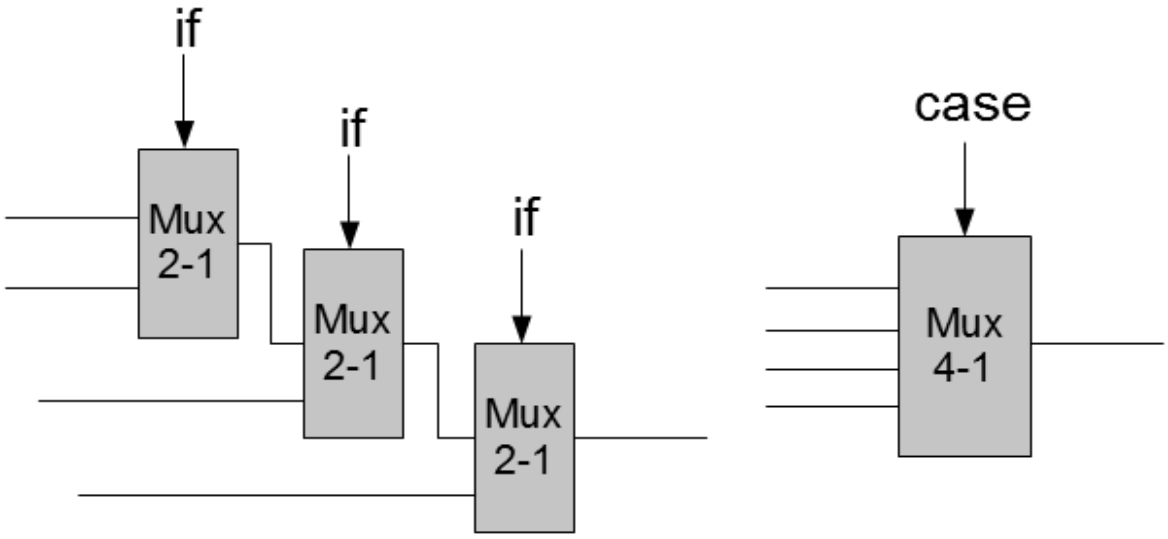


Рисунок 4.5.1 — Разница в представлении обычных условных операторов **if** и оператора **case**

Также при использовании условных операторов или оператора **case** для описания комбинационных схем необходимо запомнить, что нужно всегда реализовывать альтернативный выбор (т.е. не опускать конструкции **else** или **default** в соответствующих операторах). Это позволит вам избежать предупреждений и ошибок на этапе синтеза, связанных с появлением нежелательных защелок (latch) в проекте.

4.5.2 Блок always

Блок **always** служит для упрощения описания цифровой аппаратуры. Как было отмечено ранее, в теле модуля все конструкции выполняются параллельно, что соответствует реальному функционированию аппаратуры. Однако блок **always** позволяет отойти от этого правила и выполнять операторы последовательно. Несмотря на это, блок **always** может быть успешно синтезирован, если в процессе его описания разработчик будет следовать некоторым правилам и шаблонам, с другой стороны, использование данной техники может привести к ненужному усложнению реализации или ошибкам на стадии синтеза.

Блок **always**, описывающий поведение комбинационной схемы, может быть

представлен как черный ящик, чье поведение представлено операторами, описанными внутри данной конструкции языка. Кроме этого, для описания такого блока используется еще одна конструкция, называющаяся блокирующим присваиванием.

Упрощенный синтаксис блока **always** представлен на рисунке 4.5.2.

```
// ****************************
// Шаблон описания блока always
// ****************************

always @([sens_list])
    begin : [label]
        // Декларация локальных переменных
        [data_type] [data_name];
        ...
        // Тело блока
        [proc_statement];
        ...

    end : [label]
```

Рисунок 4.5.2 — Упрощенный синтаксис блока **always**

- [sens_list] – Список сигналов и событий, на которые реагирует постоянный блок. Это те события, при наступлении которых always блок начинает свою работу и выполняет внутренние процедурные операторы. Для комбинационных схем все входные сигналы должны быть включены в этот лист, для чего используется символ «*». Тело составлено из некоторого набора процедурных операторов. Разделители **begin** и **end** могут быть пропущены, если в теле имеется только один оператор. Можно также сказать, что такой блок представляет собой бесконечный цикл;
- [label] – Данная конструкция служит для облегчения чтения кода и является опциональной;
- [data_type] – Тип сигнала, переменной и т.д. При этом, стоит отметить, что в блоке always нельзя использовать тип сигнала **wire**. Вместо этого нужно

пользоваться типом сигнала **reg;**

- [data_name] – Название сигнала, переменной и т.д.;
- [proc_statement] – Процедурное присваивание.

Процедурное присваивание может быть использовано только внутри блока **always** или блока **initial** (о нем позже). Есть два типа присваивания: блокирующее присваивание и не блокирующее присваивание. Их базовый синтаксис представлен на рисунке 4.5.3:

```
// ****************************
// Шаблон описания присваиваний
// ****************************
// Неблокирующее присваивание
[variable_name] <= [expression];

// Блокирующее присваивание
[variable_name] = [expression];
```

Рисунок 4.5.3 — Базовый синтаксис блокирующего и неблокирующего присваиваний

- [variable_name] – Название сигнала или переменной;
- [expression] – Некоторое выражение.

В блокирующем присваивании выражение вычисляется, а затем немедленно присваивается переменной, до того как следующее выражение будет вычислено. Оно ведет себя точно так же как нормальное присваивание переменной в языке C. В не блокирующем присваивании, рассчитанное выражение присваивается после вычисления всех не блокирующих присваиваний в конце текущего временного интервала моделирования (присваивания таким образом не блокируют вычисления в других присваиваниях).

Для выбора нужного типа процедурного присваивания можно пользоваться простыми правилами:

- для комбинационных схем, используйте блокирующее присваивание;
- для последовательных схем, используйте не блокирующее присваивание.

4.6 Реализация комбинационных схем уровня RTL

Ниже, будет представлен ряд примеров, основывающихся на предыдущем разделе. Эти примеры должны помочь вам получить наглядное представление о рассмотренных ранее конструкциях языка Verilog HDL.

4.6.1 Модифицированное описание 1-битного компаратора

На рисунке 4.6.1 представлено модифицированное описание 1-битного компаратора, рассмотренного в предыдущих разделах.

В данном случае мы используем более высокоуровневые конструкции языка, чтобы улучшить наглядность кода, такие как блок **always**, относительный и условный операторы. Тем не менее, в качестве результата мы получаем абсолютно аналогичный 1-битный компаратор.

```
// *************************
// Модифицированное описание
//  1-битного компаратора
// *************************

// Описание модуля
module rtl_comparer
                // Описание портов ввода-вывода
                (
                    input input1,
                    input input2,
                    output equal
                );

                // Описание вспомогательных сигналов
                reg local_equal;

                // Описание тела модуля
                always @*
                    begin
                        if (input1 == input2)
                            local_equal = 1'b1;
                        else
                            local_equal = 1'b0;
                    end

                assign equal = local_equal;

endmodule : rtl_comparer
```

Рисунок 4.6.1 — Модифицированное описание 1-битного компаратора

4.6.2 Описание двоичного дешифратора

В данном примере будет рассмотрено описание двоичного дешифратора, который осуществляет подсчет количества единиц в 4-х разрядном входном сигнале при наличии разрешающего сигнала enable.

```
// *************************
// Описание схемы подсчета
//  единиц входного сигнала
// *************************

// Описание модуля
module rtl_decoder
                // Описание портов ввода-вывода
                (
                    input [3:0] bus,
                    input enable,
                    output [2:0] number
                );

                // Описание вспомогательных сигналов
                reg [2:0] local_number;

                // Описание тела модуля
                always @*
                        case ({enable, bus})
                            5'b11000,
                            5'b10100,
                            5'b10010,
                            5'b10001: local_number = 3'b001;
                            5'b11100,
                            5'b11010,
                            5'b11001,
                            5'b10110,
                            5'b10011,
                            5'b10101: local_number = 3'b010;
                            5'b11110,
                            5'b11011,
                            5'b10111: local_number = 3'b011;
                            5'b11111: local_number = 3'b100;
                            default:  local_number = 3'b000;
                        endcase
                assign number = local_number;

endmodule : rtl_decoder
```

Рисунок 4.6.2 — Описание схемы подсчета единиц во входном сигнале

В этом примере, наряду с уже рассмотренными конструкциями, используется блок **case** и конкатенация.

4.6.3 Описание блока АЛУ

Блок АЛУ (Арифметико-Логическое Устройство) является неотъемлемой

частью любого процессора. Данный блок, в зависимости от реализации, принимает на вход несколько операндов и в зависимости от поданной команды, осуществляет некоторую операцию и выдает результат. В этом примере данный блок будет принимать два 32-х разрядных операнда и осуществлять некоторый базовый набор операций, такой как сложение, вычитание, поразрядное И, поразрядное ИЛИ, сдвиг вправо и влево на один разряд. Если в блок будет подана не существующая команда, то блок будет выдавать сигнал об ошибке. Один из возможных вариантов построения данного блока представлен на рисунке 4.6.3.

```
// **************
// Описание АЛУ
// **************

// Описание модуля
module rtl_alu
                // Описание портов ввода-вывода
                (
                    input [31:0] operand1,
                    input [31:0] operand2,
                    input [2:0] cmd,
                    output [31:0] result,
                    output error
                );

                // Описание вспомогательных сигналов
                reg [31:0] local_result;

                // Описание тела модуля
                always @*
                    begin
                        case (cmd)
                            3'b000: local_result = operand1 + operand2;
                            3'b001: local_result = operand1 - operand2;
                            3'b010: local_result = operand1 & operand2;
                            3'b011: local_result = operand1 | operand2;
                            3'b100: local_result = operand1 << 1;
                            3'b101: local_result = operand1 >> 1;
                            default: local_result = operand1;
                        endcase
                    end

                assign error = ((cmd == 3'b110) || (cmd == 3'b111))
                    ? 1'b1 : 1'b0;

                assign result = local_result;

endmodule : rtl_alu
```

Рисунок 4.6.3 — Описание блока АЛУ

4.7 Конструкции для реализации последовательных схем уровня RTL

Последовательная цифровая схема представляет собой схему, содержащую память, которая формирует внутренние состояния. В отличие от комбинационных схем, в которых выходы являются лишь функциями входов, выходы

последовательных схем являются функциями входов и внутренних состояний. Для проектирования на FPGA используется полностью синхронный подход к проектированию, т.е. все элементы памяти контролируются (синхронизируются) глобальным тактовым сигналом и данные выбираются и сохраняются по фронту или спаду тактового сигнала. Это позволяет проектировщику отделять логику работы системы от памяти, что значительно упрощает процесс разработки. Этот подход является одним из самых важных принципов построения сложных цифровых систем.

Основными элементами памяти в проектировании цифровой аппаратуры, поддерживаемыми внутренними ресурсами FPGA, являются:

- Триггер D-типа (D Flip-flop);
- Регистр;
- Регистровый файл.

4.7.1 Триггер D-типа

Для триггеров данного типа, характерны два типа реализации: с асинхронным сбросом и без асинхронного сброса. Оба варианта представлены на рисунках 4.7.1 и 4.7.2.

```
// *************************
// Описание триггера D-типа
// *************************

module d_flip_flop(
                   input clk,
                   input data_in,
                   output reg data_out
                   );

                  // Описание вспомогательных сигналов

                  // Описание тела модуля
                  always @(posedge clk)
                      data_out <= data_in;

endmodule : d_flip_flop
```

Рисунок 4.7.1 — Описание триггера D-типа

```
// *************************
// Описание триггера D-типа
//  с асинхронным сбросом
// *************************

module d_flip_flop_with_reset(
                   input clk,
                   input reset,
                   input data_in,
                   output reg data_out
                   );

                  // Описание вспомогательных сигналов

                  // Описание тела модуля
                  always @(posedge clk, posedge reset)
                      if (reset)
                          data_out <= 0;
                      else
                          data_out <= data_in;

endmodule : d_flip_flop_with_reset
```

Рисунок 4.7.2 — Описание триггера D-типа с асинхронным сбросом

Анализируя оба примера, можно выделить две детали, отличающие описание последовательных схем от комбинационных:

- Последовательные схемы всегда описываются с помощью блока **always**;
- Лист чувствительности для последовательных схем не содержит в своем составе всех возможных сигналов (*). Это свойство характерно для последовательных схем, т.к. там блок **always** должен включаться при изменении любого сигнала, входящего в состав выражений блока **always**;
- Для описания последовательных схем используется неблокирующее присваивание.

4.7.2 Регистр

Регистр представляет из себя набор триггеров D-типа, контролируемых единым тактовым сигналом и сигналом сброса. Описание регистра идентично описанию триггера D-типа, за исключением добавления разрядности. Описание 32-х разрядного регистра представлено на рисунке 4.7.3.

```
// ******************
// Описание регистра
// ******************

module register(
                    input clk,
                    input [31:0] data_in,
                    output reg [31:0] data_out
                    );

                // Описание вспомогательных сигналов

                // Описание тела модуля
                always @(posedge clk)
                    data_out <= data_in;

endmodule : register
```

Рисунок 4.7.3 — Описание 32-х разрядного регистра

4.7.3 Регистровый файл

Регистровым файлом называется набор регистров с одним и более входным портом и одним и более выходным. Порты регистрового файла различаются по типу — для чтения или для записи. Порт также может быть сдвоенным, т.е. один порт одновременно может быть и на чтение, и на запись. Каждый порт обладает входным сигналом адреса, для выбора соответствующей ячейки памяти. Регистровый файл в основном используется как быстрое, временное хранилище. На основе регистрового файла можно также построить такие конструкции как очереди FIFO (First-In-First-Out), буферы и т.д.

Также можно отметить, что на этапе логического синтеза регистровые файлы могут быть переведены в распределенное или блочное представление. Распределенные регистровые файлы строятся на простой логике, доступной на FPGA. Блочное же представление регистрового файла отображается на физические блоки памяти, расположенные в FPGA. В случае больших объемов регистровых файлов лучше использовать блочный вариант, что приведет к существенной экономии ресурсов кристалла FPGA, тем не менее следует помнить о конечности данного ресурса. Если требуется описать небольшой регистровый файл, то подойдет распределенный вариант.

Наряду с большим разнообразием видов данной структуры, существует также огромное количество техник для ее описания. При этом, для использования блочного варианта памяти, обычно требуется указать это явно в настройках синтезатора среды проектирования.

```
// ************************************
// Вариант описания регистрового файла
// ************************************

// Описание модуля
module register_file
                // Описание портов ввода-вывода
                (
                    input clk,
                    input [1:0] addr_r,
                    input [1:0] addr_w,
                    input [7:0] data_w,
                    input we,
                    output reg [7:0] data_r
                );


    // Описание вспомогательных сигналов
    reg [7:0] ram [3:0];

    // Описание тела модуля
    always @(posedge clk)
        begin
            if (we) begin
                ram[addr_w] <= data_w;
            end
            data_r <= ram[addr_r];
        end

endmodule : register_file
```

Рисунок 4.7.4 — Описание регистрового файла с одним портом по чтению и одним портом по записи

На рисунке 4.7.4 представлен вариант описания регистрового файла с одним независимым портом для чтения и одним независимым портом для записи. В данном описании используется конструкция языка Verilog HDL, описывающая двумерный массив.

4.8 Реализация последовательных схем

Ниже, будут приведены примеры, иллюстрирующие принципы проектирования последовательных цифровых схем с помощью средств Verilog HDL.

4.8.1 Двоичный счетчик

Приведем классический пример по проектированию — двоичный счетчик. В цифровой технике можно реализовать огромное разнообразие таких счетчиков, поэтому опишем поведение счетчика для этого примера.

Блок будет прибавлять единицу к своему значению каждый такт. В случае асинхронного сброса или сигнала сброса, счетчик принимает свое значение равным нулю. В случае переполнения, счетчик продолжает считать с нулевого значения. Пример реализации подобного счетчика приведен на рисунке 4.8.1.

```
// ******************
//  Двоичный счетчик
// ******************

// Описание модуля
module counter
                // Описание портов ввода-вывода
                (
                    input clk,
                    input reset,
                    input to_zero,
                    output [7:0] data
                );

                // Описание вспомогательных сигналов
                reg data_reg;
                wire data_plus_one;

                // Описание тела модуля
                always @(posedge clk, posedge reset)
                    begin
                        if (reset)
                            data_reg <= 0;
                        else if (to_zero)
                            data_reg <= 0;
                        else
                            data_reg <= data_plus_one;
                    end

                assign data_plus_one = data_reg + 1'b1;

                assign data = data_reg;

endmodule : counter
```

Рисунок 4.8.1 — Пример реализации двоичного счетчика

На данном примере можно хорошо наблюдать разделение логики работы и элементов, реализующих память. Это улучшает понимание поведения модуля и делает код более наглядным.

4.8.2 Стек

Стек представляет собой очередь типа LIFO (Last-In-First-Out), что означает, что данные, сохраненные в буфере последними будут выбраны первыми. Также можно сказать, что в любой момент времени только ячейка с данными, находящаяся на вершине стека, является доступной. Для доступа к остальным

данным, находящимся ниже вершины стека, требуется последовательно удалить из стека все данные до нужной нам ячейки.

Любые манипуляции со стеком обычно происходят с использованием операций push и pop. Операция push добавляет новые данные на вершину стека, а операция pop служит для последовательного извлечение данных с вершины стека.

В данной реализации, стек будет иметь глубину, равную 8-ми и ширину, равную 4-м, т.е. будет содержать 8-мь 4-х битных ячеек памяти. Кроме стандартных операций push и pop, данный стек будет выполнять операцию синхронного сброса, переводящую его к известному начальному состоянию.

Что касается самой реализации аппаратной части, то стек может быть реализован несколькими способами. Одним из способов является перемещение данных между ячейками памяти и работа непосредственно с регистром, представляющим собой вершину стека. Более интересным примером, является стек со статической реализацией. При этом данные, находящиеся в стеке не перемещаются между регистрами, а вместо этого используется указатель SP (Stack Pointer). Для данного примера была выбрана реализация с использованием SP.

Для большей наглядности предлагается рассмотреть данный пример по частям. На рисунке 4.8.2 представлена часть описания стека на языке Verilog HDL, относящаяся к объявлению портов ввода-вывода и вспомогательных сигналов. Сигналы push, pop и reset представляют возможные операции над стеком. Сигнал err сигнализирует об ошибках, таких как: одновременное выполнение операций push и pop, выполнение операции pop над пустым стеком или выполнение операции push над полностью заполненным стеком.

```
// *****
// Стек
// *****

// Описание модуля
module stack (
    // Системные сигналы
    input clk,
    // Входные сигналы
    input [3:0] data_in,
    input push,
    input pop,
    input reset,
    // Выходные сигналы
    output err,
    output full,
    output empty,
    output [3:0] data_out
    );

    // Внутренние сигналы
    reg [2:0] sp_reg, sp_next;
    reg [3:0] stack [7:0];
```

Рисунок 4.8.2 — Объявление портов ввода-вывода и вспомогательных сигналов для модуля, реализующего стек

Далее, на рисунке 4.8.3 представлена часть описания модуля, содержащая поведение регистров. sp_reg представляет собой регистр, содержащий значение SP. При этом, по сигналу reset происходит сброс указателя, что приводит модуль в начальное состояние. Stack – представляет собой упрощенный вариант регистрового файла. Запись новых данных осуществляется при возникновении некоторого ряда условий, а именно: отсутствие ошибки, отсутствие заполненности стека и появление операции push.

```
// Описание регистров
always @(posedge clk)
    begin
        if (reset)
            sp_reg <= 3'b000;
        else
            sp_reg <= sp_next;

        if (push && ~full && ~err)
            stack[sp_reg] <= data_in;
        else
            stack[sp_reg] <= stack[sp_reg];
    end
```

Рисунок 4.8.3 — Описание поведения регистров модуля, реализующего стек

И, наконец, на рисунке 4.8.4 представлена заключительная часть модуля, описывающая его логику работы.

```
// Формирование сигнала ошибки
assign err = push & pop | push & full | pop & empty;
// Формирование сигнала заполненности стека
assign full = (&sp_reg) ? 1'b1 : 1'b0;
// Формирование сигнала пустоты стека
assign empty = (|sp_reg) ? 1'b1 : 1'b0;

// Формирование указателя стека
always @*
    begin
        if (push && ~err)
            sp_next = sp_reg + 4'b001;
        else if (pop && ~err)
            sp_next = sp_reg - 4'b001;
        else
            sp_next = sp_reg;
    end

// Формирование данных на выходе
assign data_out = stack[sp_reg];

endmodule : stack
```

Рисунок 4.8.4 — Описание внутренней логики модуля, реализующего стек

4.9 Константы и параметры. Проектирование IP-блоков

В данном разделе описываются такие понятия как константы и параметры, являющиеся ключевыми при написании блоков класса «soft-core», являющихся полностью параметризированными, законченными и протестированными модулями, выполняющими некоторые законченные функции. Это могут быть сопроцессоры, контроллеры интерфейсов и т.д.

4.9.1. Константы

В HDL описаниях очень часто требуется задавать значения констант. Это может потребоваться как в выражениях, так и при определении границ массивов и т.д. Для улучшения читабельности кода и для большей гибкости можно использовать одну из двух техник:

- Использование конструкции **define** препроцессора Verilog HDL. Эта техника аналогична технике применяемой в C;
- Конструкция **localparam**. С помощью этого ключевого слова описываются некоторые константы, имеющие смысл в пределах модуля.

Использование констант наилучшим образом может быть пояснено на примере. На рисунке 4.9.1 представлен пример в котором константы используются для выделения бита переноса при сложении дух операндов одинаковой разрядности.

```
// ******************************
// Пример использования констант
// ******************************

module adder_with_const(
                    input [31:0] operand1,
                    input [31:0] operand2,
                    output [31:0] sum,
                    output carry_out
                    );

                // Описание вспомогательных сигналов
                localparam CARRY = 32;
                localparam OTHER = CARRY - 1;

                wire [CARRY:0] local_sum;

                // Описание тела модуля
                assign local_sum = operand1 + operand2;

                assign sum = local_sum[OTHER:0];
                assign carry_out = local_sum[CARRY];

endmodule : adder_with_const
```

Рисунок 4.9.1 — Пример использования констант

4.9.2. Параметры

Использование констант вносит большую гибкость в описание RTL модулей, но в действительности описание можно сделать еще более удобным и гибким в использовании. Более того, можно сделать его удобным для последующего повторного использования в других проектах. Для этих целей используется конструкция, известная как **parameter**, служащая для передачи информации внутрь модуля. Это позволяет нам создавать новый экземпляр модуля варьируя некоторыми значениями, что делает модуль универсальным и многократно используемым. Тем не менее, параметры не могут быть модифицированы внутри модуля, что делает их похожими на константы.

В языке Verilog HDL, секция описания параметров должна быть вставлена в начале описания модуля, до описания портов. При этом, в описании параметров

модуля задаются значения по умолчанию. Так, описание регистрового файла, рассмотренное выше может быть модифицировано с применением параметров, что позволит нам создавать разные регистровые файлы с разной разрядность и количеством регистров. На рисунке 4.9.2 представлен пример такого регистрового файла.

```
// ************************************
// Вариант описания регистрового файла
//           с параметрами
// ************************************

// Описание модуля
module register_file_params
                // Описание параметров
                #(
                    parameter ADDR_WIDTH = 2,
                    parameter DATA_WIDTH = 8
                )
                // Описание портов ввода-вывода
                (
                    input clk,
                    input [ADDR_WIDTH-1:0] addr_r,
                    input [ADDR_WIDTH-1:0] addr_w,
                    input [DATA_WIDTH-1:0] data_w,
                    input we,
                    output reg [DATA_WIDTH-1:0] data_r
                );

    // Описание вспомогательных сигналов
    reg [DATA_WIDTH-1:0] ram [(ADDR_WIDTH*2-1):0];

    // Описание тела модуля
    always @(posedge clk)
        begin
            if (we) begin
                ram[addr_w] <= data_w;
            end
            data_r <= ram[addr_r];
        end

endmodule : register_file_params
```

Рисунок 4.9.2 — Вариант описания параметризированного регистрового файла

Также, с использованием параметров, мы можем описать схему многоразрядного компаратора, заменяющего однобитный компаратор из примера, рассмотренного выше. На рисунке 4.9.3 представлен пример описания компаратора с разрядностью N.

```
// *************************
//  N-битный компаратор
// *************************

// Описание модуля
module rtl_comparer_params
                // Описание параметров
                #(
                    parameter WIDTH = 5
                )
                // Описание портов ввода-вывода
                (
                    input [WIDTH-1:0] input1,
                    input [WIDTH-1:0] input2,
                    output equal
                );

                // Описание вспомогательных сигналов
                reg local_equal;

                // Описание тела модуля
                always @*
                    begin
                        if (input1 == input2)
                            local_equal = 1'b1;
                        else
                            local_equal = 1'b0;
                    end

                assign equal = local_equal;

endmodule : rtl_comparer_params
```

Рисунок 4.9.3 — Описание компаратора с разрядностью N

Параметры предоставляют механизм реализации «расширяемого кода», при котором ширина схемы может быть скорректирована для специфических нужд. Это делает код более мобильным и поддерживает многократное использование.

В заключении отметим, что для того, чтобы задать требуемые значения для определенного экземпляра модуля, требуется описать их, используя соответствующие конструкции языка. Пример создания экземпляра модуля с требуемыми параметрами, представлен на рисунке 4.9.4.

```
// ****************************
//  Создание экземпляра модуля,
//   содержащего параметры
// ****************************

module top
            // Описание портов ввода-вывода
                (
                    input [4:0] input1,
                    input [4:0] input2,
                    output equal
                );

            rtl_comparer_params
                #(
                    .WIDTH   (5)
                )
                special_module
                (
                    .input1  (input1),
                    .input2  (input2),
                    .equal   (equal)
                );

endmodule : top
```

Рисунок 4.9.4 — Пример создания экземпляра модуля компаратора с требуемыми параметрами

4.10 Конструкции для реализации конечных автоматов. Автоматы Мура и Мили

Конечные автоматы , FSM (Finite State Machine), используются для создания систем, которые имеют ограниченный набор состояний и перемещаются между ними при выполнении некоторых условий. Перемещение зависит от текущего состояния и внешних входов. В отличие от обычных последовательных схем, переход состояний в конечных автоматах не является простой, повторяющейся

характеристикой.

В этом разделе мы обсудим проектирование простых конечных автоматов и рассмотрим их описание с помощью HDL кода. Для практических целей, FSM в основном применяются в качестве контроллеров, которые проверяют внешние события и активизируют надлежащие контрольные сигналы для контроля операций с каналом данных.

Базовое представление конечного автомата ничем не отличается от представления обычной последовательной схемы. Его составляют регистры состояния, логика следующего состояния и некоторая логика на выходе. Конечный автомат называется автоматом Мура, если его выходы являются только функциями состояния и называется автоматом Мили, если выходы являются функциями как состояния, так и внешних входов.

