ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Физический факультет
Кафедра автоматизации физико-технических исследований

К. Ф. Лысаков

ПРОГРАММИРУЕМАЯ ЛОГИКА FPGA

Учебное пособие

Новосибирск
2009

УДК 519.682.5
ББК В185.29(HDL)я73-1
Л886

Лысаков К. Ф. Программируемая логика FPGA: Учеб. пособие / Новосиб. гос. ун-т. Новосибирск, 2009. 98 с.

Сегодня в качестве процессорных устройств для обработки данных чаще всего применяются либо универсальные, либо сигнальные процессоры, так как они обладают возможностью перепрограммирования (в отличие от заказных микросхем), что позволяет применять одно аппаратное решение для различных задач. FPGA, так же как и заказные интегральные схемы, позволяют создавать конвейер операционных блоков для реализации конкретного алгоритма. В то же время FPGA, как и сигнальные процессоры, обладает программируемостью и позволяют создавать высокопроизводительные системы, критичные к габаритным размерам и энергопотреблению. Сегодня существует тенденция к переводу трудоемких вычислений с кластерных архитектур на FPGA, однако существует нехватка специалистов в области применения и использования FPGA.

Магистранты кафедры АФТИ физического факультета НГУ, а также студенты, желающие применять FPGA для задач автоматизации или построения высокопроизводительных систем обработки данных.

Предназначено для студентов и магистрантов высших учебных заведений, а также всех желающих использовать FPGA для задач автоматизации или построения высокопроизводительных систем обработки данных.

Рецензент
доц., зам. зав. кафедрой АФТИ ФФ НГУ М. Ю. Шадрин

Учебное пособие подготовлено в рамках реализации программы развития НИУ-НГУ на 2009-2018 гг.

ОГЛАВЛЕНИЕ

1. Программируемая логика

1.1. Программируемые матрицы

В середине 80-х г. XX в. были созданы первые программируемые матрицы, которые позволяли реализовывать лишь простые логические функции («И» и «ИЛИ») и их комбинации. Но это направление электроники стало развиваться очень стремительно и сегодня современные программируемые матрицы не только достигли уровня современных процессоров, но и превзошли их в некоторых задачах.

В настоящее время существуют два основных типа программируемых матриц:

- программируемые логические матрицы (ПЛМ, PLD – Programmable Array Devices);
- вентильные матрицы, программируемые пользователем (ВМПП, FPGA – Field-programmable gate array).

В современной литературе часто употребляется название ПЛИС – программируемые логические интегральные схемы, которые объединяют оба семейства, перечисленные выше.

Программируемые логические интегральные схемы, также как и заказные интегральные схемы, позволяют создавать конвейеры операционных блоков произвольной ширины для реализации конкретного алгоритма и распараллеливание обработки данных. В то же время ПЛИС, как и сигнальные процессоры, обладают возможностью изменять в любое время заложенный алгоритм.

При этом архитектура ПЛИС такова, что позволяет создавать высокопроизводительные системы, критичные к габаритным размерам и энергопотреблению. Это достигается за счет архитектуры кристаллов, позволяющей реализовать до нескольких независимых устройств обработки данных внутри одного кристалла. При этом существует возможность реализовывать различные интерфейсные стандарты (TTL, PCI, DDR, Ethernet) для каждого из этих устройств обработки.

1.1.1. PLD

Кристаллы этого семейства представляют собой матрицу логических элементов с триггерами, в которых перемычками программируются условия возникновения на выходе логической единицы в зависимости от поданных на вход набора сигналов.

Вначале перемычки выполнялись в виде пережигаемых тонких проводников. Сейчас перемычки выполняются в виде МОП-транзисторов с плавающим затвором, т. е. как в электрически перепрограммируемом ПЗУ, ПЛМ изготовляются по технологии флэш-памяти. Большие ПЛМ (Complex PLD – CLPD) отличаются только тем, что несколько ПЛМ собраны на одном кристалле и объединены программируемым полем связей.

1.1.2. FPGA

Это семейство появилось как дальнейшее развитие ПЛМ для реализации более сложных алгоритмов. От ПЛМ ВМПП отличаются как по архитектуре, так и по объему и представляют собой матрицу логических ячеек, памяти и линий связи между ними. При этом связи внутри кристалла имеют распределенную трехуровневую архитектуру, что дало возможность создания ВМПП объемом 10 и выше миллионов вентилей.

Кристаллы ВМПП состоят из матрицы конфигурируемых логических блоков (КЛБ – CLB), которая окружена программируемыми блоками ввода-вывода. Все соединения между основными элементами осуществляются с помощью набора иерархических высокоскоростных программируемых трассировочных ресурсов. Соединение между КЛБ осуществляется с помощью главных трассировочных матриц – ГТМ.

ГТМ представляет собой матрицу программируемых транзисторных двунаправленных переключателей, расположенных на пересечении горизонтальных и вертикальных линий связи. Каждый КЛБ окружен локальными линиями связи (VersaBlockTM), которые позволяют осуществлять соединения с матрицей ГТМ [8, 9].

Роль основного логического элемента в современных ВМПП играет блок, называемый Slice. Он состоит из четырех соединенных между собой логических таблиц (LUT – look-up table), представляющих однобитное ОЗУ с четырьмя входами и одним выходом. Каждый LUT может быть сконфигурирован для реализации элементарной логической функции «И» или «ИЛИ», либо может быть просто ячейкой памяти на один бит.

Таким образом, один Slice может быть сконфигурирован как ячейка памяти на 4 бита либо как четырехбитный сумматор.

1.1.3. Рынок ПЛИС

В настоящее время можно выделить двух производителей кристаллов ПЛИС, занимающих вместе около 80 % всего рынка: Xilinx и Altera.

Оба производителя выпускают примерно одинаковую продукцию по характеристикам и быстродействию, при этом используя сопоставимые маршруты проектирования и похожие средства разработки. Поэтому теоретическую информацию и примеры на языке описания аппаратуры можно с равным успехом применять к изделиям обоих производителей.

1.2. Языки описания аппаратуры

При описании и реализации алгоритмов обработки данных для программируемой логики существует два основных подхода: схемотехнический и «программистский».

Схемотехнический подход в описании алгоритмов заключается в рисовании схем, состоящих из базовых логических элементов кристаллов ПЛИС и связей между ними. Результатом схемотехнического проектирования является схема, реализующая заданный алгоритм. Этот подход появился первым при описании алгоритмов и до сих пор имеет множество последователей.

«Программистский» подход сформировался в результате развития и появления новых языков программирования. Среди множества языков было разработано семейство языков описания аппаратуры HDL (Hardware Description Languages). Эти языки являются языками низкого уровня, оперирующими также с примитивами ПЛИС. Результатом такого проектирования является набор текстов на определенном языке.

Существующие сегодня системы разработки программного обеспечения для ПЛИС позволяют объединять в одном проекте оба подхода. Это позволяет, разбив проект на модули, реализовывать их наиболее удобным для проектировщика способом.

Язык описания аппаратуры VHDL

В конце 70-х – начале 80-х гг. Министерство Обороны США финансировало программы VHSIC (Very High Speed Integrated

Circuits – сверхбыстродействующие интегральные схемы, СБИС), основной целью которых была разработка нового поколения интегральных схем (ИС).

В процессе проектировки таких ИС встала проблема описания схем, состоящих из сотен тысяч логических элементов. В результате в 1981 г. был предложен язык VHDL (VHSIC Hardware design and Description Language – язык для описания СБИС на аппаратном уровне). Его разработчики преследовали две цели. Во-первых, создать язык, пригодный для описания крайне сложных ИС. Во-вторых, сделать возможным обмен информацией между участниками программы в общей для всех, стандартной форме. В 1986 г. VHDL был предложен как стандарт IEEE. Пройдя ряд проверок и исправлений, в декабре 1987 г., он был принят как стандарт IEEE-1076.

В схеме разработки ИС традиционно выделяют несколько уровней описания:

- алгоритмический уровень, описывающий алгоритмы с использованием формальной нотации, на языках высокого уровня;
- архитектурный уровень, дающий точное описание системы и ее структуры;
- уровень регистровой логики, дающий более подробное описание поведения в терминах регистровых передач и выполнения операций;
- уровень электрических схем, описывающий детальную электрическую структуру системы в терминах транзисторов, емкостей и резисторов;
- уровень геометрического размещения элементов, описывающий маску для фотолитографии схемы.

Язык VHDL дает возможность описывать дизайн схемы на различных уровнях представления: начиная от самого верхнего – алгоритмического, вплоть до уровня электрических схем. Также предоставляется возможность использовать вложенные друг в друга элементы дизайна, возможность объединения их в общую иерархическую структуру. Для каждого элемента схемы в языке VHDL предусмотрена возможность как поведенческого (алгоритмический уровень представления), так и структурного описания (архитектурный уровень, уровень регистровой логики, иногда уровень электрических схем). При структурном описании элемента схемы требуется указать набор входящих в него объектов

и связи между ними. Поведенческое описание включает в себя алгоритм поведения элемента схемы, не вдаваясь в его внутреннюю структуру, и часто используется для описания наиболее простых, базовых элементов.

При проектировании ПЛИС отсутствует возможность изменения всей электрической схемы, есть только возможность реконфигурации отдельных блоков, ячеек схемы. Поэтому не требуются описания детальнее уровня регистровой логики, и можно ограничиться только тремя верхними уровнями представления схемы.

1.3. Средства разработки ПЛИС

Современные пакеты для разработки и проектирования ПЛИС позволяют не только реализовывать алгоритмы, но также предоставляют средства для отладки, разводки для заданного кристалла ПЛИС, загрузки прошивки и многое другое.

1.3.1. Пакет Xilinx ISE

Программный пакет Integrated Software Environment (ISE), распространяемый фирмой Xilinx, предназначен для разработки и проектирования различных серий ПЛИС, выпускаемых этой фирмой. Данный программный пакет предоставляет широкий набор средств для разработки, среди которых отметим базовые возможности: синтез схемы устройства на основе описания, функциональное моделирование, размещение и трассировка схемы в кристалле, временное моделирование, построение программного файла и программирование кристаллов ПЛИС.

На этапе синтеза из описаний устройства создается схема, состоящая из набора базовых компонентов и примитивов, выбранных в соответствии с ресурсами используемого кристалла ПЛИС. При этом проводится значительная оптимизация схемы, удаляются дублирующие и неиспользуемые элементы схемы. На этом этапе разработчиком задается набор ограничений, таких как минимальная требуемая тактовая частота, требуемые задержки при вводе/выводе данных, распределение входных и выходных сигналов по ножкам ПЛИС.

Во время размещения и трассировки схемы происходит распределение выполняемых функций в конфигурируемые логические блоки с учетом заданных ограничений, при этом

определяются реальные времена задержек при распространении сигналов. На основе результатов трассировки вычисляются основные характеристики созданной схемы: количество использованных ресурсов ПЛИС, максимально возможная частота тактирующего сигнала и характерные временные задержки при вводе и выводе данных. На основе результатов, полученных на этапе размещения, создается программный файл, в котором содержится вся информация о конфигурации ПЛИС.

С помощью пакета ISE имеется возможность напрямую использовать некоторые компоненты и ресурсы ПЛИС. Например, на многих кристаллах фирмы Xilinx имеются блоки синхронной, 2-х портовой памяти объемом 16 Кбит, которые разработчик может напрямую использовать. Значительную часть ресурсов ПЛИС составляют конфигурируемые логические ячейки. Пакет ISE предоставляет возможность достаточно просто конфигурировать ячейки ПЛИС как различные, стандартные схемотехнические элементы: сумматоры, мультиплексоры, декодеры, триггеры, регистры, сдвиговые регистры и др. В данной работе активно использовались все эти возможности, поэтому описание аппаратной реализации дается с использованием стандартных схемотехнических терминов.

В процессе разработки часто требуется возможность тестирования, функционального и временного моделирования отдельных элементов устройства. Пакет ISE предоставляет такую возможность. Функциональное моделирование необходимо на начальном этапе разработки для выявления возможных логических ошибок в описаниях. Временное моделирование проводится после этапа трассировки и помогает находить ошибки, связанные с задержками при распространении сигналов. В целом средства моделирования позволяют выявить большинство возможных ошибок и значительно сократить время разработки устройства.

2. САПР XILINX ISE

Xilinx ISE представляет собой систему сквозного проектирования цифровых устройств на базе ПЛИС Xilinx. Этот пакет включает в себя:

- средства схемотехнического ввода;
- языки описания аппаратуры – VHDL, Verilog и Abel;
- средства моделирования;
- средства синтеза структуры кристалла;
- средства программирования (загрузки) кристаллов.

В ISE применен подход к трансляции исходных текстов, основанный на внутреннем представлении на языке описания аппаратуры. Преимуществом такого подхода перед представлением в схемотехническом виде является скорость трансляции проекта

2.1. Проект

Проект — это специальный файл, имя которого имеет расширение **.ise*. В проекте содержится информация о всех используемых исходных файлах. Среда разработки следит за изменениями файлов, перечисленных в файле проекта.

Для того чтобы начать работу с Xilinx ISE, необходимо запустить его графическую оболочку – Project Navigator. После его запуска появится окно следующего вида (рис. 1).

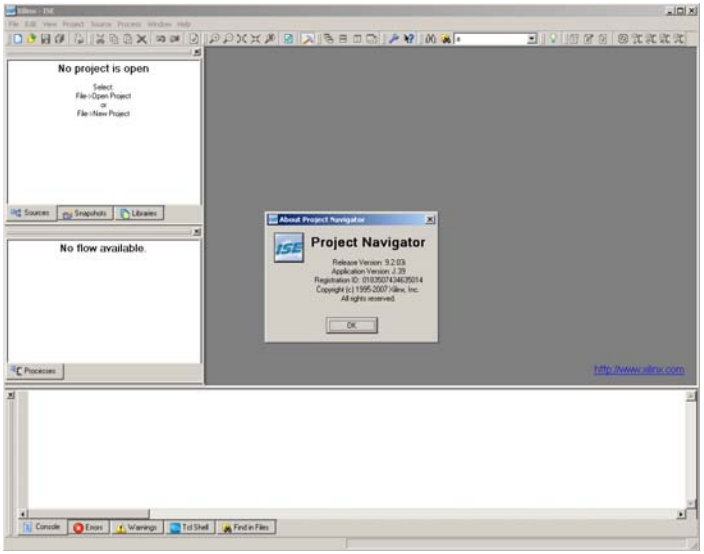

Рис. 1. Графическая оболочка Project Navigator

Для того чтобы начать новый проект, необходимо выбрать в меню File -> NewProject... После этого появится следующее диалоговое окно (рис. 2).

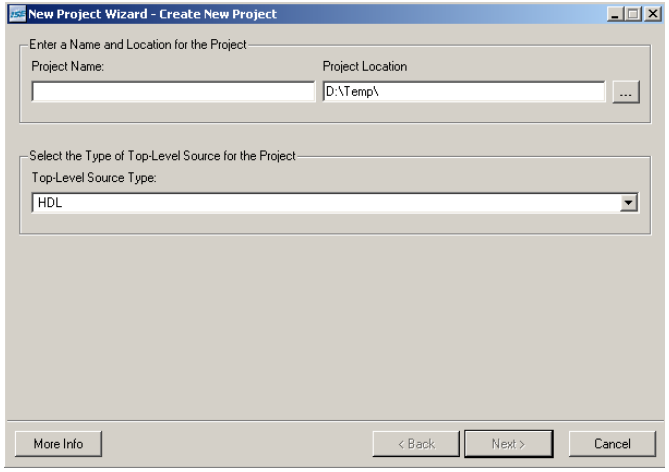

Рис. 2. Диалоговое окно создания нового проекта

Здесь предлагается выбрать название проекта, путь, где он будет находиться, и тип описания (схемотехническое или языковое).

После этого, в следующем диалоговом окне, необходимо указать параметры кристалла, для которого создается проект (рис. 3).

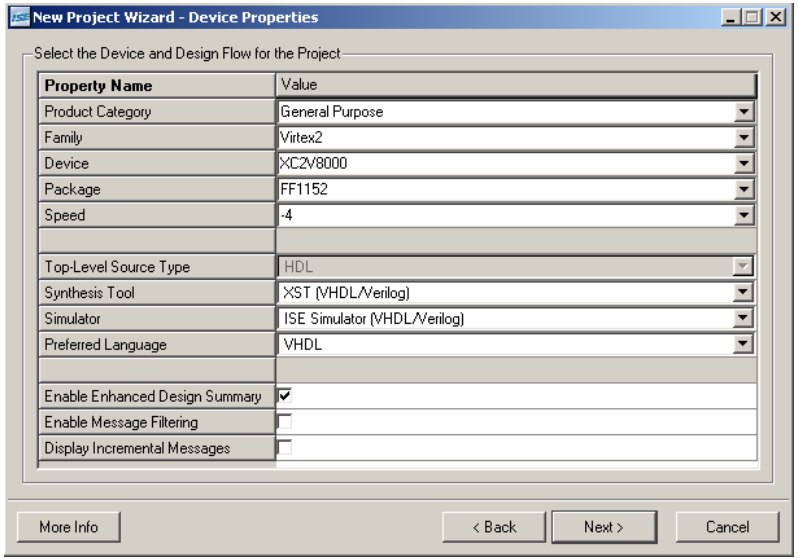

Рис. 3. Основные параметры кристалла ПЛИС

После осуществления описанных действий будет создан файл проекта, включающий в себя все заданные параметры проекта. Для того чтобы начать программное описание, необходимо добавить в проект файл (рис. 4), имеющий соответствующее расширение (при описании на языке VHDL файл имеет расширение **.vhd*).

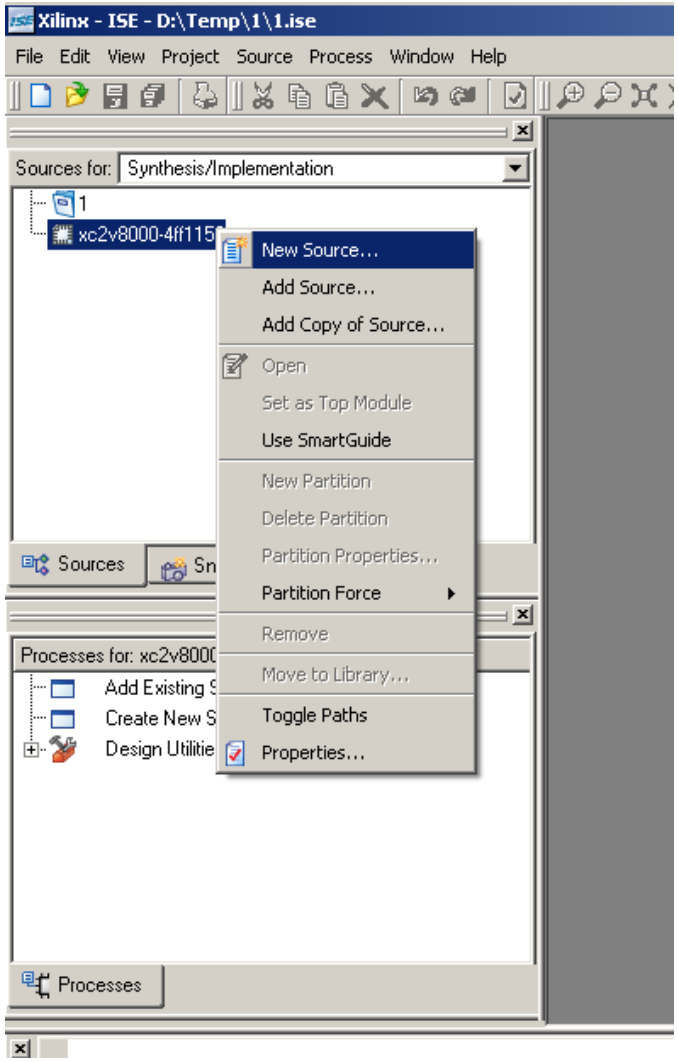

Рис. 4. Добавление файлов в проект

2.2. Структура модуля VHDL

В модуле всегда описывается некоторое устройство путем описания внешнего интерфейса и задания поведенческой модели. Ниже приведен пример модуля, реализующего триггер:

```
entity First is
    Port (
     clk  : in std_logic;
     iData  : in std_logic;
     oData  : out std_logic);
end First;
architecture First of First is
begin
  process (clk) is
  begin
    if(rising_edge(clk)) then
      oData <= iData;
    end if;
  end process;
end First;
```

Файл имеет следующую структуру:

- **[entity]** – описание внешнего интерфейса модуля. Используются идентификаторы IN и OUT для обозначения направления.
- **[architecture]** – описание внутренней архитектуры модуля. Допускается создание локальных сигналов вначале описания в этой части до слова BEGIN.
  - **[begin]** – эта структура заканчивается словом END. Внутри описывается синхранная (process) и асинхронная логика.

2.3. Файл описания связей UCF

После того как проект реализован на языке VHDL, необходимо задать условия для корректной разводки. К необходимым условиям относятся связки входов и выходов основного модуля проекта с реальными ножками кристалла.

