Министерство образования и науки Российской Федерации
Национальный исследовательский университет «МИЭТ»

Д.Н. Беклемишев, А.Н. Орлов, М.Г. Попов,
А.А. Кудров, А.Л. Переверзев

Моделирование микропроцессорных систем на базе программируемых логических интегральных схем с использованием Verilog HDL и САПР Quartus II

Учебное пособие по курсу
«Микропроцессорные средства и системы»

Под редакцией
доктора технических наук, доцента **А.Л. Переверзева**

Утверждено редакционно-издательским советом университета

Москва 2014

УДК 004.31(075.8)

М74

Рецензенты: докт. техн. наук, проф. *С.В. Гаврилов*

докт. техн. наук, проф. *А.Н. Соловьев*

Беклемишев Д.Н., Орлов А.Н., Попов М.Г., Кудров А.А.,

Переверзев А.Л.

М74 **Моделирование микропроцессорных систем на базе программируемых логических интегральных схем с использованием Verilog HDL и САПР Quartus II**: учеб. пособие по курсу «Микропроцессорные средства и системы» / Под ред. А.Л. Переверзева. - М.: МИЭТ, 2014. - 100 с.: ил.

ISBN 978-5-7256-0760-4

Рассмотрены типы и структуры ПЛИС, проанализирован маршрут проектирования систем на базе ПЛИС. Изложены основы Verilog HDL, необходимые для создания как синтезируемых описаний, так и тестовых воздействий, приведены примеры. Содержатся задания и методические указания для их выполнения в САПР Quartus II и на учебном стенде.

Для студентов факультета МП и ТК МИЭТ, обучающихся по специальностям 230100 «Информатика и вычислительная техника», 210100 «Биотехнические системы и технологии», 211000 «Конструкция и технология электронных средств», 231000 «Программная инженерия», 231300 «Прикладная математика». Может быть полезно студентам других специальностей факультета, а также колледжа электроники и информатики.

Учебное издание

Беклемишев *Дмитрий Николаевич*

Орлов *Александр Николаевич*

Попов *Михаил Геннадиевич*

Кудров *Артур Александрович*

Переверзев *Алексей Леонидович*

Моделирование микропроцессорных систем на базе программируемых логических интегральных схем с использованием Verilog HDL и САПР Quartus II

Редактор *Е.Г. Кузнецова*. Технический редактор *Л.Г. Лосякова*. Корректор *Л.Г. Лосякова.*

Подписано в печать с оригинал-макета 30.06.2014. Формат 60×84 1/16. Печать офсетная. Бумага офсетная. Гарнитура Times New Roman. Усл. печ. л. 5,80. Уч.-изд. л. 5,0. Тираж 350 экз. Заказ 36.

Отпечатано в типографии ИПК МИЭТ.

124498, Москва, Зеленоград, проезд 4806, д. 5, МИЭТ.

ISBN 978-5-7256-0760-4

Введение

Программируемые логические интегральные схемы (ПЛИС) широко применяются при построении цифровых вычислительных систем. Они используются как для замены дискретных логических элементов, так и для построения сложных систем на кристалле, включающих интерфейсную логику, процессорные ядра, элементы памяти, модули цифровой обработки сигналов, специализированные модули и т.д.

Системы, построенные на основе ПЛИС, имеют ряд важных преимуществ перед системами на микропроцессорах или дискретной логике:

• решают проблему интегральной реализации специализированных узлов системы;

• имеют более высокое быстродействие вследствие параллелизма и аппаратной реализации различных функций;

• обеспечивают возможность создания динамически реконфигурируемых систем;

• снижают количество ошибок при проектировании;

• проект ПЛИС может быть использован для разработки полузаказной или заказной ИС.

При создании проектов ПЛИС применяют высокоуровневые языки описания аппаратуры (Hardware Description Languages), такие как Verilog HDL и VHDL, являющиеся базовыми языками при разработке аппаратуры современных вычислительных систем.

Язык VHDL (Very High Speed Integrated Circuit Hardware Description Language) был разработан в 1983 году по заказу Министерства обороны США для формального описания логических схем. Синтаксис VHDL очень похож на синтаксис языка Ада.

Язык Verilog HDL был создан в 1984 году как средство моделирования аппаратуры и только в 1995 году было принято решение о его стандартизации. Синтаксис Verilog HDL очень похож на синтаксис языка С, что упрощает его освоение.

Языки описания аппаратуры включают множество конструкций, не все из которых поддерживаются средствами синтеза. Целью настоящего учебного пособия является освоение основных этапов создания, верификации и физической отладки синтезируемых Verilog-описаний на примере разработки устройства с микропрограммным управлением.

Данное учебное пособие базируется на основах цифровой схемотехники и микропроцессорной техники и состоит из трех частей.

В первой части представлены типы и структуры ПЛИС, проанализирован маршрут проектирования систем на основе ПЛИС. Рассмотрены основы Verilog HDL, необходимые для создания как синтезируемых описаний, так и тестовых воздействий, приведены примеры.

Вторая часть посвящена базовой справочной информации по САПР Quartus II.

Третья часть содержит четыре лабораторные работы и методические указания для их выполнения в системе Quartus II и на учебном стенде.