FSM обычно определяются абстрактной диаграммой состояния, отображающей набор состояний, переходы между этими состояниями и условия осуществления переходов. Также на таких диаграммах указывается начальное состояние в котором находится система на момент включения. На рисунке 4.10.1 представлен пример диаграммы состояния для конечного автомата.

Представленный конечный автомат имеет 3 состояния, причем в каждом из состояний внутренние сигналы X и Y принимают некоторые значения. До тех пор, пока не изменится состояние автомата, значения также не будут меняться. Переход между состояниями осуществляется при определенных значениях сигнала A. Так, например, чтобы перейти из состояния State2 в состояние State3, сигнал A должен принять значение больше 10.

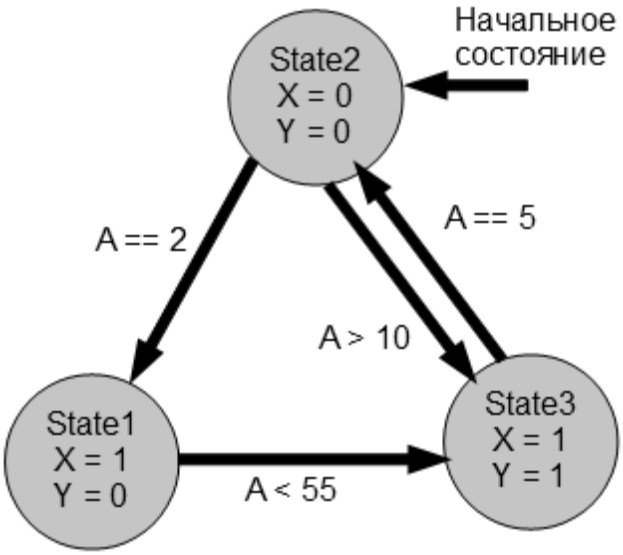


Рисунок 4.10.1 — Пример диаграммы состояния для конечного автомата

Существует большое множество вариантов описаний конечных автоматов на языке Verilog HDL. Далее будет представлен один из возможных вариантов шаблонов, описывающий такой автомат.

Для начала, на рисунке 4.10.2 представлена часть шаблона, описывающая его порты ввода-вывода.

В качестве системных сигналов представлены тактовый сигнал и сигнал асинхронного сброса, переводящий конечный автомат в начальное состояние. Сигналы input_controlN представляют собой сигналы, по которым будут осуществляться переходы, а сигналы output_controlN являются контрольными сигналами, вырабатываемыми данным автоматом.

```
// ***********************************
// Шаблон описания конечного автомата
// ***********************************

// Описание модуля
module fsm_template (
    // Системные сигналы
    input clk,
    input reset,
    // Входные контрольные сигналы
    input input_control1,
    ...
    // Выходные сигналы
    output output_control1,
    ...
    );

    // Объявление констант состояний
    localparam state1 = 0,
               state2 = 1,
               ...;

    // Объявление сигналов-состояний автомата
    reg [...] state_reg, state_next;
```

Рисунок 4.10.2 — Часть шаблона для описания конечного автомата, содержащая описание портов ввода-вывода, внутренних состояний и сигналов

На рисунке 4.10.3 представлена оставшаяся часть шаблона, дающая наглядное представление о внутренней структуре шаблона по приведенным комментариям. Отметим лишь, что как и было описано выше, результатом работы конечного автомата является генерация некоторого количества управляющих сигналов. На рисунке представлено формирование как управляющих сигналов Мура, так и Мили.

```
    // Описание состояний автомата
    always @(posedge clk)
        if (reset)
            state_reg <= 0;
        else
            state_reg <= state_next;

    // Описание переходов между состояниями
    always @*
        case (state_reg)
            // Переходы из состояния state0
            state1: begin
                if (input_control1) begin
                    ...
                end
                ...
            end
            // Переходы из состояния state1
            state2: ...
            ...
            // Если перехода нет, то возвращаемся в текущее состояние
            default: state_next = state_reg;
        endcase

    // Формирование выходных сигналов Мура
    assign output_control1 = (state_reg == 0);
    ...

    // Формирование выходных сигналов Мили
    assign output_control2 = (state_reg == 0) && ~input_control1;
    ...

endmodule : fsm_template
```

Рисунок 4.10.3 — Часть шаблона для описания конечного автомата, содержащая описание внутренней структуры

4.11. Реализация конечных автоматов

Понимание структуры и принципов функционирования конечных автоматов является одной из ключевых особенностей в проектировании сложных цифровых устройств. Для наглядности, на рисунках 4.11.1 и 4.11.2 представлен пример конечного автомата, аппаратно реализующего диаграмму состояний, рассмотренную на рисунке 4.10.1.

```
// ************************************
// Пример реализации конечного автомата
// ************************************

// Описание модуля
module fsm_example (
    // Системные сигналы
    input clk,
    input reset,
    // Входные контрольные сигналы
    input [5:0] a,
    // Выходные сигналы
    output x,
    output y
    );

    // Объявление констант состояний
    localparam state1 = 2'b01,
               state2 = 2'b10,
               state3 = 2'b11;

    // Объявление внутренних сигналов
    reg [1:0] state_reg, state_next;
    reg x_out, y_out;
```

Рисунок 4.11.1 — Описание портов ввода-вывода внутренних состояний и сигналов конечного автомата, представленного на рисунке 4.10.1

```
    // Описание состояний автомата
    always @(posedge clk)
        if (reset)
            state_reg <= state2;
        else
            state_reg <= state_next;

    // Описание переходов между состояниями
    always @* begin
        x_out = 1;  // значение по умолчанию
        y_out = 0;  // значение по умолчанию
        state_next = state_reg;
        case (state_reg)
            state2: begin
                x_out = 0;
                if (a == 2)
                    state_next = state1;
                else if (a > 10)
                    state_next = state3;
            end
            state1:
                if (a < 55)
                    state_next = state3;
            state3: begin
                y_out = 1;
                if (a == 5)
                    state_next = state2;
            end
            default: state_next = state_reg;
        endcase
    end

    // Формирование выходных сигналов
    assign x = x_out;
    assign y = y_out;

endmodule : fsm_example
```

Рисунок 4.11.2 — Описание тела конечного автомата, представленного на рисунке 4.10.1

5. Функциональное тестирование аппаратуры средствами языка Verilog HDL

Функциональное тестирование является неотъемлемым этапом любого процесса проектирования, поэтому здесь, в этой главе, мы рассмотрим этот процесс более подробно. Для осуществления тестирования проекта, требуется описать тестовое окружение (test environment), представляющее собой в самом простейшем случае модуль более высокого уровня абстракции, включающую в себя экземпляр тестируемого проекта и реализующий определенные сценарии тестирования. Основная задача тестового окружения — подавать на входные сигналы тестируемого проекта тестовые комбинации и анализировать выходные сигналы на корректность работы тестируемого проекта. В общем случае, поскольку тестовое окружение, является всего лишь надстройкой над проектом, выполняющее вспомогательную функцию, то оно не обязано быть синтезируемым. Это означает, что не требуется следовать лишь техникам и шаблонам, используемым в предыдущей главе для описания аппаратуры. В связи с этим, для реализации тестового окружения можно использовать абсолютно все конструкции языка Verilog HDL. Таким образом, процесс построения тестового окружения больше напоминает специфическое программирование.

В этой главе мы постараемся раскрыть некоторые, наиболее важные для построения тестовых окружений, конструкции языка. В завершении главы мы рассмотрим процесс тестирования модуля, реализующего стек. Поведение и описание данного модуля было приведено в главе 4.

5.1 Базовая структура тестового окружения

На рисунке 5.1.1 представлена базовая структура тестового окружения, написанного на языке Verilog HDL.

```
// ****************************
// Базовая структура тестового
//  окружения на Verilog HDL
// ****************************

// Настройки моделирования
...

// Тестовое окружение
module test;

    // Входные сигналы
    reg input1;
    ...

    // Выходные сигналы
    wire output1;
    ...

    // Генератор тактового сигнала
    ...

    // Экземпляр тестируемого модуля
    uut_instance uut (
        .input1  (input1),
        ...
        .output1 (output1),
        ...
    );

    // Тестовые последовательности
    ...

endmodule : test
```

Рисунок 5.1.1 — Базовая структура тестового окружения на языке Verilog HDL

Базовая структура тестового окружения, как и базовая структура модуля, состоит из нескольких, логически выделенных частей. Прежде всего это создание экземпляра тестируемого модуля и объявление его сигналов ввода-вывода. Кроме этого, при моделировании последовательных схем, стандартной частью является создание генератора тактового сигнала. И, наконец, заключительной частью является создание набора тестовых последовательностей для проверки

корректности работы тестируемого модуля.

До описания тестового окружения, требуется также указать некоторые настройки моделирования, которые будут использоваться соответствующими программными пакетами моделирования.

В последующих разделах будут более детально описаны все стандартные составные части тестовых окружений, а также конструкции, облегчающие их описание.

5.2 Описание настроек моделирования

В языке Verilog HDL имеется ряд конструкций, позволяющих задать некоторые параметры для среды моделирования. Такие конструкции реализованы в виде директив компилятора. Наиболее часто используемой является директива **timescale**. Данная директива позволяет установить точность моделирования, а также шаг моделирования, т.е. минимальную единицу времени при моделировании. На рисунке 5.2.1 представлен синтаксис директивы **timescale**.

```
// ******************************
// Синтаксис директивы timescale
// ******************************
`timescale [time_unit]/[time_precision]
```

Рисунок 5.2.1 — Синтаксис директивы **timescale**

- [time_unit] – Минимальная единица времени моделирования. Пример значений: 1ns;
- [time_precision] – Точность моделирования. Пример значений: 100ps.

После включения данной директивы в код тестового окружения, можно пользоваться такой конструкцией, как **#N**, где N – некоторое число. Это позволяет ждать прохождения некоторого заданного временного интервала.

На рисунке 5.2.2 представлен пример использования конструкции **#N**, для задания сигнала B через 10ns после задания сигнала A.

```
// ************************
// Пример использования #
// ************************
`timescale 1ns/1ps
...
A = 1;
#10;
B = 1;
...
```

Рисунок 5.2.2 — Пример использования конструкции языка Verilog HDL, #N

5.3 Тестовые последовательности и блок initial

Ключевым элементом в тестовом окружении является часть, создающая тестовые последовательности. Для описания этой части может использоваться блок **initial**, который представляет собой специальную конструкцию языка Verilog HDL, аналогичную блоку **always**. Вся разница заключается в том, что в отличие от блока **always**, срабатывающего по определенному событию, блок **initial** срабатывает лишь единожды — при старте моделирования. На рисунке 5.3.1 представлен шаблон для описания блока **initial**.

```
// ******************************
// Шаблон описания блока initial
// ******************************

initial
    begin : [label]
    // Декларация локальных переменных
    [data_type] [data_name];
    ...
    // Тело блока
    [proc_statement];
    ...
    end : [label]
```

Рисунок 5.3.1 — Шаблон описания блока **initial**

- [label] – Данная конструкция служит для облегчения чтения кода и является опциональной;
- [data_type] – Тип сигнала, переменной и т.д. Аналогично блоку **always**;

- [data_name] – Название сигнала, переменной и т.д.;
- [proc_statement] – Процедурное присваивание.

Стоит отметить, что кажущаяся на первый взгляд простота и бесполезность данной конструкции, обманчива. Например, пользуясь конструкцией языка Verilog HDL, **#N**, моделирующей время, можно описать удивительное множество полезного кода, тестирующего аппаратуру. Например, блок, тестирующий поведение однобитного компаратора, описанного выше, может иметь вид, представленный на рисунке 5.3.2.

```
// *************************
// Пример блока initial для
// тестирования однобитного
//       компаратора
// *************************
initial
    begin
        input1 = 1'b0;
        input2 = 1'b0;
        #1;
        input1 = 1'b0;
        input2 = 1'b1;
        #1;
        input1 = 1'b1;
        input2 = 1'b0;
        #1;
        input1 = 1'b1;
        input2 = 1'b1;
        #1;
    end
```

Рисунок 5.3.2 — Пример блока **initial**, тестирующего однобитный компаратор

5.4 Описание генератора тактового сигнала

Неотъемлемой частью тестирования последовательных схем является моделирование тактового сигнала. Для этих целей описывается отдельный блок **initial**. На рисунке 5.4.1 приведен пример описания генератора тактового сигнала. В этом описании используется новая конструкция, **forever**, представляющая собой

бесконечный цикл. При реализации тестового окружения не рекомендуется использовать множество таких циклов, т. к. это может привести к падению скорости моделирования.

```
// ***************************
// Пример описания генератора
//      тактового сигнала
// ***************************
    ...
    reg clk;
    ...
    initial begin
        clk = 0;
        forever begin
            #10 clk = ~clk;
        end
    end
```

Рисунок 5.4.1 — Пример описания генератора тактового сигнала

При реализации генераторов тактовых сигналов, для большей гибкости можно использовать константы и параметры, а также выносить описание в отдельные модули.

5.5 Вспомогательные конструкции

Кроме уже описанного минимума, можно также использовать огромное количество конструкций языка, схожих с конструкциями, применяемыми для обычного программирования. В этом разделе мы рассмотрим некоторые из них.

5.5.1 Task u function

С ростом сложности проекта, также растет и сложность тестового окружения. Одним из способов снижения сложности является разбивка всего кода тестового окружения на более мелкие части. В этом нам могут помочь такие конструкции, как **task** и **function**.

Конструкция **function** описывает функцию, принимающую на вход

некоторое количество аргументов, осуществляющую заданную операцию и возвращающую некоторое значение. Тело функции не может содержать в своем составе конструкций, относящихся ко времени, например **#N**. При этом любой вызов функции происходит «мгновенно» и не затрачивает никакого времени моделирования, т. е. реализация функций в языке Verilog HDL, аналогична реализации функции в обычных языках программирования высокого уровня. Функции в свою очередь могут вызывать другие функции. На рисунке 5.5.1 представлен синтаксис описания **function**.

```
// **********************
// Синтаксис конструкции
//       function
// **********************
module ...
    ...
    function [result_type] [name] ([args])
        begin
            [local_vars]
            ...
            [proc_statement];
            ...
        end
    endfunction
...
endmodule
```

Рисунок 5.5.1 — Синтаксис описания **function**

- [result_type] – Тип возвращаемого значения;
- [name] – Имя функции;
- [args] – Список передаваемых аргументов;
- [local_vars] – Объявление локальных переменных;
- [proc_statement] – Процедурные операторы.

В свою очередь, конструкция **task** является более гибкой. Она не возвращает значений, но может содержать любые конструкции языка, относящиеся ко времени, а также может иметь аргументы разных типов, таких как **input**, **output**

или **inout**. На рисунке 5.5.2 представлен синтаксис описания **task**.

```
// **********************
// Синтаксис конструкции
//         task
// **********************
module ...
    ...
    task [name] ([args])
        begin
            [local_vars]
            ...
            [proc_statement];
            ...
        end
    endtask
...
endmodule
```

Рисунок 5.5.2 — Синтаксис описания **task**

- [name] – Имя задания;
- [args] – Список передаваемых аргументов;
- [local_vars] – Объявление локальных переменных;
- [proc_statement] – Процедурные операторы.

Стоит отметить также, что объявление функции возможно только в пределах модуля.

5.5.2 Циклы

Язык Verilog HDL поддерживает следующие циклы: **for**, **while**, **repeat** и **forever**. Использование циклов возможно только в пределах секция кода, в которых разрешено выполнение процедурных операторов. Например внутри блоков **always** или **initial**, а также **function** и **task**. Для примера, на рисунке 5.5.3 представлен синтаксис цикла **for**.

```
// ****************
// Синтаксис цикла
//       for
// ****************

for ([init_assign]; [condition]; [step_assign]) begin
    [local_vars];
    ...
    [proc_statement];
    ...
end
```

Рисунок 5.5.3 — Синтаксис цикла **for**

- [initial_assign] – Присваивание до входа в цикл;
- [condition] – Условие выхода из цикла;
- [step_assign] – Присваивание на очередной итерации цикла;
- [local_vars] – Объявление локальных переменных;
- [proc_statement] – Процедурные операторы.

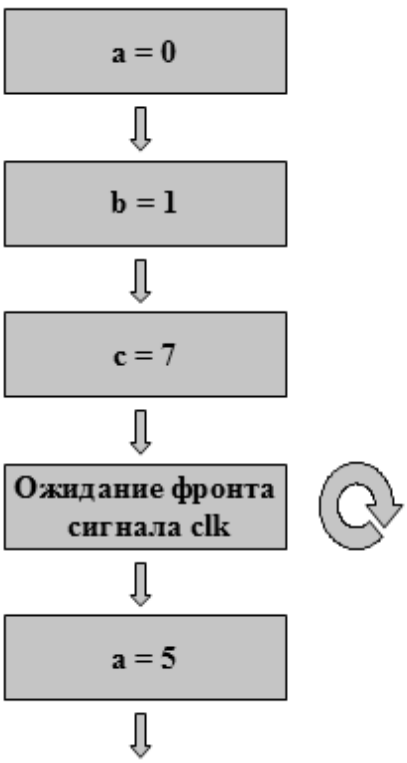
5.5.3 Ожидание событий и состояний

В тестовых окружениях нам часто требуется дождаться появления определенных событий или состояний. Для этих целей используются следующие конструкции:

- **@**([event]) – Ожидание возникновения определенного события. В роли такого события может выступать фронт (**posedge** clk) или срез тактового сигнала (**negedge** clk);
- **wait**([condition]) – Ожидание возникновения определенного состояния. В роли условия может выступать любое логическое выражение, результатом которого является 0 или 1.

Эти конструкции языка можно использовать только внутри **always** или **initial** блоков. При этом, каждый блок **always** или **initial** можно рассматривать как

отдельный программный поток. Если в блоке выполнение кода доходит до одного из описанных выше операторов, то дальнейшее выполнение кода в этом блоке будет приостановлено до появления описанного события или состояния. Данное поведение может быть продемонстрировано на рисунке 5.5.4.




```
initial begin
    a = 1;
    b = 2;
    c = 7;
    @(posedge clk);
    a = 5;
    ...
end
```

Рисунок 5.5.4 — Структура поведения конструкций ожидания языка Verilog HDL

5.5.4 Системные функции

Verilog HDL обладает богатым набором встроенных системных функций, служащих для облегчения разработки и отладки кода, а также предоставления системных сервисов. Названия всех системных функций начинаются со знака **$**. Ниже будут кратко перечислены основные группы функций и их основные представители.

- Контроль за выполнением моделирования. К таким функциям относятся

$finish и **$stop**. Функция **$finish** полностью завершает моделирование, тогда как функция **$stop** лишь его приостанавливает;

- Функции вывода. Для упрощения анализа результатов поведения аппаратуры можно использовать всевозможные функции для печати текста. К таким функциям относятся: **$display**, **$write**, **$monitor** и **$strobe**. Функция **$display** аналогична стандартной функции printf языка C;
- Функции для работы с файлами. Язык Verilog HDL реализует все обычные сервисы по работе с файлами, такие как open, close и т. д. Для каждого из сервисов существует одноименная системная функция. Примерами могут служить функции **$open**, **$close** и т. д.
- Функции генерирования псевдослучайных чисел. Ярким представителем является функция **$random**, возвращающая 32-х разрядное целое число;

5.6 Реализация законченного тестового окружения

В данном разделе будет рассмотрено тестовое окружение для модуля, реализующего стек. При этом, мы рассмотрим также результаты моделирования в виде временных диаграмм.

На рисунке 5.5.1 представлена верхняя часть тестового окружения, описывающая входные и выходные сигналы тестируемого модуля.

```
// ******************************
// Тестовое окружение для модуля,
//       реализующего стек
// ******************************

// Вспомогательные конструкции
`timescale 1ns / 1ps

// Тестовое окружение
module test;

    // Параметры
    localparam T = 20;

    // Входные сигналы
    reg reset;
    reg clk;
    reg [3:0] data_in;
    reg push;
    reg pop;

    // Выходные сигналы
    wire err;
    wire full;
    wire empty;
    wire [3:0] data_out;
```

Рисунок 5.6.1 — Верхняя часть тестового окружения для модуля, реализующего стек

Далее, на рисунке 5.6.2, представлено описание используемых функций и заданий.

```
// Описание task & function
// Инициализация
task init();
    begin
        reset = 1;
        clk = 0;
        data_in = 0;
        push = 0;
        pop = 0;
        #(5*T + T/4);
        reset = 0;
    end
endtask : init

// Запись данных в стек
task push_data(reg [3:0] data);
    begin
        push = 1;
        data_in = data;
        @(negedge clk);
        push = 0;
    end
endtask : push_data

// Чтение данных из стека
task pop_data();
    begin
        pop = 1;
        @(negedge clk);
        pop = 0;
    end
endtask : pop_data

// Генератор тактового сигнала
initial begin
    clk = 0;
    forever begin
        #(T/2) clk = ~clk;
    end
end
```

Рисунок 5.6.2 — Описание используемых в тестовом окружении функций и заданий

И, наконец, на рисунке 5.6.3 представлена оставшаяся часть тестового окружения, описывающая создание экземпляра тестируемого модуля и создание тестовых последовательностей, основанных на описанных ранее заданиях.

```
    // Экземпляр модуля, реализующего стек
    stack uut (
        .reset      (reset),
        .clk        (clk),
        .data_in    (data_in),
        .push       (push),
        .pop        (pop),
        .err        (err),
        .full       (full),
        .empty      (empty),
        .data_out   (data_out)
    );

    // Тестовые последовательности
    initial begin
        // Инициализация
        init();
        // Тестовый вектор
        push_data(4'b1111);
        // Тестовый вектор
        push_data(4'b0101);
        // Тестовый вектор
        pop_data();
        // Тестовый вектор
        push = 1;
        pop = 1;
        @(negedge clk);
        // Окончание тестирования
        $finish();
    end

    // Монитор ошибок проекта
    initial begin
        $monitor("An error occured! - %b", err);
    end

endmodule : test
```

Рисунок 5.6.3 — Заключительная часть тестового окружения, для модуля, реализующего стек

При моделировании тестового окружения средствами ISIM были получены временные диаграммы, характеризующие работу тестируемого модуля. Данные временные диаграммы представлены на рисунке 5.6.4.

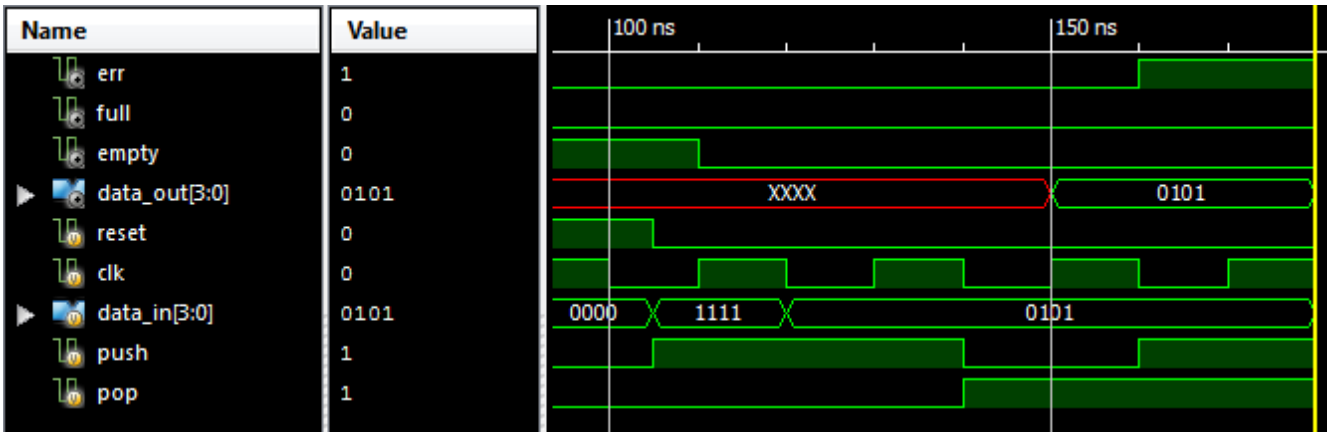


Рисунок 5.6.4 — Временные диаграммы, полученные в результате тестирования модуля, реализующего стек

Анализируя временные диаграммы можно сделать вывод, о том, что на выбранном нами наборе входных тестовых векторов, разработанный модуль работает корректно.

6. Лабораторные работы

6.1 Лабораторная работа №1
Создание проекта и выполнение всего маршрута проектирования в среде проектирования ISE фирмы Xilinx

Цель работы: изучение всего маршрута проектирования Xilinx с использованием САПР ISE, создание проекта и выполнение всех этапов маршрута проектирования Xilinx, конфигурирование тестовой платы Atlys.

После выполнения работы вы:

- научитесь создавать собственные проекты в САПР ISE;
- познакомитесь со всеми этапами маршрута проектирования для FPGA;
- познакомитесь с базовым структурным блоком языка Verilog HDL;
- научитесь конфигурировать целевую FPGA.

ВВЕДЕНИЕ

Данная лабораторная работа предоставляет обзор средств проектирования ISE. Программный продукт ISE фирмы Xilinx представляет собой интегрированную систему автоматизированного проектирования, EDA (Electronic Design Automation), цифровых систем, которая предполагает реализацию проекта с использованием программируемых логических интегральных схем (ПЛИС), производимых этой фирмой. Система позволяет описать проект как посредством ввода принципиальной схемы с использованием графического редактора, так и с помощью конструкций, описанных на языках HDL, таких как Verilog HDL или VHDL.

Вся среда проектирования представляет собой набор программных модулей,

каждый из которых используется для выполнения определенного этапа обработки проекта. В качестве таких модулей можно выделить такие программы, как графический редактор, текстовый редактор, синтезатор (XST) и т. д.

В данной лабораторной работе будет предложено реализовать простой проект, выполнив при этом все этапы маршрута проектирования, вплоть до конфигурирования целевой платы и визуального наблюдения результата на целевой плате Atlys.

ОПИСАНИЕ ЗАДАЧИ

В качестве основы данной лабораторной работы будет предложено реализовать схему, выполняющую простую коммутацию одного из светодиодов с одной из кнопок, расположенных на плате Atlys.

Структурная схема предложенного проекта представлена на рисунке 6.1.1.