Файл UCF (User Comstrains File) позволяет устанавливать связи следующим образом:

```
NET  "sys_clk_p"       LOC = "D4";
```

Также в этом файле возможно объединение существующих примитивов кристалла в группы, после чего установливаются временные ограничения на распространение сигналов между ними. Это делается следующим образом:


```
INST "DataBank" TNM = "GRP_MyRamB ";
INST "RamB_ iData_FDs" TNM = "GRP_iData";
TIMESPEC "TS_0" = FROM "MyRamB" TO "GRP_iData" 5
ns;
```

Также можно задать общее условие на глобальный тактовый сигнал:

```
TIMESPEC "TS_Clock" = PERIOD "Clock" 10 ns HIGH
50 %;
```

2.4. Трансляция проекта

Трансляция проекта состоит из трех основных этапов: синтез (Synthesize), имплементация (Implement Design) и собственно генерация программного файла (Generate Programming File).

Для трансляции всего проекта достаточно запустить процесс генерации программного файла. Недостающие для его выполнения процессы будут запущены автоматически.

На рис. 5 показана иерархия процессов, доступных для проекта на базе FPGA.

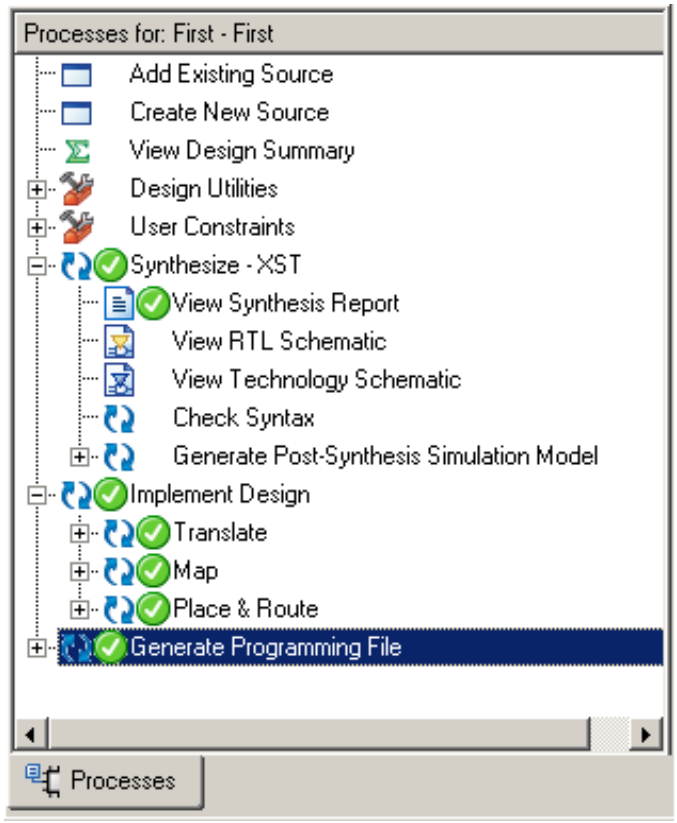

Рис. 5. Иерархия процессов для проекта на FPGA

3. САПР Active-HDL

Active-HDL – интеграционный пакет разработки и моделирования цифровых схем, созданных с помощью языков описания оборудования и C/C++ языков программирования. Он предоставляет инженерам и командам разработчиков средства для проектирования, тестирования и реализации.

В рамках данного курса этот пакет используется для моделирования и тестирования проектов.

3.1. Начало работы

Также как и в ISE, для начала работы необходимо создать проект. Но при этом в рамках Active-HDL несколько проектов могут быть объединены в одно рабочее пространство – workspace.

Ниже приведен рисунок запуска программы. При этом среда предлагает либо открыть существующее рабочее пространство (если оно есть), либо создать новое рабочее пространство (рис. 6).

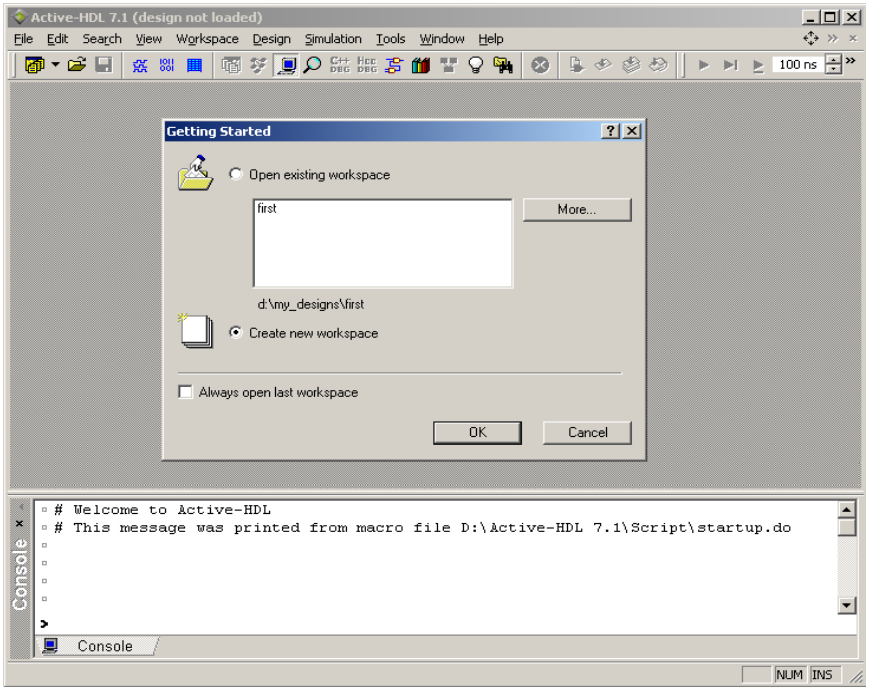

Рис.69. Запуск Active-HDL

Путем несложных операций по созданию нового рабочего пространства, добавления в него пустого проекта и добавления файла vhd можно получить следующий вид проекта (рис. 7):

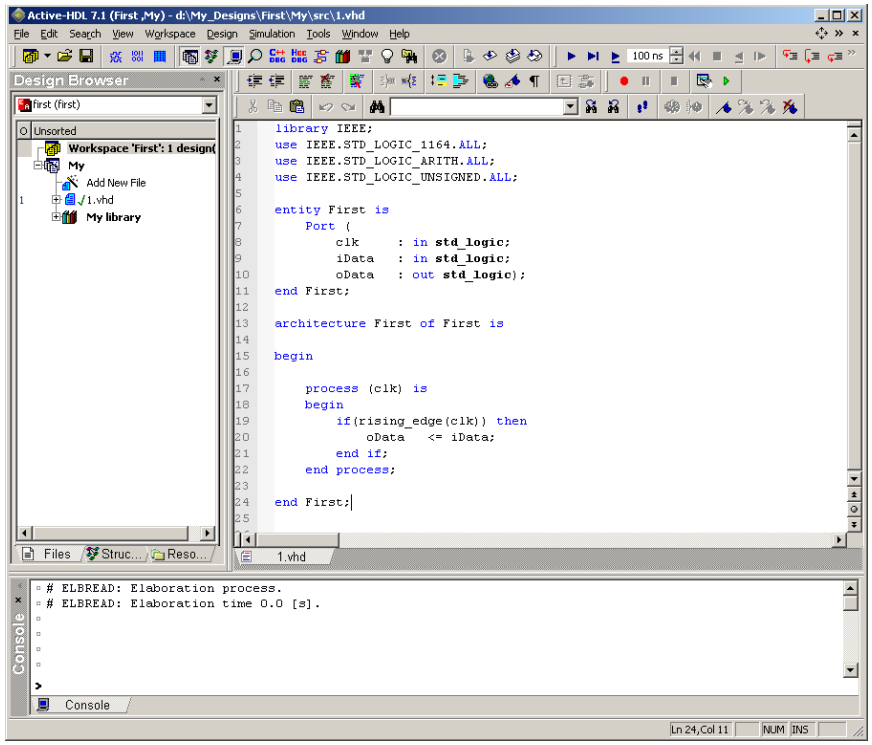

Рис. 7. Проект в среде Active-HDL

3.2. Компиляция проекта

Для того чтобы получить проект готовый к моделированию, необходимо проверить написанный код на соответсвие синтаксису и преобразовать все файлы в один.

Среда Active-HDL дает возможность как компилировать каждый файл проекта по отдельности (применяется при добавлении либо при поиске ошибок), так и скомпилировать проект целиком. Для этого применяются команды, вызываемые нажатием правой клавиши мыши по файлу проекта либо по проекту целиком. На рис. 8 приведен пример контекстного меню.

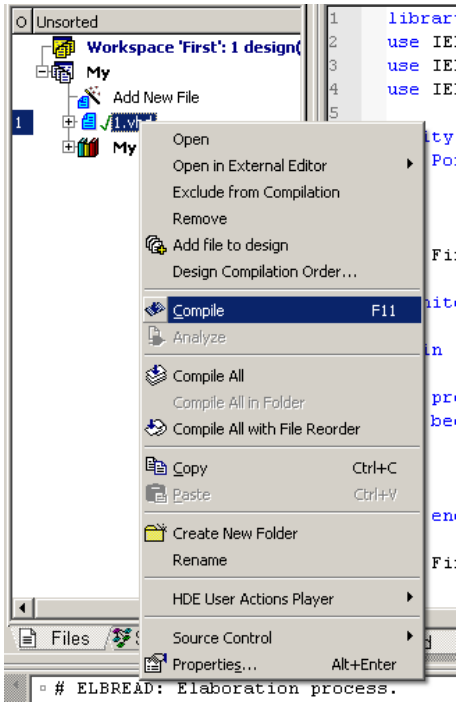

Рис. 8. Контекстное меню проекта в Active-HDL

Если в процессе компиляции возникнут ошибки, то об этом будет сообщено в нижнем окошке и указан номер строки, в которой обнаружена ошибка.

3.3. Моделирование проекта

После того как весь проект скомпилирован, можно приступать к его моделированию. Суть моделирования заключается в возможности задавать входные условия различными и видеть результат работы модулей.

Для модулирования необходимо добавить в проект специальный файл следующим образом:

```
File -> New -> Waveform
```

После этого его необходимо сохранить, только тогда он появится в списке файлов проекта. Этот файл с расширением *.awf хранит в себе правила задания входных сигналов и результаты проведенного моделирования.

3.3.1. Задание видимых сигналов для моделирования

При моделировании необходимо выбрать сигналы, которые будут видимы при моделировании. Для сигналов, являющихся входными, возможно задание различных значений. Выходные сигналы можно будет видеть либо в виде логических значений, либо в виде значений, если скомпоновать их в шины.

Для выбора сигналов необходимо перейти в закладку **Structure**, находящуюся внизу окна **Design Browser**. При этом, если в рабочем пространстве находится несколько проектов или файлов внутри проекта, необходимо выбрать моделируемый объект в верхней закладке, так как в среде имеется возможность моделировать файлы по отдельности (рис. 9).

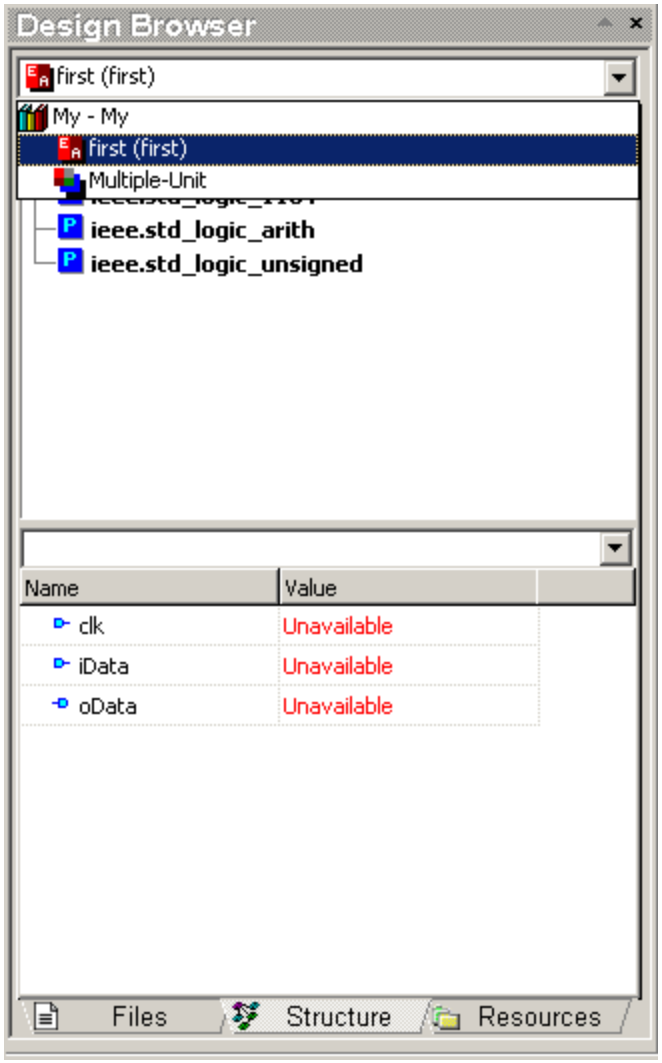

Рис. 9. Выбор объекта для моделирования в Active-HDL

После выбора моделируемого объекта в окне ниже появится список доступных для наблюдения сигналов. В случае нашего проекта их 3: тактовый сигнал, входные данные и выходные.

Для добавления сигналов в область видимости можно, зажав левую кнопку мыши на интересующем сигнале, перетащить его в левую часть области имен сигналов правого окна – область имен сигналов. Проделав это для всех трех сигналов, мы получим следующий вид файла моделирования, готового для использования (рис. 10).

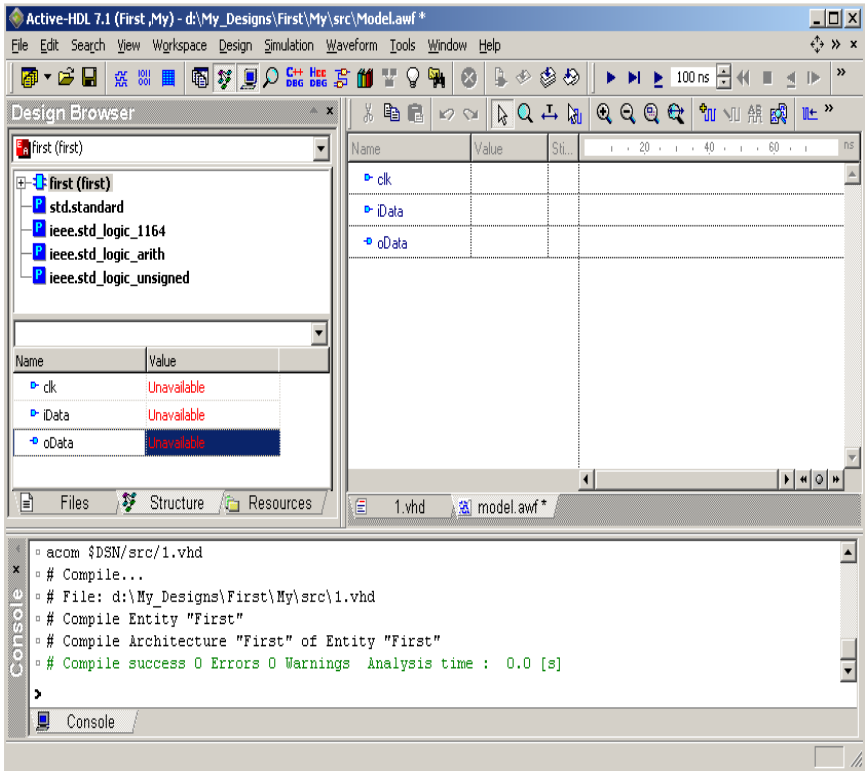

Рис. 10. Моделирование проекта в Active-HDL

3.3.2. Задание входных сигналов

Для задания входного сигнала необходимо, нажав на нем правую кнопку мыши, вызвать контекстное меню и выбрать **Simulators** (рис. 11):

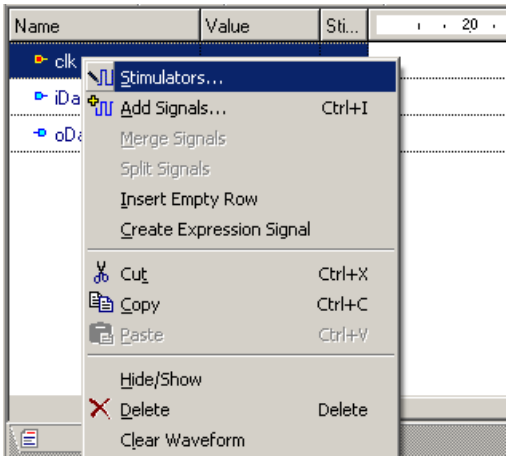

Рис. 11. Задание входных сигналов

После этого отобразится меню типов задания входных сигналов (рис. 12).

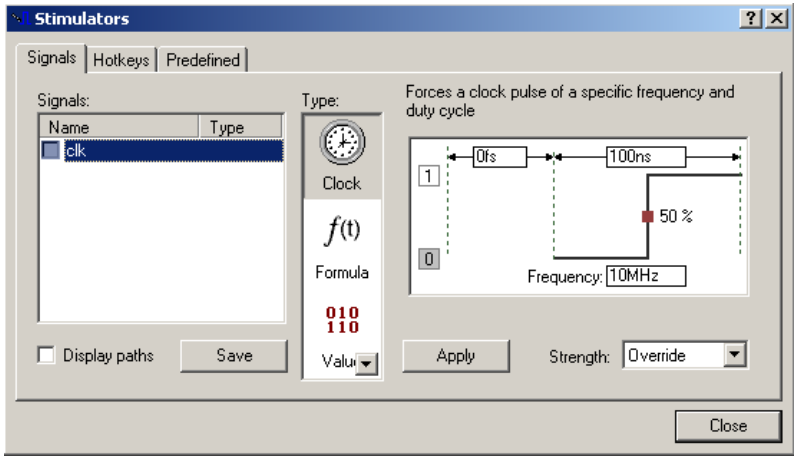

Рис. 12. Выбор типа симуляторов для входного сигнала

В системе существует несколько типов симуляторов:

- [Clock] – периодический сигнал;
- [Formula] – текстовая формула зависимости от времени;
- [Value] – константное значение;

- [Hotkey] – устанавливает изменение сигнала при нажатии клавиши;
- [Counter] – счетчик;
- [Custom] – задание сигнала непосредственно на диаграмме;
- [Predefined] – задание предопределенного сигнала с использованием другого заданного;
- [Random] – произвольное значение с заданием периода изменения.

Для задания зависимости уровня сигнала от времени формулой используется следующий синтаксис:

```
1 0 ns, 0 60 ns, 1 100 ns
```

Другими словами задается значение уровня сигнала и момент времени, когда происходит указанное изменение. Далее до следующего изменения (перечисляется через запятую) сигнал остается в указанном значении.

3.3.3. Запуск моделирования и результаты

После того как определены все значимые входные значения, можно запускать моделирование. На рис. 13 приведен пример моделирования вышеописанного проекта со следующими условия на входные данные:

- Clock – периодический сигнал с частотой 20 МГц;
- iData – задано следующей формулой: «1 0 ns, 0 60 ns, 1 100 ns».

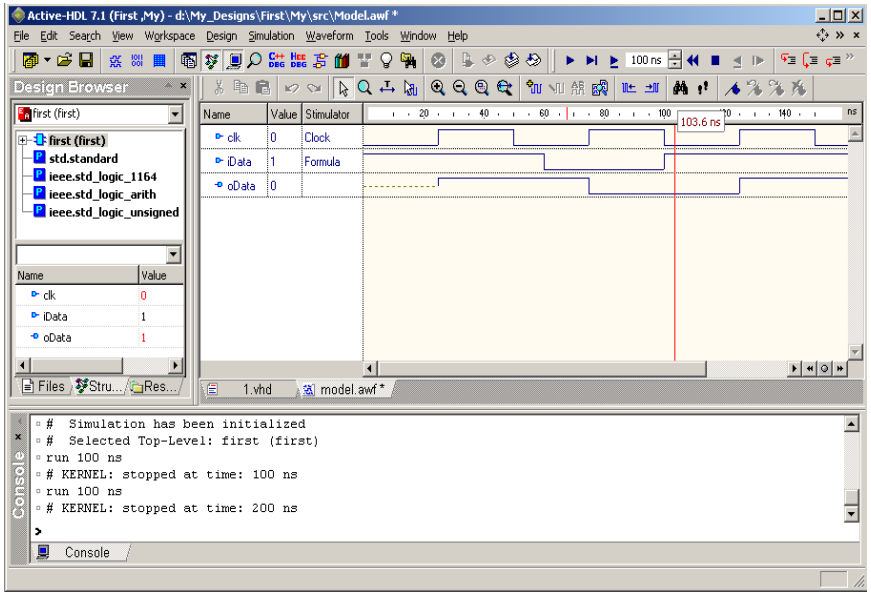

Рис. 13. Результат моделирования проекта на Active-HDL

В командах управления диаграммой присутствуют кнопки изменения масштаба, быстрого сдвига на момент срабатывания триггера, которые упрощают поиск нужного момента времени.

4. Отладка с помощью ChipScope

ChipScope – программный пакет компании Xilinx, предназначенный для внутрисхемной отладки ПЛИС FPGA фирмы Xilinx.

В пакете имеются возможности по созданию IP-ядер (**Core Generator** и **Core Inserter**) и программа для наблюдения за тестовыми сигналами с различными возможностями по запуску и синхронизации (**Analyzer**).