Часть 1. Типы и структуры ПЛИС, маршрут проектирования. Основы создания синтезируемых описаний на Verilog HDL

1.1. Основные типы и структуры ПЛИС

Цифровые системы обработки информации, как правило, состоят из стандартных элементов, таких как процессоры, контроллеры, память, интерфейсные микросхемы и т.д. Однако практически в каждой разработке есть необходимость в реализации специализированных модулей, характерных для конкретного устройства или типа устройств. Исторически такие специализированные блоки реализовывались на основе микросхем средней и малой степени интеграции, что увеличивало количество применяемых в устройстве корпусов, усложняло монтаж, снижало надежность и быстродействие. Реализация специфических компонентов системы на базе заказной микросхемы затруднительна, поскольку связана с большими затратами средств и времени на проектирование.

Данная проблема нашла решение в середине 70-х годов XX века с появлением БИС и СБИС с реконфигурируемой структурой. Первыми представителями таких ИС стали программируемые логические матрицы (ПЛМ; Programmable Logic Array - PLA), программируемая матричная логика (ПМЛ; Programmable Array Logic - PAL) и базовые матричные кристаллы (БМК). Последние также называют вентильными матрицами (ВМ; Gate Array - GA).

Позже ИС PLA- и PAL-типов объединили термином Complex Programmable Logic Device (CPLD), а ИС GA-типа получили продолжение в виде Field Programmable Gate Array (FPGA). История развития ИС с реконфигурируемой структурой проиллюстрирована на рис.1.1.

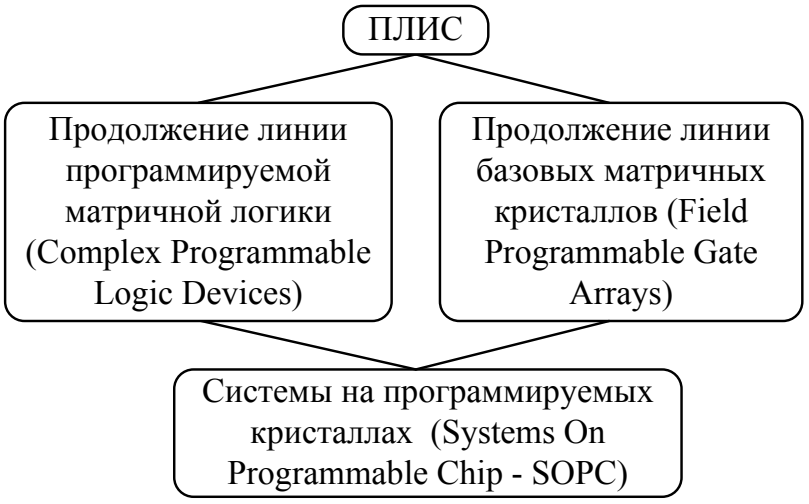


Рис.1.1. Типы ПЛИС

Программируемость ПЛИС реализуется благодаря наличию множества ключевых элементов, состояние которых задается разработчиком. Различают однократно программируемые и репрограммируемые ПЛИС.

ПЛИС CPLD-типа бывают как однократно программируемыми, так и репрограммируемыми.

Для современных ПЛИС типа FPGA характерна репрограммируемая структура на основе триггерной памяти конфигурации (SRAM-based). В этом случае в проектируемой системе необходимо иметь постоянное запоминающее устройство (ПЗУ) для хранения конфигурационной информации. Существуют и другие решения, например, фирма Xilinx выпускает семейство ПЛИС Spartan3AN со встроенной флеш-памятью. Фирма Actel производит ПЛИС на основе так называемой Flash-технологии. При этом ключевой элемент и ячейка ПЗУ интегрированы между собой, т.е. конфигурационная память распределена по всему кристаллу.

Рассмотрим обобщенные структуры двух основных типов современных ПЛИС, т.е. CPLD и FPGA.

Обобщенная структура ПЛИС типа CPLD (рис.1.2) состоит из коммутационной матрицы, массива функциональных блоков (ФБ) и блоков ввода/вывода на периферии кристалла.

Каждый ФБ содержит конфигурируемую многовходовую матрицу элементов И, формирующую конъюнктивные термы от входных переменных, и группу элементов ИЛИ, между которыми распределяются выработанные термы. Помимо этого, ФБ может содержать и другие элементы, расширяющие его функциональные возможности, например мультиплексоры, триггеры и т.п.

Рис.1.2. Обобщенная структура ПЛИС CPLD-типа

Типовая коммутационная матрица позволяет соединять выход любого ФБ с любым входом любого другого ФБ или входом блока ввода/вывода. Коммутационная матрица CPLD имеет регулярную и непрерывную структуру, что гарантирует предсказуемое время задержек распространения сигналов по линиям связей (рис.1.3).