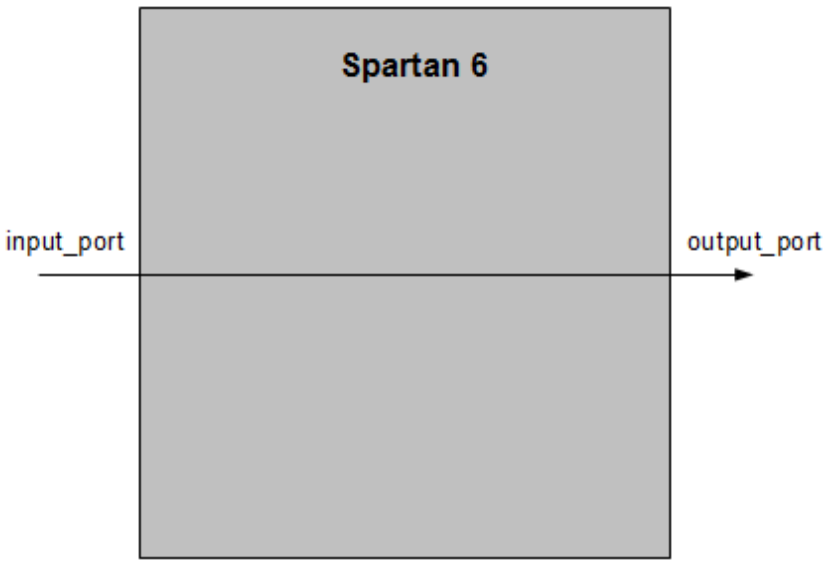


Рисунок 6.1.1 - Структурная схема проекта лабораторной работы №1

ВЫПОЛНЕНИЕ РАБОТЫ

Шаг 1. Создание нового проекта в среде ISE.

Разрабатываемое устройство представляется в системе ISE как проект.

Вначале любых действий со средой ISE требуется либо создать проект, либо открыть уже существующий. Любой проект содержит настройки семейства и типа целевой FPGA.

1-1. Создание нового проекта для целевой FPGA Spartan 6, находящейся на плате Atlys.

1-1-1. Запустите ISE: Выберете Start → All Programs → Xilinx ISE Design Suite 13.1 → ISE Design
Tools → Project Navigator.

1-1-2. В Project Navigator, выберете File → New Project. После этого откроется окно меню создания нового проекта (Рисунок 6.1.2).

1-1-3. Выберете путь к вашим будущим проектам и имя вашего нового проекта.

1-1-5. Нажмите Next.

1-1-6. Выберете следующие опции для проекта и нажмите Next (Рисунок 6.1.3):
Device Family: **Spartan6**
Device: **XC6SLX45**
Package: **CSG324**
Speed Grade: **–3**
Synthesis Tool: **XST (VHDL/Verilog)**
Simulator: **ISim (VHDL/Verilog)**
Preferred Language: **Verilog**

1-1-7. Вы увидите окно Project Summary. Нажмите Finish.

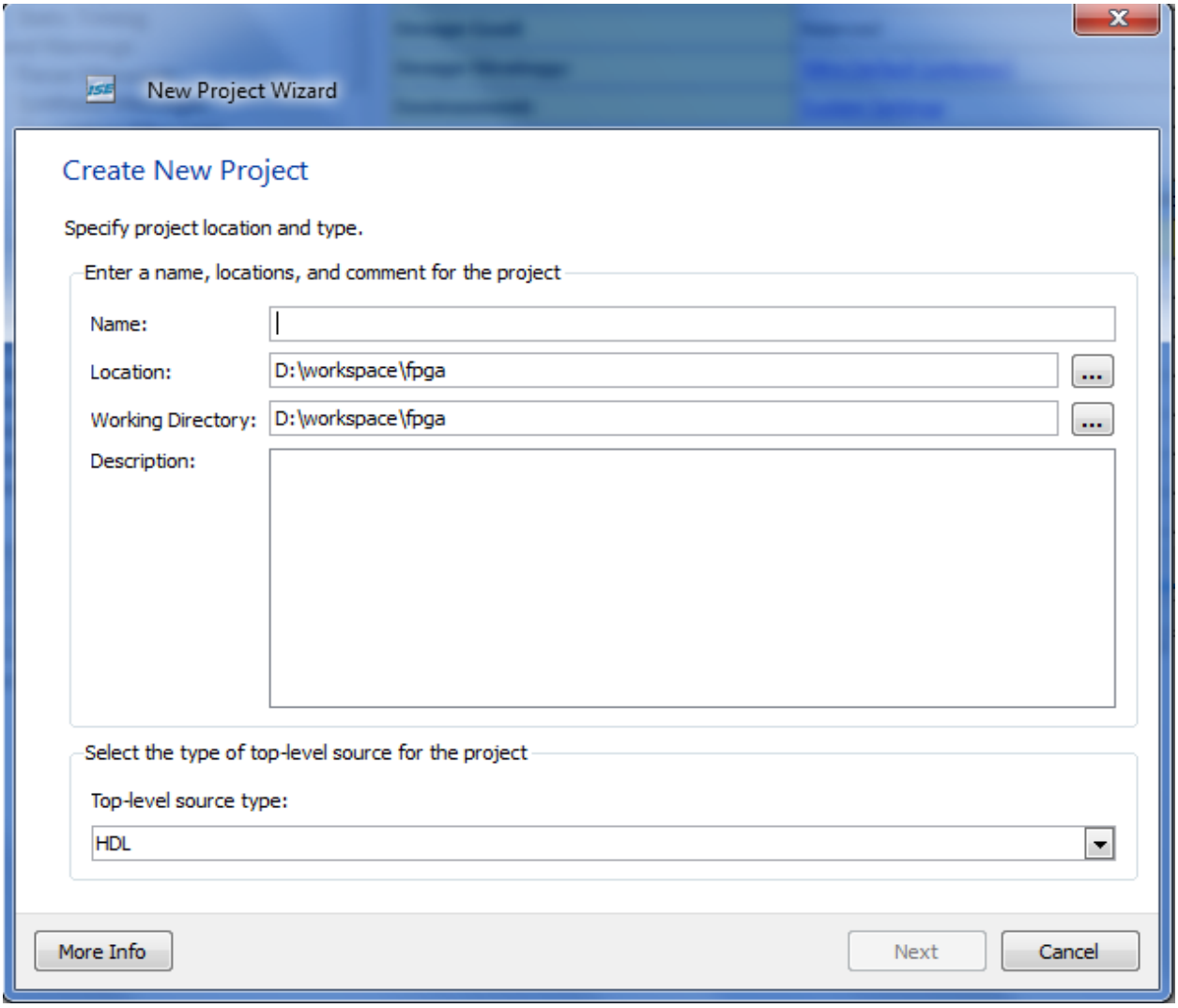


Рисунок 6.1.2 — Окно создания нового проекта в среде ISE

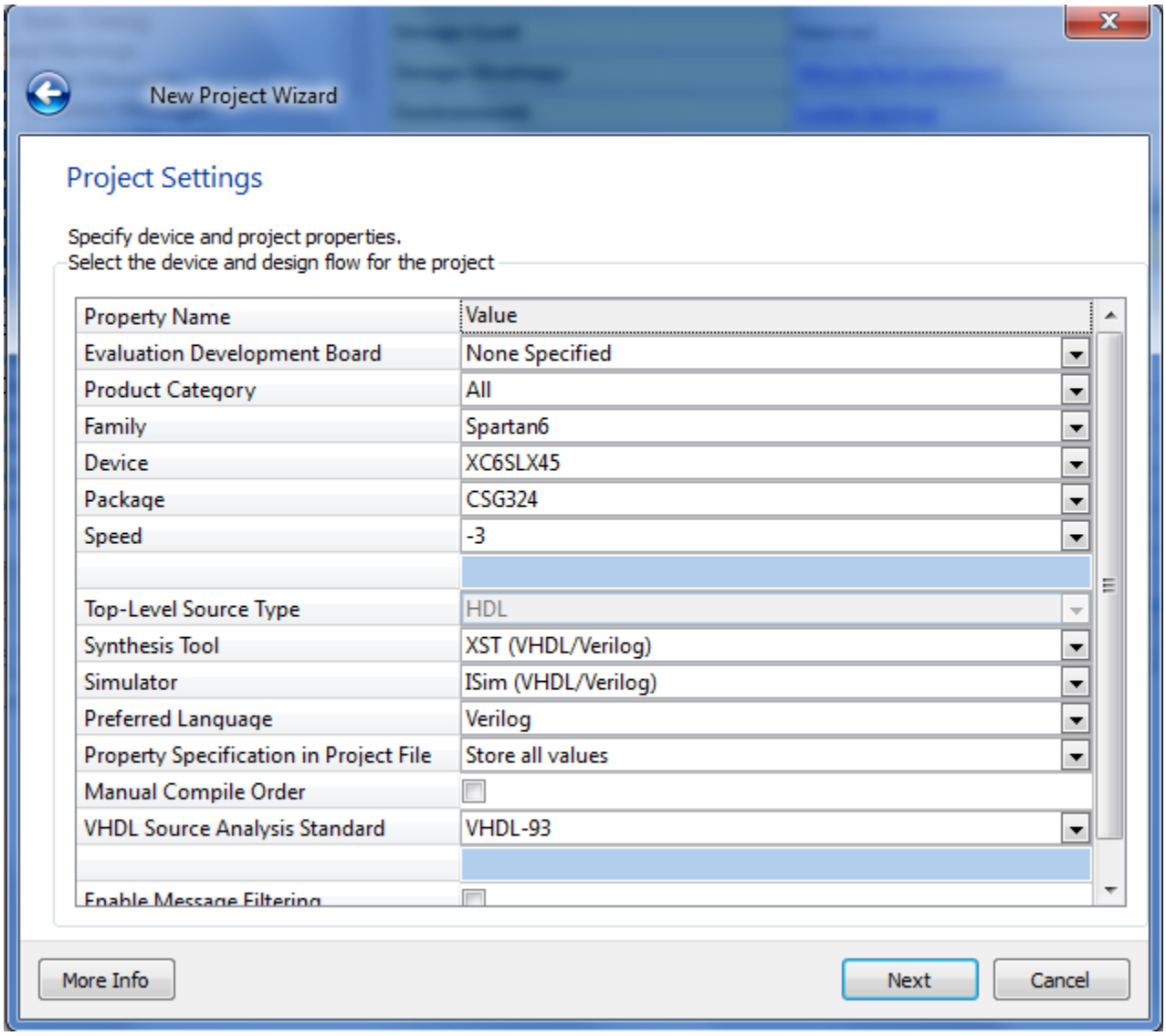


Рисунок 6.1.3 — Выбор опций нового проекта в среде ISE

Шаг 2. Создание и описание файлов проекта с помощью языка Verilog HDL в среде ISE.

Любой проект для FPGA представляет из себя набор файлов, реализующих разнообразные функции. На данном шаге требуется описать модуль верхнего уровня. Данный модуль будет представлять основу проекта. Для более подробного ознакомления со структурой проекта для FPGA просмотрите главу 2 (Понятие проекта. Составные части проекта. Иерархия проекта).

В качестве языка для описания модуля верхнего уровня будет использоваться язык Verilog HDL. Вам потребуется ознакомиться со структурой базового модуля, представленной в главе 4 (Понятие модуля. Базовая структура модуля).

2-1. Создание нового модуля верхнего уровня с помощью среды ISE.

2-1-1. Выберете Project → New source. После этого откроется окно меню создания нового файла для проекта (Рисунок 6.1.4).

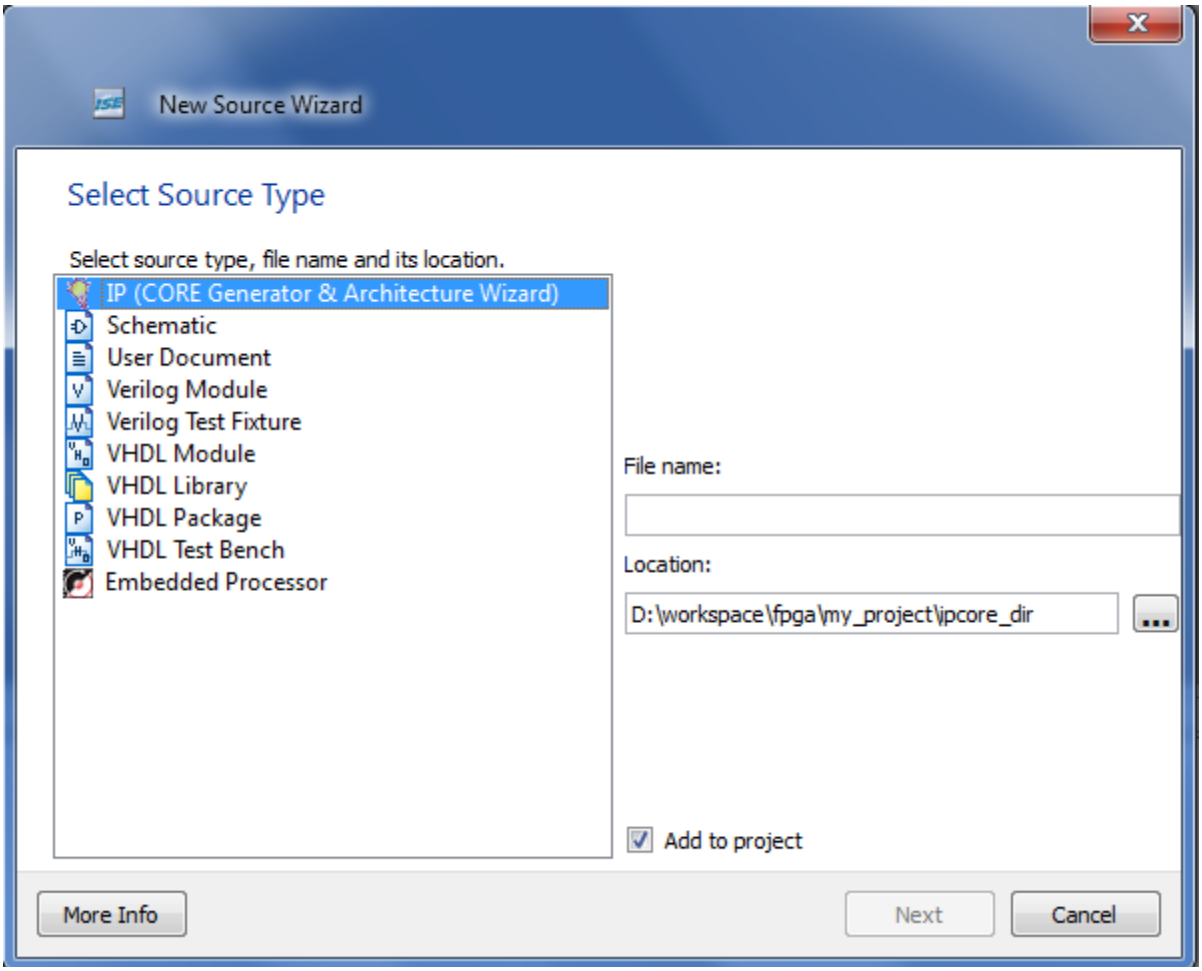


Рисунок 6.1.4 — Окно меню создания нового файла для проекта в среде ISE

2-1-2. В окне выберете Verilog Module. Также укажите название модуля. Нажмите 2 раза Next и Finish. После выполнения этих действий вы увидите ваш модуль в общей иерархии проекта (Рисунок 6.1.5).

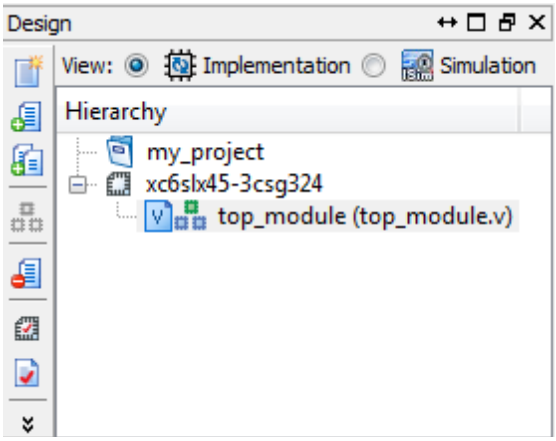


Рисунок 6.1.5 — Вид иерархии проекта с созданным модулем в среде ISE

2-2. Описание разрабатываемого модуля с помощью Verilog HDL.

2-2-1. Дважды нажмите на созданный файл левой кнопкой мыши. Вы перейдете во встроенный текстовый редактор.

2-2-2. Введите описание проектируемого модуля.

2-2-3. Сохраните описание, выбрав File→ Save. Вы можете увидеть окно Project Navigator после выполнения этого шага на рисунке 6.1.6.

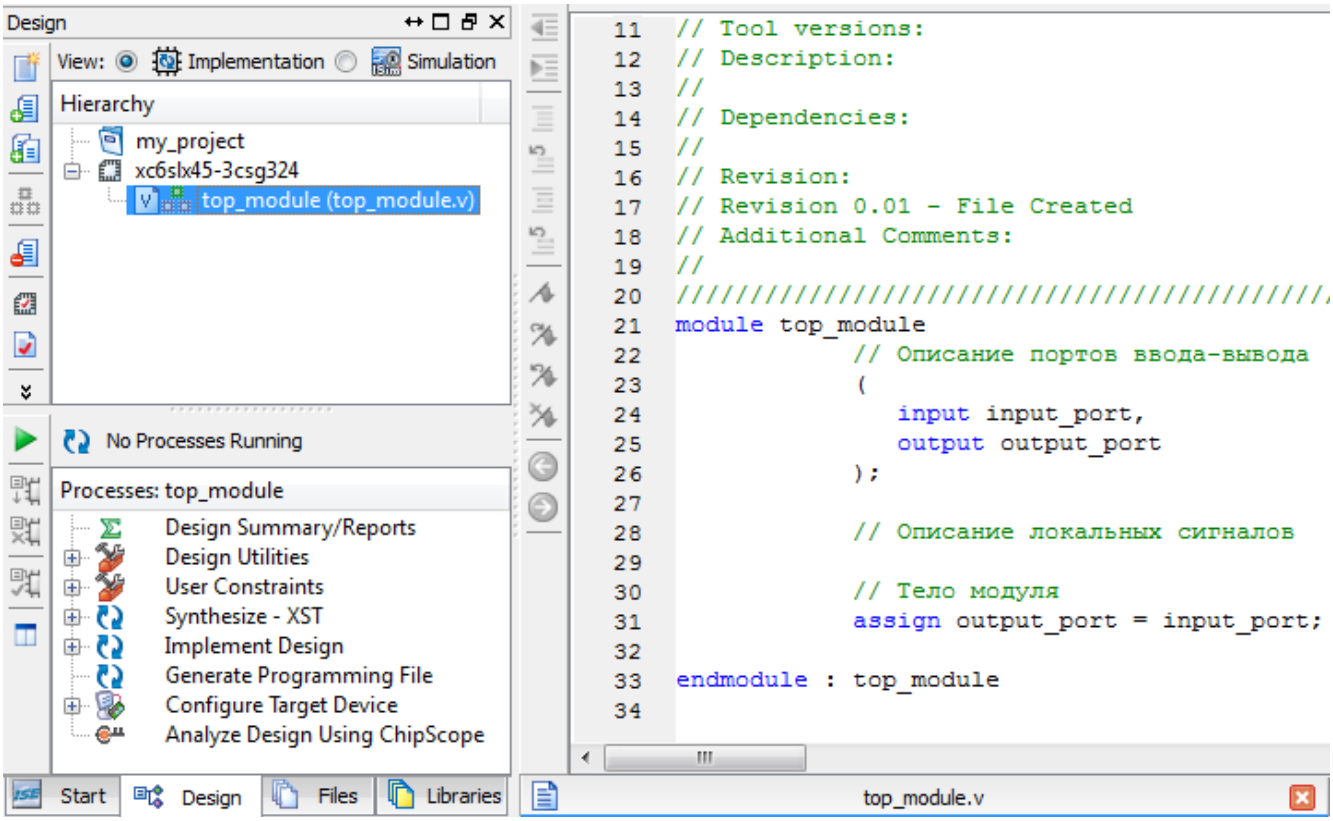


Рисунок 6.1.6 — Вид окна Project Navigator после добавления описания модуля

Шаг 3. Создание файла, содержащего ограничения проекта по размещению.

Наряду с модулем верхнего уровня, любой проект для FPGA содержит файл, описывающие ограничения проекта, называемый UCF (User Constraints File). Этот файл содержит множество ограничений, предъявляемых к проекту. Одними из ограничений, описываемых в этом файле, являются ограничения на размещение логики проекта на физическом уровне. Существует большое количество видов ограничений по размещению. В данном проекте мы создадим базовый набор ограничений, которые будут накладываться на сигналы ввода-вывода нашего модуля верхнего уровня. Эти ограничения укажут среде, при выполнении последующих шагов маршрута проектирования, к каким физическим выводам микросхемы Spartan 6 требуется подключить наши сигналы ввода-вывода модуля верхнего уровня.

В данном проекте нам требуется соединить входной сигнал с одной из кнопок, а выходной сигнал с одним из светодиодов.

3-1. Создание нового файла, содержащего ограничения проекта с помощью среды ISE.

3-1-1. Выберете Project → New source. После этого откроется окно меню создания нового файла для проекта.

3-1-2. В окне выберите Implementation Constraints File и введите название файла. Нажмите Next, затем Finish.

3-2. Задание в проекте ограничений по размещению средствами PlanAhead.

3-2-1. Выберете Tools → PlanAhead→ I/O Pin Planning - Pre-Synthesys. После этого откроется окно PlanAhead (Рисунок 6.1.7).

3-2-2. Откройте документ Atlys_rm.pdf. Найдите в нем название выводов для одного из светодиодов и одной из кнопок на плате Atlys.

3-2-3. В PlanAhead выберите поиск в Package Pins. Введите найденные сигналы в стору поиска и назначьте им соответствующие сигналы вашего проекта.

3-2-4. Сохраните изменения и выйдите из PlanAhead.

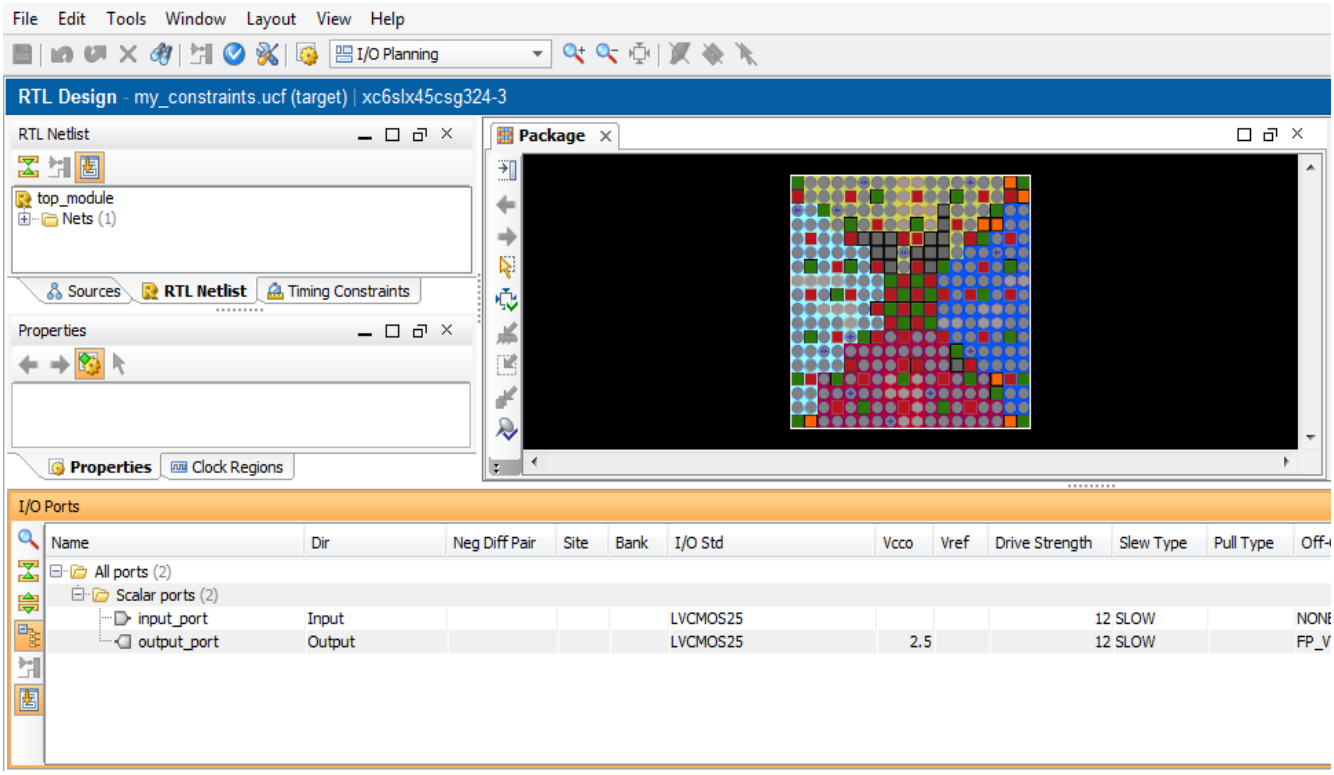


Рисунок 6.1.7 — Окно PlanAhead для задания ограничений на размещение сигналов ввода-вывода

Шаг 4. Проведение синтеза всего проекта.

На данном шаге требуется выполнить синтез всего проекта для получения NGC файла и продвижения по маршруту проектирования.

4-1. Проведение синтеза всего проекта.

4-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Synthesize – XST.

4-1-2. Дождитесь окончания синтеза.

4-1-3. По окончанию синтеза во вкладке Design должна загореться зеленая галочка.

4-1-4. В случае возникновения ошибок требуется исправить описание проектируемого модуля.

Шаг 5. Проведение имплементации всего проекта.

На данном шаге требуется выполнить имплементацию всего проекта, что

позволит нам на следующем шаге создать конфигурационный файл для целевой FPGA.

5-1. Проведение имплементации всего проекта.

5-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Implement Design.

5-1-2. Дождитесь окончания имплементации.

5-1-3. По окончанию имплементации во вкладке Design должна загореться еще одна зеленая галочка.

5-1-4. В случае возникновения ошибок требуется исправить файл, содержащий ограничения по размещению проекта.

Шаг 6. Создание конфигурационного файла для целевой FPGA.

Этот шаг является заключительным в маршруте проектирования для FPGA. Мы создадим конфигурационный файл, который затем загрузим в плату Atlys.

6-1. Создание конфигурационного файла для целевой FPGA.

6-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Generate Programming File.

6-1-2. Дождитесь окончания создания конфигурационного файла.

6-1-3. По окончанию имплементации во вкладке Design должна загореться еще одна зеленая галочка.

6-1-4. Прежде чем переходить к следующим шагам, убедитесь, что вкладка Design имеет вид, схожий с Рисунком 6.1.8.

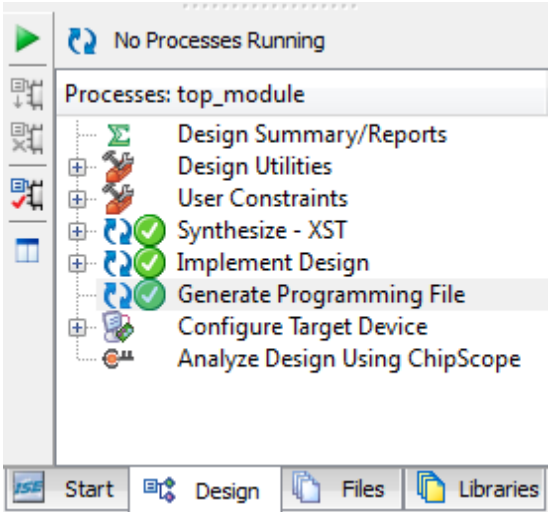


Рисунок 6.1.8 — Вид вкладки Design после окончания процессов синтеза, имплементации и создания конфигурационного файла.