Система работает посредством внедрения в проект IP-ядер логического анализатора, шинного анализатора и виртуального ввода/вывода, позволяя наблюдать за любым заданным внутренним сигналом или узлом, включая встроенные аппаратные или софт-процессоры. Сигналы захватываются со скоростью, допустимой хост-компьютером, и передаются через интерфейс JTAG, таким образом освобождая программируемые выводы ПЛИС для использования разработчиком.

Следует отметить, что по интерфейсу JTAG производится и загрузка конфигурационной последовательности в ПЛИС, соответственно для отладки не требуется какое-либо дополнительное оборудование. Захваченные сигналы могут быть проанализированы с помощью логического анализатора, входящего в состав программы ChipScope.

4.1. Создание виртуального логического анализатора

Существует два основных способа внедрения в проект виртуального логического анализатора:

- **Core Generator** позволяет создать отдельный компонент, который программист самостоятельно вставляет в свой проект;
- **Core Inserter** позволяет автоматически внедрить в проект анализатор, указав файл проекта ISE.

Для задач обучения предпочтительным является использование первого способа, благодаря которому программист максимально самостоятельно управляет своим проектом. Этот подход и будет рассматриваться далее.

Для использования виртуального логического анализатора необходимо наличие двух компонентов:

- Integrated Logic Analyser (**ILA**);
- Integrated Controller (**ICON**).

Оба эти компоненты создаются с помощью программы ChipScope Core Generator, входящей в состав пакета ChipScope (рис. 14).

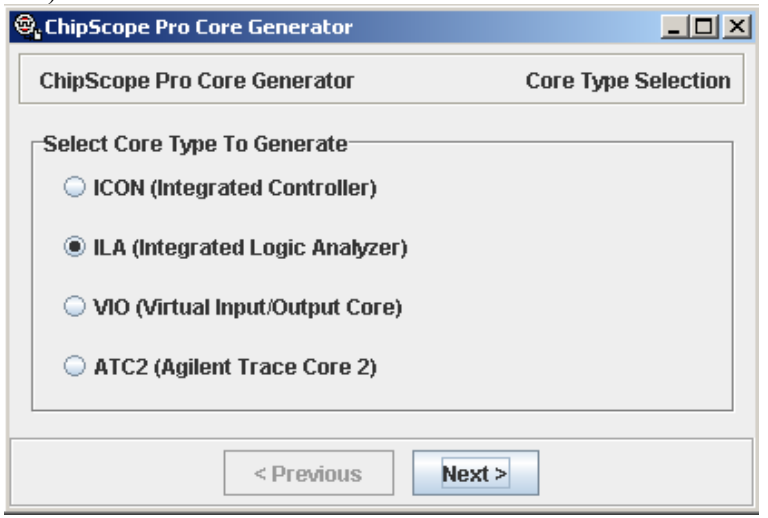

Рис. 14. Программа ChipScope Core Generator

4.1.1. Integrated Logic Analyser

Компонент ILA является ключевым и предназначен для описания тестовых отладочных сигналов и сигнала их тактирования. Рассмотрим по шагам создание данного компонента.

Первый шаг. Предлагается выбрать следующие параметры (рис. 15):

- место, куда будут помещены файлы компонента;
- кристалл, для которого будет предназначен компонент;
- фронт тактового сигнала для тактирования тестовых сигналов.

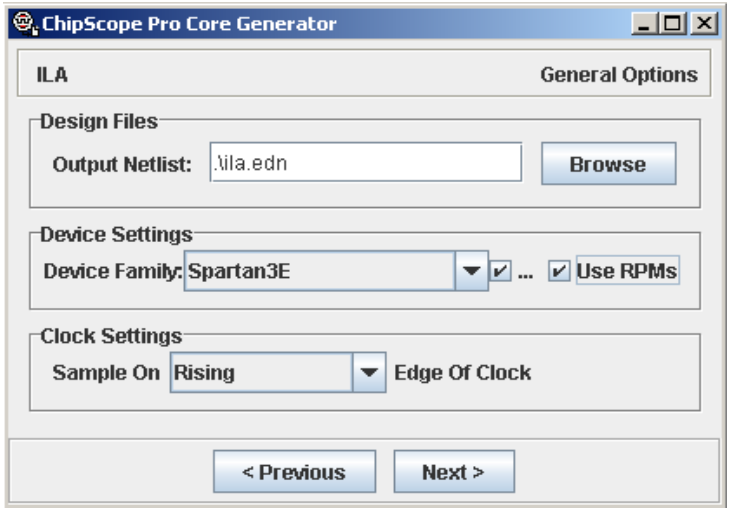

Рис. 15. Создание компонента ILA – шаг 1

Второй шаг. На данном этапе пользователем задаются условия запуска анализатора для записи данных, в частности количество портов, конфигурация каждого порта и условия запуска по каждому порту. Для примера создаваемый анализатор будет отслеживать два порта: один однобитовый и один 32-разрядный. При этом условия по запуску позволяют отслеживать значения на портах триггера, а на однобитном порту в дополнение еще и изменения фронта (рис. 16).

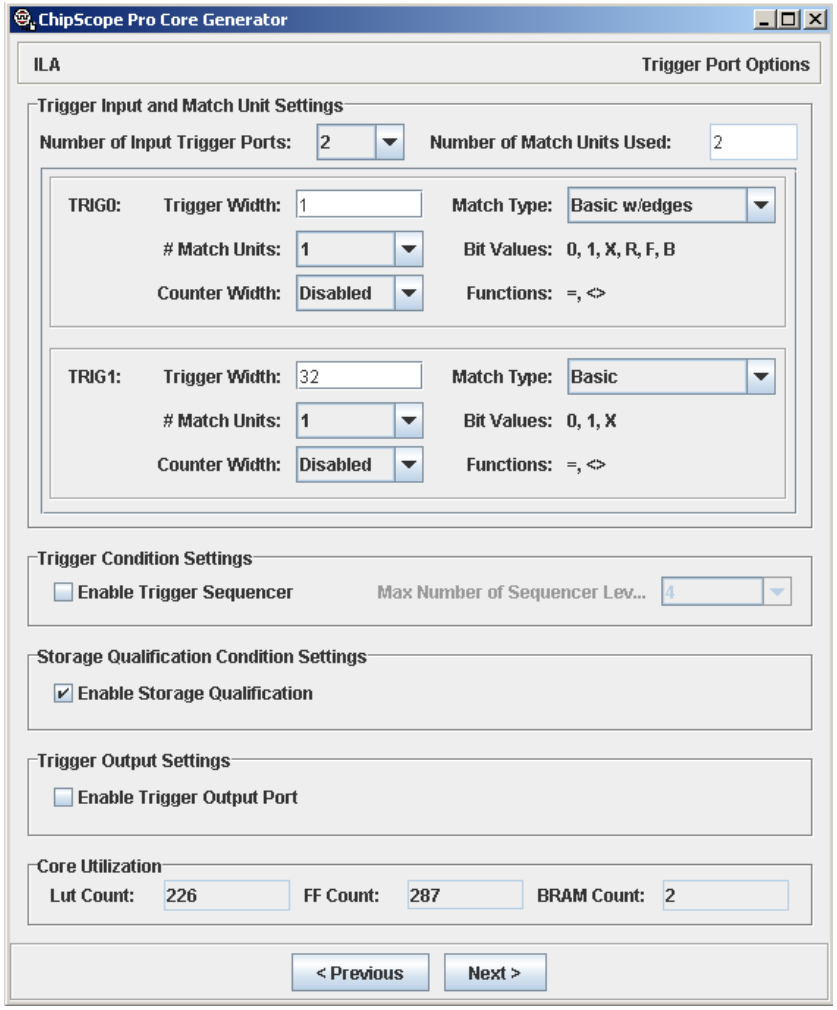

Рис. 16. Создание компонента ILA – шаг 2

Внизу окна указывается, какой объем логики кристалла будет задействован для компонента виртуального анализатора. Но эта оценка не является окончательной и зависит также от следующих этапов задания параметров.

Третий шаг. Задается объем отсчетов, которые будут записаны по условию срабатывания триггера на обозначенных портах и

разрядность данных, доступных для просмотра в логическом анализаторе.

Важно помнить, что невозможно задавать условия запуска по данной шине. Но при просмотре записанных анализатором данных доступны обозначенные только на этом шаге. Другими словами, портами триггера, не выбранными для записи данных, можно только задавать условия запуска, но не существует возможности смотреть их состояние.

В нашем примере мы будет использовать для записи данных созданный 32-разрядный порт триггера, не создавая отдельной шины данных (рис. 17).

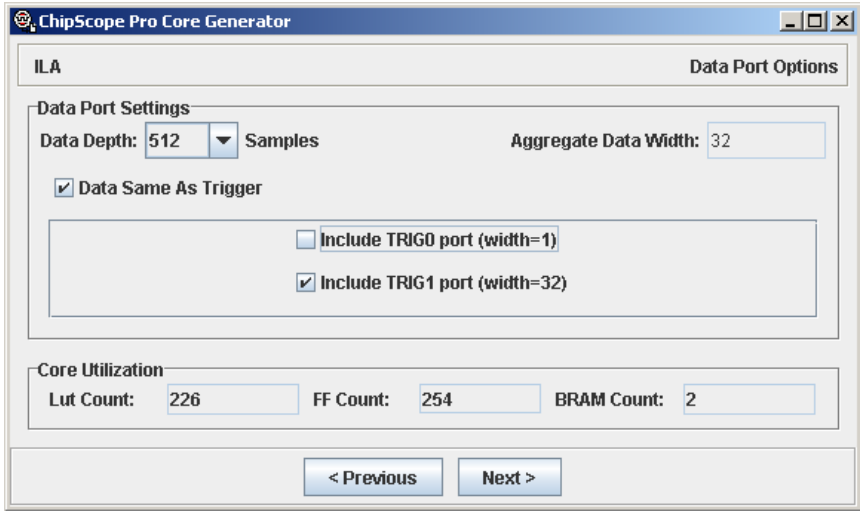

Рис. 17. Создание компонента ILA – шаг 3

Запись данных производится во внутренние блоки памяти ПЛИС и лишь затем передается в ПК, так как скорости передачи данных по интерфейсу JTAG может быть недостаточно.

В данном примере происходит запись 512 отсчетов выбранных сигналов. Чаще всего этого достаточно для тестирования и отладки проекта, но в некоторых случаях необходимо записать как можно больше отсчетов. Максимальный объем определяется емкостью кристалла. При этом необходимо помнить, что определенное количество логики и блоков памяти занимает собственно проект, и можно распоряжаться лишь оставшимся количеством.

Четвертый шаг. Задается язык описания компонента и средство для его синтеза.

Результатом будут файлы ***ila.edn*** и ***ila.ncf***, которые необходимо скопировать в проект.

Сгенерированный в результате описанных действий компонент будет иметь следующий вид:

```
component ila
  port
  (
    control  : in std_logic_vector(35 downto
0);
    clk    : in std_logic;
    trig0  : in std_logic_vector(0 downto
0);
    trig1  : in std_logic_vector(31 downto
0)
  );
end component;
```

4.1.2. Integrated Controller

В созданном компоненте **ila** присутствует порт **control**, который предназначен для управления работой виртуального анализатора. Управление заключается в возможности задавать условия срабатывания триггера и т. д. Таким образом, компонент **icon** предназначен именно для управления компонентом анализатора.

Особенность компоненты заключается в возможности управлять несколькими компонентами виртуальных анализаторов. Это задается количеством портов управления.

Создание компонента происходит за два шага и по сути аналогично первому и последнему шагу при создании компонента анализатора.

В результате происходит создание двух файлов для их включения в проект. Созданный компонент управления одним анализатором выглядит следующим образом:

```
component icon
port
(
  control0 : out std_logic_vector(35 downto
0)
);
end component;
```

4.2. Использование виртуального логического анализатора

Программа ChipScope Analyzer предназначена для управления виртуальным логическим анализатором ChipScope.

После того как компоненты анализаторы вставлены в проект, сгенерирован программный файл и загружен в кристалл, не отсоединяя кабель загрузчика, запустите приложение ChipScope Analyzer.

Открывшееся окно будет иметь следующий вид (рис. 18).

Рис. 18. Приложение ChipScope Analyzer

При первом запуске не загружено никакого проекта, поэтому автоматически создается новый проект.

Для использования виртуального логического анализатора необходимо выполнить команду подключения, указав тип соединения с устройством (рис. 19).

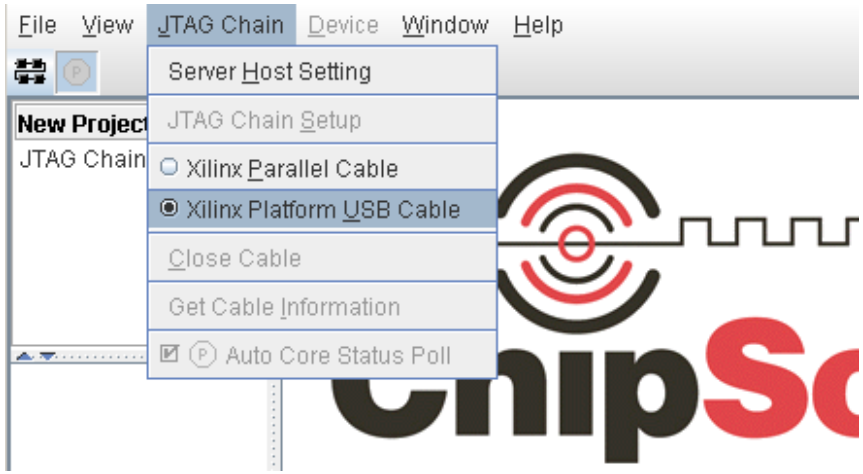

Рис. 19. Выбор типа соединения с устройством

Далее необходимо указать скорость передачи данных. После этого запустится поиск виртуального анализатора на выбранном соединении и попытка подключения к нему на выбранной скорости. В случае успеха будет выведено окно со списком обнаруженных кристаллов ПЛИС и предложено выбрать один из них для тестирования (рис. 20).

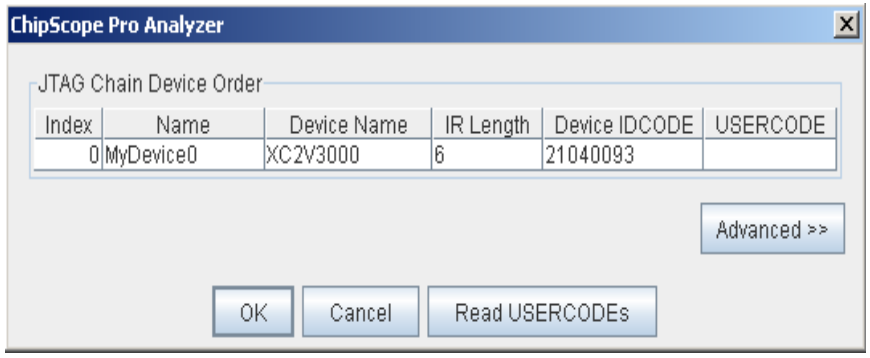

Рис. 20. Выбор устройства для тестирования

При открытии анализатора доступны следующие окна (рис. 21).

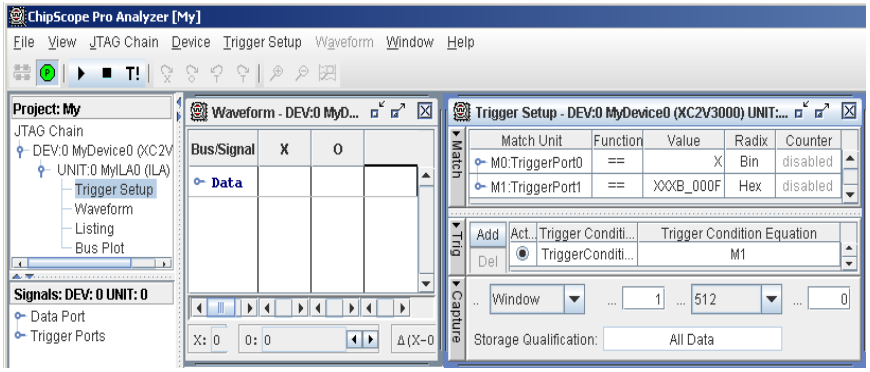

Рис. 21. Окна открытого проекта ChipScope

- Project – главное окно. В нем отображаются доступные окна для просмотра и редактирования;
- Signals – доступные сигналы данных и триггеров;
- Waveform – окно временных диаграмм выбранных сигналов;
- Trigger Setup – здесь задаются условия запуска анализатора.

В верхней части главного окна доступна команда запуска логического анализатора и остановки записи данных. В случае выбранного активного окна временных диаграмм также задается масштабирование и переход к заданным меткам на диаграмме.

4.2.1. Выбор сигналов и просмотр

Сигналы, находящиеся в списке в окне ***Waveform***, доступны к просмотру на диаграмме. В этот список можно поместить лишь те сигналы, которые находятся в списке сигналов данных в окне ***Signals***.

Для выбора сигналов существует 2 способа:

- перетащить мышью конкретный сигнал или шину из окна ***Signals*** в окно ***Waveform***;
- вызвать меню на сигнале или шине в окне ***Signals*** (правой клавишей мыши) и выбрать «***Add To View***».

При просмотре диаграмм существует возможность выбирать режим представления данных: бинарный, шестнадцатеричный и т.

д. Это делается вызовом меню на интересующем сигнале или шине и выборе Bus Radix (рис. 22).

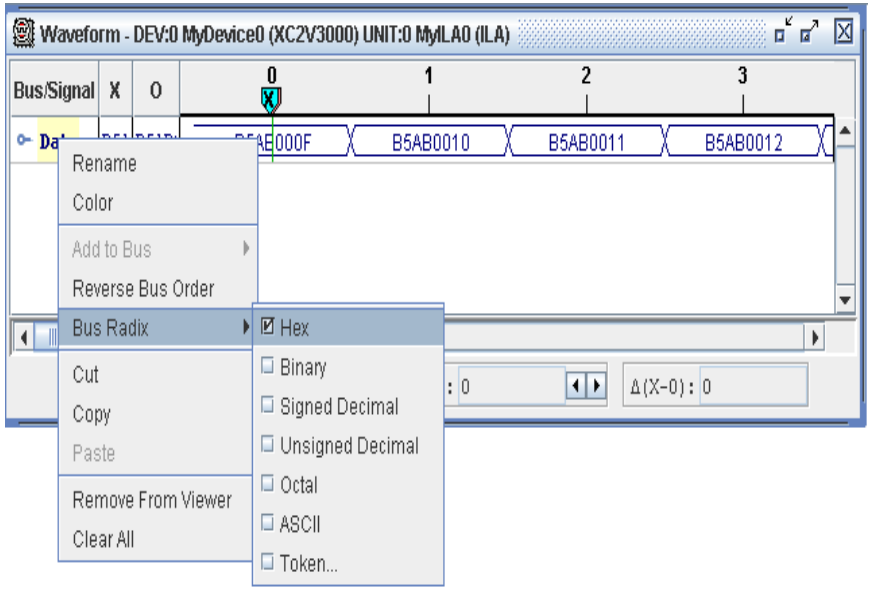

Рис. 22. Меню представления сигнала или шины

4.2.2. Создание условия запуска

Управление условиями запуска осуществляется в окне ***Trigger Setup***, состоящем из трех основных частей:

- ***Match*** – здесь можно увидеть все заданные сигналы и шины, по которым можно задавать условия запуска;
- ***Trig*** – выбирается активный триггер или комбинация для запуска;
- ***Capture*** – задаются параметры записи данных, в том числе количество отсчетов.

Общий вид всего окна ***Trigger Setup*** приведен на рис. 23.

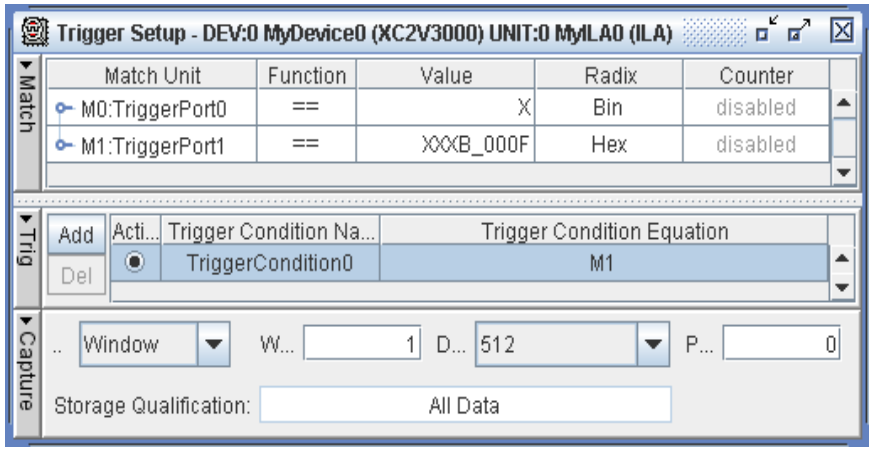

Рис. 23. Окно управления условиями запуска

В части ***Match*** можно задавать условия срабатывания триггера. При этом значение можно указывать в шестнадцатеричной, восьмеричной, двоичной, знаковой и беззнаковой форме – это выбирается в столбце ***Radix***. Указание величины «**X**» означает любое значение.

В части ***Trig*** можно создавать несколько условий и при необходимости выбирать активное. В приведенном примере создано только одно условие ***TriggerCondition0***. Во втором столбце ***Trigget Condition Equation*** можно задавать следующие условия на существующие триггеры: объединение по «И», объединение по «ИЛИ» и отрицание любого из условий.