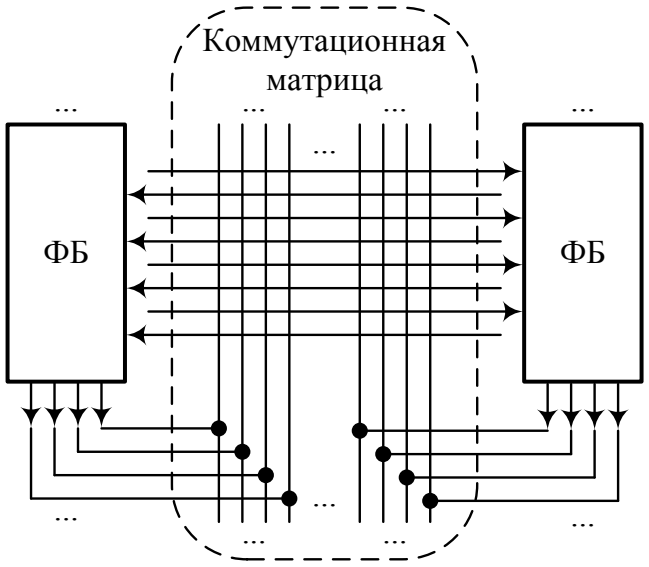


Рис.1.3. Фрагмент схемы коммутации ФБ

Лидерами по производству ПЛИС являются фирмы Altera и Xilinx, которые выпускают широкий спектр ПЛИС типа CPLD с различными возможностями по быстродействию и количеству логических вентилей. Типичными их представителями являются семейства FLEX, MAX (Altera) и XC9500, CoolRunner (Xilinx). Быстродействие современных CPLD достигает 250 МГц, а емкость - десятков тысяч вентилей.

Обобщенная структура ПЛИС типа FPGA (рис.1.4) состоит из блоков ввода/вывода, расположенных по периферии кристалла, схем управления тактовыми сигналами, блочной памяти, массива конфигурируемых логических блоков (КЛБ) и трассировочных ресурсов.

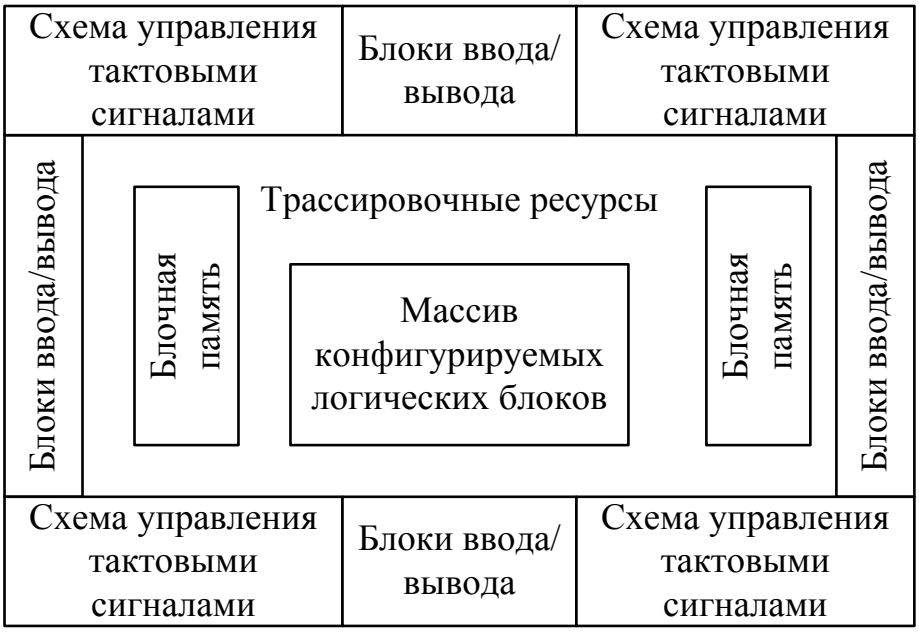


Рис.1.4. Обобщенная структура ПЛИС FPGA-типа

Как правило, блоки ввода/вывода содержат буферы с тремя состояниями, мультиплексоры, линии задержки, регистры. Все эти элементы обеспечивают гибкую организацию взаимодействия проекта ПЛИС с внешними устройствами.

Схемы управления тактовыми сигналами могут включать модули фазовой автоподстройки и синтезаторы частот.

Блочная память представляет собой конфигурируемые по ширине и глубине модули статической памяти. Общий объем этого внутреннего ОЗУ может достигать нескольких мегабит. Помимо этого, часть КЛБ может быть использована в качестве так называемой распределенной памяти.

Основу КЛБ составляет один или несколько табличных функциональных логических преобразователей. В англоязычной литературе такие преобразователи называют Lookup Table (LUT). LUT представляют собой программируемые ПЗУ, которые адресуются аргументами реализуемых логических функций. Параметры LUT зависят от конкретного семейства ПЛИС, например, в ПЛИС семейства Spartan3 фирмы Xilinx используют 4-входовые преобразователи.

Помимо функциональных преобразователей, реализующих логические функции, КЛБ содержат мультиплексоры, позволяющие гибко коммутировать между собой преобразователи, входы и выходы. Кроме того, в КЛБ содержатся регистры с программируемой полярностью тактового сигнала.

Трассировочные ресурсы FPGA, как правило, сегментированы и имеют иерархическую структуру, т.е. состоят из проводящих участков разных длин, соединенных ключевыми элементами. Сложная структура трассировочных ресурсов FPGA приводит к усложнению контроля над задержками распространения сигналов по цепям межсоединений по сравнению с ПЛИС CPLD-типа.

Типичными представителями ПЛИС FPGA-типа являются семейства Stratix, Cyclone (Altera) и Virtex, Spartan (Xilinx). Быстродействие современных FPGA достигает 500 МГц, а емкость - нескольких миллионов вентилей.