Шаг 7. Подключение платы Atlys к компьютеру.

На данном шаге требуется подключить тестовую плату Atlys к компьютеру.

7-1. Подключение платы Atlys к компьютеру.

7-1-1. Подключите плату Atlys к источнику питания.

7-1-2. Включите плату с использованием переключателя, расположенного на плате.

7-1-3. Подключите плату с помощью USB кабеля к компьютеру.

7-1-4. При возникновении затруднений, обратитесь к документу Atlys_rm.pdf, поставляемым вместе с платой Atlys.

Шаг 8. Загрузка созданного конфигурационного файла на плату Atlys.

На данном шаге мы произведем загрузку конфигурационного файла в плату Atlys для конфигурирования, находящегося на ней, FPGA Spartan 6.

8-1. Загрузка созданного конфигурационного файла на плату Atlys.

8-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мышки на Configure Target Device. При этом откроется меню программы IMPACT, представленное на рисунке 6.1.9.

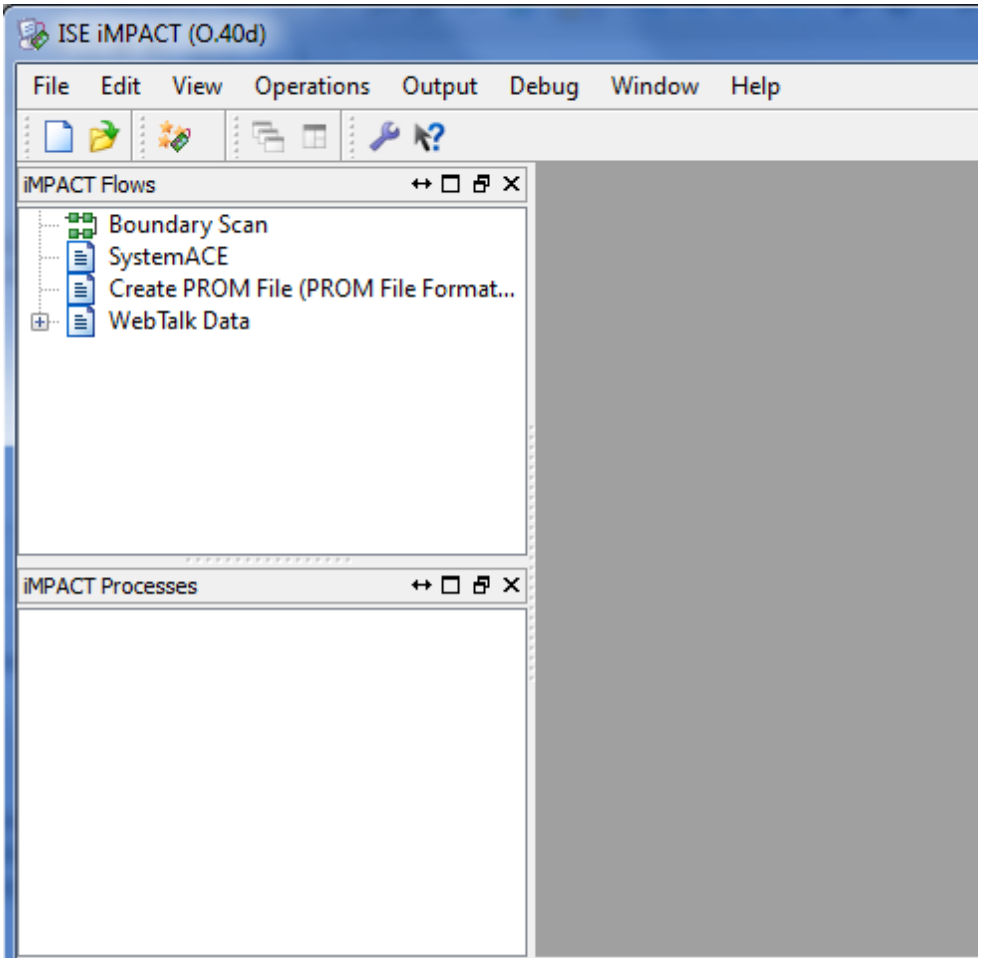


Рисунок 6.1.9 — Меню программы IMPACT для загрузки конфигурационного файла в целевую FPGA

8-1-2. Выберите Edit → Launch Wizard. В появившемся меню нажмите Ok.

8-1-3. Выберите Output → Cable Setup. В разделе Cable Plug-in поставьте галочку и введите digilent_plugin.

8-1-4. После этого окно Cable Communication Setup должно иметь вид, схожий с Рисунком 6.1.10. Нажмите Ok.

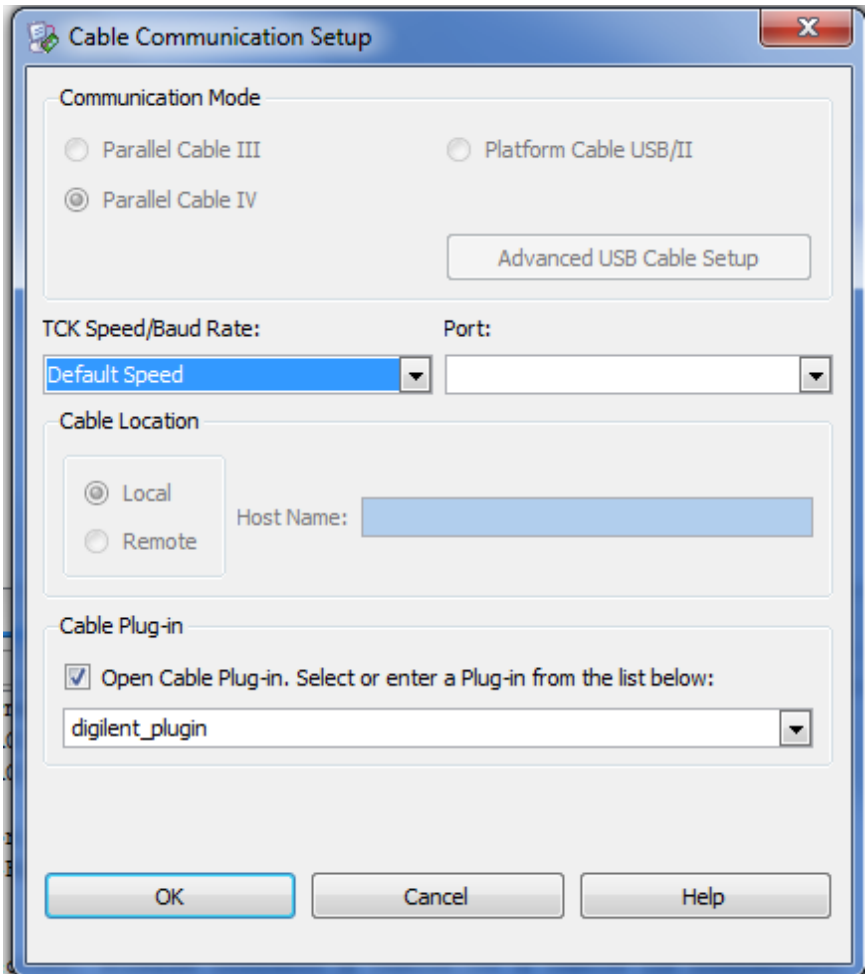


Рисунок 6.1.10 — Вид окна Cable Communication Setup при выборе правильных настроек

8-1-5. Выберите File → Initialize Chain. В выпавшем окне нажмите Yes. После чего вы перейдете в меню выбора конфигурационного файла для последующей загрузке на плату Atlys.

8-1-6. Выберите нужный конфигурационный файл по названию вашего проекта. Далее, в выпадающих окнах нажмите No, затем Yes.

8-1-7. Нажмите правой кнопкой мыши на изображение микросхемы. В выпадающем меню выберите Program.

8-1-8. Дождитесь окончания загрузки конфигурационного файла на FPGA.

Шаг 9. Наблюдение работы проекта на плате Atlys.

На этом шаге вам предлагается понаблюдать за поведением платы Atlys.

9-1. Наблюдение работы проекта на плате Atlys.

9-1-1. Для получения визуального результата требуется нажать на выбранную в ходе проекта кнопку и увидеть один из светодиодов горящим. Это будет свидетельствовать об успешном выполнении проекта.

Проекты для самостоятельной работы

- При нажатии одной из 3-х кнопок загораются все 8-мь светодиодов, расположенных на плате Atlys;
- При нажатии первой кнопки загораются четные светодиоды, при нажатии второй кнопки загораются нечетные светодиоды, при нажатии третьей кнопки, загораются все 8-мь светодиодов. При нажатии двух или трех кнопок сразу, светодиоды не загораются.

ЗАКЛЮЧЕНИЕ

По окончании данной лабораторной работы вы получили представление как о среде проектирования Xilinx ISE, так и о маршруте проектирования для FPGA. Также вы познакомились с базовыми конструкциями языка Verilog HDL. Кроме этого вы научились загружать созданный в ходе работы над проектом конфигурационный файл в FPGA.

6.2 Лабораторная работа №2

Проектирование простых комбинационных схем. Моделирование проекта

Цель работы: изучение возможностей языка Verilog HDL по описанию комбинационных схем, изучение принципов построения тестовых окружений, выполнение функционального тестирования разрабатываемого устройства.

После выполнения работы вы:

- научитесь пользоваться операторами языка Verilog HDL для создания комбинационных схем;
- научитесь применять конструкции языка Verilog HDL для реализации иерархического проектирования;
- познакомитесь с понятием тестового окружения;
- научитесь создавать простейшие тестовые окружения для разрабатываемых устройств.

ВВЕДЕНИЕ

Целью данной лабораторной работы является получение более глубокого представления о средствах и методах проектирования комбинационных схем с помощью языка Verilog HDL. При этом в ходе работы над предлагаемым проектом будут использоваться конструкции языка, основанные на блоке **always**. Также данная лабораторная работа представляет собой введение в функциональное тестирование, реализуемое с помощью программы Xilinx ISIM. При этом, для проведения функционального тестирования в ходе выполнения работы потребуется описать тестовое окружение, которое будет включать в себя как тестируемый модуль, так и некоторую логику генерации базовых тестовых последовательностей.

При этом, в ходе выполнения работы, будут получены как результаты

моделирования, так и полностью функционирующее устройство, реализующее предложенные функции.

ОПИСАНИЕ ЗАДАЧИ

В качестве базовой задачи для данной лабораторной работы будет являться создание упрощенного блока АЛУ (Арифметико-Логическое Устройство), осуществляющего некоторый набор арифметических и логических операций над 4-х разрядными входными операндами. В качестве выполняемых операций разрабатываемый блок будет реализовывать: сложение, вычитание, побитовое И, побитовое ИЛИ, а также операцию сравнения операндов, с выбором наибольшего операнда в качестве результата. Выбор операций будет задаваться тремя контрольными сигналами, т.к. количество выполняемых операций равно пяти. Код операции, задающий конкретную операцию может быть выбран произвольным образом. Четвертый контрольный сигнал будет определять выбор второго операнда между значением, поступающим на вход АЛУ и некоторой жестко заданной константой. В качестве результата работы, блок будет выдавать 4-х разрядный результат арифметических или логических операций. Кроме этого, выходными сигналами блока будут являться 2 флага: флаг переноса и флаг равенства результатов при операции сравнения.

Входные операнды будут задаваться с помощью 8-ми переключателей, а управляющие сигналы будут задаваться с помощью кнопок, расположенных на плате Atlys. Результат работы блока будет отображаться на 6-ти светодиодах (4-ре светодиода служат для отображения результата, оставшиеся 2 для отображения флагов).

Структурная схема проекта этой лабораторной работы представлена на рисунке 6.2.1.

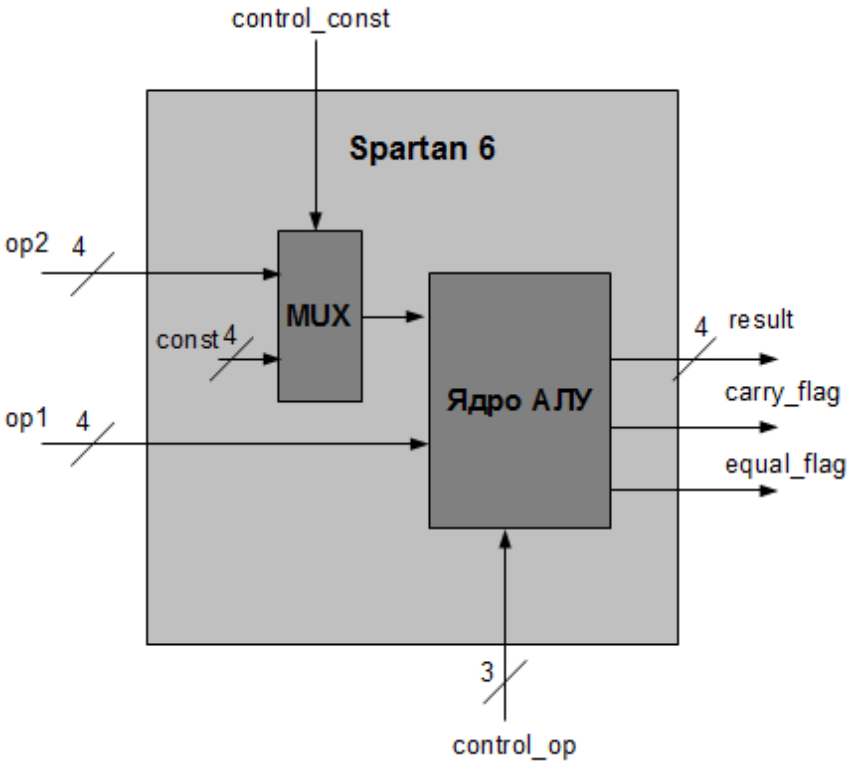


Рисунок 6.2.1 — Структурная схема проекта лабораторной работы №2

ВЫПОЛНЕНИЕ РАБОТЫ

Шаг 1. Создание нового проекта в среде ISE.

Для начала выполнения данной работы требуется создать новый проект, выполнив при этом определенную, аналогичную представленную в лабораторной работе №1, последовательность действий.

1-1. Создание нового проекта для целевой FPGA Spartan 6, находящейся на плате Atlys.

1-1-1. Запустите ISE и создайте новый проект в котором укажите все требуемые опции. При возникновении трудностей при создании нового проекта, обратитесь к лабораторной работе №1.

Шаг 2. Создание и описание файлов проекта с помощью языка Verilog HDL в среде ISE.

На данном шаге требуется описать поведение разрабатываемого модуля. При этом, для описания поведения модуля будут использоваться все конструкции языка Verilog HDL, применимые для описания комбинационных схем. Вам потребуется

ознакомиться с этими конструкциями, обратившись к главе 4.

2-1. Создание нового модуля верхнего уровня с помощью среды ISE.

2-1-1. Создайте новый модуль верхнего уровня, используя графический интерфейс среды ISE. За подробным описанием процесса создания нового модуля обратитесь к лабораторной работе №1.

2-2. Описание разрабатываемого модуля с помощью Verilog HDL.

2-2-1. Введите описание проектируемого модуля с помощью встроенного текстового редактора. Вы можете увидеть окно Project Navigator после выполнения этого шага на рисунке 6.2.2.

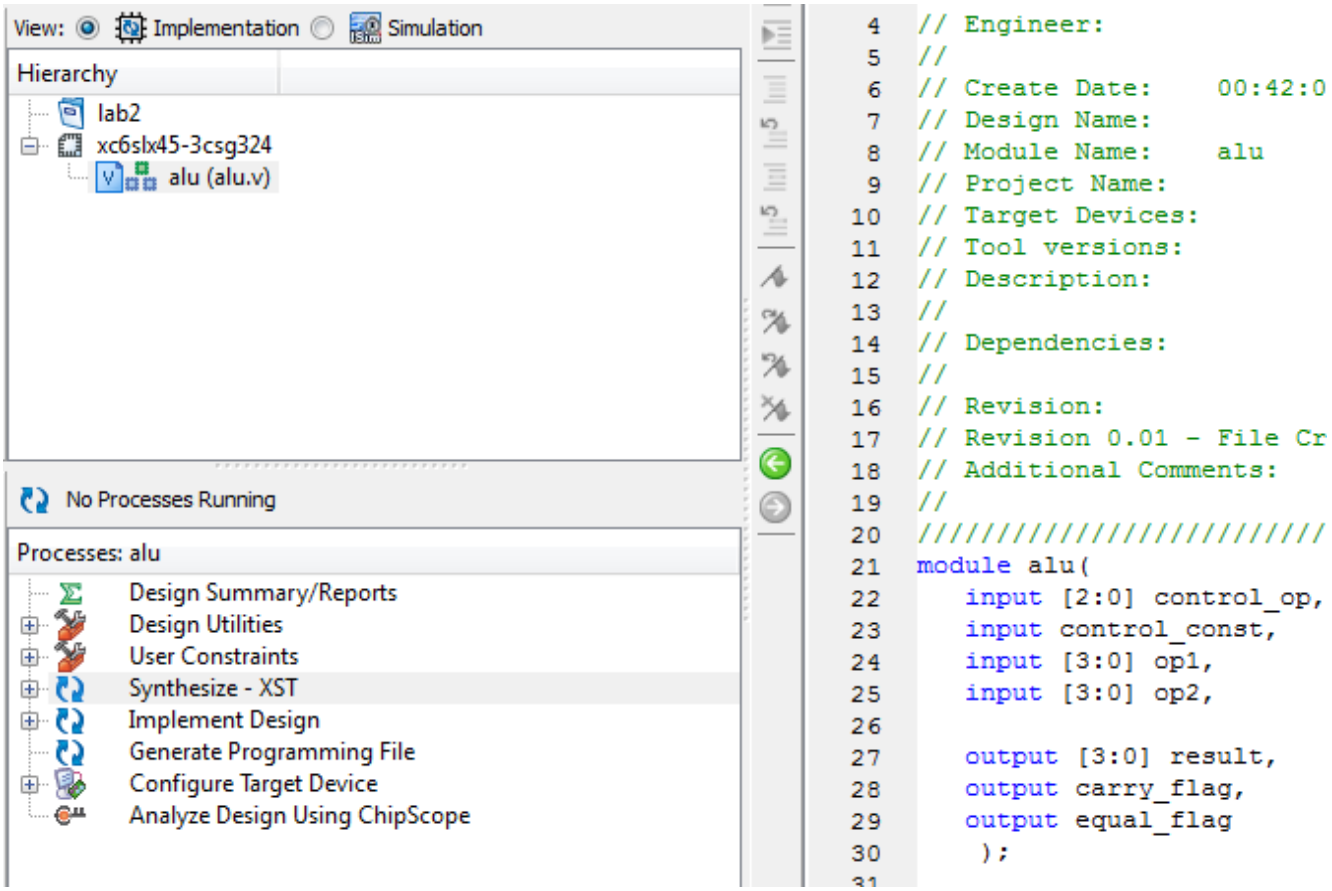


Рисунок 6.2.2 — Вид окна Project Navigator после добавления описания модуля

Шаг 3. Создание тестового окружения для проектируемого модуля

Для любых проектов, более сложных, чем описание нескольких логических функций, требуется реализовать процесс функционального тестирования. Данный

процесс подразумевает моделирование поведения разрабатываемого блока на некотором входном наборе тестовых значений. При этом, для реализации процесса тестирования требуется реализовать некоторую абстракцию, которая будет управлять входными сигналами тестируемого блока и анализировать его выходные сигналы. Такая абстракция и называется тестовым окружением.

Также стоит отметить, что тестовое окружение, в своем самом простом исполнении, представляет собой описание модуля, в котором могут использоваться все конструкции языка Verilog HDL.

3-1. Создание нового файла, содержащего описание тестового окружения с помощью среды ISE.

3-1-1. Выберете **Project → New source. После этого откроется окно меню создания нового файла для проекта (Рисунок 6.2.3).**

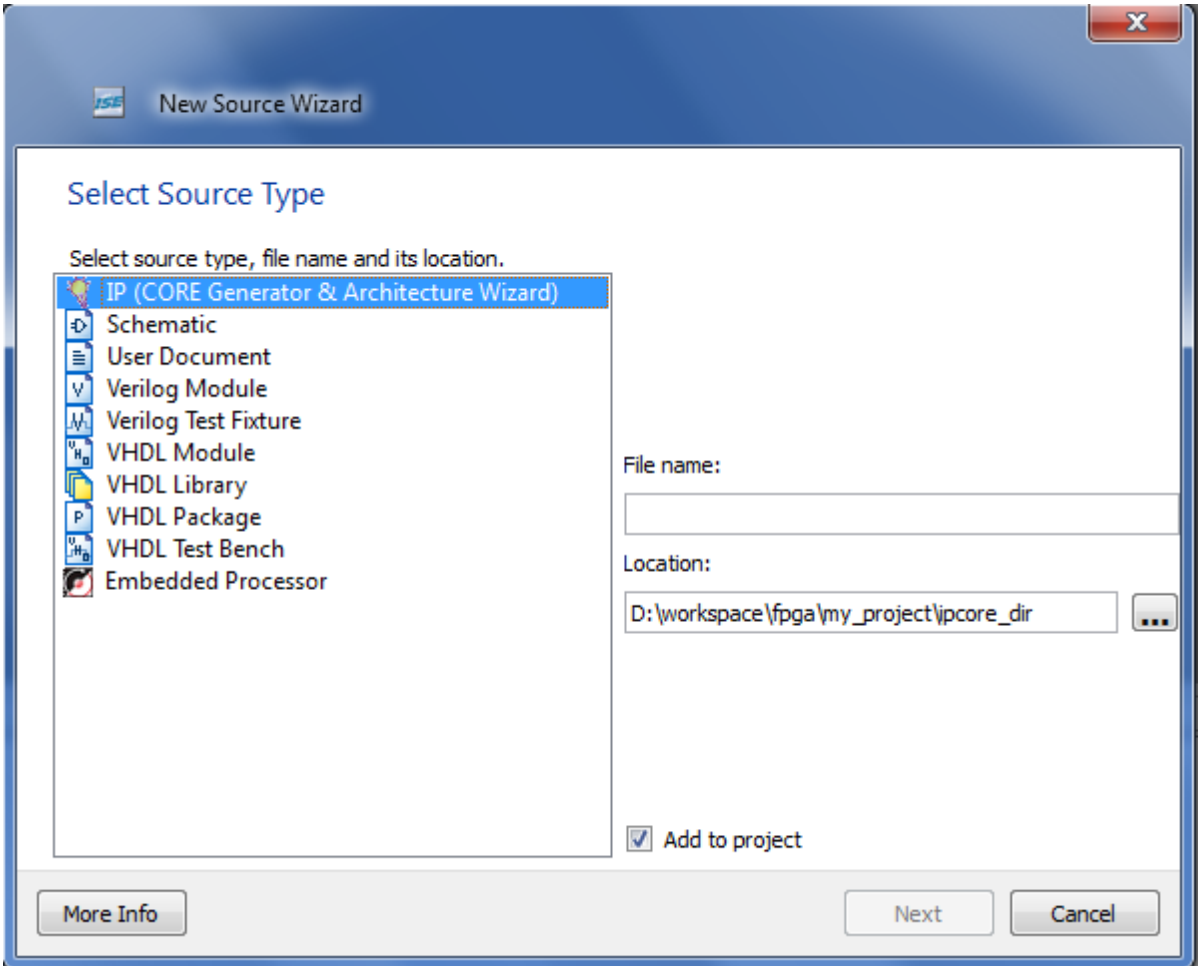


Рисунок 6.2.3 — Окно меню создания нового файла для проекта в среде ISE

3-1-2. В окне выберете Verilog Test Fixture. Также укажите название модуля. Нажмите 2 раза Next и Finish.

3-1-3. В окне Project Navigator перейдите на вкладку Simulation. После этого вы увидите ваш модуль, созданный для описания тестового окружения. Данный переход представлен на рисунке 6.2.4.

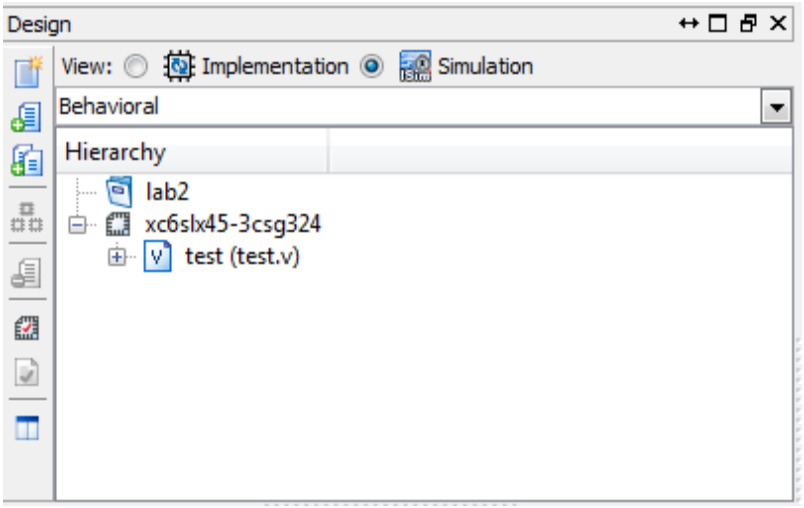


Рисунок 6.2.4 — Переход на вкладку, посвященную моделированию проекта

3-2. Реализация тестового окружения с помощью среды ISE.

3-2-1. Введите описание тестового окружения для проектируемого модуля с помощью встроенного текстового редактора.

Шаг 4. Моделирование проекта средствами ISIM.

После реализации тестового окружения требуется промоделировать проект и проанализировать его поведение под действием тестовых последовательностей. Для этого мы будем использовать встроенную в ISE программу для моделирования, ISIM.

4-1. Моделирование проекта.

4-1-1. Во вкладке Design в иерархии проекта нажмите левой кнопкой мыши на файле, содержащем описание тестового окружения.

4-1-2. Раскройте меню Isim Simulator и нажмите дважды левой кнопкой мыши на Simulate Behavioral Model. При этом, откроется новое окно, представляющее собой симулятор ISIM. Основные панели окна симулятора представлены на рисунке 6.2.5.

4-1-3. Перейдите во вкладку Default.wcfg. Выберите View → Zoom → To full view. После этого вы увидите временные диаграммы, представляющие собой результаты моделирования. Пример временных диаграмм представлен на рисунке 6.2.6.

4-1-4. Проанализируйте результаты моделирования на наличие ошибок в работе разрабатываемого модуля. В случае обнаружения ошибок, исправьте описание модуля.