В части ***Capture*** существует возможность задавать количество отсчетов для записи данных – при формировании компонента логического анализатора указывается максимальное количество отсчетов.

5. Проектирование на VHDL

Язык описания аппаратуры VHDL является достаточно мощным средством абстрактного описания цифровых устройств, освобождающим программиста от необходимости комбинировать нужную схему из готовых библиотечных компонентов.

Все цифровые узлы, создаваемые в ПЛИС, теоретически могут быть реализованы только с использованием встроенных триггеров и комбинаторной логики. Но очевидно, что построение сложной схемы (конечного автомата, счетчика с условиями растра и т. д.) достаточно трудоемко произвести по классической методологии графического описания создаваемой схемы. Применение языков HDL позволяет описать поведение создаваемой схемы в алгоритмическом виде, что существенно облегчает анализ работы такого устройства, его модификацию и отладку.

5.1. Синтаксис VHDL

Синтаксис языка во многом схож с типовыми языками высокоуровневого программирования. Но, являясь языком описания аппаратуры, VHDL оперирует с базовыми элементами кристаллов ПЛИС: триггерами, логическими операциями и т. п.

По своей сути язык VHDL является структурным, так как позволяет разделять программу на модули, создавать свои типы данных и оперировать с ними.

5.1.1. *Сигналы*

Все переменные, описываемые на языке VHDL, предназначенные для описания схемы в ПЛИС, имеют два типа:

- ***std_logic*** – для описания логического битового сигнала;
- ***std_logic_vector*** – комбинирование нескольких сигналов std_logic в одну шины для более удобного их использования.

Для описания новых переменных используется следующий синтаксис:

```
My_Var : std_logic;
My_Cmpl  : std_logic_vector(7 downto 0);
```

Введение новых переменных происходит в теле описания архитектуры модуля до ключевого слова **begin**.

При описании интерфейсных сигналов модуля на VHDL (интерфейса) дополнительно используются идентификаторы направления сигнала: ***IN, OUT, INOUT***.

При этом описание интерфейса модуля выглядит следующим образом:

```
entity First is
  Port (
  clk    : in std_logic;
  iData  : in std_logic_vector(3 downto 0);
  oData  : out std_logic_vector(3 downto 0));
end First;
```

При работе с переменными типа ***std_logic*** используют значения **1** и **0**, заключенные в одинарные кавычки (**'0'**).

При присвоении значений переменным типа ***std_logic_vector*** и их сравнении с константными значениями чаще всего используют двоичную и шестнадцатеричную системы исчисления. При этом для записи значения в двоичном виде используются двойные ковычки (“11001”), а в шестнадцатиричном виде перед значением указывается символ **X** (X”5A”).

5.1.2. *Операторы сравнения и присвоения*

Все логические выражения в VHDL представляются с помощью следующих операторов:

- [AND] – логическое «И»;
- [OR] – логическое «ИЛИ»;
- [XOR] – исключающее «ИЛИ»;
- [NOT] – логическое отрицание;
- [=] – отношение «равно»;
- [/=] – отношение «не равно».

В языке VHDL используются разные символы для присвоения значений и для проверки равенства. Это позволяет избежать многих ошибок, невольно возникающих у программиста в таких языках, как C/C++. Для присвоения значения используется следующий синтаксис:


```
My_Cmpl    <= "A5";
```

А если необходимо сравнить значение этой переменной с указанной константой, то запись выглядеть так:

```
My_Cmpl    = "A5";
```

5.1.3. *Синхронная и асинхронная логика*

Как говорилось выше, вся поведенческая модель компонента описывается в его теле: между ключевыми словами **begin** и **end**.

Для описания синхронной логики используется блок, определенный ключевым словом **process**, имеющий обозначенное начало и конец. Все, что описано внутри этого блока, исполняется только по изменению фронта указанного сигнала. Описание блока происходит следующим образом:

```
process (clk) is
  begin
    -- тело процесса
  end process;
```

Здесь ***clk*** – это тактовый сигнал, по изменению фронта которого происходят указанные операции.

Допускается использовать два идентификатора для определения фронта сигнала:

- ***rising_edge()*** – восходящий фронт: сигнал меняет значение с 0 на 1;
- ***falling_edge()*** – низходящий фронт: сигнал меняет значение с 1 на 0.

Все, что описано вне процесса, является асинхронным и всегда исполняется с учетом задержек на распространение сигналов.

5.1.4. *Условные операторы*

Средства языка предлагают два условных оператора: **if** и **when**. Необходимо помнить, что область применения этих операторов различна: первый применяется для синхронной логики, а второй – только для асинхронной.

Синтаксис оператора **if** таков, что выполняет указанные действия, если условие в скобках является истиной. Также оператор позволяет сколько угодно разветвлять оператор, используя служебные слова **else** или **elsif**. Необходимо помнить, что оператор всегда заканчивается служебным словом **endif**:

```
if(reset = '1') then
  a <= '0';
elsif(rising_edge(clk)) then
  a <= a + '1';
end if;
```

Оператор **when** также позволяет разветвлять условия и используется следующим образом:

```
b <=X"1" when ch = X"A" else
    X"2" when ch = X"B" else
    X"0";
```

5.1.5. *Переключатели*

Существующий в языке переключатель **case** позволяет сделать выбор значения переменной между обозначенными значениями. По сути действие этого переключателя аналогично условному оператору **if**, так как применяется только в процессах. Но зачастую использование именно **case** позволяет сделать написанный код более легким для понимания и отладки. Синтаксис его применения таков:

```
case(my_var) is
  when X"1" =>
    a <= "0001";
  when X"2" =>
    a <= "0010";
  when others =>
    a <= "0000";
end case;
```


5.1.6. *Собственные типы данных*

Средства языка позволяют создавать собственные именованные типы данных, что дает возможность получить более информативный код и значительно упростить этапы моделирования и отладки.

Собственные типы данных часто применяются при описании конечных автоматов, так как каждое состояние имеет свой смысл, а оперировать именами этих состояний гораздо удобнее. Ниже приведен пример заведения переменной на примере конечного автомата:

```
type State_type is ( S_Idle,
                     S_Operation,
                     S_Finish);
Signal CurState : State_type;
```

Таким образом, переменная ***CurState*** может принимать три описанных буквенных значения, а значит, сравниваться и приравнивать к ним.

5.1.7. *Массивы*

В языке VHDL существует возможность заводить массивы данных. Массивы могут быть как одномерными, так и многомерными. Введение массивов похоже на заведение собственного типа данных и происходит следующим образом:

```
  type memory is array (1 to 5) of
std_logic_vector (7 downto 0);
  signal MyMem : memory;
```

Двумерный массив описывается так:

```
  type memory2 is array (1 to 5, 1 to 5) of
std_logic_vector (7 downto 0);

  signal MyMem2 : memory2;
```

Обращение к переменным происходит с указанием индекса элемента:

```
MyMem(1)    <= X"01";
MyMem2(1,4)  <= X"01";
```

5.2. Реализация основных цифровых узлов

В данном разделе рассматривается реализация основных узлов, используемых в цифровой схемотехнике, на языке VHDL. Для большей наглядности и компактности часть кодов, относящихся к подключению библиотек, будет опущена.

5.2.1. *Дешифратор*

Дешифратор можно рассматривать как несколько отдельных логических элементов, выходы которых образуют многоразрядный сигнал, рассматриваемый в совокупности отдельных образующих его логических линий. Например, дешифратор D2_3E из стандартной библиотеки имеет следующую внутреннюю структуру (рис. 24).

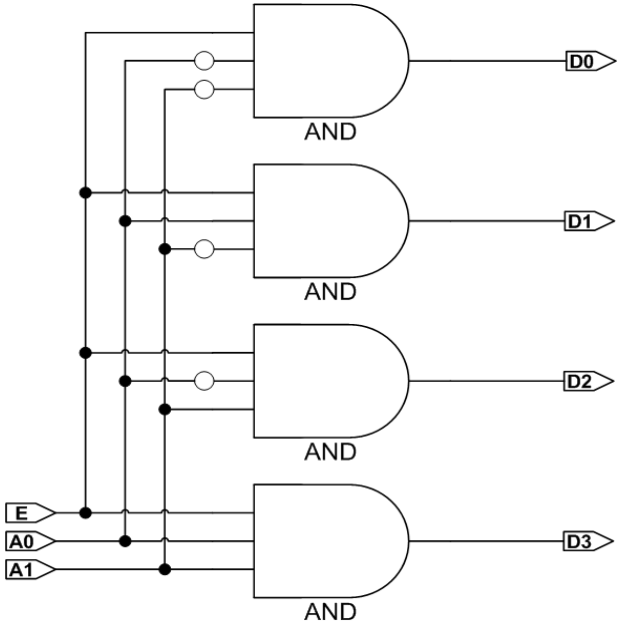

Рис. 24. Эквивалентная схема дешифратора

Описанная схема устанавливает сигнал логической единицы на одном из выходов D0-D3 в соответствии с двоичной комбинацией на входах A0-A1. При этом вход E действует как вход глобального разрешения работы.

Соответствующий VHDL код для реализации указанного дешифратора в асинхронном виде выглядит следующим образом:

```
entity MyDecoder is
    Port (
    E : in std_logic;
    A : in std_logic_vector(1 downto 0);
    D : out std_logic_vector(3 downto 0));
end MyDecoder;
architecture MyDecoder of MyDecoder is
begin
  D <="0000" when E = '0' else
      "0001" when A = "00" else
      "0010" when A = "01" else
      "0100" when A = "10" else
      "1000";-- when A = ”11”
end MyDecoder;
```

5.2.2. *Шифратор*

Это обратная задача дешифратору. Приведем код синхронного шифратора:

```
  entity MyCoder is
      Port (
      clk : in std_logic;
      E   : in std_logic;
      A   : in std_logic_vector(3 downto 0);
      D   : out std_logic_vector(1 downto 0));
  end MyCoder;

  architecture MyCoder of MyCoder is
  begin

  process(clk)
  begin
    if(rising_edge(clk)) then
```

```
      if (E = '0') then
        D <= "00";
      else
        case (A) is
          when "0001" => D <= "00";
          when "0010" => D <= "01";
          when "0100" => D <= "10";
          when "1000" => D <= "11";
          when others => D <= "00";
        end case;
      end if;
    end if;
  end process;

  end MyCoder;
```

5.2.3. *Счетчики*

Счетчики очень часто используются в различных цифровых схемах. В их основе лежит регистр, хранящий значение счетчика. Регистр завязан через сумматор обратной связью на самого себя.

Часто в реальных задачах бывают условия по сбросу счетчиков и загрузке предустановленного значения. В качестве примера разберем счетчик, имеющий, помимо этого, еще и управляющий сигнал, разрешающий счет – **CE**:

```
  entity MyCounter is
      Port (
      clk    : in std_logic;
      reset    : in std_logic;
      CE     : in std_logic;
      Load   : in std_logic;
      LoadVal  :  in  std_logic_vector(7  downto
0);
      Counter  :  out  std_logic_vector(7  downto
0));
  end MyCounter;

  architecture MyCounter of MyCounter is
  signal MyCounter :  std_logic_vector(7  downto
0);
```



```
begin
  process(clk, reset)
  begin
    if(reset = '1') then
      MyCounter  <= X"00";
    elsif(rising_edge(clk) AND CE = '1') then
      if(Load = '1') then
        MyCounter  <= LoadVal;
      else
        MyCounter  <= MyCounter + '1';
      end if;
    end if;
  end process;
end MyCounter;
```

Необходимо отметить, что в описанном примере использован асинхронный сброс (**reset**). В некоторых задачах это бывает критичным, если, например, тактовый сигнал не является постоянным – генерируется только при работе компонента. В таком случае привести компонент в начальное состояние возможно не всегда.

5.2.4. *Делители частоты*

Если возникает необходимость поделить частоту имеющегося тактового сигнала на величину, равную степени 2, то это делается достаточно просто с использованием счетчика. Счетчик, работающий от тактового сигнала, изменяет свои биты с уменьшенной частотой в следующем порядке: самый младший бит (нулевой) делит частоту вдвое, первый бит – на 4, второй – на 8 и т. д.

Приведем пример модуля, делящего частоту на 8 и 32 (на выходе модуля имеется два тактовых сигнала):

```
  entity ClkDiv is
      Port (
      clk      : in std_logic;
      reset    : in std_logic;
      clk_div8 : out std_logic;
      clk_div32  : out std_logic);
```

```
end ClkDiv;

architecture ClkDiv of ClkDiv is

signal Counter : std_logic_vector(4 downto 0);

begin

  process(clk, reset)
  begin
    if(reset = '1') then
      Counter    <= (others => '0');
    elsif(rising_edge(clk)) then
      Counter    <= Counter + '1';
    end if;
  end process;

  clk_div8 <= Counter(2);
  clk_div32  <= Counter(4);

end ClkDiv;
```

Из нового необходимо отметить использование конструкции **(others => '0')**. Такая запись позволяет приравнять все биты сигнала заданному значению, вне зависимости от их количества.

Наличия сигнала **reset** в модулях является хорошим тоном, и в данном пособии автор настоятельно рекомендует использовать его во всех модулях и для всех заведенных сигналов. Это обеспечит возможность привести модуль в исходное состояние в любой момент времени.

5.3. Использование существующих модулей

Язык VHDL по своей сути похож на языки структурного программирования, так как позволяет разделять программу на модули, и использовать существующие. Помимо модулей, создаваемых самим проектировщиком, кристаллы ПЛИС имеют ряд компонентов, которые не находят прямого отражения в средствах языка, присутствуя при этом физически на кристаллах программируемой логики.

Для использования существующего модуля его, во-первых, необходимо включить в проект. Для этого надо добавить существующий файл ***vhd***, в котором он описан и реализован. Рассмотрим для примера использование двух описанных ранее модулей: делителя частоты и дешифратора. Проект должен уменьшить внешнюю тактовую частоту в 8 раз и на этой частоте запустить дешифратор.

Исходные данные задачи: имеются 2 vhdl-файла: MyDecoder.vhd и ClkDiv.vhd, с реализованными внутри моделями, описанными выше (применен синхронный шифратор). Верхний модуль в иерархии проекта обычно называют ***root***. Для использования существующие модули необходимо объявить в описании архитектуры модуля (до слова ***begin***). Объявление происходит с использованием ключевого слова **component**.

После объявления компонента необходимо указать, каким именно образом он используется. Для этого создается экземпляр компонента с уникальным именем CurClkDiv:ClkDiv port map(…);. Допускается создавать сколько угодно экземпляров компонента с различными именами экземпляров. В скобках через запятую указывается, каким образом соединяются интерфейсные сигналы. В ситуациях, когда выходные сигналы не нужны, употребляется ключевое слово **open**. В случаях, когда входные сигналы являются незначимыми, употребляется символ **'-'**, имеющий тип std_logic.

После правильного использования компонентов в окне исходных файлов проекта файлы выстраиваются в соответствии с реализованной иерархией (рис. 25).

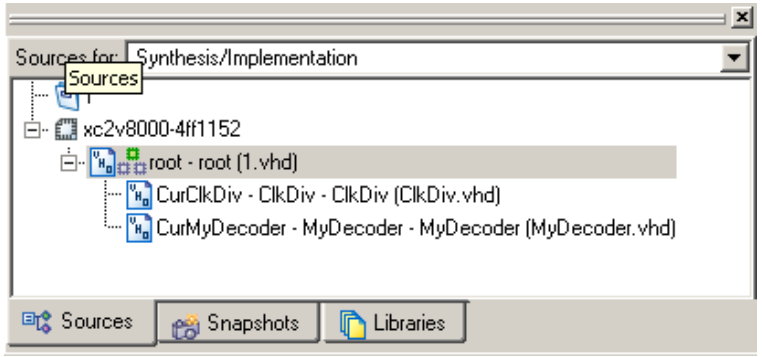

Рис. 25. Иерархия модулей в проекте

Ниже приведен код для реализации обозначенного проекта:

```
entity root is
    Port (
    clk    : in std_logic;
    reset  : in std_logic;
    A      : in std_logic_vector(1 downto 0);
    D      : out std_logic_vector(3 downto 0));
end root;
architecture root of root is

component ClkDiv is
    Port (
    clk      : in std_logic;
    reset    : in std_logic;
    clk_div8   : out std_logic;
    clk_div32    : out std_logic);
end component;

component MyDecoder is
    Port (
    clk  : in std_logic;
    E    : in std_logic;
    A    : in std_logic_vector(1 downto 0);
    D    : out std_logic_vector(3 downto 0));
end component;

signal clk_int : std_logic;

begin

CurClkDiv : ClkDiv
  Port map(  clk        => clk,
             reset      => reset,
             clk_div8   => clk_int,
             clk_div32    => open);

CurMyDecoder : MyDecoder
```



```
  Port map(  clk => clk_int,
             E   => '1',
             A   => A,
             D   => D);
end root;
```

5.3.1. *Память*

В современных кристаллах программируемой логики имеются блоки встроенной синхронной статической памяти. Объем и размер блоков зависит от конкретного кристалла. Для определенности далее будем рассматривать кристалл FPGA Xilinx XC3S500E, используемый в учебном аппаратном решении SLED.

Блоки памяти в Xilinx имеют двухпортовую независимую архитектуру. Это позволяет независимо записывать и вычитывать данные, причем возможно применение различных тактовых сигналов. Объем таких блоков памяти для XC3S500E составляет 18 Кб, общее количество памяти на кристалле 360 Кб (20 блоков памяти).

Особенностью блоков памяти RAMB (RAM Blocks) Xilinx является возможность ее реконфигурирования с учетом требований разрядности данных. Так, память RAMB кристалла XC3S500E позволяет использовать следующие конфигурации:

- ***RAMB16_S36*** – 9 бит адрес, 32 бита данные, 4 бита четности;
- ***RAMB16_S18*** – 10 бит адрес, 16 бит данные, 2 бита четности;
- ***RAMB16_S9*** – 11 бит адрес, 8 бит данные, 1 бит четности;
- ***RAMB16_S4*** – 12 бит адрес, 4 бита данные;
- ***RAMB16_S2*** – 13 бит адрес, 2 бита данные;
- ***RAMB16_S1*** – 14 бит адрес, 1 бит данные.

В случае конфигурирования памяти в двухпортовом режиме возможен как симметричный вариант (RAMB16_S9_ S9 и т. п.), так и режим различной разрядности в любых сочетаниях (по правилам сначала указывается меньшая разрядность: RAMB16_S2_ S36).

Интерфейс памяти имеет следующие сигналы:

- ***CLK*** – тактовый сигнал работы памяти;
- ***EN*** – сигнал разрешения чтения и записи;
- ***ADDR*** – адрес ячейки памяти;

- ***DI*** – входные данные;
- ***DIP*** – входные биты четности;
- ***WE*** – сигнал разрешения записи данных;
- ***SSR*** – сигнал выдачи заданной константы;
- ***DO*** – выходные данные;
- ***DOP*** – выходные биты четности.

В конце названий прибавляется название порта – A или B (***DIPA, SSRB***).

На рис. 26 приведен интерфейс блока двухпортовой памяти.

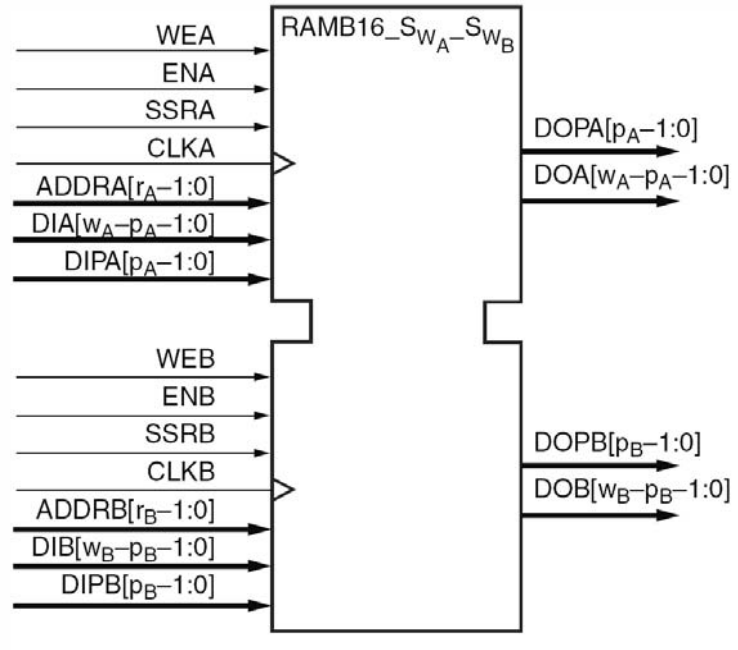

Рис. 26. Интерфейс двухпортовой памяти RAMB

Так как память является синхронной, то все операции чтения и записи происходят по восходящему фронту тактового сигнала. При записи данных на одном такте необходимо выставить адрес, данные и сигнал разрешения записи. При чтении данные появляются на следующем такте после выставления адреса ячейки памяти.

5.3.2. *Умножители*

Реализация умножителей на уровне регистров является достаточно трудоемкой задачей, но при этом нередко возникает при реализации самых разных алгоритмов обработки данных. В современных кристаллах FPGA реализованы примитивы умножителей, способных перемножать два 18-битных числа. Результат разрядностью 36 бит появляется на следующем такте.