ПЛИС FPGA-типа отличаются от ПЛИС CPLD-типа в первую очередь структурой и функциональностью ФБ и КЛБ. Считается, что FPGA имеет большую зернистость по сравнению с CPLD. Это значит, что КЛБ имеют более сложную структуру, чем ФБ, что позволяет реализовывать на основе КЛБ более сложные функции и приводит к упрощению программируемой части соединений. При этом сложнее оптимально использовать возможности каждого КЛБ, что приводит к потерям быстродействия и площади кристалла.

Другим отличием ПЛИС FPGA от ПЛИС CPLD является большее максимальное количество логических вентилей. Если современные CPLD имеют до нескольких тысяч логических вентилей на кристалл, то FPGA содержат от 50 тысяч до нескольких миллионов вентилей.

1.2. Маршрут проектирования систем на основе ПЛИС

Рассмотрим обобщенный маршрут проектирования систем на основе ПЛИС (рис.1.5).

Этап получения и/или анализа технического задания включает моделирование устройства на поведенческом и функциональном уровне. На основе результатов моделирования формируются иерархия проекта, требования, предъявляемые к проекту в целом и к его отдельным модулям, различные ограничения и т.д.

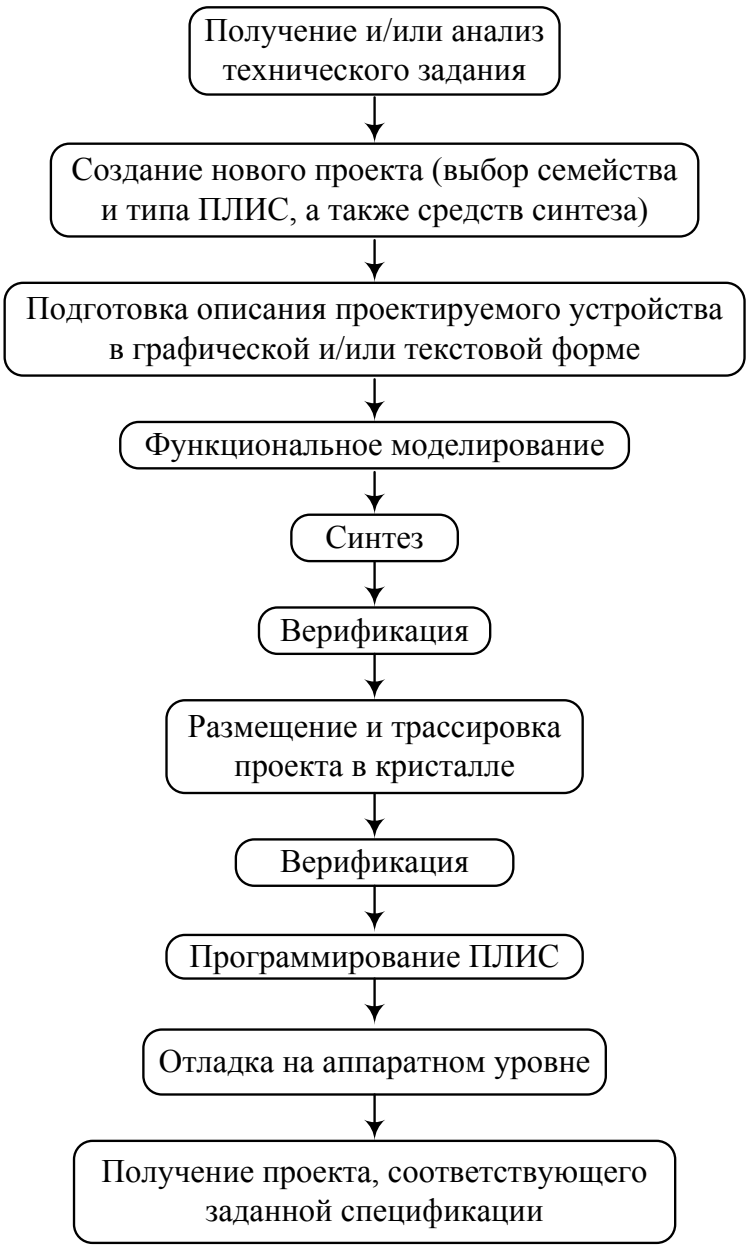


Рис.1.5. Маршрут проектирования систем на ПЛИС

На *этапе создания проекта* выполняется выбор типа, семейства, модели ПЛИС с учетом предъявляемых к устройству требований.

После формирования спецификаций для отдельных модулей и выбора семейства ПЛИС *разрабатывается синтезируемое описание проектируемого устройства*, которое часто называют RTL-описанием (Register Transfer Level). Это описание может быть представлено полностью на HDL (Verilog, VHDL) либо в графическом виде (схема, граф и т.п.). На практике часто применяют следующий подход: модуль верхнего уровня представляют в графическом виде, а входящие в него модули описывают на HDL. Такая орга-

низация проекта упрощает понимание взаимосвязей модулей всей иерархической структуры проекта, а также алгоритмов функционирования устройства в целом.