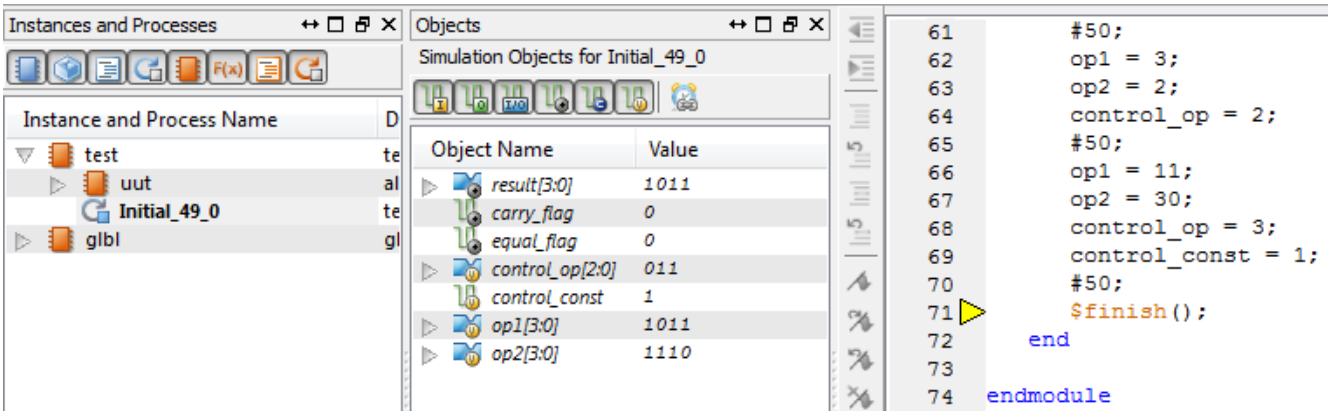


Рисунок 6.2.5 — Основные панели окна симулятора ISIM

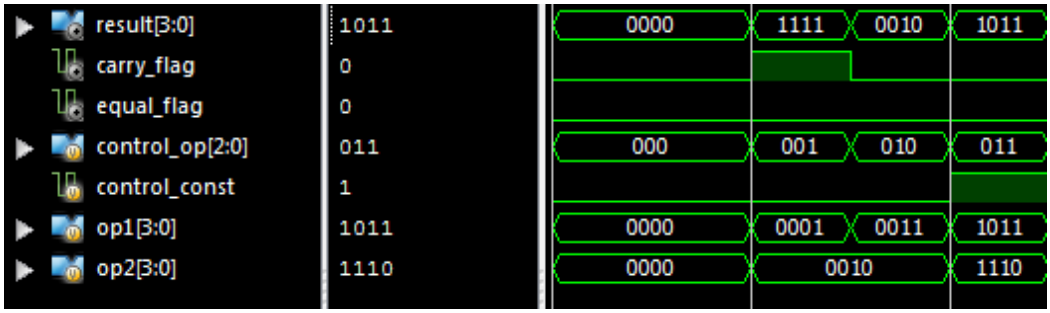


Рисунок 6.2.6 — Пример временных диаграмм

Шаг 5. Создание файла, содержащего ограничения проекта по размещению.

На данном этапе требуется создать файл, содержащий ограничения проекта (User Constraints File). В данном проекте нам требуется соединить входные сигналы как с переключателями, так и с кнопками, а выходные сигнал с набором светодиодов, представленных на плате Atlys.

5-1. Создание нового файла, содержащего ограничения проекта с помощью среды ISE.

5-1-1. Создайте файл, содержащий ограничения проекта по размещению. Для этого воспользуйтесь программой PlanAhead или встроенным текстовым редактором, для ручного ввода конструкций, описывающих ограничения проекта. В случае возникновения затруднений при использовании PlanAhead, обратитесь к лабораторной работе №1.

Шаг 6. Проведение синтеза и имплементации всего проекта. А также создание конфигурационного файла.

На данном шаге требуется выполнить синтез и имплементацию всего проекта. Кроме этого, требуется создать конфигурационный файл.

6-1. Проведение синтеза всего проекта.

6-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Synthesize – XST. В случае возникновения ошибок требуется исправить описание проектируемого модуля.

6-1-2. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Implement Design. В случае возникновения ошибок требуется исправить файл, содержащий ограничения по размещению проекта или описание проектируемого модуля.

6-1-3. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Generate Programming File.

6-1-4. Прежде чем переходить к следующим шагам, убедитесь, что вкладка Design имеет вид, схожий с Рисунком 6.2.7. Проект также может иметь некоторое количество предупреждений, которые необходимо проанализировать и принять решение о их исправлении или игнорировании.

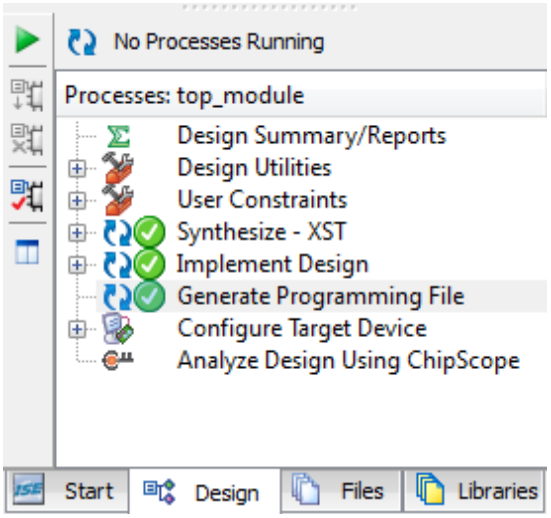


Рисунок 6.2.7 — Вид вкладки Design после окончания процессов синтеза, имплементации и создания конфигурационного файла.

Шаг 7. Подключение платы Atlys к компьютеру.

На данном шаге требуется подключить тестовую плату Atlys к компьютеру.

7-1. Подключение платы Atlys к компьютеру.

7-1-1. Подключите плату Atlys к источнику питания.

7-1-2. Включите плату с использованием переключателя, расположенного на плате.

7-1-3. Подключите плату с помощью USB кабеля к компьютеру.

7-1-4. При возникновении затруднений, обратитесь к документу Atlys_rm.pdf, поставляемым вместе с платой Atlys.

Шаг 8. Загрузка созданного конфигурационного файла на плату Atlys.

На данном шаге мы произведем загрузку конфигурационного файла в плату Atlys для конфигурирования, находящегося на ней, FPGA Spartan 6.

8-1. Загрузка созданного конфигурационного файла на плату Atlys.

8-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мышки на Configure Target Device. При этом откроется меню программы IMPACT. Осуществите необходимую настройку программы IMPACT. В случае возникновения затруднений, обратитесь к описанию лабораторной работы №1.

8-1-2. Сконфигурируйте FPGA Spartan 6, расположенный на плате Atlys.

Шаг 9. Наблюдение работы проекта на плате Atlys.

На этом шаге вам предлагается понаблюдать за поведением платы Atlys.

9-1. Наблюдение работы проекта на плате Atlys.

9-1-1. Для получения визуального результата требуется выбрать значения входных операндов с помощью переключателей, а также выбрать выполняемую разработанным модулем операцию с помощью кнопок. Результат работы будет отображаться на наборе светодиодов, расположенных на плате Atlys. Корректные результаты визуального наблюдения будут свидетельствовать об успешном выполнении проекта.

Проекты для самостоятельной работы

- Добавить флаг нулевого результата для операций сложения и вычитания. При этом значение флага будет отображаться на одном из светодиодов, расположенном на тестовой плате Atlys;
- Добавить операции логического сдвига влево и вправо на один разряд;
- Реализовать операции сложения и вычитания на одном и том же сумматоре.

ЗАКЛЮЧЕНИЕ

По окончании данной лабораторной работы вы получили представление о конструкциях языка Verilog HDL, используемых для описания комбинационных схем. Также вы научились создавать базовые тестовые окружения для разрабатываемых блоков и осуществлять процесс моделирования. Кроме этого вы получили навыки работы с временными диаграммами для анализа корректности функционирования разрабатываемых модулей.

6.3 Лабораторная работа №3

Проектирование простых последовательных схем. Реализация тактирования

Цель работы: изучение принципов построения цифровых схем, содержащих элементы памяти, изучение возможностей языка Verilog HDL по описанию последовательных схем, реализация тактирования в FPGA и использование глобальных временных ограничений.

После выполнения работы вы:

- научитесь пользоваться операторами и конструкциями языка Verilog HDL для создания последовательных схем;
- научитесь проектировать цифровые устройства, содержащие память;
- получите представление о глобальных временных ограничениях;
- получите базовое представление об утилите Core Generator;
- научитесь реализовывать тактирование в FPGA.

ВВЕДЕНИЕ

Целью данной лабораторной работы является получение базового представления о проектировании цифровых устройств, содержащих в своем составе элементы памяти. При этом в ходе работы над предлагаемым проектом будут использоваться конструкции языка и шаблоны, служащие для описания триггеров и регистров. Описание подобных элементов памяти базируется на блоках **always**, «срабатывающих» по некоторым событиям, основанным на переключении тактового сигнала (**posedge**, **negedge**). Реализация самого тактового сигнала в свою очередь предполагает использование специальных встроенных в FPGA блоков, таких как CMT (содержащих в своем составе блоки DCM и PLL). Доступ к подобным встроенным блокам осуществляется через специальную утилиту, называющуюся Core Generator.

Как и в предыдущих лабораторных работах, в ходе выполнения работы, будут получены как результаты моделирования, так и полностью функционирующее устройство, реализующее поставленную задачу.

ОПИСАНИЕ ЗАДАЧИ

Задачей данной лабораторной работы будет являться создание блока, изменяющего свои выходные сигналы с определенной частотой. В текущей реализации, такой блок будет иметь единственный выходной сигнал, разрядностью в 1 бит, который будет инвертировать свое значение раз в секунду. Кроме этого, данный блок будет поддерживать 8-ми разрядный входной сигнал, отвечающий за увеличение частоты изменения выходного сигнала. Т. е. чем больше значение будет подано на вход блока, тем чаще будет происходить изменение выходного сигнала.

Входное 8-ми разрядное значение будут задаваться с помощью 8-ми переключателей. При этом выходной 1-но разрядный сигнал будет соединен с любым из светодиодов, расположенных на плате Atlys. Также в проекте потребуется реализовать тактовый сигнал и сигнал асинхронного или синхронного сброса всех триггеров.

При этом главной задачей проекта будет являться наблюдение мигания светодиода с различной, зрительно различимой, частотой.

Ядром данного проекта будет служить обыкновенный счетчик. Также проект будет содержать аппаратный блок CMT для реализации тактирования. Структурная схема проекта этой лабораторной работы представлена на рисунке 6.3.1.

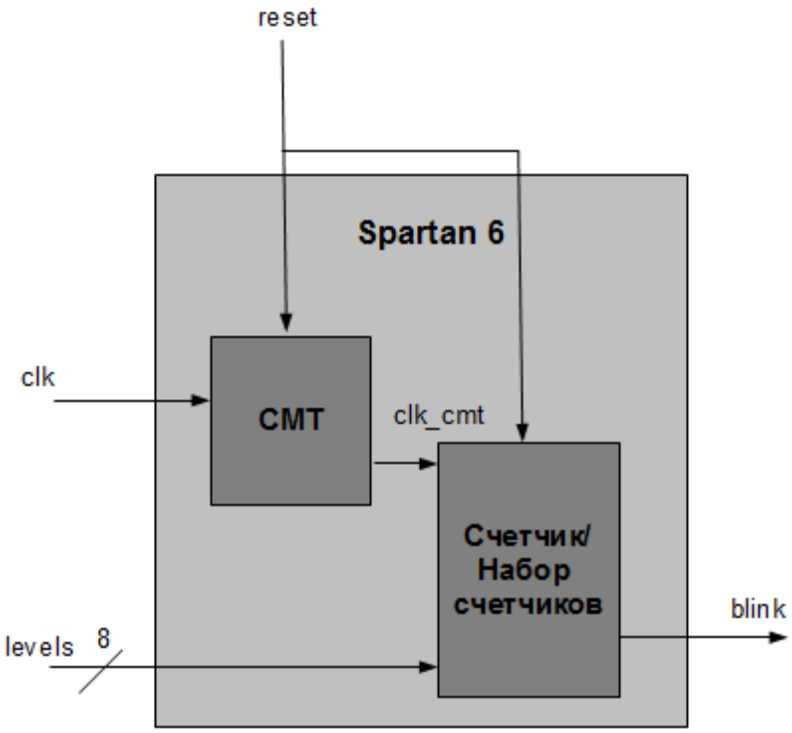


Рисунок 6.3.1 — Структурная схема проекта лабораторной работы №3

ВЫПОЛНЕНИЕ РАБОТЫ

Шаг 1. Создание нового проекта в среде ISE.

Для начала выполнения данной работы требуется создать новый проект, выполнив при этом определенную, аналогичную представленную в лабораторной работе №1, последовательность действий.

1-1. Создание нового проекта для целевой FPGA Spartan 6, находящейся на плате Atlys.

1-1-1. Запустите ISE и создайте новый проект в котором укажите все требуемые опции. При возникновении трудностей при создании нового проекта, обратитесь к лабораторной работе №1.

1-1-2. Создайте новый модуль верхнего уровня, используя графический интерфейс среды ISE. За подробным описанием процесса создания нового модуля обратитесь к лабораторной работе №1.

Шаг 2. Создание экземпляра аппаратного блока CMT

Для любых проектов, содержащих элементы памяти, требуется реализовывать

тактирование. Тактирование заключается в подаче тактового сигнала, генерируемого внешним осциллятором, на один из заранее выделенных для этих целей входов FPGA. После этого поданный тактовый сигнал может использоваться для тактирования любых элементов памяти, содержащихся в FPGA. При этом, стоит отметить, что нельзя сразу использовать для тактирования поданный тактовый сигнал. Преждевременно его нужно обработать специальными аппаратными блоками, содержащимися во всех современных FPGA. В случае Spartan 6, таким блоком является модуль CMT. CMT служит как для фильтрации входного тактового сигнала, так и для генерации используемых внутри FPGA тактовых сигналов различной частоты и фазы. На данном шаге требуется добавить такой модуль в разрабатываемый проект. Добавление подобного модуля осуществляется через утилиту Core Generator.

2-1. Конфигурирование нового аппаратного блока CMT.

2-1-1. В окне Hierarchy щелкните правой кнопкой мышки и выберите Create New Source.

2-1-2. В окне New Source выберите IP (CORE Generator & Architecture Wizard). Введите название модуля и нажмите Next.

2-1-3. В окне Select IP раскройте вкладку FPGA Features and Design Clocking и выберите Clocking Wizard. Далее нажмите Next и Finish. Данная операция представлена на рисунке 6.3.2.

2-1-4. В появившемся диалоге, в окне Clocking Features / Input Clocks оставьте включенными опции по умолчанию и нажмите Next.

2-1-5. В окне Output Clock Settings введите 100 MHz в качестве выходной частоты и нажмите Next.

2-1-6. В окне I/O and Feedback уберите галочку с RESET и нажмите Next. Далее нажмите 3 раза Next, а затем Generate.

2-1-7. Дождитесь успешной генерации блока. Блок должен появиться во вкладке Hierarchy в среде ISE.

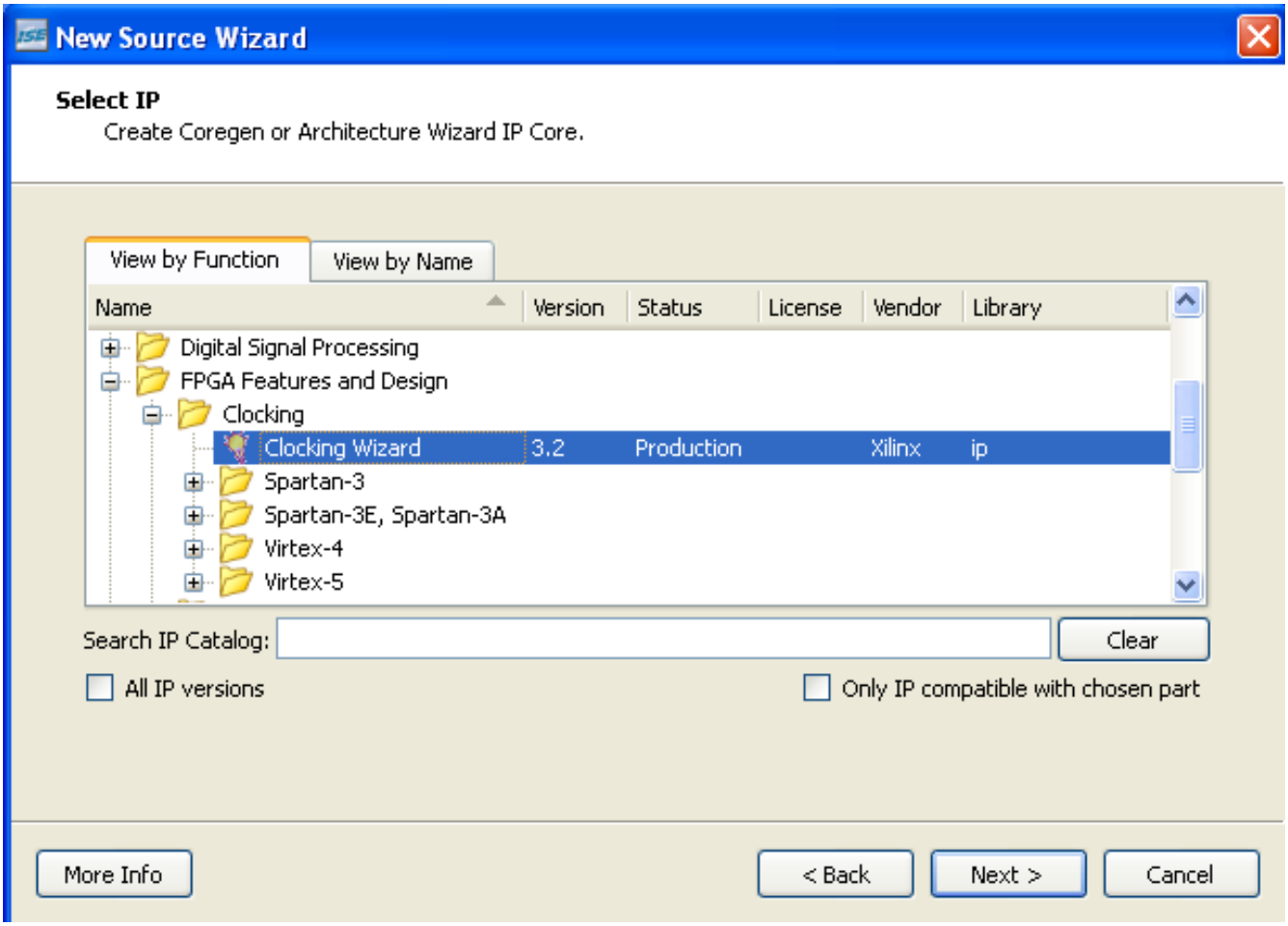


Рисунок 6.3.2 — Вид вкладок в окне New Source Wizard для выбора модуля CMT

2-2. Создание экземпляра модуля CMT в описании разрабатываемого проекта.

2-2-1. В окне Hierarchy щелкните левой кнопкой мышки на созданный модуль CMT. Далее во вкладке Processes нажмите дважды левой кнопкой мышки на View HDL Instantiation Template. Скопируйте открывшееся описание в разрабатываемый проект.

2-2-2. В описании разрабатываемого проекта подключите все требуемые сигналы ввода-вывода к экземпляру модуля CMT. Также укажите имя созданного экземпляра модуля CMT.

2-2-3. Обратите особое внимание на то, что начиная с этого момента все элементы памяти в проекте будут тактироваться выходным сигналом из модуля CMT, а не входным тактовым сигналом.

Шаг 3. Описание файлов проекта с помощью языка Verilog HDL в среде ISE.

На данном шаге требуется описать поведение разрабатываемого модуля. При этом, для описания поведения модуля будут использоваться все конструкции языка Verilog HDL, применимые как для описания комбинационных, так и последовательных схем. Вам потребуется ознакомиться с этими конструкциями, обратившись к главе 4. При этом, в ходе описания проекта требуется также подключить созданный на предыдущем шаге блок CMT.

3-1. Описание разрабатываемого модуля с помощью Verilog HDL.

3-1-1. Введите описание проектируемого модуля с помощью встроенного текстового редактора. Вы можете увидеть окно Project Navigator после выполнения этого шага на рисунке 6.3.3.

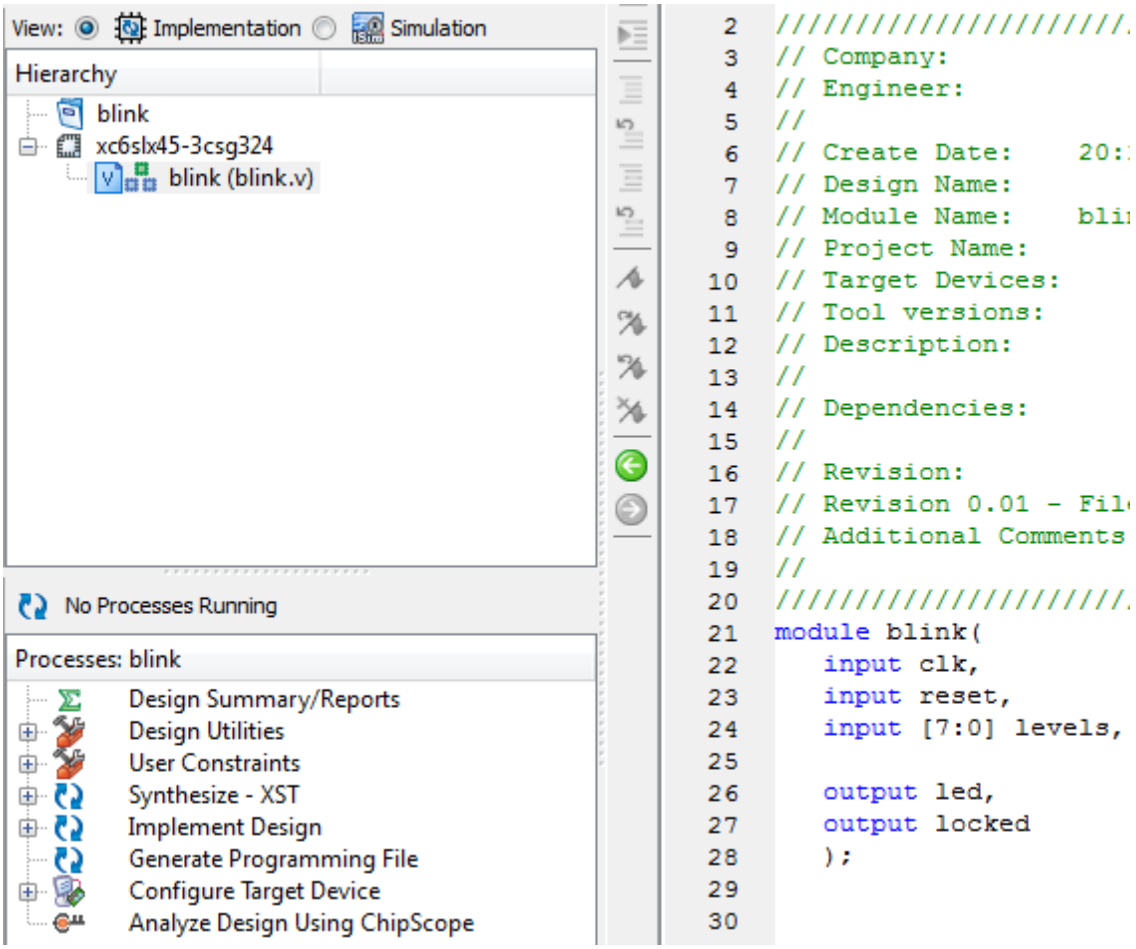


Рисунок 6.3.3 — Вид окна Project Navigator после добавления описания модуля

Шаг 4. Создание тестового окружения для проектируемого модуля

Для данного модуля требуется создать тестовое окружение и осуществить моделирование проекта. Т. к. значимые операции в данном проекте выполняются с крайне низкой частотой (порядка 1Hz), то ожидание появления результатов может занять довольно длительное время. Для большей эффективности моделирования требуется более тщательно продумать процесс тестирования и использовать ряд вспомогательных конструкций языка Verilog HDL.

4-1. Создание нового файла, содержащего описание тестового окружения с помощью среды ISE.

4-1-1. Создайте новый модуль для описания тестового окружения посредством графического интерфейса среды ISE. За подробным описанием процесса создания нового модуля для описания тестового окружения обратитесь к лабораторной работе №2.

4-1-2. Введите описание тестового окружения для проектируемого модуля с помощью встроенного текстового редактора.

Шаг 5. Моделирование проекта средствами ISIM.

После реализации тестового окружения требуется произвести моделирование проекта и проанализировать его поведение под действием тестовых последовательностей. Весь процесс моделирования аналогичен описанному в лабораторной работе №2.

5-1. Моделирование проекта.

5-1-1. Проведите моделирование проекта средствами ISIM. За подробным описанием процесса моделирования обратитесь к лабораторной работе №2.

Шаг 6. Создание файла, содержащего ограничения проекта по размещению.

На данном этапе требуется создать файл, содержащий ограничения проекта (User Constraints File). В данном проекте нам требуется соединить входной 8-ми разрядный сигнал с переключателями, входной тактовый сигнал с жестко заданным входом L15 (местом подключения осциллятора к FPGA), входной сигнал

сброса с одной из кнопок, а выходной сигнал с одним из светодиодов, расположенных на плате Atlys.

6-1. Создание нового файла, содержащего ограничения проекта с помощью среды ISE.

6-1-1. Создайте файл, содержащий ограничения проекта по размещению. Для этого воспользуйтесь программой PlanAhead или встроенным текстовым редактором, для ручного ввода конструкций, описывающих ограничения проекта. В случае возникновения затруднений при использовании PlanAhead, обратитесь к лабораторной работе №1.

Шаг 7. Добавление в файл, описывающий ограничения проекта, глобальных временных ограничений.

На данном шаге требуется дополнить созданный UCF файл глобальными временными ограничениями. Данные ограничения служат в качестве управляющих значений для этапов синтеза и имплементации и влияют как на результирующую пиковую частоту проекта, так и на занимаемую площадь. Более подробно о глобальных временных ограничениям можно прочитать в главе 3.

7-1. Добавление глобальных временных ограничений в проект.

7-1-1. В окне Processes среды ISE раскройте вкладку User Constraints и дважды щелкните левой кнопкой мышки на Create Timing Constraints. На рисунке 6.3.4 представлен вид окна Processes при выборе Create Timing Constraints.