Количество таких однотактных умножителей обычно совпадает с количеством блоков памяти RAMB и для кристалла XC3S500E составляет 20 штук.

Компонент умножителя имеет описанный ниже интерфейс:

```
component MULT18X18SIO is
  generic (
      B_INPUT  : string;
      AREG   : integer;
      BREG   : integer;
      PREG   : integer);
  port (
      CLK    : in std_logic;
      A    : in std_logic_vector(17 downto 0);
      B    : in std_logic_vector(17 downto 0);
      P    : out std_logic_vector(35 downto 0);
      BCOUT  :  out  std_logic_vector(17  downto
0);
      BCIN   :  in  std_logic_vector(17  downto
0);
      CEA    : in std_logic;
      CEB    : in std_logic;
      CEP    : in std_logic;
      RSTA : in std_logic;
      RSTB : in std_logic;
      RSTP : in std_logic
   );
end component;
```

В блоке generic описываются настройки конкретного объекта:

- ***AREG, BREG, PREG*** – разрешают входные регистры на портах, принимаемые значения 0 либо 1;

- ***B_INPUT*** – использование умножителя: DIRECT (прямое включение) либо CASCADE (каскадирование умножителей).

Таким образом, предусмотрено использование умножителей путем их каскадирования. Для этого в интерфейсе присутствуют сигналы **BCOUT** и **BCIN**, соединяемые в случае каскадирования. При этом на всех, кроме конечного умножителя, устанавливается атрибут **B_INPUT => "CASCADE"**, а на последнем – **B_INPUT => "DIRECT"**.

Сигналы **CE** разрешают динамически работу регистров на указанном порту, а **RST** производит перегрузку порта.

5.3.3. *DCM*

Блоки DCM (Digital Clock Manager) позволяют управлять тактовыми сигналами кристаллов следующим образом:

- умножение и деление частоты;
- синхронизация по фазе с заданным тактовым сигналом;
- сдвиг фазы тактового сигнала на указанную величину;
- получение сигналов, сдвинутых на 90, 180 и 270 градусов по фазе.

При помощи блоков DCM можно получить тактовый сигнал нужной частоты. У блоков DCM есть один основной вход CLKIN и два основных выхода: CLKDV и CLKFX. Выход CLKDV предназначен для деления частоты. В зависимости от установленного параметра CLKDV_DIVIDE, частота на этом выходе может быть равна входящей частоте, делённой на 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 7.5, 8, 9, 10, 11, 12, 13, 14, 15 или 16. На выход CLKFX подаётся синтезированный тактовый сигнал, частоты которого можно задавать в гораздо большем диапазоне, чем у CLKDV, однако синтезированный сигнал отличается несколько меньшей стабильностью (явление так называемого джиттера). Частота сигнала на выходе CLKFX задаётся двумя параметрами CLKFX_MULTIPLY и CLKFX_DIVIDE, и равна f_{CLKIN} * CLKFX_MULTIPLY / CLKFX_DIVIDE. CLKFX_MULTIPLY может быть любым целым числом в диапазоне от 2 до 32 включительно, CLKFX_DIVIDE – от 1 до 32.

На рис. 27 показана функциональная схема блоков и сигналов DCM.

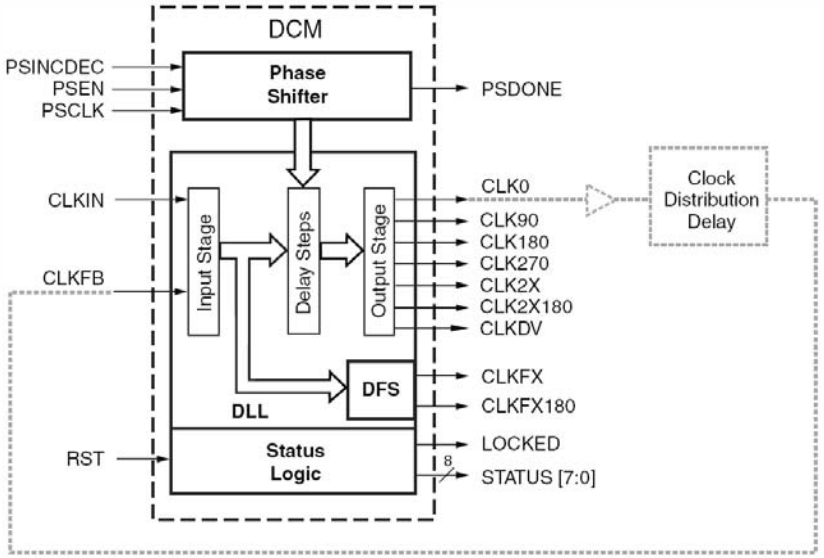

Рис. 27. Функциональная схема и сигналы DCM

Блоки DCM можно соединять последовательно друг с другом для получения недоступных для одного DCM частот.

Автоматически сгенерировать VHDL-описание каскада из двух DCM вместе с цепями автоматического их перезапуска в случае потери синхронизации можно при помощи программы CORE Generator, входящей в пакет Xilinx ISE.

5.3.4. *Примитив FDDRRSE*

При необходимости вывести данные из ПЛИС на удвоенной частоте существует два основных подхода:

- умножить внутри ПЛИС тактовый сигнал вдвое и вывести на его частоте данные;
- увеличить внутри ПЛИС ширину данных вдвое и, используя специализированные примитивы, вывести сигнал наружу на двойной частоте.

Учитывая особенности архитектуры ПЛИС, более эффективным в такой задаче является второй подход. Это объясняется тем, что сама архитектура создана для всевозможного распараллеливания, а

вот умножения частоты нередко невозможно, поскольку кристалл практически всегда работает на максимально возможной частоте.

Примитив FDDRRSE представляет собой триггер, преобразующий пару входных битов в однобитовый сигнал двойной частоты. Использование этого компонента необходимо в тех случаях, когда нужно выдать из ПЛИС данные на двойной частоте. Примером является стандарт DDR.

Схемотехнический вид компонента приведен на рис. 28.

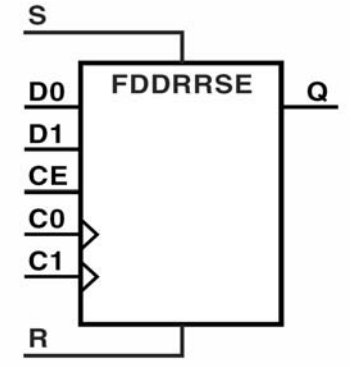

Рис. 28. Компонент FDDRRSE

Как видно, компонент имеет два порта входных данных (***D0***, ***D1***), два тактовых сигнала для этих данных (***C0***, ***C1***), а также сигнал разрешения работы CE. Помимо этого, имеются синхронные сигналы ***S*** (Set) и ***R*** (Reset).

Ниже на рис. 29 приведена схема использования такого элемента при выводе данных на двойной частоте.

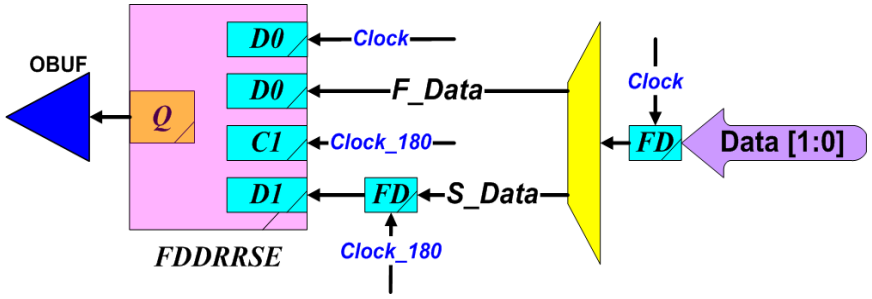

Рис. 29. Вывод данных на двойной частоте

Здесь используются два тактовых сигнала, сдвинутые относительно друг друга на половину фазы – 180 градусов.

Узким местом в такой реализации вывода данных является путь между двумя триггерами, составляющий всего половину периода тактового сигнала. Но так как никаких промежуточных действий с этим сигналом не происходит (нет асинхронной логики), то это является допустимым подходом.

5.3.5. *IOBUF, IBUFG*

При необходимости завести внешний сигнал в ПЛИС или вывести наружу существует стандартный путь решения, когда коммутация происходит напрямую. При этом при разводке проекта в эти места автоматически вставляются примитивы ***OBUF***, ***IBUF*** или ***IBUFG***.

Однако в ряде случаев возникает необходимость явно указывать тип ввода или вывода. Это необходимо, например, для явного указания стандартов выхода (уровень логики, время нарастания сигнала и т. п.). В качестве примера можно привести следующие типы: ***PCIX***, ***LVTTL***, ***SSTL2_II_DCI***, ***LVDS_25*** и т. д.

IBUFG используется в случае, если сигнал необходимо развести по глобальным путям внутри кристалла. Это является обязательным требованием при разводке тактового сигнала.

К сожалению, в некоторых версиях разводчиков по умолчанию тактовые сигналы, например с выхода компонента ***DCM***, могут быть разведены по обычным путям данных. В результате элементы, которые должны работать на одном фронте, начинают работать со сдвигом по времени. Как следствие, это может привести к выходу из строя работоспособности всей схемы, поведение которой к тому же будет зависеть от разводки к разводке. Поэтому настоятельно рекомендуется всегда применять глобальные буферы в местах, где программист это предусматривает, не полагаясь на программные приложения оптимизации и разводки.

Ниже показан пример VHDL-кода по использованию DCM. Пример является рабочим и используется в существующих проектах.

```
entity Clock_DCM is
  port (
       Ext_Clock  : in std_logic;
```

```
        Ext_Reset  : in std_logic;
        Clock      : out std_logic;
        Reset      : out std_logic);
  end Clock_DCM;

  architecture Clock_DCM of Clock_DCM is
  component DCM
    port (
        CLKIN  : in STD_ULOGIC;
        RST    : in STD_ULOGIC;
        CLKFB  : in STD_ULOGIC;
        CLK0   : out STD_ULOGIC;
        LOCKED : out STD_ULOGIC);
  end component;

  attribute CLK_FEEDBACK      : string;
  attribute DLL_FREQUENCY_MODE  : string;
  attribute CLKOUT_PHASE_SHIFT  : string;
  attribute PHASE_SHIFT       : real;
  attribute FACTORY_JF        : string;
  attribute STARTUP_WAIT      : string;
attribute CLK_FEEDBACK of MyDCM : label is "1X";
attribute DLL_FREQUENCY_MODE of MyDCM :
                                label is "LOW";
attribute CLKOUT_PHASE_SHIFT of MyDCM :
                                label         is
                               "FIXED";
attribute PHASE_SHIFT of MyDCM  : label is 64.0;
attribute FACTORY_JF of MyDCM   : label       is
"FFFF";
attribute STARTUP_WAIT of MyDCM :    label    is
"TRUE";
  signal MyDCM_Locked   : std_logic;
  signal MyDCM_Reset      : std_logic;

  type MyDCM_State_STATE_TYPE
        is (MyDCM_SetReset,
            MyDCM_Wait,
            MyDCM_SeeForLock);
  signal MyDCM_State : MyDCM_State_STATE_TYPE;
```



```
signal MyDCM_Counter  :       STD_LOGIC_VECTOR(15
downto 0);

  signal MyDCM_Clock_FromDLL  : std_logic;
  signal MyDCM_Clock        : std_logic;

  --
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$
  signal Ext_Clock_Int      : std_logic;
  signal Ext_Clock_G        : std_logic;
  signal Ext_Reset_Int      : std_logic;

  --
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$
  begin

  ExtClock_IBUFG: IBUFG_LVCMOS33
    port map(
      I => Ext_Clock,
      O => Ext_Clock_Int );
  ExtClock_BufGMUX : BUFGMUX
  port map ( O => Ext_Clock_G,
             I0=> Ext_Clock_Int,
             I1=> '-',
             S => '0');
  Ext_RST_IBUF: IBUF
    port map(  I => Ext_Reset,
             O => Ext_Reset_Int );

  MyDCM: DCM
    port map(
      CLKIN    => Ext_Clock_G,
      RST      => MyDCM_Reset,
      CLKFB    => MyDCM_Clock,
      CLK0     => MyDCM_Clock_FromDLL,
      LOCKED   => MyDCM_Locked
      );
```

```
  MyDCM_Clock_BUFG: BUFGMUX
    port map(I0 => MyDCM_Clock_FromDLL,
             I1=> '-',
             S => '0',
             O => MyDCM_Clock);

  Clock <= MyDCM_Clock;

  --
||||||||||||||||||||||||||||||||||||||||||||||||||||
|

  process (Ext_Clock_G, Ext_Reset_Int) is
  begin
    if(Ext_Reset_Int = '1') then
      MyDCM_State  <= MyDCM_SetReset;

    elsif(rising_edge(Ext_Clock_G)) then

      case MyDCM_State is

        when MyDCM_SetReset =>
            MyDCM_State  <= MyDCM_Wait;

        when MyDCM_Wait =>
          if(MyDCM_Counter = X"FFFF") then
            MyDCM_State  <= MyDCM_SeeForLock;
          else
            MyDCM_State  <= MyDCM_Wait;
          end if;

        when MyDCM_SeeForLock =>
          if(MyDCM_Locked = '0') then
            MyDCM_State  <= MyDCM_SetReset;
          else
            MyDCM_State  <= MyDCM_SeeForLock;
          end if;

        when others =>
            MyDCM_State  <= MyDCM_SetReset;
```



```
      end case;
    end if;
  end process;

MyDCM_Reset  <=    '1'    when    MyDCM_State    =
MyDCM_SetReset
             else'0';

  process (Ext_Clock_G) is
  begin
    if(rising_edge(Ext_Clock_G)) then
      if(MyDCM_State = MyDCM_Wait) then
        MyDCM_Counter  <= MyDCM_Counter + '1';
      else
        MyDCM_Counter  <= (others => '0');
      end if;
    end if;
  end process;

  --|||                                          Reset
||||||||||||||||||||||||||||||||||||||||

  Reset    <= '1' when Ext_Reset_Int = '1' OR
                  MyDCM_Locked = '0' else
           '0';

  end Clock_DCM;
```

Отличительной особенностью приведенного примера использования DCM является наличие встроенного конечного автомата, контролирующего работу примитива DCM. В случае необходимости происходит его перегрузка.

5.3.6. *Оператор generate*

Оператор **generate** часто используется при необходимости создать несколько экземпляров одного компонента. В качестве примера можно привести работу с двунаправленной внешней шиной (данные на шине PCI или памяти): имеется двунаправленная внешняя 32-разрядная шина данных ***BUS_Data***. В дизайне

используется два 32-битных сигнала ***IData*** и ***OData*** и сигнал управления ***OE*** – разрешение выхода данных на шину.

Существующий примитив ***IOBUF*** в кристаллах FPGA Xilinx позволяет описать однобитную двунаправленную шину. В нашем случае необходимо использовать 32 таких примитива для описания нашей 32-разрядной шины. При этом сигнал ***OE*** является единым для всех битов. Схемотехнический вид примитива ***IOBUF*** приведен на рис. 30.

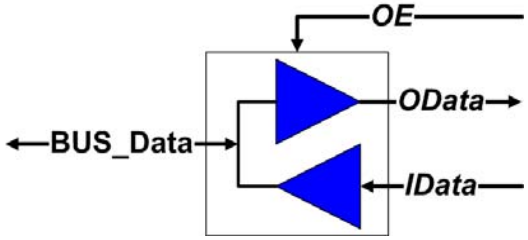

Рис. 30. Работа с двунаправленной шиной

Вот фрагмент кода, каким образом, используя оператор ***generate***, можно сократить код:

```
IOBs: for I in 0 to 31 generate
  IOB_I: IOBUF
  port map ( I => IData(I),
             T => OE,
             O => OData(I),
             IO=> BUS_Data(I));
end generate;
```

5.4. Последовательная передача данных RS-232

Интерфейс RS-232 является стандартным последовательным интерфейсом для ПК. Сопряжение устройства на базе ПЛИС с ПК, выполненное без дополнительных внешний устройств (за исключением преобразователей электрических уровней), предоставляет новые возможности как разработчику на этапе проектирования, так и пользователю созданного устройства. Кроме того, разработка и реализация контроллера RS-232 позволяет продемонстрировать довольно эффективный прием проектирования на VHDL – реализацию конечных автоматов для выполнения относительно сложных операций.

Стандарт RS-232 описывает электрический интерфейс для последовательной двунаправленной передачи данных между различным оборудованием. Стандарт был введён в 1962 г. Ассоциацией электронной промышленности (IAE) и изначально предназначался для подключения модемов к механическим телетайпам и компьютерным терминалам. В настоящее время широко используется для подключения различного оборудования к персональным компьютерам через последовательный порт.

Стандарт RS-232 определяет набор используемых сигналов, их функциональное назначение, электрические и временные характеристики.

Протокол RS-232 является асинхронным, другими словами, данные в нем передаются без сопровождения тактовым сигналом. Для передачи используются две сигнальные линии:

- ***TD*** (Transmit Data) – линия данных к устройству;
- ***RD*** (Receive Data)– линия данных к ПК.

Помимо этого, существуют следующие управляющие сигналы:

- ***RTS*** (Request To Send) – запрос на передачу данных от ПК;
- ***CTS*** (Clear To Send) – ответ на запрос от устройства.

Как видно, стандарт несимметричен: он предполагает, что передача данных от устройства к компьютеру возможна в любой момент, а обратная передача от компьютера к устройству будет возможна только после прохождения «рукопожатия» – процедуры установки связи с помощью сигналов ***RTS*** и ***CTS***. Подобная асимметрия была когда-то актуальна при обмене данными между модемом и компьютером. В ряде задач процедура установки связи может смело игнорироваться: устройство может полностью игнорировать запросы на установку связи от компьютера (игнорируется сигнал ***RTS***) и при этом непрерывно заявлять о готовности к приёму данных от компьютера (сигнал ***CTS*** всегда установлен в логический '0').

Для электрического сопряжения сигналов требуется использовать стандартные преобразователи уровней RS-232, лежащих в диапазоне от -12 до +12 В, в логические уровни TTL. После преобразования к выводам ПЛИС окажутся подключены два сигнала, которые чаще всего обозначают как **rx** и **tx**.

Протокол передачи данных по интерфейсу RS-232 выглядит следующим образом (рис. 31):

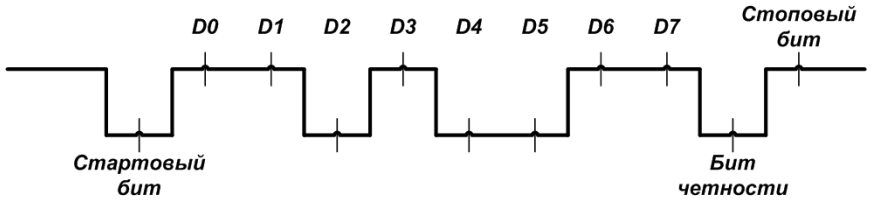

Рис. 31. Временная диаграмма передачи данных по протоколу RS-232

- передача начинается установкой низкого логического уровня на входе приемника. Низкий уровень сохраняется на время длительности одного бита, который называется стартовым;
- происходит передача битов данных, количество которых зависит от настроек протокола. Передача начинается с младшего бита;
- после передачи последнего бита данных вводится бит четности, формируемый по позитивной или негативной четности, в зависимости от настроек протокола. Также настройками протокола этот бит может быть отключен;
- в завершение передается стоповый бит высокого логического уровня, после которого линия остается в состоянии логической единицы вплоть до следующей посылки данных. В некоторых ситуациях допускается передача не одного, а двух стоповых битов.

Асинхронный протокол передачи накладывает достаточно жесткие требования на стабильность интервалов времени, в течение которых передаются отдельные биты. Для повышения надежности передачи желателен прием отдельных битов как можно ближе к середине этих интервалов. С той же целью можно использовать довольно эффективный прием многократного считывания состояния линии с последующим подсчетом количества принятых нулей и единиц.

При реализации приемника в ПЛИС необходимо в первую очередь решить проблему задания временных интервалов. Длительность отдельных бит может быть определена исходя из настроек протокола, а именно скоростью передачи данных.

5.5. Реализация алгоритмов фильтрации данных

Достаточно часто при обработке изображений необходимо произвести фильтрацию. Зачастую такие задачи требовательны как к временным, так и к аппаратным ресурсам. При этом сама идея фильтрации достаточно проста и легко реализуется на базе программируемой логики, позволяющей распараллеливать вычисления и использовать конвейеризацию.

Рассмотрим два основных фильтра: линейный (строчный) и пространственный (двумерный).

5.5.1. *Линейная фильтрация*

При предлагаемой реализации линейной фильтрации строки данных все коэффициенты фильтра, также как и входные данные, должны быть целочисленными. На выходе фильтра производится нормализация данных с заданным коэффициентом **Nr**.

Рассмотрим реализацию одномерного фильтра со следующими параметрами:

- разрядность входных данных – N бит;
- длина строки данных – A точек;
- длина фильтра – K точек.