На *этапе функционального моделирования* разрабатываются тестовые воздействия для созданного HDL-описания, проверяется соответствие результатов работы устройства идеальным результатам, полученным при моделировании или расчетах. Тестовые воздействия могут быть представлены в графическом (временные диаграммы) либо в текстовом (testbench) виде. Как правило, файл тестовых воздействий создается с помощью того же языка, на котором написан проект.

Этап синтеза выполняется в автоматическом режиме. Входными данными для синтезатора являются RTL-описание, технологическая библиотека, определяемая семейством и моделью ПЛИС, ограничения на проект. В результате получают описание в определенном технологическом базисе.

Этап верификации после синтеза и последующие верификации могут быть выполнены с использованием тестовых воздействий, разработанных при функциональном моделировании. На данном этапе проводят статический временной анализ по критическим путям, т.е. по путям прохождения сигнала от входа до выхода схемы, имеющим наибольшую задержку.

Этап размещения и трассировки может быть выполнен как в автоматическом, так и в полуавтоматическом режимах. Результатом данного этапа является топология проекта с учетом выбранного кристалла ПЛИС.

После получения топологии проекта выполняют *верификацию с учетом топологических задержек*. Верификация после каждого этапа проектирования может выявить ошибки в проекте, что приводит к повторению этапа трассировки и размещения с иными ограничениями или к коррекции RTL-описания и спецификаций на модули устройства. Поэтому процесс разработки проекта для ПЛИС является итерационным.

После верификации с учетом топологических задержек выполняют *физическую отладку*, т.е. программируют ПЛИС конфигурационным файлом разрабатываемого проекта и анализируют работу реального устройства.

1.3. Основы разработки синтезируемых описаний на языке Verilog

Verilog HDL - это язык описания аппаратуры, который позволяет осуществить проектирование, верификацию и реализацию аналоговых, цифровых и смешанных электронных систем на различных уровнях абстракции.

Первый стандарт языка Verilog HDL был принят в 1995 году (IEEE Std.1364-1995), в 2001 и 2005 годах были утверждены его очередные версии (IEEE Std.1364-2001 и -2005). Стандарт Verilog HDL определяет синтаксис, все возможные конструкции, объекты и расширения языка. При этом не все конструкции и операторы поддерживаются средствами разработки.

Настоящий раздел посвящен краткому изложению основ разработки и применения синтезируемых конструкций Verilog HDL, т.е. тех конструкций, которые поддерживаются большинством средств синтеза.

Уровни абстракции при проектировании. Verilog HDL поддерживает четыре уровня абстракции:

1) поведенческий уровень (Behavioral Level). Это самый высокий уровень абстракции, он представляет собой описание алгоритма функционирования устройства без привязки к аппаратной реализации. Проектирование на этом уровне подобно обычному программированию на других языках высокого уровня;

2) уровень потока данных (Dataflow Level). На этом уровне описываются обмен данными между регистрами и операции, выполняемые с этими данными. Данный уровень иногда называют уровнем регистровых пересылок (Register Transfer Level - RTL). При этом в описании используются только синтезируемые конструкции;

3) вентильный уровень (Gate Level). Описание на этом уровне подобно графическому представлению проекта, т.е. состоит из описания логических вентилей и соединений между ними;

4) транзисторный, или ключевой уровень (Switch Level). На этом самом низшем уровне абстракции описываются ключевые транзисторы и соединения между ними.

Основные синтаксические правила. Исходный текстовый файл Verilog имеет расширение .v и содержит последовательность символов, к которым относятся пробелы, комментарии, операторы числа, строки, идентификаторы и ключевые слова. В Verilog-описании различаются прописные и строчные буквы. Пробельные символы (пробел, табуляция, переход на новую строку) используются для разделения других конструкций языка и игнорируются синтезатором.

В Verilog приняты две формы для ввода комментариев. Однострочные комментарии начинаются с символов // и заканчиваются концом строки. Блочные комментарии начинаются с символов /* и заканчиваются символами */.

Представление чисел в синтезируемых описаниях. Основным типом синтезируемых констант являются целочисленные константы, которые могут быть безразмерными и размерными.

Формат размерного числа в общем случае имеет вид:

<разрядность>'<формат><значение>,

где **<разрядность>** - число бит в двоичном представлении числа; поле **<формат>** может принимать значения **d, D,** если число десятичное; **h, H,** если число шестнадцатеричное; **o, O,** если число восьмеричное; **<значение>** - цифры 0 - 9 или буквы a, b, c, d, e, f в зависимости от указанного формата.

Пример 1. Представление чисел в Verilog HDL.

```
3 'b010 // 3-битное двоичное число
8 'h1A // 8-битное шестнадцатеричное число
4 'd3 // 4-битное десятичное число
-4 'd3 // 4-битное отрицательное десятичное число
8 'o6 // 8-битное восьмеричное число
10 // безразмерное десятичное число
'hA5 // безразмерное шестнадцатеричное число
```

Для логического разделения разрядов при записи констант может быть использован символ подчеркивания (_). Verilog игнорирует данный символ на любой позиции константы, кроме первого символа. Например, с точки зрения Verilog HDL 8'b1010_1110 и 8'b10101110 - это одна и та же константа.

Синтезируемые типы данных. При разработке синтезируемых RTL-описаний в основном применяются типы **wire**, **tri**, **reg**.