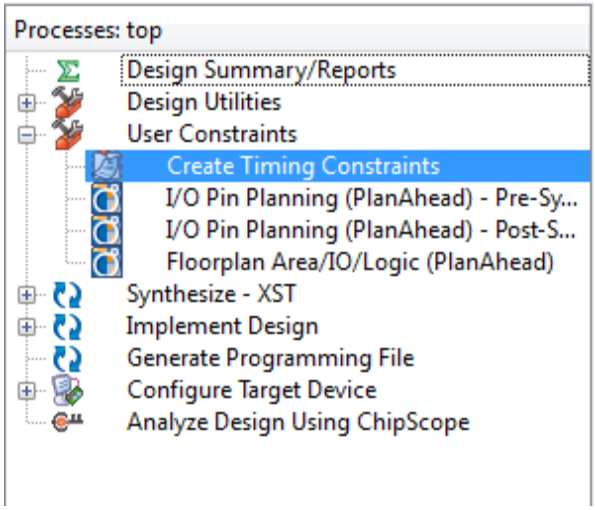


Рисунок 6.3.4 – Вид вкладок в окне Process при выборе Create Timing Constraints

7-1-2. В окне Timing Constraints во вкладке Constraint Type выберите Clock Domains. На рисунке 6.3.5 представлен вид вкладки Constraint Type при выборе Clock Domains.

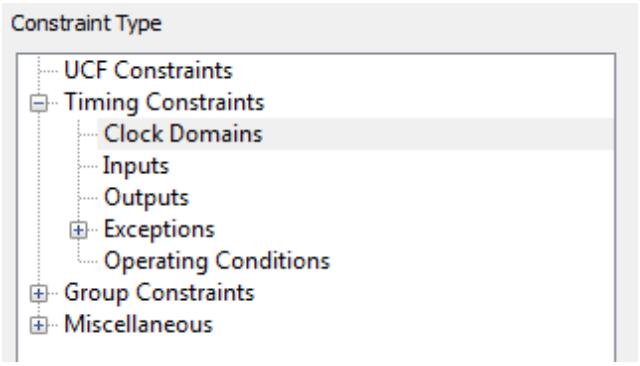


Рисунок 6.3.5 – Вид вкладок в окне Constraint Type при выборе Clock Domains

7-1-3. Нажмите дважды левой кнопкой мышки на Clk в окне Unconstrained Clocks. При этом действии будет показано новое диалоговое окно — Clock Period. Вид данного диалогового окна представлен на рисунке 6.3.6.

7-1-4. Напишите новое значение настройки Time, равное 10ns (т. к. входной тактовый сигнал равен 100 MHz). Далее нажмите Ok.

7-1-5. В окне Constraint Type нажмите дважды левой кнопкой мышки на вкладке Inputs. Нажмите Next. Далее нажмите Finish. Стоит отметить, что проект, предложенный в данной лабораторной работе не содержит существенных ограничений OFFSET IN. Выполненные в текущем пункте действия служат больше в ознакомительных целях.

7-1-6. В окне Constraint Type нажмите дважды левой кнопкой мышки на вкладке Outputs. Нажмите Ok. Стоит отметить, что проект, предложенный в данной лабораторной работе не содержит существенных ограничений OFFSET OUT. Выполненные в текущем пункте действия также служат больше в ознакомительных целях.

7-1-7. Сохраните ограничения и закройте текущее диалоговое окно.

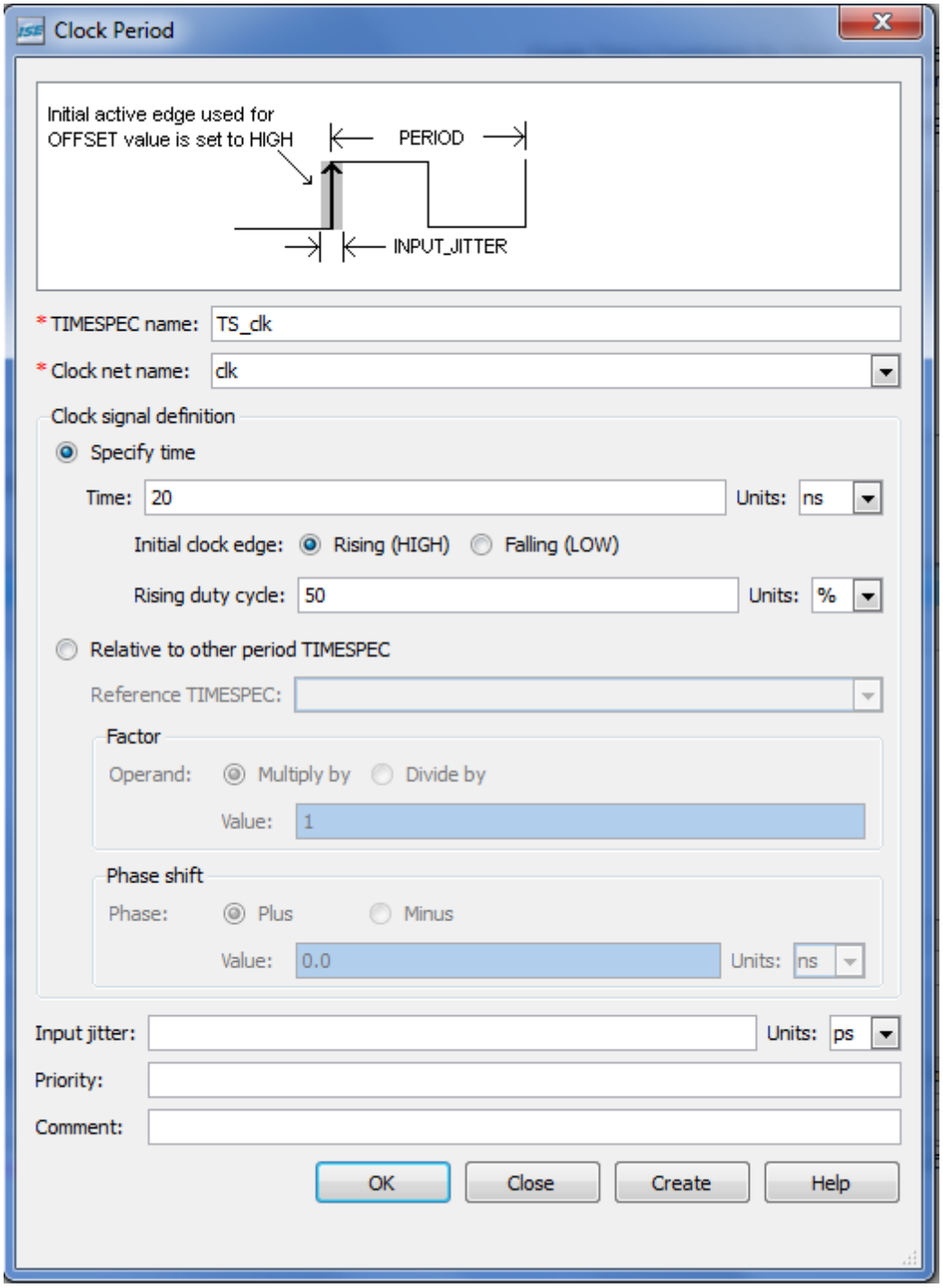


Рисунок 6.3.6 – Вид диалогового окна Clock Period

Шаг 8. Проведение синтеза и имплементации всего проекта. А также создание конфигурационного файла.

На данном шаге требуется выполнить синтез и имплементацию всего проекта. Кроме этого, требуется создать конфигурационный файл.

8-1. Выполнение оставшихся шагов маршрута проектирования.

8-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Synthesize – XST. В случае возникновения ошибок требуется исправить описание проектируемого модуля.

8-1-2. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Implement Design. В случае возникновения ошибок требуется исправить файл, содержащий ограничения по размещению проекта или описание проектируемого модуля.

8-1-3. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Generate Programming File.

8-1-4. Прежде чем переходить к следующим шагам, убедитесь, что вкладка Design имеет вид, схожий с Рисунком 6.3.7. Проект также может иметь некоторое количество предупреждений, которые необходимо проанализировать и принять решение о их исправлении или игнорировании.

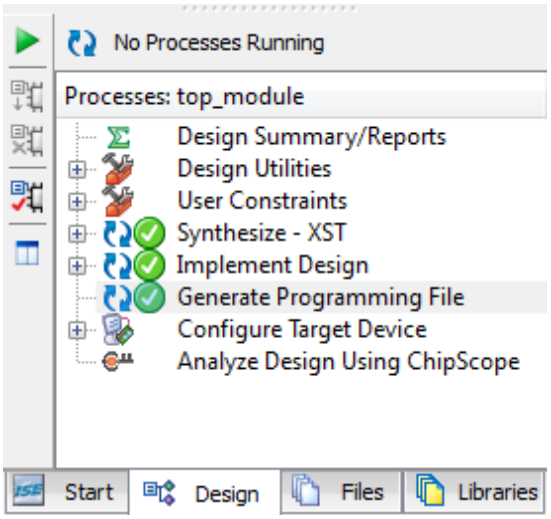


Рисунок 6.3.7 — Вид вкладки Design после окончания процессов синтеза, имплементации и создания конфигурационного файла.

Шаг 9. Подключение платы Atlys к компьютеру.

На данном шаге требуется подключить тестовую плату Atlys к компьютеру.

9-1. Подключение платы Atlys к компьютеру.

9-1-1. Подключите плату Atlys к источнику питания.

9-1-2. Включите плату с использованием переключателя, расположенного на плате.

9-1-3. Подключите плату с помощью USB кабеля к компьютеру.

9-1-4. При возникновении затруднений, обратитесь к документу Atlys_rm.pdf,

поставляемым вместе с платой Atlys.

Шаг 10. Загрузка созданного конфигурационного файла на плату Atlys.

На данном шаге мы произведем загрузку конфигурационного файла в плату Atlys для конфигурирования, находящегося на ней, FPGA Spartan 6.

10-1. Загрузка созданного конфигурационного файла на плату Atlys.

10-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мышки на Configure Target Device. При этом откроется меню программы IMPACT. Осуществите необходимую настройку программы IMPACT. В случае возникновения затруднений, обратитесь к описанию лабораторной работы №1.

10-1-2. Сконфигурируйте FPGA Spartan 6, расположенный на плате Atlys.

Шаг 11. Наблюдение работы проекта на плате Atlys.

На этом шаге вам предлагается понаблюдать за поведением платы Atlys.

11-1. Наблюдение работы проекта на плате Atlys.

11-1-1. Для получения визуального результата требуется произвести сброс схемы, нажатием на соответствующую, выбранную в ходе проекта, кнопку. Далее можно наблюдать мигание светодиода с частотой, примерно равной 1Hz. При изменении значения входного 8-ми разрядного сигнала, путем установки соответствующих переключателей, можно наблюдать увеличение частоты мигания светодиода. Корректные результаты визуального наблюдения будут свидетельствовать об успешном выполнении проекта.

Проекты для самостоятельной работы

- Реализовать счетчик, понижения частоты с использованием каскада счетчиков более низкой разрядности;
- Произвести параметризацию модуля, реализующего счетчик, по разрядности и использовать несколько экземпляров данного модуля с разными параметрами при реализации предыдущего задания.

ЗАКЛЮЧЕНИЕ

По окончании данной лабораторной работы вы научились проектировать цифровые устройства, содержащие элементы памяти, и получили представление о конструкциях языка Verilog HDL, используемых для описания последовательных схем. Также вы научились реализовывать тактирование и дополнять ограничения проекта глобальными временными ограничениями. Кроме этого, вы научились пользоваться утилитой Core Generator для использования встроенных аппаратных блоков в ваших проектах.

6.4 Лабораторная работа №4
Проектирование конечных автоматов

Цель работы: анализ возможностей, предоставляемых конечными автоматами, изучение принципов построения цифровых схем, содержащих конечные автоматы, изучение возможностей языка Verilog HDL и применяемых техник по описанию конечных автоматов, изучение возможностей статического временного анализа (Static Timing Analysis).

После выполнения работы вы:

- научитесь пользоваться операторами и конструкциями языка Verilog HDL для создания конечных автоматов;
- научитесь проектировать цифровые устройства, содержащие конечные автоматы;
- научитесь генерировать и просматривать отчеты статического временного анализа.

ВВЕДЕНИЕ

Целью данной лабораторной работы является получение базового представления о проектировании цифровых устройств, содержащих в своем составе конечные автоматы. Конечные автоматы представляют собой устройства, содержащие некоторое количество заранее определенных состояний и функций переходов между ними. При этом, реализация конечных автоматов представляет собой некоторый набор регистров, содержащих текущее состояние, а также набор вспомогательных логических элементов, реализующих переходы между реализованными состояниями. Соответственно, описание подобных модулей с помощью языка Verilog HDL не привносит никаких новых конструкций языка и базируется на обычных шаблонах и решениях для описания комбинационных и

последовательных схем. Стоит отметить, что часто при описании конечных автоматов удобно использовать конструкции Verilog HDL, описывающие константы. Т.е. такие конструкции как **localparam**, **`define** и т.д.

Как и в предыдущих лабораторных работах, в ходе выполнения работы, будут получены как результаты моделирования, так и полностью функционирующее устройство, реализующее поставленную задачу.

ОПИСАНИЕ ЗАДАЧИ

В данной лабораторной работе будет происходить фильтрация входного сигнала, представляющего собой результат нажатия на одну из кнопок, расположенных на плате Atlys.

При нажатии на любую из кнопок пользователем, ожидается, что произойдет единичная смена логического уровня. Т.е. при нажатии не активной кнопки, предполагается, что логический уровень изменится однократно с логического «0» до логической «1», а затем снова до логического «0». На самом деле это не совсем так. При нажатии кнопки, в течение некоторого интервала времени во время и после нажатия, возникает серия помех (glitch), которая может оказаться нежелательной при взаимодействии пользователя с разработанным устройством. Данная ситуация представлена на рисунке 6.4.1.

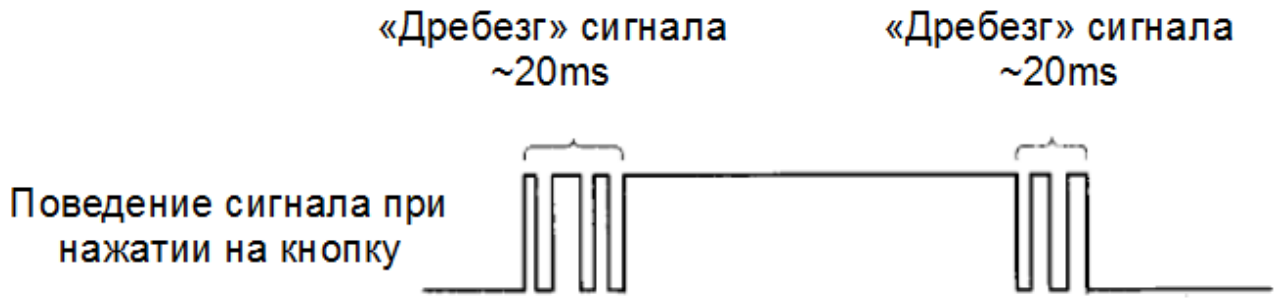


Рисунок 6.4.1 – Поведение сигнала, вызванное однократным нажатием на кнопку

Одним из возможных вариантов решения данной проблемы является

создание модуля, фильтрующего нежелательные помехи и вырабатывающего единичный импульс при однократном нажатии кнопки. Соответственно, целью данной работы является создание подобного модуля.

Ядром такого модуля будет являться конечный автомат, диаграмма переходов которого представлена на рисунке 6.4.2.

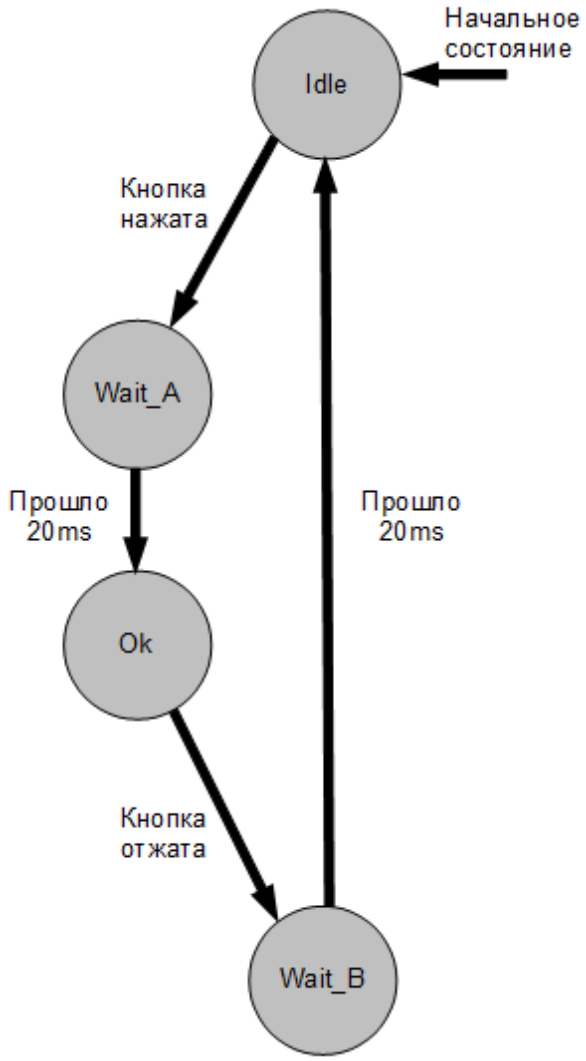


Рисунок 6.4.2 – Диаграмма переходов конечного автомата

Также модуль будет содержать набор счетчиков, служащих как для подсчета интервалов времени для фильтрации помех, так и для подсчета числа нажатий на выбранную в ходе проекта кнопку. Блок-схема модуля представлена на рисунке 6.4.3.

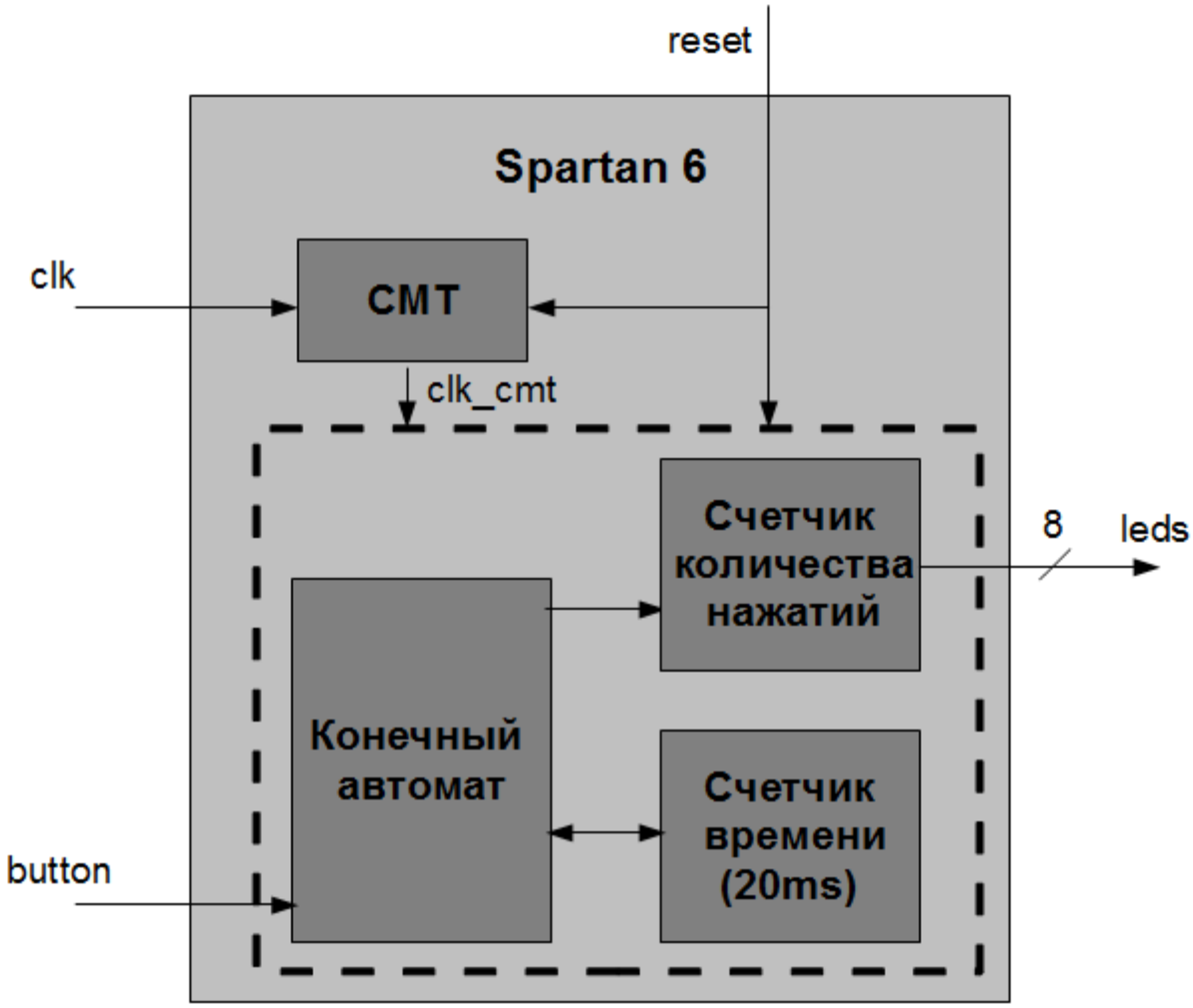


Рисунок 6.4.3 — Структурная схема проекта лабораторной работы №4

Как видно из рисунка 6.4.3 разрабатываемый модуль содержит 1 анализируемый входной сигнал (button), подключенный к одной из кнопок, а также 2 служебных входных сигнала, представляющие собой тактовый сигнал и сигнал асинхронного или синхронного сброса. Также модуль содержит выходную 8-ми разрядную шину (leds), соединенную с 8-мью светодиодами, расположенными на плате Atlys, и представляющую собой результат подсчета нажатий на анализируемую кнопку.

Корректность работы проекта будет определяться следующим визуально анализируемым положением – одно нажатие на кнопку, ведет за собой единичное приращение счетчика.

ВЫПОЛНЕНИЕ РАБОТЫ

Шаг 1. Создание нового проекта в среде ISE.

Для начала выполнения данной работы требуется создать новый проект, выполнив при этом определенную, аналогичную представленную в лабораторной работе №1, последовательность действий.

1-1. Создание нового проекта для целевой FPGA Spartan 6, находящейся на плате Atlys.

1-1-1. Запустите ISE и создайте новый проект в котором укажите все требуемые опции. При возникновении трудностей при создании нового проекта, обратитесь к лабораторной работе №1.

1-1-2. Создайте новый модуль верхнего уровня, используя графический интерфейс среды ISE. За подробным описанием процесса создания нового модуля обратитесь к лабораторной работе №1.

Шаг 2. Создание экземпляра аппаратного блока CMT

Поскольку в данном проекте мы будем использовать элементы памяти, то требуется обеспечить их правильное тактирование посредством создания экземпляра блока CMT. Соответственно, на данном шаге требуется добавить такой модуль в разрабатываемый проект с помощью утилиты Core Generator.

2-1. Конфигурирование нового аппаратного блока CMT.

2-1-1. Сконфигурируйте блок CMT посредством утилиты Core Generator и создайте экземпляр данного модуля в описании проекта. Для получения более подробной информации обратитесь к лабораторной работе №3.

2-2. Создание экземпляра модуля CMT в описании разрабатываемого проекта.

2-2-1. Создайте экземпляр модуля CMT в описании разрабатываемого модуля. Для получения более подробной информации обратитесь к лабораторной работе №3.

2-2-2. Обратите особое внимание на то, что начиная с этого момента все элементы памяти в проекте будут тактироваться выходным сигналом из модуля CMT, а не входным тактовым сигналом.

Шаг 3. Описание файлов проекта с помощью языка Verilog HDL в среде ISE.

На данном шаге требуется описать поведение разрабатываемого модуля. При этом, для описания поведения модуля будут использоваться все конструкции языка Verilog HDL, применимые как для описания комбинационных, так и последовательных схем. Также для облегчения описания состояний конечного автомата рекомендуется использовать вспомогательные конструкции языка, такие как **localparam** и т.д. Вам потребуется ознакомиться с этими конструкциями, обратившись к главе 4. При этом, в ходе описания проекта требуется также подключить созданный на предыдущем шаге блок CMT.

3-1. Описание разрабатываемого модуля с помощью Verilog HDL.

3-1-1. Введите описание проектируемого модуля с помощью встроенного текстового редактора. Вы можете увидеть окно Project Navigator после выполнения этого шага на рисунке 6.4.4.

3-1-2. Обратите внимание на тот факт, что при описании поведения модуля рекомендуется выделить описание конечного автомата вместе со счетчиком временных интервалов в один файл. Эта часть текущего проекта будет использоваться в следующей лабораторной работе.

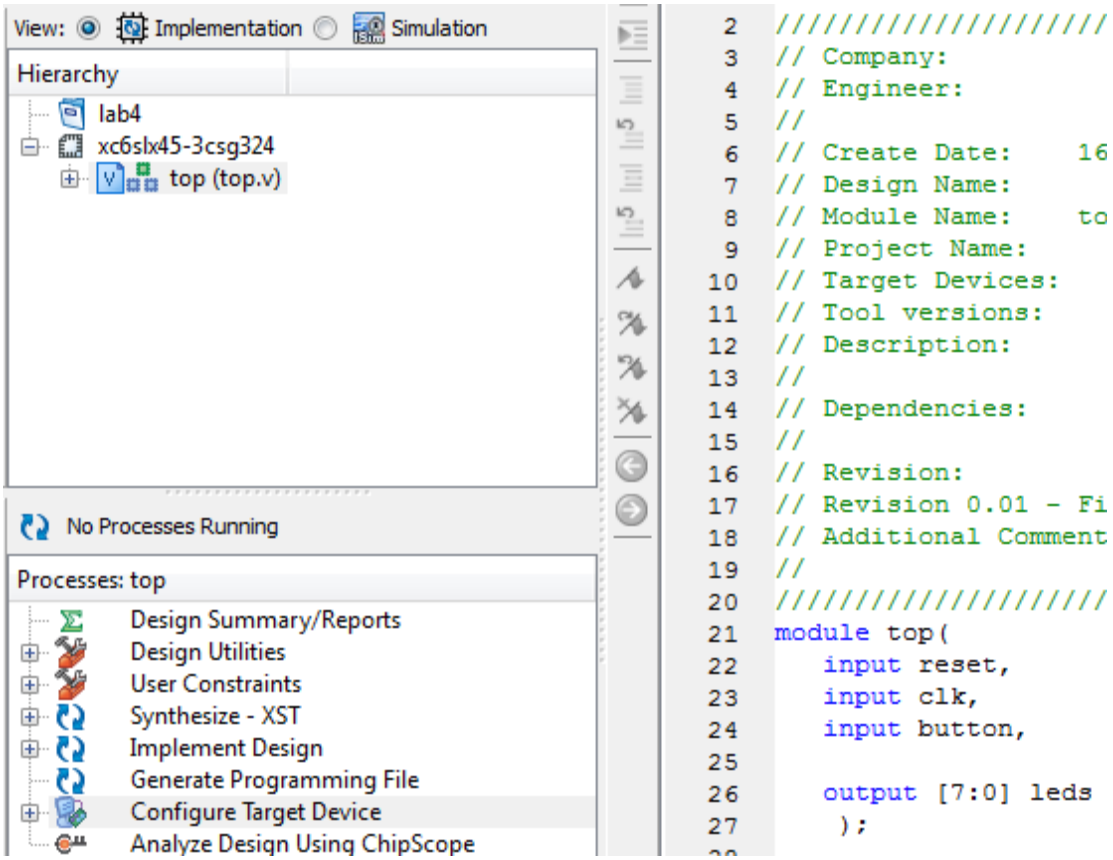


Рисунок 6.4.4 — Вид окна Project Navigator после добавления описания модуля

Шаг 4. Создание тестового окружения для проектируемого модуля

Для данного модуля требуется создать тестовое окружение и осуществить моделирование проекта, которое позволит получить наглядное представление о правильности работы как спроектированного конечного автомата, так и всего блока в целом.