На рис. 32 приведена блок-схема для реализации фильтра:

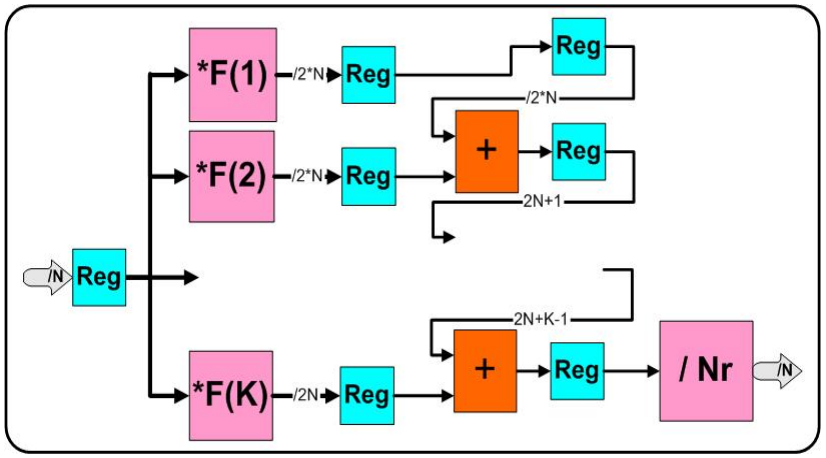

Рис. 32. Блок-схема линейного фильтра

Глубина конвейера равна (K + 2). Другими словами, фильтрованное значение точки появляется на выходе через указанное количество тактов. Так как других задержек фильтр не имеет, то фильтрация строки целиком произойдет за (A + K + 2) тактов.

В фильтре используются (K – 1) сумматор. Разрядность используемых сумматоров изменяется от (2N) до (2N + K – 1). В качестве умножителей используются встроенные блоки, позволяющие производить за один такт умножение двух целых чисел с разрядностью 18 бит. Для реализации нормализации используется табличный делитель, реализованный на основе внутренней памяти кристалла программируемой логики.

Таким образом, для реализации указанного конвейера необходимо:

- K умножителей и 1 нормализатор;
- (K – 1) сумматор различной разрядности от (2N) до (2N + K – 1);
- 1 регистр для хранения N-разрядного значения и K регистров для хранения 2N-разрядных значений;
- K регистров для хранения значений разрядности от 2N до (2N + K) бит.

Применимо к архитектуре семейства Virtex2 / Spartan3E для реализации указанного конвейера линейной фильтрации необходимо:

- K умножителей;
- (2N + K) / 10 RamB (18Kb) для реализации нормализатора;
- $\frac{2K^2 + 12NK - 3N - K}{8}$ Slices.

Пример

Фильтр размером 20 точек для строки длиной 2000 точек, 10 бит. Применимо к архитектуре Virtex2xc6000, для реализации необходимо:

- 20 умножителей – 100 %;
- 4 RamB – 20 %;
- 344 Slices – 7,4 %.

Время фильтрации составляет около 20 мкс (при частоте 100 МГц). Производительность конвейера составляет 1,011 тактов на пиксель.

5.5.2. *Двумерная фильтрация*

При предлагаемой реализации двумерной фильтрации строки данных, все коэффициенты фильтра, также как и входные данные, должны быть целочисленными. На выходе фильтра производится нормализация данных с заданным коэффициентом **Nr**.

Рассмотрим реализацию одномерного фильтра со следующими параметрами:

- разрядность входных данных – N бит;
- длина строки данных – A точек;
- количество строк данных – B точек;
- длина фильтра – K точек;
- ширина фильтра – L точек.

Основная идея алгоритма заключается в том, что осуществляется фильтрация текущей строки изображения по всем строкам фильтра одновременно. При этом запоминается в памяти ненормализованный результат. Таким образом, в памяти необходимо хранить (L – 1) строк данных, разрядностью (2N + K) бит.

На рис. 33 приведена блок-схема для реализации конвейера фильтра с указанными параметрами.

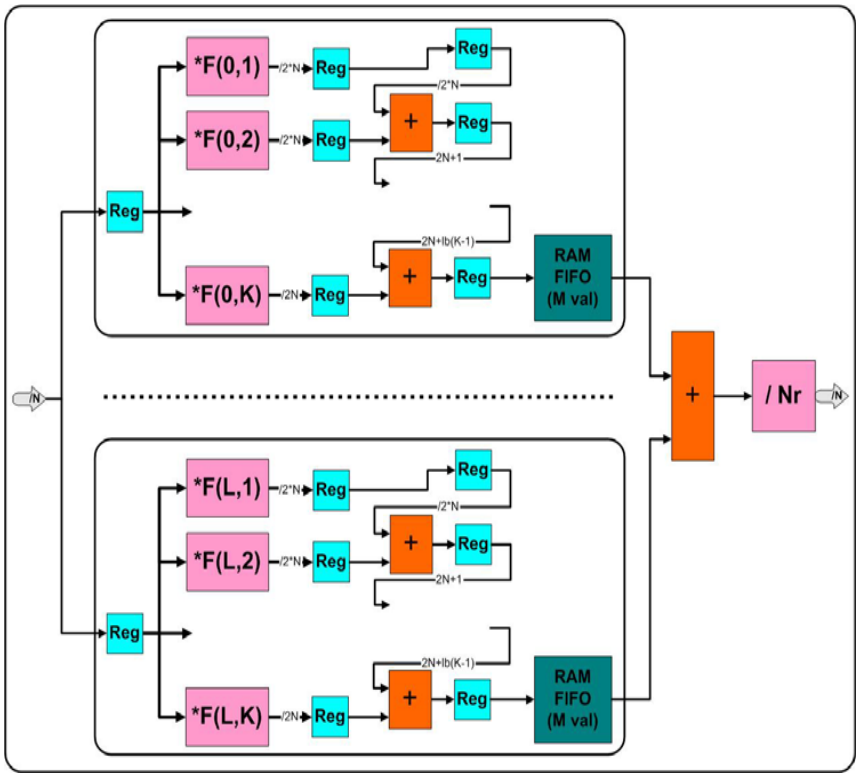

Рис. 33. Блок-схема двумерного фильтра

Глубина конвейера равна (AL + K + 2). Целиком фильтрация строки произойдет за (AB + AL + K + 2) тактов.

Применимо к архитектуре семейства Virtex2, для реализации указанного конвейера линейной фильтрации необходимо:

- (L*K) умножителей;
- L(M(2N + K) / 18432) RamB (18 Kb) для хранения результатов фильтрации строк;
- (2N + K) / 10 RamB (18 Kb) для реализации нормализатора;
- $L * \frac{2K^2 + 12NK - 3N - K}{8}$ Slices.

Пример

Реализация фильтра размером 20 точек на 10 строк для изображения размером 2000 x 2000 точек, 10 бит. Применимо к архитектуре Virtex2xc6000 необходимо:

- 100 умножителей – 70 %;
- 83 RamB; – 58 %;
- 1800 Slices – 5,4 %.

Время фильтрации составляет около 40,3 мс (при частоте 100 МГц). Производительность конвейера составляет 1,005 тактов на пиксель.

5.6. Простой программируемый контроллер

С помощью конечных автоматов можно решать достаточно большой класс задач. Однако необходимость каждый раз описывать поведение конечного автомата на языке VHDL является затратным по времени занятием. Поэтому часто используется метод, при котором используется программируемый конечный автомат по работе с внешней памятью, в которой можно хранить последовательность переходов из состояний и условия этих переходов. Фактически использование такого приема превращает ПЛИС в специализированный процессор.

Приведем пример разработки программируемого конечного автомата. Для того чтобы не перегружать демонстрационный пример лишней информацией, ограничимся следующими переменными состояния:

- ***ip*** – счетчик команд;
- ***r_clk*** – регистр тактового сигнала;
- ***r_data*** – регистр данных.

Таким образом, эти два однобитных регистра могут формировать какой-либо последовательный программный протокол обмена.

Система команд автомата должна содержать следующие операции:

- ***0*** – нет операции;
- ***1*** – загрузка нуля в ***r_clk***;
- ***2*** – загрузка единицы в ***r_clk***;
- ***3*** – загрузка нуля в ***r_data***;
- ***4*** – загрузка единицы в ***r_ data***;

- ***5*** – ожидание логической единицы на входе ***d_read***;
- ***128-255*** – переход по адресу, заданному младшими семью битами.

```
entity fsm is
    Port (
     clk    : in std_logic;
     reset  : in std_logic;
     cmd    : in std_logic_vector(7 downto 0);
     ip     :  inout  std_logic_vector(7  downto
0);
     r_clk  : inout std_logic;
     r_data : inout std_logic;
     d_read : in std_logic);
end fsm;

architecture fsm of fsm is

begin

  process(clk, reset)
  begin
    if(reset = '1') then
      ip<= X"00";
    elsif(rising_edge(clk)) then
      case cmd is
        when X"00" =>
          ip <= ip + '1';
        when X"01" => r_clk <= '0';
          ip <= ip + '1';
        when X"02" => r_clk <= '1';
          ip <= ip + '1';
        when X"03" => r_data  <= '0';
          ip <= ip + '1';
        when X"04" => r_data  <= '1';
          ip <= ip + '1';
        when X"05" =>
          if(d_read = '1') then
            ip <= ip + '1';
          end if;
```



```
        when others =>
          ip <= '0' & cmd(6 downto 0);
      end case;
    end if;
  end process;
end fsm;
```

Комбинируя приведенные команды, можно программно реализовать несложный протокол обмена, использующий одну линию данных и линию синхронизации. Одноразрядный вход **d_read** может быть использован для организации циклов ожидания готовности внешнего устройства. Причем алгоритм опроса состояния этой линии и цикл с предусловием реализованы аппаратно. Выполнение единственной операции приведет конечный автомат к повторному исполнению одной и той же команды, поскольку условием продолжения работы является появление логической единицы на контролируемом входе.

5.7. Арифметико-логическое устройство

Проектирование более сложного конечного автомата практически не отличается от описанного в прошлой главе. Однако процессоры, кроме загрузки и выгрузки значений из регистров и выполнения переходов, могут выполнять еще и математические операции над числами, хранящимися в регистрах. Для этой цели в состав процессоров вводится арифметическое логическое устройство (АЛУ).

Разработка АЛУ проще, чем это может показаться на первый взгляд. Простейшее АЛУ должно иметь входы для аргументов (***a*** и ***b***), вход для кода операции (***op***) и выход результата (***q***). Пример АЛУ с минимальным набором операций:

```
entity alu is
    Port (
     a    : in std_logic_vector(7 downto 0);
     b    : in std_logic_vector(7 downto 0);
     op   : in std_logic_vector(2 downto 0);
     q    : out std_logic_vector(7 downto 0));
end alu;
```

```
architecture alu of alu is
begin

  q <= (a)     when op = "000" else
       (b)     when op = "001" else
       (a + b)    when op = "010" else
       (a - b)    when op = "011" else
       (a AND b)  when op = "100" else
       (a OR b)   when op = "101" else
       (a XOR b)  when op = "110" else
       X"00";
end alu;
```

Необходимо отметить, что реализация процессоров и контроллеров на ПЛИС становится все более актуальной задачей. Многие фирмы, в том числе и Xilinx, предлагают разработанные специально для FPGA процессорные ядра MicroBlaze (32-разрядные) и PicoBlaze (8-разрядные).

Синтезируемые процессоры следует отличать от специализированных ресурсов, вводимых в составе ПЛИС.

6. Работа с динамической памятью

6.1. Виды памяти

Все запоминающие устройства (ЗУ) по способу доступа к хранимым данным делятся на три основные группы: адресные, последовательные и ассоциативные.

При адресном доступе код на адресном входе указывает на ячейку, с которой производится обмен. Все ячейки адресной памяти в момент обращения равнодоступны.

Адресные ЗУ делятся на RAM (Random Access Memory) и ROM (Read-Only Memory). RAM хранят данные, участвующие в обмене при исполнении текущей программы, которые могут быть изменены в произвольный момент времени. В ROM содержимое практически никогда не изменяется (за исключением специальных режимов записи данных).

6.1.1. *Статическая и динамическая память*

RAM делятся на статические и динамические. Статические ЗУ называются SRAM (Static RAM), а динамические – DRAM (Dynamic RAM).

В статической памяти элементы (ячейки) построены на различных вариантах триггеров – схем с двумя устойчивыми состояниями. После записи бита в такую ячейку она может пребывать в этом состоянии очень долго – необходимо только наличие питания. При обращении к микросхеме статической памяти на нее подается полный адрес, который при помощи внутреннего дешифратора преобразуется в сигналы выборки конкретных ячеек. Ячейки статической памяти имеют малое время срабатывания (единицы-десятки наносекунд), однако микросхемы на их основе имеют низкую удельную плотность данных (порядка единиц Мбит на корпус) и высокое энергопотребление.

В динамической памяти ячейки построены на основе областей с накоплением зарядов, занимающих гораздо меньшую площадь, нежели триггеры, и практически не потребляющих энергии при хранении. При записи бита в такую ячейку в ней формируется электрический заряд, который сохраняется в течение нескольких миллисекунд; для постоянного сохранения заряда ячейки необходимо регенерировать – перезаписывать содержимое для восстановления зарядов. Ячейки микросхем динамической памяти

организованы в виде прямоугольной (обычно - квадратной) матрицы; при обращении к микросхеме на ее входы вначале подается адрес строки матрицы, сопровождаемый сигналом RAS (Row Address Strobe – строб адреса строки), затем через некоторое время – адрес столбца, сопровождаемый сигналом CAS (Column Address Strobe – строб адреса столбца). При каждом обращении к ячейке регенерируют все ячейки выбранной строки, поэтому для полной регенерации матрицы достаточно перебрать адреса строк. Ячейки динамической памяти имеют большее время срабатывания (десятки-сотни наносекунд), но большую удельную плотность (порядка десятков Мбит на корпус) и меньшее энергопотребление.

Обычные виды SRAM и DRAM называют также асинхронными – потому, что установка адреса, подача управляющих сигналов и чтение / запись данных могут выполняться в произвольные моменты времени – необходимо только соблюдение временных соотношений между этими сигналами. В эти временные соотношения включены так называемые охранные интервалы, необходимые для стабилизации сигналов, которые не позволяют достичь теоретически возможного быстродействия памяти. Существуют также синхронные виды памяти, получающие внешний синхросигнал, к импульсам которого жестко привязаны моменты подачи адресов и обмена данными; помимо экономии времени на охранных интервалах, они позволяют более полно использовать внутреннюю конвейеризацию и блочный доступ.

SIMM – Single In-Line Memory Module – вид модулей оперативной памяти, имеющий 72 либо 30 контактов.

6.1.2. *SDRAM и DDR*

SDRAM (Synchronous DRAM – синхронная динамическая память) – память с синхронным доступом, работающая быстрее обычной асинхронной (FPM/EDO/BEDO). Помимо синхронного метода доступа, SDRAM использует внутреннее разделение массива памяти на два независимых банка, что позволяет совмещать выборку из одного банка с установкой адреса в другом банке. SDRAM также поддерживает блочный обмен. Основная выгода от использования SDRAM состоит в поддержке последовательного доступа в синхронном режиме, где не требуется дополнительных тактов ожидания. При случайном доступе SDRAM работает практически с той же скоростью, что и FPM/EDO.

DDR (Double Data Rate) представляет собой дальнейшее развитие технологий динамической памяти с целью увеличения пропускной способности. Стандарт DDR предусматривает передачу данных на двойной частоте тактового сигнала, т. е. передача данных осуществляется по обоим фронтам сигнала.

Стандартные модули памяти DDR, используемые в ПК, имеют ширину шины данных 64 бита. Для этого чаще всего используются либо 8 микросхем, либо 4. Каждая микросхема памяти является независимой, но команды и адреса раздаются одновременно на все микросхемы.

Пропускная способность модуля памяти DDR с шириной шины данных 64 бита, при тактовой частоте 100 МГц, составляет 1,49 ГБ/с.

6.2. Контроллер SDRAM памяти

В качестве наглядного примера далее будет рассматриваться память Samsung K4S511632, используемая в комплексе SLED.

В SLED установлен один чип SDRAM памяти Samsung K4S511632 объёмом 512 мегабит, работающий на частотах до 133 мегагерц. Он имеет 16-разрядную шину данных и 13-разрядную шину адреса. Память разбита на 4 банка, в каждом из которых по 2^{13} строк, в каждой строке 2^{10} 16-битных слов. Такая организация позволяет экономить ширину шины адреса. К тому же, имеется возможность отдавать команды какому-либо банку, не дожидаясь окончания выполнения команд на других, что сильно увеличивает пропускную способность памяти при произвольном доступе.

Помимо шины адреса (MA12…MA0 – здесь и далее названия сигналов даны по обозначениям на принципиальной схеме SLED), двунаправленной шины данных (MDQ15…MDQ0) и 2-битной шины выбора банка (BA1…BA0) есть линия подачи тактового сигнала MCLK и ещё три линии, посредством которых отдаются команды. Эти линии называются RAS (Row Address Strobe), CAS (Column Address Strobe) и WE (Write Enable), низкий уровень на них является активным.

6.2.1. *Инициализация*

Перед началом работы с памятью, нужно выполнить определённую стартовую последовательность, показанную на рисунке 36. Сначала нужно подать команду предварительной

зарядки выходных буферов для всех банков (precharge all), затем две команды автоматической регенерации данных (auto refresh) подряд. В конце стартовой последовательности нужно дать команду установки содержимого регистра режимов (mode register set), эта команда подаётся путём установки низкого (активного) уровня на RAS, CAS и WE одновременно с подачей нужного содержимого регистра на шины адреса и выбора банка. В регистре режимов хранятся настройки чипа памяти, и таким образом мы задаём все необходимые параметры (рис. 34).

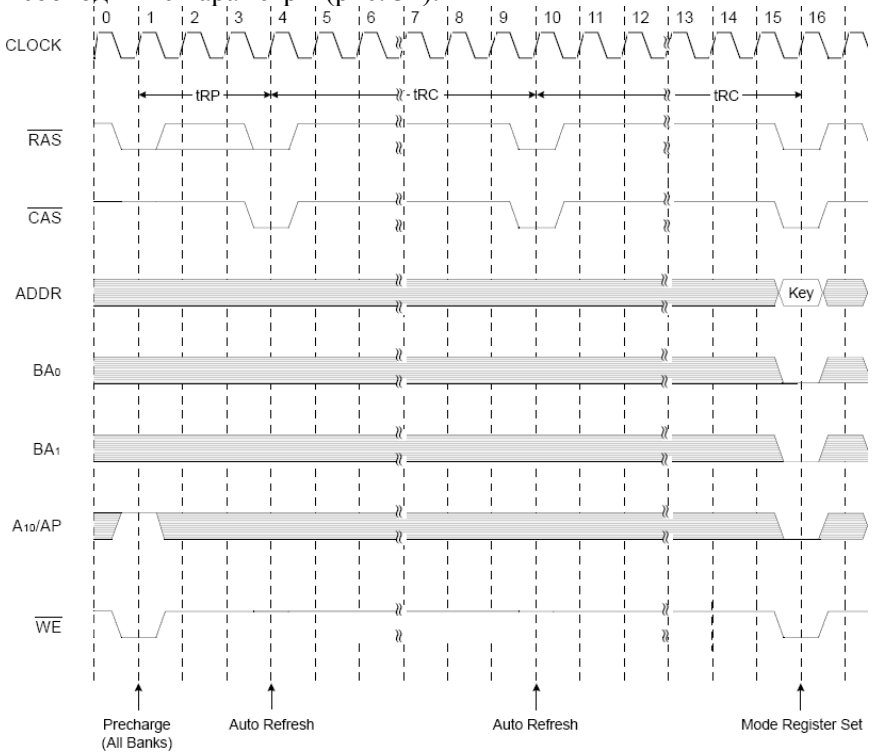

Рис. 34. Стартовая последовательность SDRAM

Структура регистра режимов описана в таблице 1. В нём задаются:

- Битами адреса MA2…MA0 – длина блока для блочных операций чтения/записи.
- Битом MA3 – последовательность слов для блочных операций чтения/записи, если выбрана длина блока 4 или 8 (для остальных длин доступен только последовательный режим).
- MA6…MA4 – длина задержки чтения (CAS Latency), она определяет, через сколько тактов после подачи команды чтения на выходах MDQ появятся данные. Значение 2 доступно только на частотах ниже 100 МГц.
- MA8…MA7 – тестовые режимы, биты должны быть установлены в логический 0.
- MA9 – операции записи могут быть отдельно объявлены как одиночные (НЕ блочные) установкой этого бита в 1.
- MA12…MA10, BA1..BA0 – эти биты зарезервированы для будущего использования и должны быть установлены в 0.