Типы **wire** и **tri** относятся к типу данных «цепь», который используется для описания соединений в схеме. Цепи не могут хранить присвоенную им величину, к ним необходимо прилагать непрерывное воздействие. Тип **wire** применяется, когда цепь подключена только к одному источнику сигнала. Тип **tri** допускает подключение одной цепи к нескольким источникам.

Тип **reg** используется для описания переменных, хранящих свои значения между процедурными присваиваниями, например, данный тип может применяться для описания аппаратного регистра. В то же время **reg** можно использовать и для описания комбинационной логики.

Указанные типы данных могут объявляться как векторы, т.е. быть многоразрядными. Если при объявлении разрядность не указана, то по умолчанию цепь или переменная считается однобитной. В Verilog-описании допускается обращение к отдельным битам векторов.

В Verilog HDL разрешено объявление одномерных массивов цепей и переменных. Элемент массива может быть как скалярным, так и многоразрядным. При этом побитное обращение к элементам массива невозможно. Массивы в основном используются для описания памяти и регистровых файлов.

Пример 2. Объявление цепей, переменных, векторов, массивов и обращений к ним.

wire a; // объявление скалярной цепи типа **wire**

reg b; // объявление скалярной переменной типа **reg**

wire [3:0] c; // объявление 4-битного вектора типа **wire**

reg [1:0] d; // объявление 2-битного вектора типа **reg**

wire [7:0] e [0:7]; // массив из восьми 8-битных векторов типа **wire**

reg [2:0] f [0:9]; // массив из десяти 3-битных векторов типа **reg**

d[1]; // обращение к 1-му биту вектора d

f[3]; // обращение к 3-му вектору массива f

wire signed [7:0] g; //объявление 8-битного знакового вектора типа **wire**

reg signed [7:0] i; // объявление 8-битного знакового вектора типа **reg**

В языке Verilog данные могут принимать одно из четырех возможных состояний: 1 - логическая единица или значение true; 0 - логический нуль или значение false; z - состояние высокого импеданса; x - неизвестное логическое состояние.

Для объявления цепей и переменных со знаком используется ключевое слово **signed**.

Параметры. Для объявления констант внутри модуля в Verilog HDL применяют ключевое слово **parameter**. Параметры не могут использоваться как переменные или цепи.

Пример 3. Использование параметров.

parameter width=8; // объявление параметра width

reg [width-1:0] a; // объявление 8-разрядной переменной типа **reg**

Операторы предназначены для указания, какие действия необходимо проводить с операндами. Операнд может быть константой, переменной, цепью, одним или несколькими битами вектора типа **wire** или **reg**, элементом массива. Комбинации из операторов и операндов образуют выражения.

В языке Verilog операторы могут быть трех видов: унарный, бинарный и тернарный. *Унарный оператор* ставится перед операндом, *бинарный* - между двумя операндами, *тернарный* имеет в своем составе два подоператора для разделения трех операндов.

Операторы объединения и повторения используются для объединения нескольких размерных операндов. Объединяемые операнды указываются в фигурных скобках через запятую. Перед фигурными скобками может стоять константа повторения, тогда комбинация операндов, перечисленных в скобках, повторяется указанное число раз.

Пример 4. Операторы объединения и повторения.

```
reg a=1 'b1;
reg [3:0] b=4 'b1010;
reg [5:0] c;
c={a, b[1:0], a, b[3:2]}; // c=6 'b110110
c={2{a}, b[3:0]}; // c=6 'b111010
```

Арифметические операторы могут быть унарными и бинарными. Унарные операторы (+ и –) используются для определения знака одного операнда. При этом отрицательные числа представляются в дополнительном коде. Бинарные арифметические операторы выполняют арифметические действия сложения (+), вычитания (–), умножения (*), деления (/), возведения в степень (**) и вычисления по модулю (%) над двумя операндами.

Отметим, что использование в синтезируемых Verilog-описаниях операторов сложения, вычитания и умножения приводит к синтезу сумматоров и умножителей. В то же время большинство синтезаторов не поддерживают операции деления, вычисления по модулю и возведения в степень. Исключение составляют операции со степенями 2.

Пример 5. Арифметические операторы.

```
reg [3:0] a=4 'b0010, b=4 'b0011, c=4 'b001x;
d=a+b; // d=4 'b0101
d=a-b; // d=4 'b1111
d=2**b; // d=4 'b1000
d=a*b; // d=4 'b0110
d=a+c; // d=4 'bxxxx
```

Операторы отношения. Выражения, использующие операторы больше (>), больше либо равно (>=), меньше (<), меньше либо равно (<=), возвращают логический нуль, если заданное отношение ложно, или логическую единицу, если отношение истинно.

Пример 6. Операторы отношения.

```
reg [3:0] a=4 'b0010, b=4 'b0011, c=4 'b001z;
a>=b; // возвращает логический нуль (0)
```

a<c; // возвращает неопределенное состояние (x)

Логические операторы отрицания (!), И (&&), ИЛИ (||) могут применяться как к операндам, так и к выражениям, результаты которых трактуются как операнды.

Пример 7. Логические операторы.

reg [3:0] a=4 'b0010, b=4 'b0011, c=4 'b001x;

(a>0) && (b>=0); // возвращает логическую единицу (1)

(a>0) || (c>=0); // возвращает неопределенное состояние (x)

Операторы равенства/неравенства сравнивают операнды побитно и возвращают логическую единицу, если результат сравнения истина, или логический нуль, если результат сравнения ложь. Если операнды имеют разную разрядность, то недостающие биты дополняются нулями до нужной разрядности.