4-1. Создание нового файла, содержащего описание тестового окружения с помощью среды ISE.

4-1-1. Создайте новый модуль для описания тестового окружения посредством графического интерфейса среды ISE. За подробным описанием процесса создания нового модуля для описания тестового окружения обратитесь к лабораторной работе №2.

4-1-2. Введите описание тестового окружения для проектируемого модуля с помощью встроенного текстового редактора.

Шаг 5. Моделирование проекта средствами ISIM.

После реализации тестового окружения требуется произвести моделирование

проекта и проанализировать его поведение под действием тестовых последовательностей. Весь процесс моделирования аналогичен описанному в лабораторной работе №2.

5-1. Моделирование проекта.

5-1-1. Проведите моделирование проекта средствами ISIM. За подробным описанием процесса моделирования обратитесь к лабораторной работе №2.

Шаг 6. Создание файла, содержащего ограничения проекта по размещению.

На данном этапе требуется создать файл, содержащий ограничения проекта (User Constraints File). В данном проекте нам требуется соединить входной 1-но разрядный сигнал с одной из кнопок, входной тактовый сигнал с жестко заданным входом L15 (местом подключения осциллятора к FPGA), входной сигнал сброса с еще одной кнопкой, а выходную 8-ми разрядную шину с набором светодиодов, расположенных на плате Atlys.

6-1. Создание нового файла, содержащего ограничения проекта с помощью среды ISE.

6-1-1. Создайте файл, содержащий ограничения проекта по размещению. Для этого воспользуйтесь программой PlanAhead или встроенным текстовым редактором, для ручного ввода конструкций, описывающих ограничения проекта. В случае возникновения затруднений при использовании PlanAhead, обратитесь к лабораторной работе №1.

Шаг 7. Добавление в файл, описывающий ограничения проекта, глобальных временных ограничений.

На данном шаге требуется дополнить созданный UCF файл глобальными временными ограничениями.

7-1. Добавление глобальных временных ограничений в проект.

7-1-1. Дополните файл User Constraints File, глобальными временными ограничениями. В случае возникновения затруднений при работе с глобальными временными ограничениями, обратитесь к лабораторной работе №3.

7-1-2. При работе с глобальными временными ограничениями выберите соответствующие значения: PERIOD = 10ns; OFFSET IN = 10ns; OFFSET OUT = 10ns.

Шаг 8. Проведение синтеза и имплементации всего проекта. А также создание конфигурационного файла.

На данном шаге требуется выполнить синтез и имплементацию всего проекта. Кроме этого, требуется создать конфигурационный файл.

8-1. Выполнение оставшихся шагов маршрута проектирования.

8-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Synthesize – XST. В случае возникновения ошибок требуется исправить описание проектируемого модуля.

8-1-2. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Implement Design. В случае возникновения ошибок требуется исправить файл, содержащий ограничения по размещению проекта или описание проектируемого модуля.

8-1-3. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Generate Programming File.

8-1-4. Прежде чем переходить к следующим шагам, убедитесь, что вкладка Design имеет вид, схожий с Рисунком 6.4.5. Проект также может иметь некоторое количество предупреждений, которые необходимо проанализировать и принять решение о их исправлении или игнорировании.

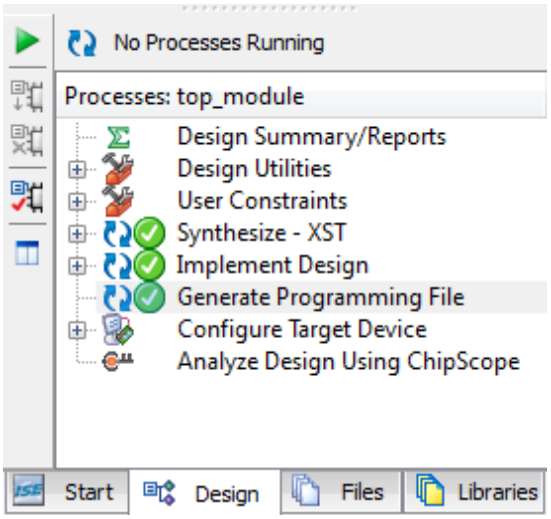


Рисунок 6.4.5 — Вид вкладки Design после окончания процессов синтеза, имплементации и создания конфигурационного файла.

Шаг 9. Проведение статического временного анализа.

На данном шаге требуется произвести статический временной анализ с целью просмотра всех критических цепей для данного проекта, т.е. цепей, обладающих наибольшей задержкой. Сам анализ осуществляется соответствующими утилитами, встроенными в среду ISE. Результатом работы подобных утилит служит некоторый отчет, содержащий подробные количественные комментарии по всем критическим цепям проекта. На основании таких отчетов можно сделать выводы о потенциальной возможности повысить частоту всего проекта, либо о необходимости произвести изменения в HDL описании некоторых частей проекта, обладающих высокой концентрацией критических цепей.

9-1. Сгенерировать и проанализировать отчет статического временного моделирования.

9-1-1. В окне среды ISE, Processes, раскройте вкладку Implement Design. Далее раскройте вкладку Map и нажмите дважды левой кнопкой мышки на Generate Post-Map Static Timing Analysis. Вид вкладок представлен на рисунке 6.4.6.

9-1-2. Перейдите в окно Design Summary и выберите во вкладке Secondary Reports, Post-Map Static Timing Analysis. Проанализируйте отчет. Вид окон среды ISE при анализе отчета статического временного моделирования представлен на рисунке 6.4.7.

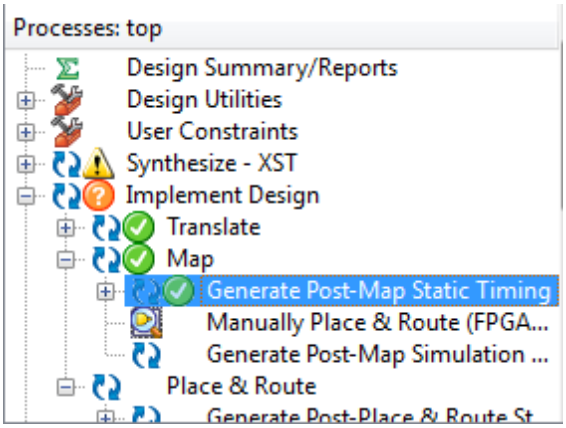


Рисунок 6.4.6 — Вид вкладок в окне Processes при осуществлении статического временного анализа

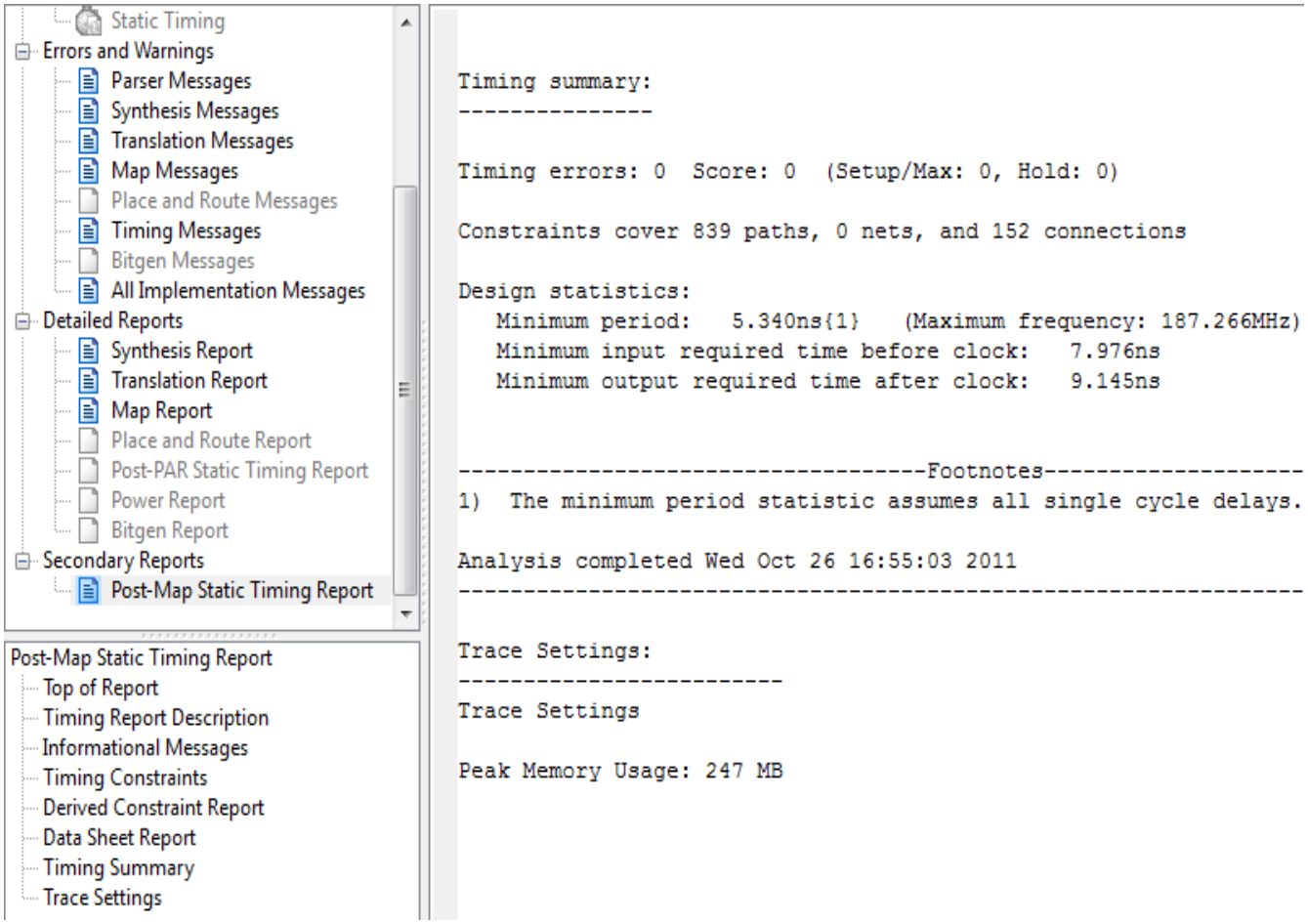


Рисунок 6.4.7 — Вид окон среды ISE при анализе отчета статического временного моделирования

9-1-3. На данном шаге производилось статическое временное моделирование после стадии отображения. Также можно произвести статическое временное моделирование после операции размещения и разводки. Стоит отметить, что для генерации отчета в этом случае потребуется несколько больше времени.

Шаг 10. Подключение платы Atlys к компьютеру.

На данном шаге требуется подключить тестовую плату Atlys к компьютеру.

10-1. Подключение платы Atlys к компьютеру.

10-1-1. Подключите плату Atlys к источнику питания.

10-1-2. Включите плату с использованием переключателя, расположенного на плате.

10-1-3. Подключите плату с помощью USB кабеля к компьютеру.

10-1-4. При возникновении затруднений, обратитесь к документу Atlys_rm.pdf,

поставляемым вместе с платой Atlys.

Шаг 11. Загрузка созданного конфигурационного файла на плату Atlys.

На данном шаге мы произведем загрузку конфигурационного файла в плату Atlys для конфигурирования, находящегося на ней, FPGA Spartan 6.

11-1. Загрузка созданного конфигурационного файла на плату Atlys.

11-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мышки на Configure Target Device. При этом откроется меню программы IMPACT. Осуществите необходимую настройку программы IMPACT. В случае возникновения затруднений, обратитесь к описанию лабораторной работы №1.

11-1-2. Сконфигурируйте FPGA Spartan 6, расположенный на плате Atlys.

Шаг 12. Наблюдение работы проекта на плате Atlys.

На этом шаге вам предлагается понаблюдать за поведением платы Atlys.

12-1. Наблюдение работы проекта на плате Atlys.

12-1-1. Для получения визуального результата требуется произвести сброс схемы, нажатием на соответствующую, выбранную в ходе проекта, кнопку. Далее требуется нажимать на выбранную и анализируемую кнопку и наблюдать соответствующее число нажатий на наборе светодиодов. Корректные результаты визуального наблюдения будут свидетельствовать об успешном выполнении проекта.

Проекты для самостоятельной работы

- Ввести в описание конечного автомата параметры для реализации более гибкого изменения временных интервалов ожидания (по умолчанию ~20ms);
- Измените глобальные временные ограничения, увеличив требуемую частоту работы модуля. Сконфигурируйте новый блок CMT, генерирующий требуемый тактовый сигнал. Определите, на какую максимальную частоту вы можете рассчитывать в вашем проекте.

ЗАКЛЮЧЕНИЕ

По окончании данной лабораторной работы вы научились проектировать цифровые устройства, содержащие в своем составе конечные автоматы. При этом вы освоили применяемые техники и шаблоны для описания подобных модулей с использованием языка Verilog HDL. Также вы познакомились со статическим временным моделированием и научились генерировать и анализировать соответствующие отчеты.

6.5 Лабораторная работа №5
Проектирование устройств, содержащих IP-блоки

Цель работы: создание проекта среднего размера, изучение принципов работы с проектом, содержащим несколько файлов, проектирование устройства, содержащего IP-блоки.

После выполнения работы вы:

- научитесь ориентироваться в более крупных проектах;
- научитесь использовать встроенные в FPGA аппаратные блоки, на примере использования модуля блочной памяти;
- научитесь конфигурировать IP-блоки посредством утилиты Core Generator;
- опробуете возможность использовать ранее разработанные модули в новых проектах.

ВВЕДЕНИЕ

Целью заключительной лабораторной работы является получение более общего представления о проектировании цифровых устройств. В данном проекте будет предложено спроектировать блок, содержащий как новые модули, описанные с помощью языка Verilog HDL, так и разработанные в предыдущих лабораторных работах. Также проект будет содержать IP-блок, базирующийся на встроенных в микросхему FPGA аппаратных модулях блочной памяти.

Как и в предыдущих лабораторных работах, в ходе выполнения работы, будут получены как результаты моделирования, так и полностью функционирующее устройство, реализующее поставленную задачу.

По окончанию данной серии лабораторных работ вы будете обладать всеми необходимыми навыками для создания собственных цифровых устройств и иметь представление о всех ключевых составляющих цифровой техники.

ОПИСАНИЕ ЗАДАЧИ

Задачей данной лабораторной работы будет являться создание цифрового устройства среднего размера, ядром которого является блок FIFO, представляющий собой буфер, выполняющий 2 операции, а именно: запись данных в буфер и чтение данных из буфера. При этом все записанные элементы всегда считываются в том порядке, в каком они были записаны в буфер. Упрощенное представление буфера FIFO представлено на рисунке 6.5.1.

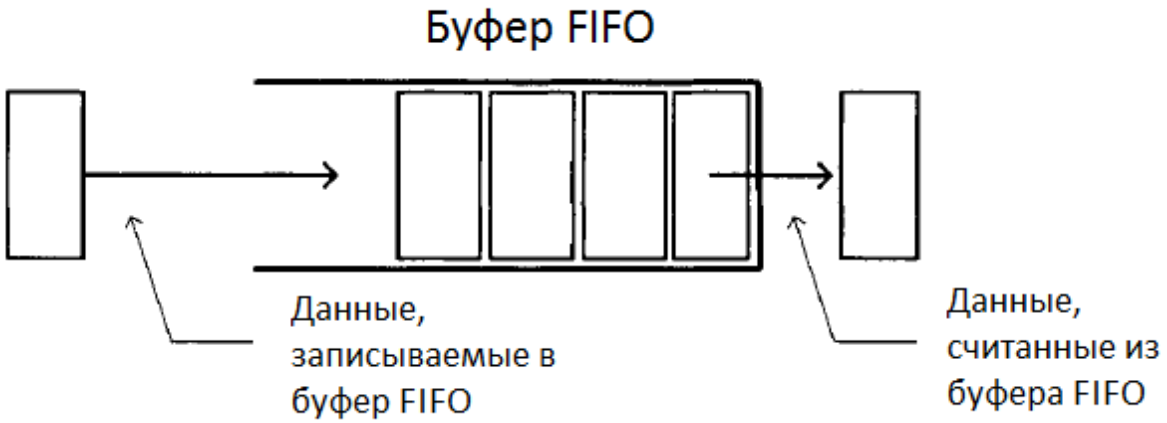


Рисунок 6.5.1 – Упрощенное представление буфера FIFO

Разрабатываемое устройство будет принимать на вход некоторое значение и записывать его по входному разрешающему сигналу в буфер FIFO. При возникновении входного разрешающего сигнала чтения, записанное значение будет выводиться на выходную шину. Все входные сигналы разрешения чтения/записи должны быть подключены к кнопкам, расположенным на плате Atlys и представлять собой единичные импульсы. В связи с этим, для каждой из таких разрешающих сигналов потребуется модуль, убирающий дребезг сигналов, разработанный в предыдущей лабораторной работе. В общем виде, блок-схема проекта данной лабораторной работы представлена на рисунке 6.5.2.

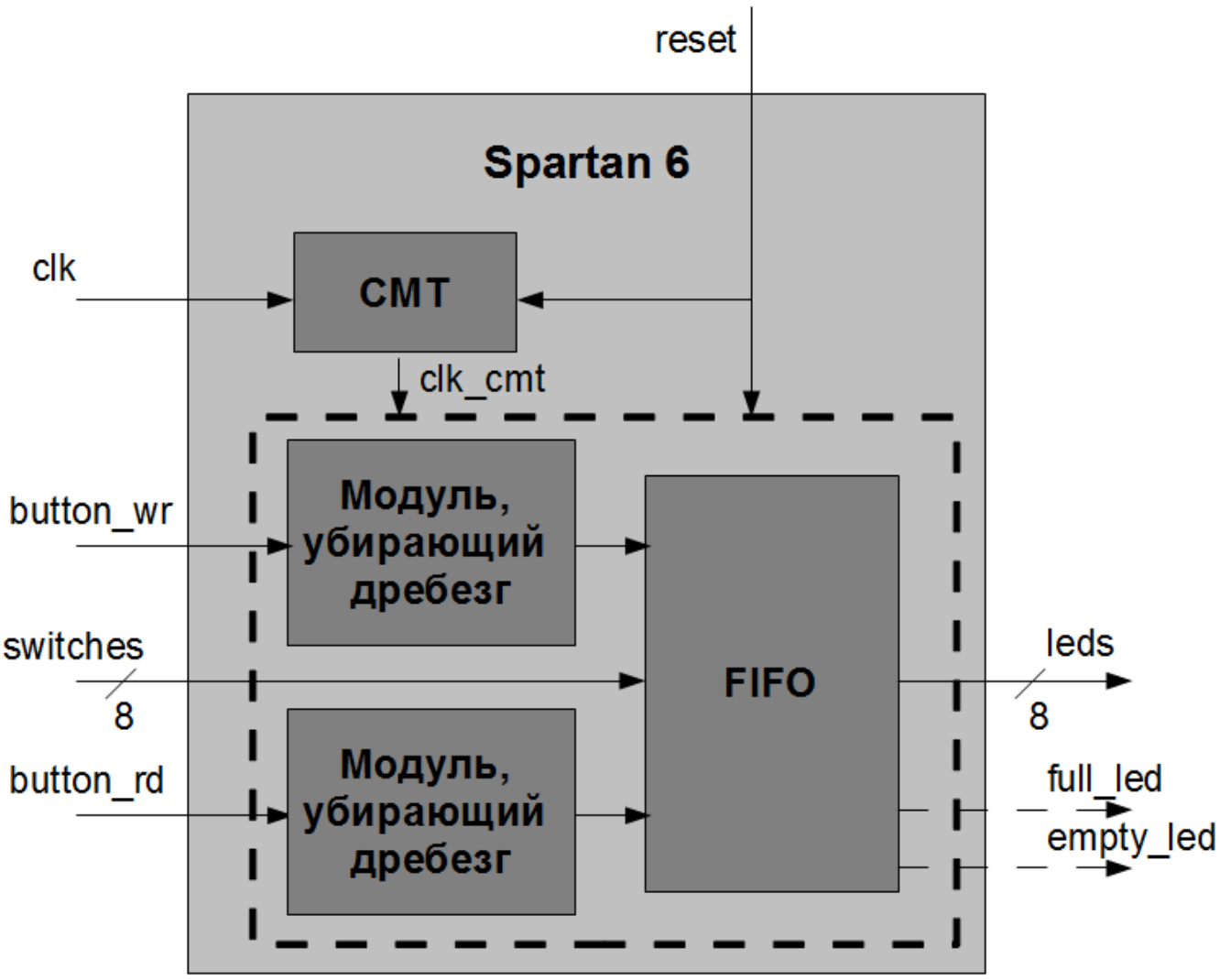


Рисунок 6.5.2 — Структурная схема проекта лабораторной работы №5

Как видно из рисунка 6.5.2 разрабатываемый модуль содержит входную 8-ми разрядную шину (switches), представляющую значение для записи в буфер FIFO, и соединенную с 8-мью переключателями, выходную 8-ми разрядную шину (leds), представляющую собой результат операции чтения, соединенную со светодиодами, а также 2 входных разрешающих сигнала чтения/записи (button_rd/button_wr), соединенных с кнопкам. Все переключатели, светодиоды и кнопки располагаются непосредственно на плате Atlys. Также модуль содержит 2 служебных входных сигнала, представляющие собой тактовый сигнал и сигнал асинхронного или синхронного сброса. Кроме этого опциональными сигналами могут являться сигналы полноты (full_led) и пустоты (empty_led) буфера FIFO.

Для определения корректности работы требуется убедиться, что данные, находящиеся внутри разрабатываемого модуля отображаются в порядке записи.

ВЫПОЛНЕНИЕ РАБОТЫ

Шаг 1. Создание нового проекта в среде ISE.

Для начала выполнения данной работы требуется создать новый проект, выполнив при этом определенную, аналогичную представленную в лабораторной работе №1, последовательность действий.

1-1. Создание нового проекта для целевой FPGA Spartan 6, находящейся на плате Atlys.

1-1-1. Запустите ISE и создайте новый проект в котором укажите все требуемые опции. При возникновении трудностей при создании нового проекта, обратитесь к лабораторной работе №1.

1-1-2. Создайте новый модуль верхнего уровня, используя графический интерфейс среды ISE. За подробным описанием процесса создания нового модуля обратитесь к лабораторной работе №1.

Шаг 2. Создание экземпляра аппаратного блока CMT

Поскольку в данном проекте мы будем использовать элементы памяти, то требуется обеспечить их правильное тактирование посредством создания экземпляра блока CMT. Соответственно, на данном шаге требуется добавить такой модуль в разрабатываемый проект с помощью утилиты Core Generator.

2-1. Конфигурирование нового аппаратного блока CMT.

2-1-1. Сконфигурируйте блок CMT посредством утилиты Core Generator и создайте экземпляр данного модуля в описании проекта. Для получения более подробной информации обратитесь к лабораторной работе №3.

2-2. Создание экземпляра модуля CMT в описании разрабатываемого проекта.

2-2-1. Создайте экземпляр модуля CMT в описании разрабатываемого модуля. Для получения более подробной информации обратитесь к лабораторной работе №3.

2-2-2. Обратите особое внимание на то, что начиная с этого момента все элементы памяти в проекте будут тактироваться выходным сигналом из модуля CMT, а не входным тактовым сигналом.

Шаг 3. Создание экземпляра IP-блока FIFO

Не смотря на то, что возможно описать модуль FIFO в виде обычного модуля, с помощью языка Verilog HDL, здесь мы воспользуемся утилитой Core Generator. С помощью этой утилиты можно задействовать встроенные в FPGA аппаратные блоки, в данном случае, мы задействуем модуль блочной памяти.

3-1. Конфигурирование нового IP-блока FIFO.

3-1-1. В окне Hierarchy щелкните правой кнопкой мышки и выберите Create New Source.

3-1-2. В окне New Source выберите IP (CORE Generator & Architecture Wizard). Введите название модуля и нажмите Next.

3-1-3. В окне Select IP раскройте вкладку Memories & Storage Elements, затем раскройте вкладку FIFOs и выберите Fifo Generator. Далее нажмите Next и Finish. Данная операция представлена на рисунке 6.5.3.

3-1-4. В появившемся диалоге, на первой страницу (Page 1) выберите тип интерфейса Native и нажмите Next.

3-1-5. На второй странице (Page 2) выберите Common Clock Block RAM и нажмите Next.

3-1-6. На третьей странице (Page 3) выберите ширину шины данных (Write Width), равную 8-ми и глубину буфера (Write Depth), равную 16-ти и нажмите Next.

3-1-7. Далее нажмите 4 раза Next, а затем Generate.

3-1-8. Дождитесь успешной генерации блока. Блок должен появиться во вкладке Hierarchy в среде ISE.