Address	BA0 ~ BA1	An ~ A10/AP	A9	A8	A7
Function	RFU	RFU	W.B.L	TM	

A6	A5	A4	A3	A2	A1	A0
CAS Latency			BT	Burst Length		

Test Mode			CAS Latency			
A8	A7	Type	A6	A5	A4	Latency
0	0	Mode Register Set	0	0	0	Reserved
0	1	Reserved	0	0	1	Reserved
1	0	Reserved	0	1	0	2
1	1	Reserved	0	1	1	3
Write Burst Length			1	0	0	Reserved
A9	Length		1	0	1	Reserved
0	Burst		1	1	0	Reserved
1	Single Bit		1	1	1	Reserved

Burst Type		Burst Length				
A_3	Type	A_2	A_1	A_0	BT = 0	BT = 1
0	Sequential	0	0	0	1	1
1	Interleave	0	0	1	2	2
		0	1	0	4	4
		0	1	1	8	8
		1	0	0	Reserved	Reserved
		1	0	1	Reserved	Reserved
		1	1	0	Reserved	Reserved
		1	1	1	Full Page	Reserved

Табл.1. Содержимое регистра режима SDRAM

6.2.2. *Активация и деактивация строки*

Чтобы производить операции чтения и/или записи с какой-либо строкой, её надо сначала активировать. Активация производится командой выбора строки – низкий уровень на линии RAS, одновременно с адресом строки на MA12…MA0 и номером банка, строку которого активируем на BA1 и BA0. В каждом банке может быть активировано не более одной строки. После выполнения всех необходимых операций или по истечении определённого времени строку нужно деактивировать. Деактивация производится командой precharge. Precharge бывает трёх типов:

- Precharge all – деактивирует все открытые строки во всех банках, подаётся низким уровнем на RAS и WE при высоком уровне на MA10.
- Auto precharge – подаётся вместе с последней операцией чтения или записи заданием высокого уровня на MA10, закрывает строку, на которой происходила операция. Если при подаче команды чтения или записи MA10 установлен в низкий уровень, автоматического закрытия строки по завершении команды не происходит.
- Precharge – закрывает активную строку на заданном банке, подаётся низким уровнем на RAS, WE и MA10 одновременно с номером банка на BA1 и BA0.

6.2.3. *Запись данных в память*

Команда записи подаётся низким уровнем на CAS и WE одновременно с адресом колонки, в которую производится запись, на шине адреса, номером банка на шине выбора банка и записываемыми данными на шине данных. Адрес колонки занимает только 10 младших бит шины адреса, остальные рекомендуется устанавливать в логический 0, за исключением линии MA10, которая при этом управляет автоматическим закрытием строки (см. предыдущий раздел).

Если память настроена на одиночную запись, то на этом процедура записи заканчивается, нет даже необходимости дожидаться окончания операции, как это нужно после большинства других команд. Можно, например, на следующем же такте произвести запись по другому адресу в той же строке.

Если же чип настроен на блочную запись, то в последующих за командой тактах должны следовать ещё данные. Общее количество данных должно соответствовать выбранной при инициализации памяти длине блока. Данные записываются в последовательно идущие адреса, задавать адрес для каждого элемента блока отдельно нельзя.

6.2.4. *Чтение данных из памяти*

Команда чтения подаётся низким уровнем на CAS одновременно с номером банка на BA1 и BA0 и адресом колонки на линиях MA9..MA0. На MA10 подаётся 1, если нужно закрыть текущую строку по завершении операции (auto precharge, см. раздел Активация и деактивация строки), или 0, в случае если этого делать не требуется. Линии MA12 и MA11 рекомендуется при этом устанавливать в 0. По прошествии CAS latency тактов (2 либо 3 в зависимости от настройки, см. раздел Стартовая последовательность) на MDQ15…MDQ0 появятся считанные данные. Если чип памяти настроен на блочные операции чтения (см. раздел Стартовая последовательность), то данные будут считываться по последовательным адресам и выдаваться каждый такт, пока не будет считан весь блок. Поскольку SDRAM имеет конвейерную архитектуру, в случае одиночного (не блочного) чтения подавать команды можно каждый такт, не дожидаясь появления данных для каждой отдельной команды.

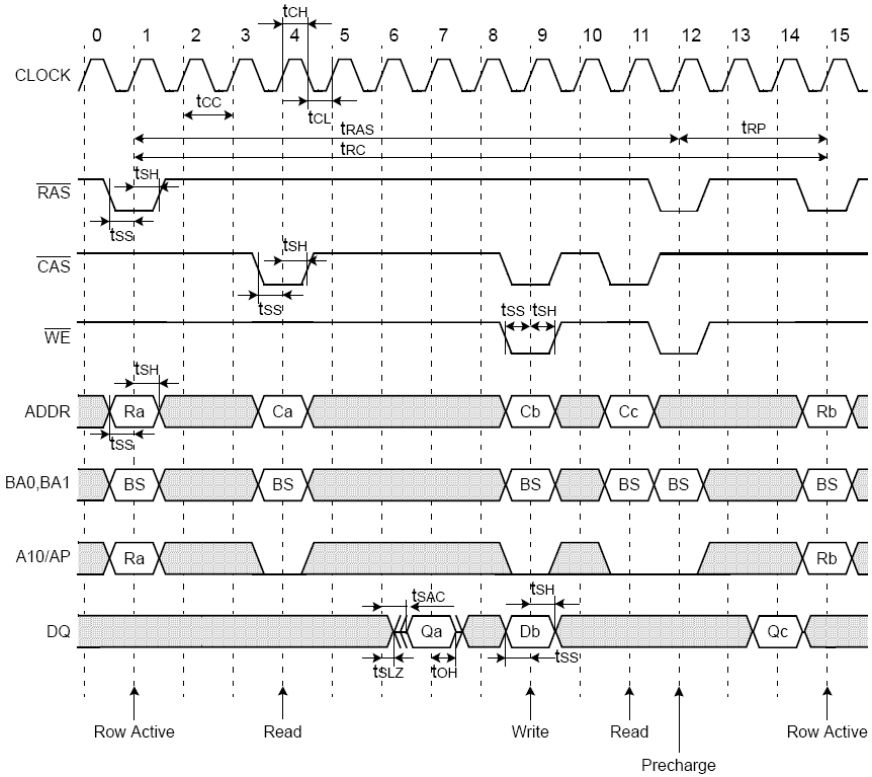

Рис. 35. Цикл чтения-записи-чтения (CAS Latency=3, Burst Length=1)

На рис. 35 для примера приведена временная диаграмма серии операций чтения и записи для одиночного случая, на рис. 36 – для блочного.

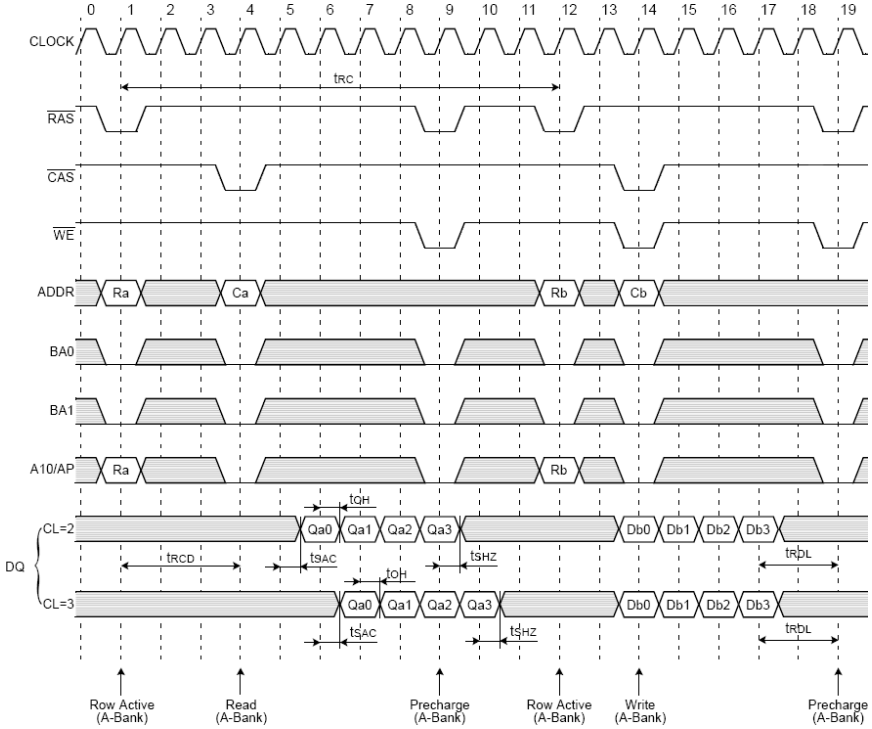

Рис. 36. Цикл чтения-записи-чтения на одном банке (Burst Length=4, tRDL=2CLK)

6.2.5. *Обновление данных в памяти*

Память SDRAM является энергозависимой, и у данных в ней есть конечный срок жизни, составляющий для установленного в SLED (как и для многих других) чипа 64 миллисекунды. Однако, при каждом обращении к строке, происходит обновление данных в ней, что позволяет, при некоторых дополнительных накладных расходах, хранить данные неопределённо долгий срок. Для упрощения этой процедуры, в современных чипах памяти существует механизм автоматического обновления. Благодаря наличию внутри чипа специального счётчика обновлений, достаточно лишь периодически подавать команду auto refresh и дожидаться окончания её выполнения, чип сам определит следующую строку в очереди на обновление, и восстановит данные в ней. При этом обновление будет происходить одновременно в четырёх строках (по одной в каждом банке).

Команда auto refresh подаётся низким уровнем на линии RAS и CAS. Чтобы все данные успели обновиться, эту команду нужно подавать не реже, чем 2^{13} (= 8192, по количеству строк в банке) раз за каждые 64 миллисекунды. Обновление можно производить как пакетно (8192 команды auto refresh подряд, раз в 64ms), так и по одной строке (по одной команде auto refresh в среднем раз в 64ms/8192 = 7.8 us), в зависимости от того, что удобнее в конкретном устройстве.

Отдельно следует рассмотреть случаи, когда всё содержимое памяти периодически полностью вычитывается или перезаписывается. Если это происходит чаще, чем ~16 раз в секунду (1 / 64 ms = 15.625 Гц), отдельного цикла обновления не требуется. К примеру, это условие выполняется, если SDRAM используется исключительно в качестве видеопамяти, в этом случае все используемые ячейки будут вычитываться не реже, чем с частотой смены кадров на мониторе, которая обычно существенно больше 16 Гц.

6.2.6. *Временные параметры*

Чтобы чип памяти успевал корректно обрабатывать все подаваемые команды, нужно соблюдать ряд временных ограничений, которые приведены в таблице 2.

t_{RRD}(min)	15 ns	Row active to row active delay. Минимальное время между двумя командами активации строки.
t_{RCD}(min)	20 ns	RAS to CAS delay. Минимальное время между командой активации строки и командой чтения или записи.
t_{RP}(min)	20 ns	Row precharge time. Время предварительной зарядки строки – промежуток, который нужно выждать после подачи команды precharge или precharge all, перед тем как подавать другие команды.
t_{RAS}(min)	45 ns	Minimum row active time. Минимальное время, которое может быть открыта строка, то есть промежуток от команды активации строки, до команды precharge или precharge all на ней.
t_{RAS}(max)	100 us	Maximum row active time. Максимально возможное время, которое может быть открыта строка. Даже если по истечении этого времени с момента активации строки всё ещё требуется производить операции чтения/записи на ней, строку нужно закрыть, а затем открыть снова.
t_{RC}(min)	65 ns	Row cycle time. Время цикла строки, равняется $t_{RAS} + t_{RP}$ и характеризует то, насколько быстро можно произвести точечное чтение (или точечную запись) из случайного места памяти.
t_{RDL}(min)	2 такта	Last data in to row precharge. Сколько должно пройти времени от подачи данных на вход до команды деактивации строки. При работе памяти на частотах 100 МГц и ниже, t_{RDL}(min) = 1 такт.
t_{MRS}(min)	2 такта	Mode register set time. Столько нужно выждать после команды mode register set, прежде чем подавать дугие команды.

Табл.2. Временные ограничения при работе с SDRAM

Естественно, все паузы между командами на практике можно выдерживать только в целых значениях периода тактового сигнала, подаваемого на чип памяти. Чтобы перевести приведённые в таблице значения в такты, нужно поделить их на период тактового сигнала и округлить полученное значение вверх. Тактовая частота установленного в SLED тактового генератора составляет 40 МГц, что соответствует периоду 25 наносекунд. При желании, с помощью блоков DCM, имеющихся внутри ПЛИС, можно преобразовать тактовый сигнал к другой частоте (даже к большей, чем исходная). Чип памяти, установленный в SLED, может работать на частотах до 133 МГц.

6.2.7. *Замечания по реализации контроллеров*

- Шина данных (MDQ15…MDQ0) является двунаправленной, т.е. данные на неё могут подаваться как со стороны ПЛИС, так и со стороны чипа памяти. Если на какую-либо линию одновременно подаётся с одной стороны высокий уровень, а с другой низкий, это приводит к короткому замыканию и протеканию по внутренним цепям микросхем недопустимо больших токов и даже может привести к их перегоранию. Чтобы этого избежать, нужно

подключать внутреннюю логику ПЛИС к шине данных через специальные примитивы IOBUF. К этому примитиву нужно подключить четыре линии: IO (её нужно подключить непосредственно к пину ПЛИС, выходящему на двунаправленную шину), I и O (на I нужно подавать данные для записи, а с O считывать данные при чтении) и T (переключает пин в состояние входа (T = 1) или выхода (T = 0)). Нужно внимательно следить за тем, когда шина работает на вход, а когда на выход и переключать вход T соответствующим образом. Вообще, во избежание проблем, рекомендуется как можно больше времени держать линии шины данных в режиме входа (T = 1) и переключать их в состояние выхода только тогда, когда это действительно нужно.

- Чип памяти захватывает значения данных, команд, адреса и пр. по восходящему фронту тактового сигнала. Поскольку тактовый сигнал не идеален, и его фронты могут сдвигаться во времени, система будет работать надёжнее всего в случае, если данные (и пр.) будут оставаться неизменными как можно большее время до и после восходящего фронта. Для этого можно подавать на чип памяти тот же тактовый сигнал, по которому данные изменяются внутри ПЛИС, но инвертированным, таким образом на стороне чипа памяти изменения данных будут происходить в районе нисходящего фронта тактового сигнала, что и требовалось.
- Поскольку прохождение через внутренние элементы ПЛИС вносит задержку в проходящий сигнал, рекомендуется проектировать контроллер памяти таким образом, чтобы команды, данные и т.п. выводились из кристалла ПЛИС через асинхронную логику (то есть через D-триггер, по фронту тактового сигнала). В этом случае среда разработки расположит выходные триггеры непосредственно около пинов микросхемы, и сигнал будет выходить наружу без задержки. Чтобы тактовый сигнал также выходил наружу синхронно с данными, его нужно выводить наружу через примитив **DDRRSE**.

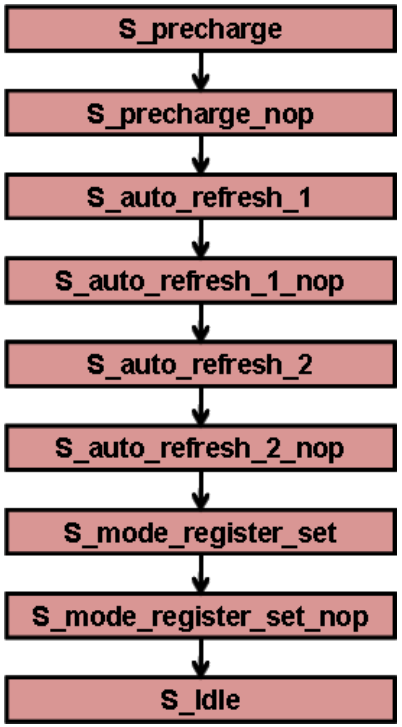

*Рис. 37*онечный автомат, отвечающий за стартовую последовательность

- Контроллер памяти удобнее всего реализовать на основе двух конечных автоматов, примерная структура которых изображена на рис. 37 и рис. 38. Первый автомат, изображённый на рисунке 37, отвечает за стартовую последовательность. Второй, на рисунке 38, начинает работу после перехода первого в состояние S_idle и представляет основной цикл работы памяти. Состояния с именами, заканчивающимися на **_nop**, представляют паузы необходимые после подачи команд, в каждом из этих состояний конечный автомат должен находиться соответствующее количество тактов. Кроме того, при реализации необходим счётчик, который будет отсчитывать, когда нужно подать команду auto refresh, и ещё один, считающий, сколько таких команд было пропущено по причине выполнения других команд. После попадания в состояние ожидания, контроллер должен автоматически выполнять всё пропущенное обновление. Систему следует проектировать таким

образом, чтобы в любой момент времени число пропущенных команд автообновления не превышало 2^{13} (8 192).

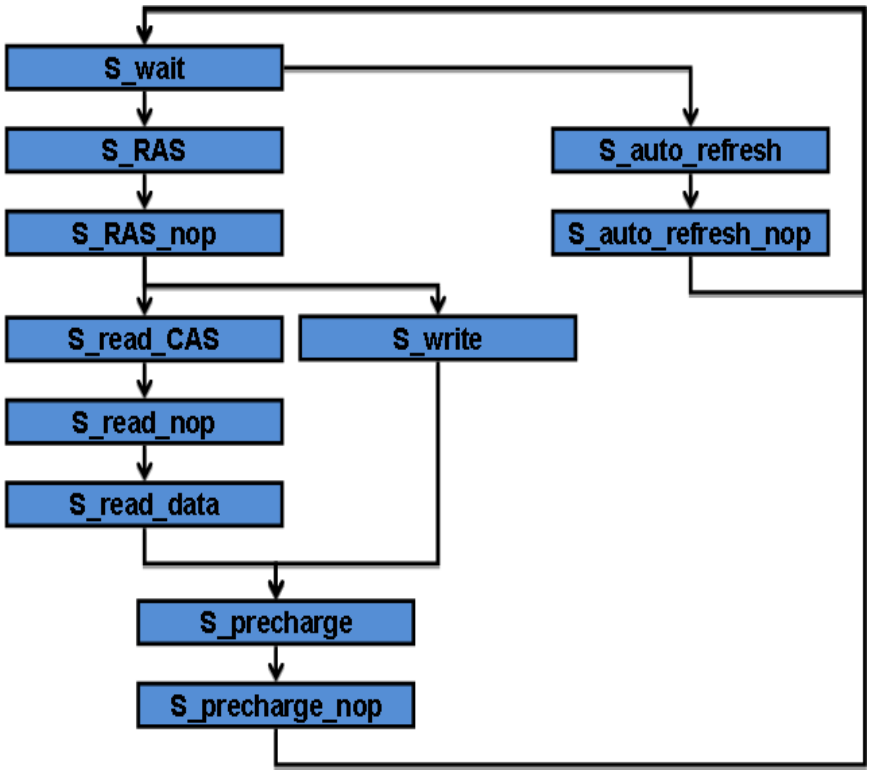

Рис. 38. Конечный автомат основного цикла работы SDRAM

7. Работа со звуком

Для воспроизведения аналоговых звуковых данных используются специализированные цифро-аналоговые преобразователи (ЦАП), и для записи в цифровом виде – аналого-цифровые (АЦП). В этой главе будет рассмотрен классический контроллер PCM3001, который совмещает в одном корпусе преобразователи ЦАП и АЦП.

7.1. Шина I2S

I2S (Inter-IC Signal bus) – синхронный последовательный интерфейс для передачи цифровых звуковых данных с временным разделением каналов. Физически он состоит из двух линий для приёма и передачи данных (DIN, DOUT – здесь и далее названия линий передачи даны в соответствии с принципиальной схемой SLED) и двух линий для тактовых сигналов. Один из тактовых сигналов, LRCK (Left/Right Clock), подаётся с частотой, равной частоте дискретизации, и предназначен для разделения каналов (данные левого канала передаются при высоком уровне LRCK, а правого – при низком), второй – BCK (Bit Clock) – тактирует передачу битов данных. В соответствии со спецификацией, при использовании PCM3001 допускаются частоты BCK, равные 32, 48 или 64 частотам LRCK.

В зависимости от разновидности интерфейса I2S данные могут быть по-разному выравнены относительно фронтов LRC и могут иметь разный порядок бит. В SLED PCM3001 настроен на выравнивание по левому фронту данных АЦП и по правому – ЦАП, первым должен идти наиболее значащий бит (рис. 39).

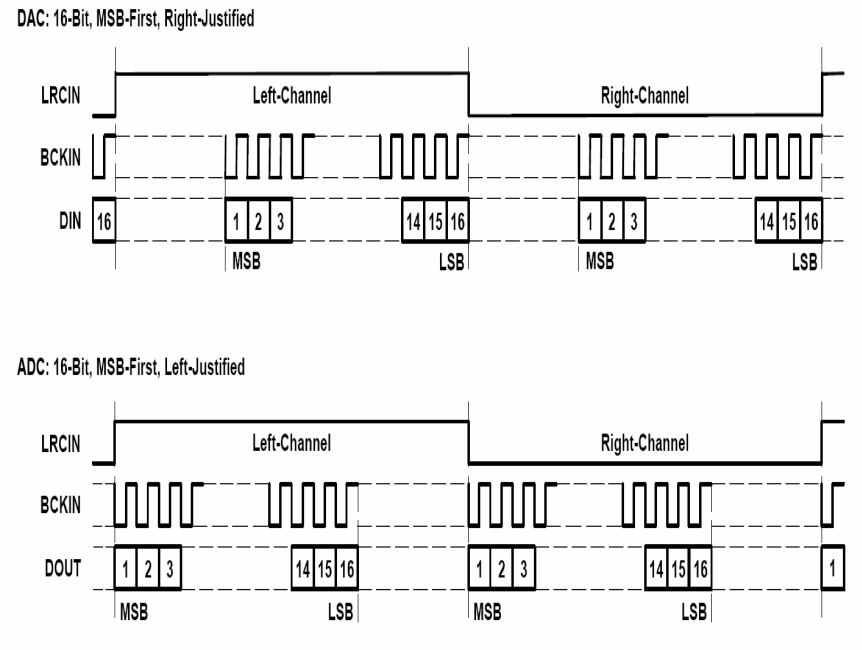

Рис. 39. Диаграмма передачи данных по интерфейсу I2S

7.2. Тактовые сигналы ЦАП-АЦП

Для работы собственно АЦП/ЦАП необходим базовый тактовый сигнал CLK12. Его частота должна быть равна 256, 384 или 512 частотам дискретизации.