Различают логические операторы равенства/неравенства (==, !=), которые возвращают неопределенное состояние (x), если операнды содержат биты со значением x, и условные операторы равенства/неравенства (===, !==), которые сравнивают операнды с учетом значений x, z и возвращают значения 0 и 1.

Пример 8. Операторы равенства/неравенства.

reg [3:0] a=4 'b0010, b=4 'b0011, c=4 'b001x, d=4 'b001x;

a==b; // возвращает логический нуль (0)

a!=b; // возвращает логическую единицу (1)

{a[3:1], a[1]} == b; // возвращает логическую единицу (1)

c==d; // возвращает неопределенное состояние (x)

c===d; // возвращает логическую единицу (1)

Побитовые операторы выполняют логические операции отрицания (~), И (&), ИЛИ (|), исключающее ИЛИ (^) с соответствующими битами операндов. Если операнды имеют разную разрядность, то старшие биты более короткого операнда дополняются нулями.

Пример 9. Побитовые операторы.

reg [3:0] a=4 'b0010, b=4 'b0011, c=4 'b001x;

d=a&b; // d=4 'b0010

d=a|b; // d=4 'b0011

d=a|c; // d=4 'b001x

d=a^b; // d=4 'b0001

d=~a; // d=4 'b1101

Операторы свертки являются унарными и выполняют побитовые логические операции над векторным операндом, результатом является один бит.

Пример 10. Операторы свертки.

reg [3:0] a=4 'b1010;

reg [1:0]b=2 'b11;

d=~|a; // d=1 'b0

d=&b; // d=1 'b1

d=~&a; // d=1 'b1

Операторы сдвига осуществляют сдвиг операнда на заданное количество бит влево или вправо. Существуют два типа операторов сдвига: логические операторы (<< и >>) и арифметические (<<< и >>>).

Пример 11. Операторы сдвига.

reg [3:0] a=4 'b0010;

reg signed [3:0] b=4 'b1011;

c=a << 1; // c=4 'b0100

c=b >> 2; // c=4 'b0010

c=b >> 2; // c=4 'b1000

Условный оператор - это единственный в Verilog тернарный оператор. В общем случае он записывается в виде

<условие> **?** <выражение 1> **:** <выражение 2>;

Сначала проверяется указанное условие. Если оно истинно, то выполняется выражение 1; если ложно, то выполняется выражение 2.

Пример 12. Условный оператор.

d=c **?** a **:** b; // мультиплексор 2 в 1

g[7:0]=f **?** e[7:0] **:** 8 'bz; // 8-разрядный буфер с третьим состоянием

В табл.1.1 перечислены операторы в порядке убывания приоритета (слева направо).

Таблица 1.1

Приоритет операторов Verilog HDL

Уровень приоритета	13	12	11	10	9	8	7	6	5	4	3	2	1
Операторы	+ – ! ~	**	* / %	+ –	<< >> <<< >>>	< <= > >=	== != === !==	& ~&	^ ^~ ~^	| ~	&&	||	?:

Непрерывные присваивания используются для присваивания каких-либо величин скалярным или векторным цепям. Считается, что непрерывное присваивание - одна из основных конструкций Verilog HDL для описания на уровне потока данных. Эта конструкция позволяет описывать логические функции на более высоком уровне, чем соединения между логическими вентилями.

Непрерывные присваивания выполняются с помощью ключевого слова **assign**. В общем случае синтаксис выглядит следующим образом:

assign <сила сигнала> #<задержка> <присваивание>,

однако параметры в данной конструкции являются несинтезируемыми, поэтому при составлении синтезируемых описаний будем использовать следующий формат:

assign <присваивание>

Verilog допускает применение неявных непрерывных присваиваний. В этом случае ключевое слово **assign** заменяется на **wire**, т.е. используется конструкция вида **wire** <присваивание>.

Пример 13. Непрерывные присваивания.

```
reg [3:0] a;
wire [1:0]  b;
wire c, d;
assign c = d;
assign c = d | a[2] ;
wire e = b[1]; // неявное непрерывное присваивание
assign b = {a[3], a[1]};
```

Процедурные присваивания, структурная конструкция always. Процедурные присваивания используются для присваивания каких-либо величин переменным в структурных конструкциях. Существует два типа процедурных присваиваний: блокирующие (=) и неблокирующие (<=). Блокирующие присваивания выполняются в том порядке, в каком они описаны в блоке. Неблокирующие присваивания выполняются параллельно.

В Verilog HDL определены две основные структурные конструкции, которые обозначаются ключевыми словами **initial** и **always**. Verilog-модуль может содержать любое количество таких конструкций, все они выполняются параллельно, и порядок их описания в модуле не имеет значения. Вложенность процедурных конструкций не допускается. Кроме того, запрещены процедурные присваивания одной переменной в разных структурных конструкциях. Исключение составляет инициализация переменной с помощью конструкции **initial** и присваивание ей значений в конструкции **always**.

Структурная конструкция **initial** выполняется только один раз, начиная с нулевого момента времени. В синтезируемых Verilog-описаниях данная конструкция используется для инициализации элементов памяти.