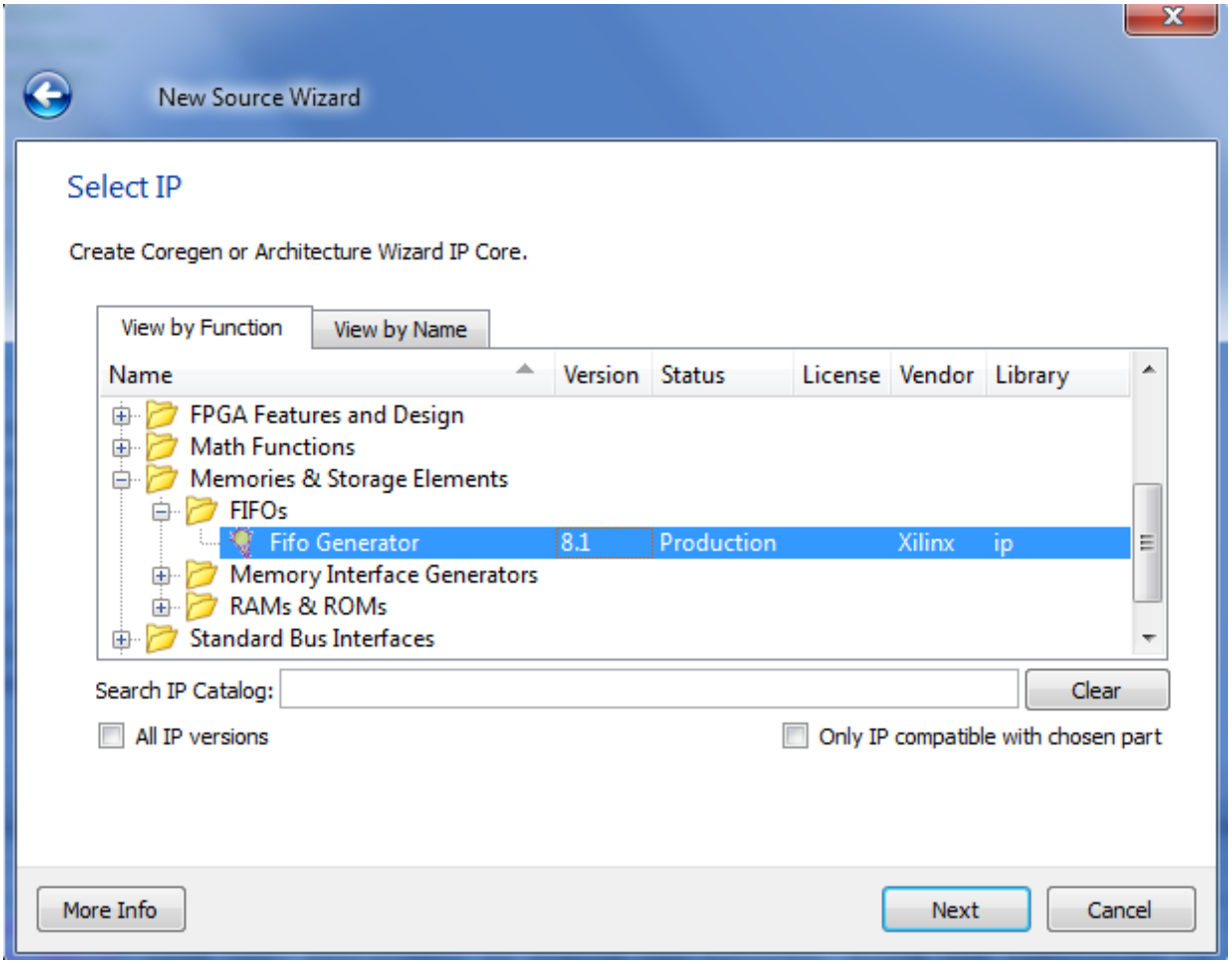


Рисунок 6.5.3 — Вид вкладок в окне New Source Wizard для выбора модуля FIFO

3-2. Создание экземпляра модуля FIFO в описании разрабатываемого проекта.

3-2-1. В окне Hierarchy щелкните левой кнопкой мышки на созданный модуль FIFO. Далее во вкладке Processes нажмите дважды левой кнопкой мышки на View HDL Instantiation Template. Скопируйте открывшееся описание в разрабатываемый проект.

3-2-2. В описании разрабатываемого проекта подключите все требуемые сигналы ввода-вывода к экземпляру модуля FIFO. Также укажите имя созданного экземпляра модуля FIFO.

Шаг 4. Копирование описания модуля, убирающего дребезг входного сигнала, в разрабатываемый проект.

На данном шаге требуется скопировать описание модуля, убирающего дребезг входного сигнала и вырабатывающего единичный импульс при смене логического

уровня, разработанного в предыдущей лабораторной работе.

4-1. Копирование разработанного в предыдущей лабораторной работе модуля в текущий проект.

4-1-1. Выберете Project → Add Copy of Source. В открывшемся диалоговом окне выберите путь к копируемому файлу.

Шаг 5. Описание файлов проекта с помощью языка Verilog HDL в среде ISE.

На данном шаге требуется описать модуль верхнего уровня. При этом, описание модуля верхнего уровня будет в основном состоять из связи уже созданных, либо сконфигурированных модулей, находящихся в проекте. Как и в предыдущей работе, для описания поведения модуля будут использоваться все конструкции языка Verilog HDL, применимые как для описания комбинационных, так и последовательных схем.

5-1. Описание разрабатываемого модуля с помощью Verilog HDL.

5-1-1. Введите описание проектируемого модуля с помощью встроенного текстового редактора. Вы можете увидеть окно Project Navigator после выполнения этого шага на рисунке 6.5.4.

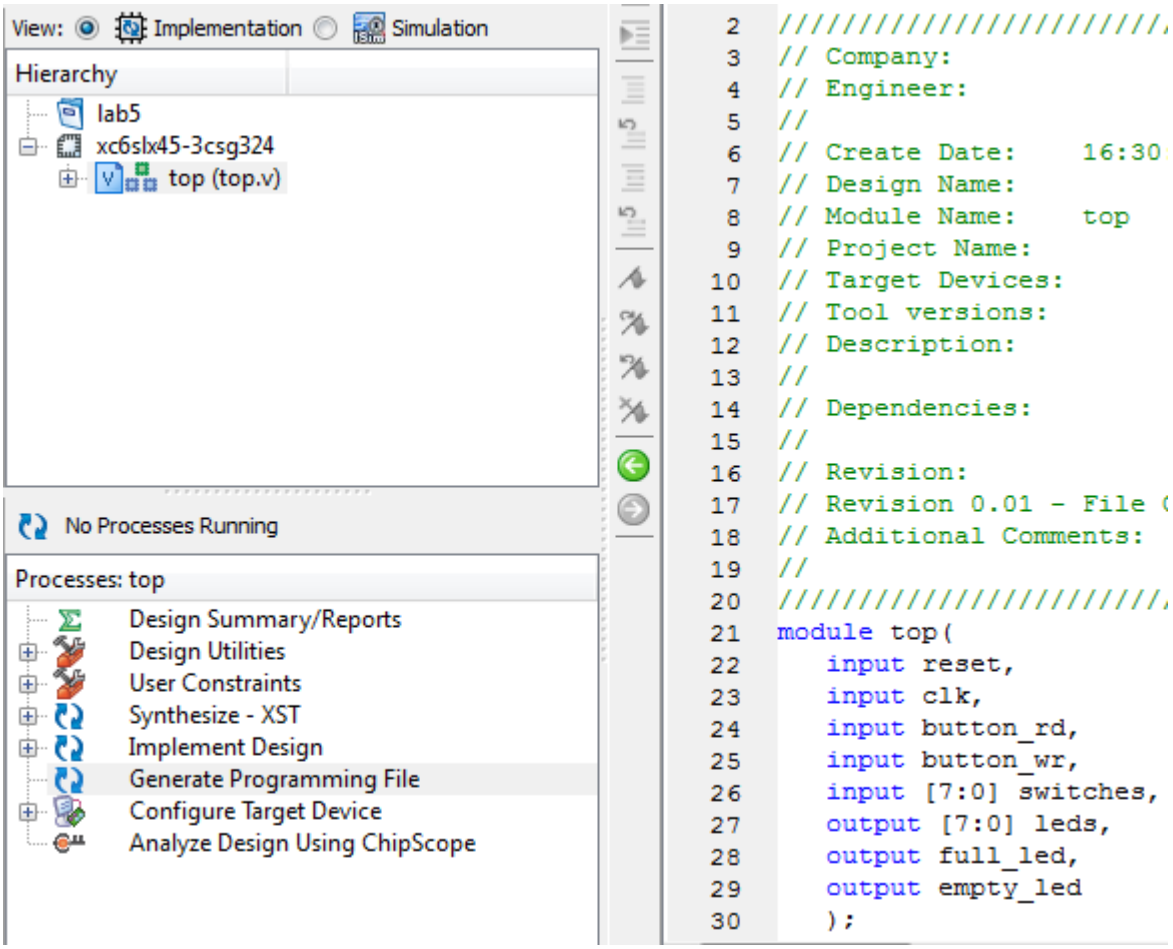


Рисунок 6.5.4 — Вид окна Project Navigator после добавления описания модуля

Шаг 6. Создание тестового окружения для проектируемого модуля

Для данного модуля требуется создать тестовое окружение и осуществить моделирование проекта, которое позволит получить наглядное представление о правильности работы всего блока в целом и позволит исключить ошибки в поведении блока.

6-1. Создание нового файла, содержащего описание тестового окружения с помощью среды ISE.

6-1-1. Создайте новый модуль для описания тестового окружения посредством графического интерфейса среды ISE. За подробным описанием процесса создания нового модуля для описания тестового окружения обратитесь к лабораторной работе №2.

6-1-2. Введите описание тестового окружения для проектируемого модуля с помощью встроенного текстового редактора.

Шаг 7. Моделирование проекта средствами ISIM.

После реализации тестового окружения требуется произвести моделирование проекта и проанализировать его поведение под действием тестовых последовательностей. Весь процесс моделирования аналогичен описанному в лабораторной работе №2.

7-1. Моделирование проекта.

7-1-1. Проведите моделирование проекта средствами ISIM. За подробным описанием процесса моделирования обратитесь к лабораторной работе №2.

Шаг 8. Создание файла, содержащего ограничения проекта по размещению.

На данном этапе требуется создать файл, содержащий ограничения проекта (User Constraints File). В данном проекте нам требуется соединить входную 8-ми разрядную шину с набором переключателей, выходную 8-ми разрядную шину с набором светодиодов, входные разрешающие сигналы с кнопками, входной тактовый сигнал с жестко заданным входом L15 (местом подключения осциллятора к FPGA), а входной сигнал сброса с еще одной кнопкой расположенной на плате Atlys.

8-1. Создание нового файла, содержащего ограничения проекта с помощью среды ISE.

8-1-1. Создайте файл, содержащий ограничения проекта по размещению. Для этого воспользуйтесь программой PlanAhead или встроенным текстовым редактором, для ручного ввода конструкций, описывающих ограничения проекта. В случае возникновения затруднений при использовании PlanAhead, обратитесь к лабораторной работе №1.

Шаг 9. Добавление в файл, описывающий ограничения проекта, глобальных временных ограничений.

На данном шаге требуется дополнить созданный UCF файл глобальными временными ограничениями.

9-1. Добавление глобальных временных ограничений в проект.

9-1-1. Дополните файл User Constraints File, глобальными временными ограничениями. В случае возникновения затруднений при работе с глобальными временными ограничениями, обратитесь к лабораторной работе №3.

9-1-2. При работе с глобальными временными ограничениями выберите соответствующие значения: PERIOD = 10ns; OFFSET IN = 10ns; OFFSET OUT = 20ns.

Шаг 10. Проведение синтеза и имплементации всего проекта. А также создание конфигурационного файла.

На данном шаге требуется выполнить синтез и имплементацию всего проекта. Кроме этого, требуется создать конфигурационный файл.

10-1. Выполнение оставшихся шагов маршрута проектирования.

10-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Synthesize – XST. В случае возникновения ошибок требуется исправить описание проектируемого модуля.

10-1-2. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Implement Design. В случае возникновения ошибок требуется исправить файл, содержащий ограничения по размещению проекта или описание проектируемого модуля.

10-1-3. В среде ISE во вкладке Design дважды нажмите левой кнопкой мыши на Generate Programming File.

10-1-4. Прежде чем переходить к следующим шагам, убедитесь, что вкладка Design имеет вид, схожий с Рисунком 6.5.5. Проект также может иметь некоторое количество предупреждений, которые необходимо проанализировать и принять решение о их исправлении или игнорировании.

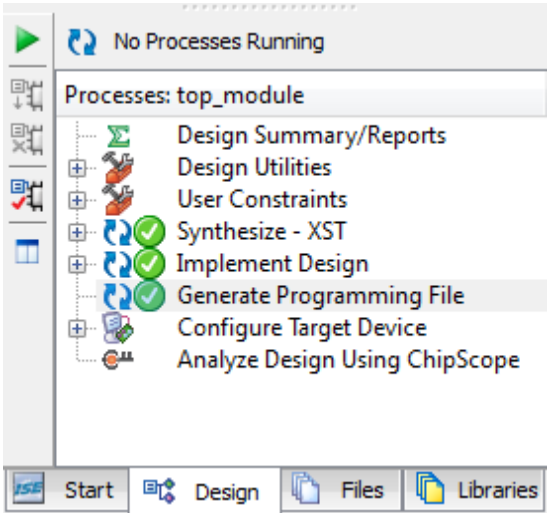


Рисунок 6.5.5 — Вид вкладки Design после окончания процессов синтеза, имплементации и создания конфигурационного файла.

Шаг 11. Подключение платы Atlys к компьютеру.

На данном шаге требуется подключить тестовую плату Atlys к компьютеру.

11-1. Подключение платы Atlys к компьютеру.

11-1-1. Подключите плату Atlys к источнику питания.

11-1-2. Включите плату с использованием переключателя, расположенного на плате.

11-1-3. Подключите плату с помощью USB кабеля к компьютеру.

11-1-4. При возникновении затруднений, обратитесь к документу Atlys_rm.pdf, поставляемым вместе с платой Atlys.

Шаг 12. Загрузка созданного конфигурационного файла на плату Atlys.

На данном шаге мы произведем загрузку конфигурационного файла в плату Atlys для конфигурирования, находящегося на ней, FPGA Spartan 6.

12-1. Загрузка созданного конфигурационного файла на плату Atlys.

12-1-1. В среде ISE во вкладке Design дважды нажмите левой кнопкой мышки на Configure Target Device. При этом откроется меню программы IMPACT. Осуществите необходимую настройку программы IMPACT. В случае возникновения затруднений, обратитесь к описанию лабораторной работы №1.

12-1-2. Сконфигурируйте FPGA Spartan 6, расположенный на плате Atlys.

Шаг 13. Наблюдение работы проекта на плате Atlys.

На этом шаге вам предлагается понаблюдать за поведением платы Atlys.

13-1. Наблюдение работы проекта на плате Atlys.

13-1-1. Для получения визуального результата требуется произвести сброс схемы, нажатием на соответствующую, выбранную в ходе проекта, кнопку. Далее требуется выбирать с помощью переключателей данные для записи в буфер FIFO и записывать их по нажатию на соответствующую кнопку, представляющую сигнал разрешения записи. После этого необходимо считывать записанные значения, нажатием на соответствующую кнопку, представляющую собой сигнал разрешения чтения, и анализировать их на соответствующем наборе светодиодов. Корректные результаты визуального наблюдения будут свидетельствовать об успешном выполнении проекта.

Проекты для самостоятельной работы

- Опишите самостоятельно модуль FIFO, используя конструкции языка Verilog HDL;
- Измените глобальные временные ограничения, уменьшив значение ограничения OFFSET OUT до 10ns. Произведите повторный синтез и имплементацию. Проанализируйте полученные результаты.

ЗАКЛЮЧЕНИЕ

По окончании данной лабораторной работы вы научились проектировать цифровые устройства среднего размера, содержащие в своем составе экземпляры IP-блоков. При этом вы освоили работу над проектом, содержащим несколько описаний модулей различных типов, как созданных в виде описаний на языке Verilog HDL, так и сгенерированных утилитой Core Generator.

7. Приложение

7.1 Краткое описание лабораторного макета Atlys

Лабораторный макет Atlys представляет собой законченную и готовую к использованию платформу, основанную на микросхеме FPGA фирмы Xilinx семейства Spartan 6 LX45. Микросхема FPGA, вместе с большим набором периферийных модулей, таких как Gb Ethernet, HDMI Video, 128MByte 16-bit DDR2, USB и т. д., делают данную платформу идеальной для реализации широкого спектра цифровых систем. Данная платформа также полностью подходит для использования при реализации встроенных систем, включающих высокопроизводительный процессор общего назначения класса “soft-core”, MicroBlaze. Также стоит отметить, что данная платформа полностью поддерживается соответствующими САПР фирмы Xilinx, включая ChipScope, EDK, ISE и т. д.

Полный список ресурсов, предоставляемых данной платформой приведен ниже:

• Spartan-6 LX45 FPGA, 324-pin BGA;

• 128Mbyte DDR2;

• 10/100/1000 Ethernet PHY;

• порты USB2 для программирования и передачи данных;

• порты USB-UART и USB-HID (для подключения клавиатуры или мышки);

• 2 входных порта и 2 выходных порта HDMI;

• AC-97 аудио кодек;

• 16Mbyte x4 SPI Flash для хранения конфигурации и пользовательских данных;

• 100MHz CMOS осциллятор;

• 48 I/O разведенных для использования платы расширения;

• GPIO включают в себя 8 светодиодов, 6 кнопок и 8 переключателей.

На рисунке 7.1.1 представлена блок-схема лабораторного макета Atlys.

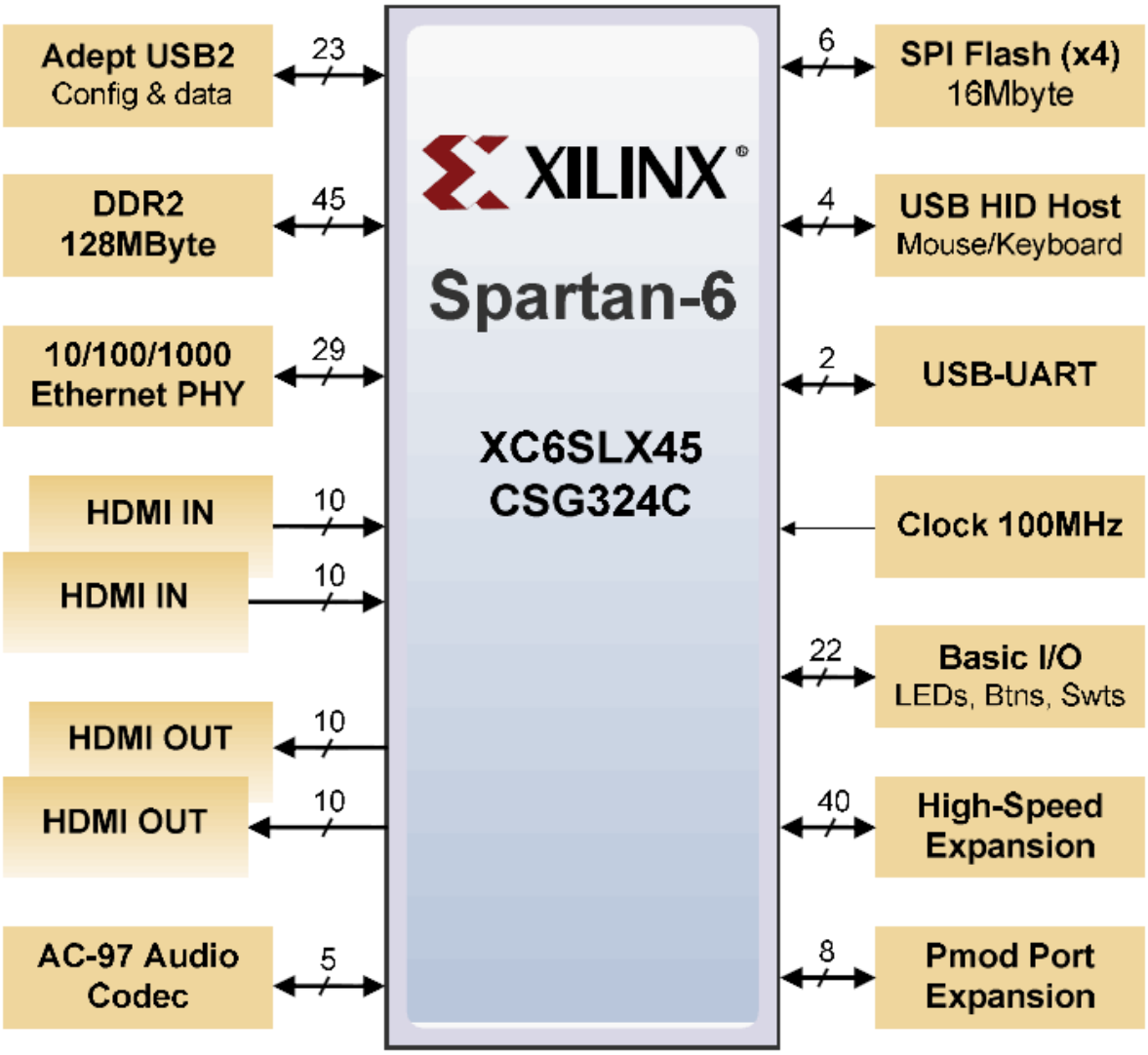


Рисунок 7.1.1 — Блок-схема лабораторного макета Atlys

На рисунке 7.1.2 приведен внешний вид лабораторного макета Atlys.

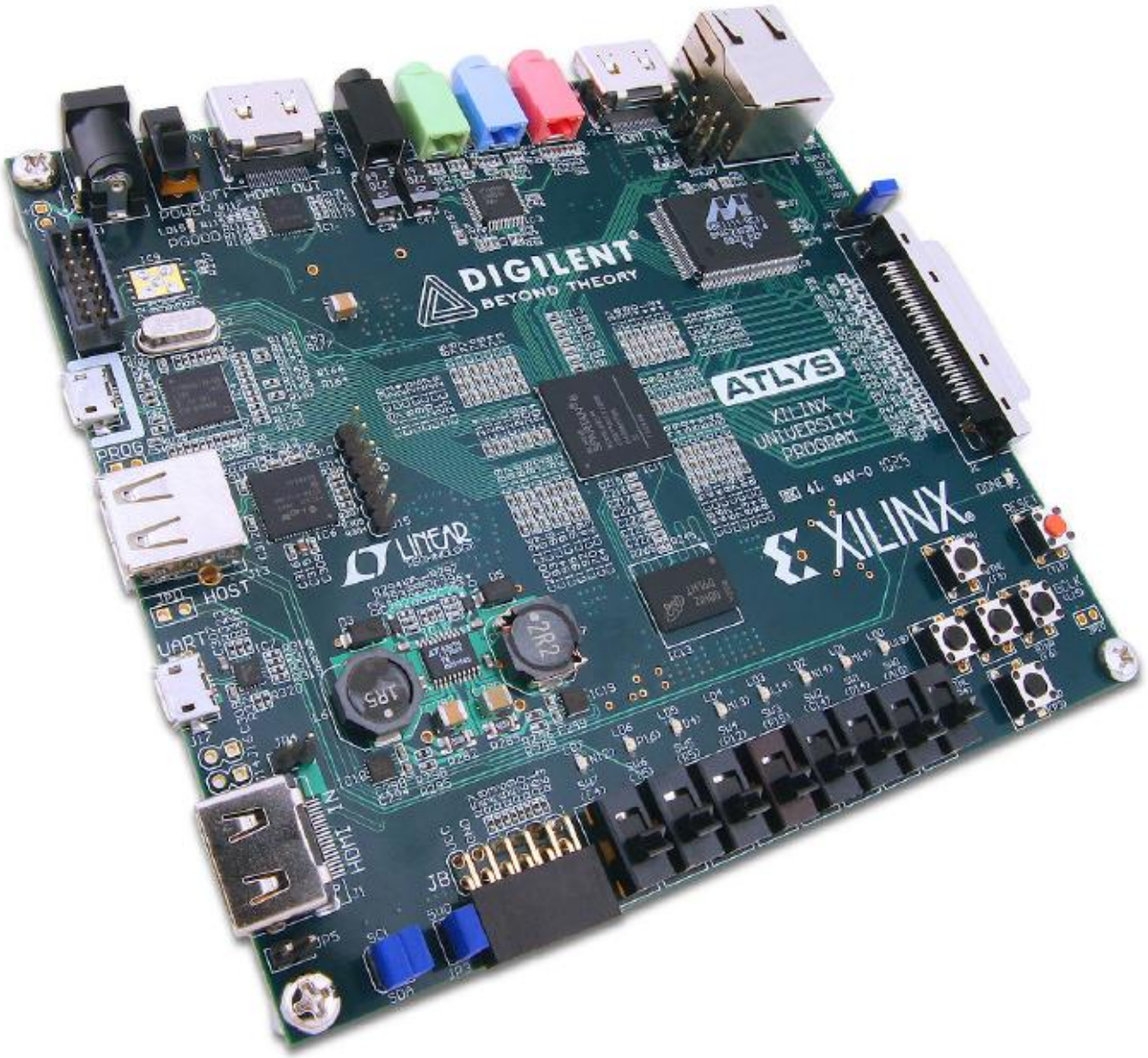


Рисунок 7.1.2 — Внешний вид лабораторного макета Atlys

За более подробной информацией по лабораторному макету Atlys требуется обратиться к документации. Ссылка на документацию приведена в списке рекомендуемой литературы.

7.2 Правила оформления отчетов по лабораторным работам

Перед выполнением каждой лабораторной работы слушатель должен предоставить отчет, который не должен превышать одного листа А4 и содержать следующие поля:

- Название лабораторной работы;
- ФИО слушателя;
- Дата проведения лабораторной работы;
- Краткое описание лабораторной работы;
- Блок-схема проекта, разрабатываемого в данной лабораторной работе;
- Три графы для подписи: присутствие на занятии, выполнение лабораторной работы, защита лабораторной работы.

7.3 Правила оформления файлов проектов по лабораторным работам

Для слушателей, выполняющих курс лабораторных работ, существуют требования к оформлению файлов проекта:

- Каждый модуль, написанный на языке Verilog HDL должен располагаться в отдельном файле;
- Каждый файл с исходным кодом должен содержать пролог, который включает в себя: дату создания файла, ФИО разработчиков, а также название лабораторной работы. Шаблон пролога представлен на рисунке 7.3.1;
- Весь исходный код модулей и тестовых окружений на языке Verilog HDL должен быть отформатирован (добавлены отступы и т. д.);
- Исходный код модулей должен содержать комментарии, отражающие мысли разработчика.

```
////////////////////////////////////////////////////////////////////////
//
//  Author(s): Ivanov Ivan, Petrov Petr
//
//  Project: Labx
//  Created on: xx.xx.xxxx
//  Description:
//
////////////////////////////////////////////////////////////////////////
```

Рисунок 7.3.1 — Шаблон пролога для каждого файла проекта

8. Рекомендуемая литература

1. Максфилд К. Проектирование на ПЛИС. Архитектура, средства и методы // М.: «Додека», 2007. - 407с.
2. Pong P. Chu. FPGA prototyping by Verilog Examples // New Jersey: «Willey», 2008. - 508с.
3. Verilog // «Википедия» — универсальная энциклопедия

   URL: http://ru.wikipedia.org/wiki/Verilog.
4. IEEE 1364 Standard for Verilog Hardware Description Language.
5. Пакет проектирования компании Xilinx ISE //

   URL: http://www.xilinx.com/tools/webpack.htm.
6. Ресурс по языку SystemVerilog и современным методологиям верификации // URL: http://www.testbench.in.
7. Ресурс по современным языкам, участвующим в реализации проектирования интегральных микросхем и современным методологиям верификации // URL: http://www.doulous.com.
8. Ресурс по современным средствам и методам проектирования цифровой техники //

   URL: http://www.demosondemand.com.
9. Ресурс по современным методологиям верификации //

   URL: http://www.verification-academy.mentor.com.
10. Описание лабораторного макета Atlys //

    URL: http://www.digilentinc.com.

11. D. Harris & S. Harris Digital Design and Computer Architecture // Canada: «Morgan Kaufmann», 2009. - 459c.
12. Hennesy & Patterson Computer Architecture: a Quantitative Approach // Canada: «Morgan Kaufmann», 2006. - 704c.