Для управления из FPGA доступен ещё один сигнал – RSTB – это сигнал перезагрузки PCM3001 с активным низким уровнем. Во время работы АЦП/ЦАП этот сигнал дожжен поддерживаться в высоком (неактивном) состоянии. Импульсы низкого уровня на нём не должны быть короче 40 наносекунд, то есть двух тактов 40-мегагерцового кварцевого генератора, установленного в SLED. После возвращения RSTB в неактивное состояние PCM3001 начинает процедуру перезагрузки, которая длится 1024 периода CLK12.

PCM3001 поддерживает частоты дискретизации от 4 до 48 килогерц. В современных устройствах наиболее распространены частоты 8, 44.1 и 48 кГц. Соответствующие этим частотам частоты LRCK, BCK и CLK12 приведены в таблице 3.

ЧАСТОТА ДИСКРЕТИЗАЦИИ, КГЦ	ЧАСТОТА CLK12, МГЦ	ЧАСТОТА LRCK, КГЦ	ЧАСТОТА BCK, КГЦ		
Fs	= 256 Fs	= Fs	= 32 Fs	= 48 Fs	= 64 Fs
8	2.048	8	256	384	512
44.1	11.2896	44.1	1411.2	2116.8	2822.4
48	12.288	48	1536	2304	3072

Табл.3. Частоты тактовых сигналов для PCM3001

При реализации контроллера звука следует помнить, что данные захватываются PCM3001 по восходящему фронту BCK, и поэтому для повышения общей устойчивости системы изменения состояния DIN должны происходить как можно дальше по времени от этого фронта, например в районе фронта нисходящего.

8. Работа с графическими дисплеями

В настоящее время всё ещё широко распространён стандарт аналоговой передачи видео VGA (Video Graphics Array). Вообще говоря, термин VGA изначально обозначал только один стандарт, выпущенный компанией IBM в 1987 году и описывавший ряд режимов с небольшими по современным меркам разрешениями экрана (до 640x480 пикселей), но в настоящее время под ним часто понимают все режимы передачи видеосигнала через 15-контактный D-sub разъём.

Для передачи собственно видеосигнала в VGA задействовано пять сигнальных линий – по трём из них передаётся непосредственно информация о цвете (красный, зелёный и синий аналоговые каналы, интенсивность цвета задаётся уровнем сигнала на соответствующей линии), а ещё по двум осуществляется управление развёрткой монитора (линии вертикальной и горизонтальной синхронизации). Передача данных о цвете каждого пикселя осуществляется последовательно построчно слева направо сверху вниз. В паузах между строками подаётся сигнал горизонтальной синхронизации (т.н. горизонтальный синхроимпульс), а в паузах между кадрами – вертикальной. Синхроимпульсы могут иметь различную полярность.

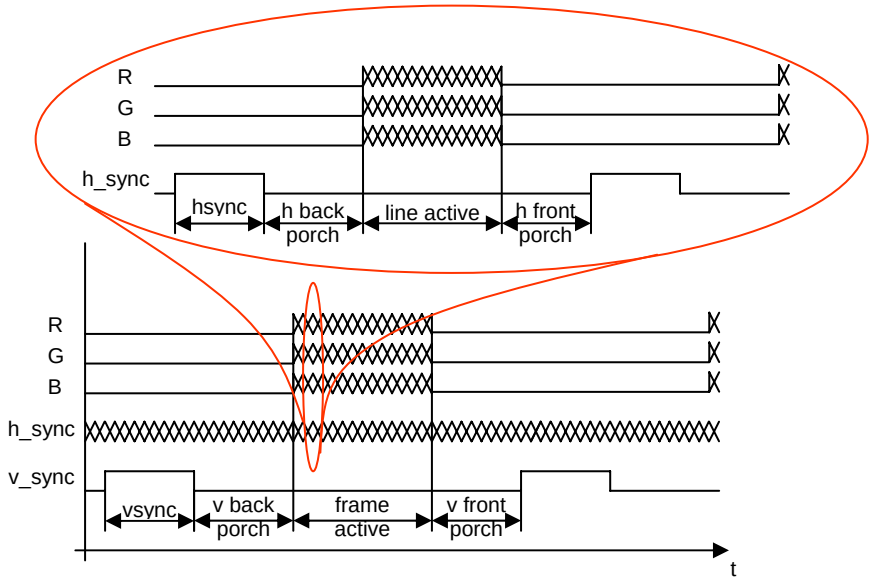

Рис. 40. Диаграмма передачи данных через VGA-интерфейс

На рис. 40 приведена временная диаграмма передачи данных через VGA-интерфейс. Длительности каждой из фаз для режима 800x600x60 Гц указаны в таблице 4.

General Timing	Screen refresh rate	Vertical refresh	Pixel freq.
	60 Hz	37.878788 kHz	40.0 MHz

Horizontal timing (line)			**Vertical timing (frame)**		
Line part	Pixels	Time [µs]	Frame part	Lines	Time [ms]
Visible area	800	20	Visible area	600	15.84
Front porch	40	1	Front porch	1	0.0264
Sync pulse	128	3.2	Sync pulse	4	0.1056
Back porch	88	2.2	Back porch	23	0.6072
Whole line	1056	26.4	Whole frame	628	16.5792

Табл.4. Временные характеристики режима SVGA 800x600x60 Гц

При реализации не следует забывать, что синхроимпульсы должны идти всё время работы адаптера, потому что даже кратковременное отключение какой-либо из синхронизаций может быть воспринято монитором как сигнал к переходу в один из режимов энергосбережения.

8.1. Сигналы синхронизации

Сигналы, используемые в видео-контроллерах:

- R(/8), G(/8) B(/8) – 8-битные параллельные шины для подачи текущих значений цвета на THS8135.
- VCLK – линия для подачи тактового сигнала на THS8135. Частота сигнала должна быть равна частоте смены пикселей соответствующего режима. (40 МГц для режима 800x600x60 Гц). При реализации не стоит забывать о задержках распространения сигнала внутри ПЛИС, и о том, что эти задержки могут быть разными для разных сигналов. Чтобы избежать расхождения во времени тактового сигнала и сигналов на шинах цветов, имеет смысл, во-первых, поставить дополнительный выходной буфер из D-триггеров на шинах цветов, а во-вторых, выводить тактовый сигнал из ПЛИС при помощи примитива **FDDRRSE**. Кроме того, следует помнить, что ЦАП, которому мы передаём данные, захватывает их по восходящему фронту тактового сигнала, что означает, что эти данные должны быть стабильны (не должны изменяться) какое-то время до и после фронта, во избежание возможных проблем, связанных с нестабильностью частоты тактового сигнала. Для этого можно выдавать через FDDRRSE не точную копию внутреннего тактового сигнала, а инвертированную, тогда для ЦАП данные на шинах будут изменяться в районе нисходящего фронта тактового сигнала – на максимально возможном расстоянии от фронта восходящего.
- VBLANK – управляющий вход THS8135, предназначенный для принудительной установки нулевого уровня на выходах ЦАП, независимо от данных на шинах цветов. На этом входе низкий уровень является активным.

- VSYNC и VSYNCT – вместе с VBLANK предназначены для вставки синхроимпульсов на аналоговые линии цветов. Такая синхронизация не требуется в обычном VGA-контроллере, поэтому на них можно просто установить логическую единицу.
- INT_VHSYNC и INT_VSYNCV – непосредственно линии управления горизонтальной и вертикальной развёрткой соответственно.

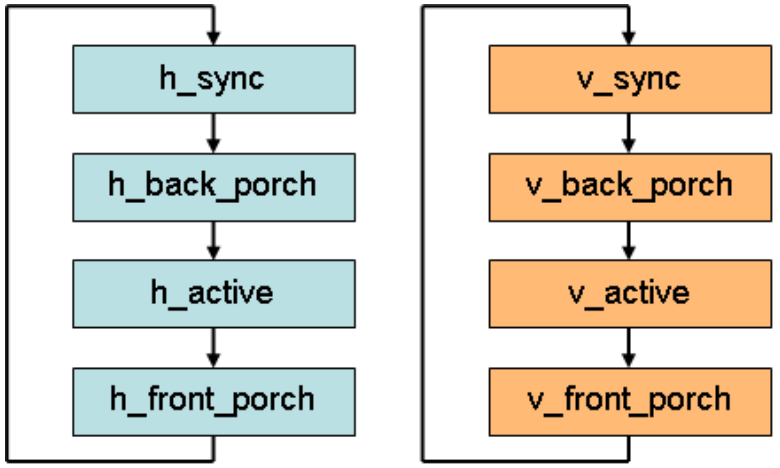

Рис. 41. Конечные автоматы для реализации VGA-интерфейса

Реализовать VGA-контроллер удобнее всего на основе двух конечных автоматов, показанных на рис. 41. При этом состояние линий горизонтальной и вертикальной синхронизации однозначно задаётся соответствующими конечными автоматами. Данные на линиях цвета должны присутствовать только в то время, когда оба автомата находятся в активном состоянии, что обозначает видимую область экрана. Всё остальное время на линии цвета должен быть подан нулевой уровень, для этого можно применять управляющий вход THS8135, называемый в VBLANK.

9. АППАРАТНОЕ РЕШЕНИЕ SLED

Аппаратное решение SLED (Educational Device от компании SoftLab-NSK) разработано для обучения программированию ПЛИС на примере кристалла Xilinx семейства Spartan3E XC3S500E.

С помощью SLED в обучении решаются следующие задачи:

- общение с ПК через последовательный порт COM;
- использование динамической памяти стандарта SDRAM;
- звуковые кодеки, запись, воспроизведение и обработка звука;
- контроллер VGA для вывода информации на монитор.

Также на устройстве имеется встроенная клавиатура и сегментный индикатор для отображения четырех цифр.

9.1. Структурное описание

В качестве основного вычислителя и контроллера используется кристалл ПЛИС семейства Spartan3E XC3S500E. На рис. 42 приведена блок-схема внутреннего устройства SLED.

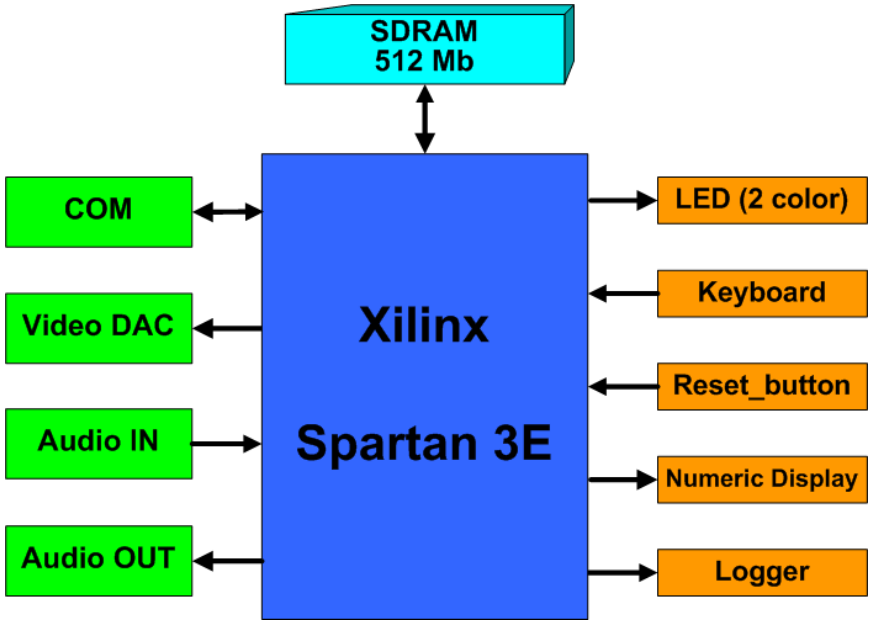

Рис. 42. Блок-схема структурного решения SLED

Такое структурное решение позволяет осуществлять следующие действия с использованием SLED:

- передачи данных в ПК и прием данных из ПК через COM-порт;
- возможность хранения до 512 Мб данных;
- прием звука в оцифрованном виде;
- воспроизведение звука в цифровом виде;
- вывод информации на монитор с использованием видео-ЦАП;
- индикация внутреннего состояния устройства;
- ввод данных со встроенной клавиатуры;
- возможность сброса в начальное состояние;
- вывод отладочной информации на логический анализатор (логгер).

9.2. Аппаратное описание

Память

В устройстве SLED имеется встроенная динамическая память стандарта SDRAM объемом 512 Мб.

Установленная память Samsung K4S511632 обладает следующими характеристиками:

- напряжение питания 3,3 В;
- внутренняя архитектура 32Mx16, 4 банка;
- максимальная частота 133 МГц;
- корпус 54pins TSOP(II).

Максимальная пропускная способность составляет 253 МБ/с.

Звук

В устройстве имеется один аналоговый стереовход и один аналоговый стереовыход. Для преобразования аналоговых сигналов в цифровые и обратно используется микросхема PCM 3001 от Texas Instruments, обладающая следующими характеристиками:

- напряжение питания 5 В;
- встроенные 18-битные стереопреобразователи ЦАП и АЦП;
- динамический диапазон АЦП – 94 ДБ, ЦАП – 97 ДБ;
- поддержка форматов I2S и DSP;
- частота дискретизации от 4 до 48 КГц;
- корпус 28-pin SSOP.

При частоте дискретизации 48 КГц, выделяя 2 стерео-канала, результирующий поток составляет 48К*2*18бита = 211 КБ/с.

Видео

Используя встроенный видео-ЦАП, пользователь может выводить данные на монитор посредством VGA. Используемые кварцевый генератор на 40 МГц позволяет выводить данные на монитор с разрешением 800 x 600 точек с частотой 60 Гц.

Используемый видео-ЦАП THS8135 от Texas Instruments обладает следующими характеристиками:

- 10-битный преобразователь ЦАП;
- поддержка входных форматов YCbCr и RGB;
- напряжение питания 3,3 В (аналоговое) и 1,8 В (цифровое);
- корпус TQFP-48.

COM-порт

В современных компьютерах все реже встречается порт COM, который при относительно малой пропускной способности является достаточно простым и надежным для реализации управления либо передачи небольших объемов данных.

В устройстве SLED общение с ПК происходит через порт USB, однако внутри с помощью специального контроллера происходит эмуляция порта COM. Это позволяет видеть общение с ПК со стороны кристалла ПЛИС как общение со стандартным COM-портом, при этом сохраняется возможность использования устройства практически с любым современным компьютером.

Упрощённая схема обмена данными между компьютером и SLED представлена на рис. 43.

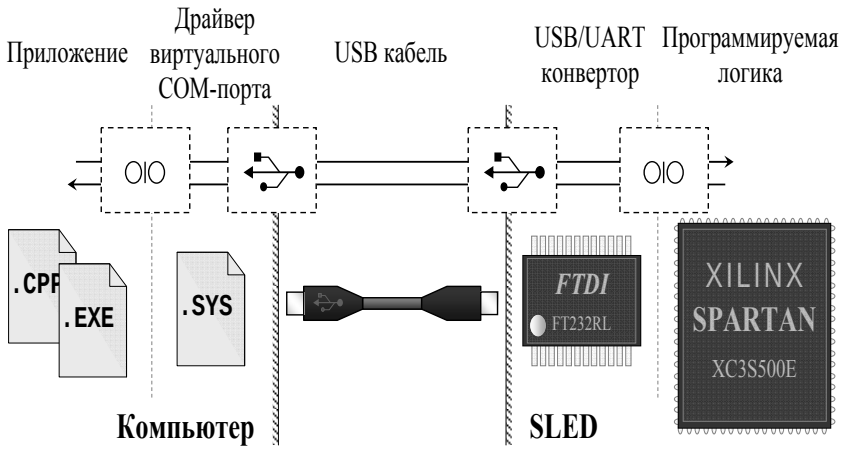

Рис. 43. Схема обмена данными между ПК и SLED

Аппаратная эмуляция последовательного интерфейса со стороны SLED достигается благодаря использованию контроллера ***FT232RL*** фирмы «Future Technology Devices». Этот контроллер установлен на плате SLED и представляет собой USB/UART конвертор (UART – универсальный асинхронный приемо-передатчик). Основные интерфейсы контроллера FT232RL показаны на рис. 44.

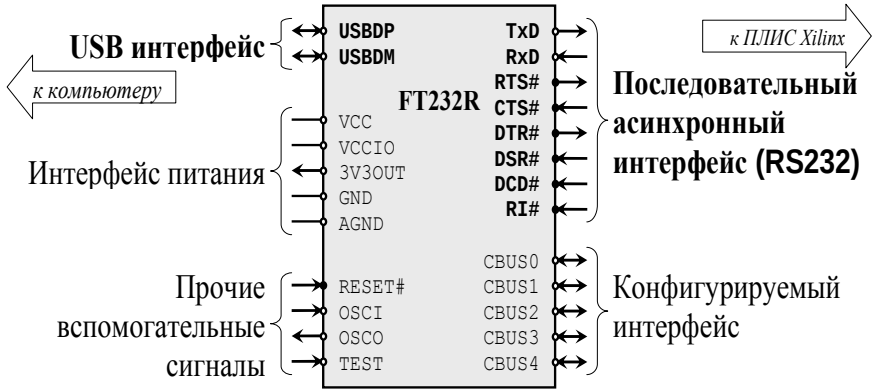

Рис. 44. Интерфейс контроллера FT232RL

Программная эмуляция последовательного интерфейса со стороны компьютера достигается установкой драйвера виртуального COM-порта, разработанного фирмой «Future Technology Devices» специально для контроллера ***FT232RL***. Этот драйвер устанавливается в операционной системе и при подключении USB-устройства на основе ***FT232RL*** создаёт в системе дополнительный COM-порт, с которым посредством стандартного API может работать любое пользовательское приложение. В качестве примера можно использовать бесплатно распространяемую программу ***Tera Term***.

Таким образом, несмотря на то, что физически SLED подключается к компьютеру через USB-кабель, какое-либо использование USB полностью исключается, и создаётся полноценная эмуляция обмена данными через последовательный асинхронный интерфейс.

FT232RL обладает следующими характеристиками:

- интерфейс USB;
- встроенный кварцевый генератор 12 МГц;
- встроенный входной буфер на 256 байт и 128 байт для выхода;
- корпус 28-pin SSOP.

Индикация

На плате установлен двухцветный индикаторный светодиод с общим катодом. Он позволяет получить два различных цвета, а следовательно, добиться более информативной индикации о внутреннем состоянии устройства.

Клавиатура

Установленная клавиатура Accord AK-1604 имеет 16 клавиш в формате 4 x 4. Матричный тип клавиатуры позволяет иметь лишь 8 контактов для определения нажатой клавиши.

Numeric display

Имеющийся в составе SLED дисплей позволяет отображать 4 цифры с точками и представляет собой объединенные в одном корпусе четыре 7-сигментных индикатора.

Установленный дисплей от Kingbright CA04-41EWA обладает следующими характеристиками:

- размер цифр 10,16 мм;
- отображаемый цвет – красный;
- рабочее напряжение 5 В;
- 20 управляющих контактов (цифры объединены в пары).

Загрузка программных файлов

Загрузка осуществляется через специальный кабель, имеющий с одной стороны разъем LPT для подключения к ПК, с другой стороны – разъем DB-9, подключаемый непосредственно к SLED.

В качестве программного обеспечения для загрузки используется входящий в состав пакета Xilinx ISE – iMPACT.

СПИСОК ЛИТЕРАТУРЫ

1. Мячев А. А, Степанцов В. Н. ПЭВМ и микроЭВМ. М.: Радио и связь, 1991.
2. Пятибратов А. П. Вычислительные машины, системы и сети. М.: Финансы и статистика, 1991.
3. Villasenor J., Mangione-Smith W. Configurable Computing, Scientific American. June 1997.
4. Тарасов И. Е. Разработка цифровых устройств на основе ПЛИС Xilinx® с применением языка VHDL. М.: Горячая линия - Телеком, 2005.
5. Зотов Ю. В. Проектирование встраиваемых микропроцессорных систем на основе ПЛИС фирмы XILINX®. М.: Горячая линия – Телеком, 2006.
6. Гадзиковский В. И. Методы проектирования цифровых фильтров. М.: Горячая линия – Телеком, 2007.
7. Ганеев Р. М. Математические модели в задачах обработки сигналов. М.: Горячая линия – Телеком, 2002.
8. Зотов В. Ю. Проектирование цифровых устройств на основе ПЛИС фирмы XILINX в САПР WebPACK ISE. М.: Горячая линия – Телеком, 2003.
9. Потехин Д. С. Разработка систем цифровой обработки сигналов на базе ПЛИС. М.: Горячая линия – Телеком, 2007.
10. Кун С. Матричные процессоры на СБИС. М.: Мир, 1991.
11. Угрюмов Е. П. Цифровая схемотехника. СПб.: БХВ-Петербург, 2002.
12. Новиков Ю. В. Основы цифровой схемотехники. М.: Мир, 2001.
13. Douglas Perry. VHDL. McGraw-Hill, 1998.