Структурная конструкция **always** повторяется циклически, начиная с нулевого момента времени. Данная конструкция и непрерывные присваивания являются основными инструментами для создания синтезируемого Verilog-описания. В синтезируемых описаниях используется следующая форма записи процедурных блоков:

always @ (событие)
begin
    <процедурные присваивания/операторы>
end

Под событием подразумевается изменение уровня указанного сигнала или появление положительного (**posedge**) либо отрицательного (**negedge**) перепада. При таком описании все присваивания и операторы блока будут выполняться при каждом возникновении какого-либо события. В одном блоке могут использоваться несколько событий, для этого применяется ключевое слово **or**. Однако в одном блоке нельзя комбинировать изменения уровня сигнала и перепады. Для описания изменения уровня сигнала в качестве события указывается имя сигнала.

Описание события-перепада в общем случае имеет вид:

posedge/**negedge** <сигнал> **or posedge**/**negedge** <сигнал> **or** …

В случае, когда событием является изменение уровня любого сигнала, используемого в блоке, можно использовать конструкцию

always @ *
begin
    <процедурные присваивания/операторы>
end

Пример 14. Блоки always.

```
reg [3:0] a;
wire [3:0]  b;
wire c;
always @ (a or b) begin // этот и следующий блок always
    a <= &b; // эквивалентны,
end // оба описывают комбинационную схему

always @ *
```

a <= &b;

always @ (posedge c) // данный блок эквивалентен триггеру,

a <= b; // фиксирующему состояние сигнала b по переднему

// фронту сигнала c.

Внутри блоков **initial** и **always** могут использоваться не только операторы и присваивания, но и поведенческие конструкции Verilog. Основными поведенческими конструкциями для создания синтезируемых Verilog-описаний являются условные конструкции (**if-else**) и конструкции ветвления (**case**).

Синтезируемая поведенческая условная конструкция (if-else) аналогична условным конструкциям других языков высокого уровня. При ее выполнении проверяется указанное условие и в зависимости от результата проверки выполняется то или иное действие.

В общем случае условная конструкция имеет вид:

if (<условие>)

<процедурная операция 1>;

else

<процедурная операция 2>;

Допускается использование оператора **if** без оператора **else**. Если необходима проверка нескольких условий, то возможно использование сложных конструкций вида

if (<условие 1>)

<процедурная операция>;

else if (<условие 2>)

else

<процедурная операция>;

Синтезируемая поведенческая конструкция ветвления (case). Для проверки множества условий применяют конструкции ветвления, обозначаемые ключевыми словами **case**, **endcase**, **default**. Ветвь **default** необязательна и выполняется, если проверка всех условий дала результат ложь.

В общем случае конструкция имеет вид:

case (<выражение>)

значение 1: <процедурная операция 1>;

значение 2: <процедурная операция 2>;

default: <процедурная операция по умолчанию>;

endcase

Структура Verilog-модуля. Модуль является основной иерархической единицей Verilog-описания. Модуль взаимодействует с другими модулями через свои порты ввода/вывода (входы, выходы, двунаправленные порты). Внутри модуля объявляют константы, цепи, переменные, указывают параметры портов ввода/вывода, описывают непрерывные присваивания и структурные конструкции. Кроме того, внутри модуля могут быть подключены другие модули для создания иерархической структуры.

Синтезируемый Verilog-модуль может иметь структуру вида

module <имя модуля> (<список портов>);

<объявление параметров>

<объявление входных портов>

<объявление выходных портов>

<объявление двунаправленных портов>

<объявление цепей>

<объявление переменных>

<подключение модулей нижнего уровня иерархии>

<непрерывные присваивания>

<структурные блоки>

endmodule

Ниже приведен пример модуля, в котором описан двухвходовый логический элемент И. Далее на его основе создана иерархическая структура из двух элементов И. Отметим, что при подключении модулей and_2 внутри модуля and_4, помимо названия модуля, указаны уникальные имена m1 и m2, это необходимо для именования множества модулей в иерархической структуре (в том числе имеющих одно и то же описание).

Стандарт Verilog-2001 допускает упрощенную запись объявления портов, в которой объединяются список портов, их объявление и указывается их тип (цепь или переменная). Так, заголовок модуля and_2 можно переписать в виде **module** and_2 (**input wire** a, **input wire** b, **output wire** c).

Пример 15. Модуль с иерархической структурой.

```
module and_2 (a, b, c);
input a;
input b;
output c;
assign c = a&b;
endmodule
module and_4 (a, b, c, d, e);
```

```
input a;
input b;
input c;
input d;
output e;
wire r_1, r_2;
and_2 m1 ( .(a), .(b), .(r_1) );
and_2 m2 ( .(c), .(d), .(r_2) );
assign e = r1&r2;
endmodule
```

При создании иерархической структуры и подключении портов ввода/вывода необходимо соблюдать следующие правила:

• входы внутри модуля могут быть только цепью, а подключаться извне могут как к цепям, так и к переменным;

• выходы внутри модуля могут быть как цепью, так и переменной, а подключаться извне могут только к цепям;

• двунаправленные порты внутри модуля могут быть только цепью и подключаться могут только к цепям.

1.4. Примеры синтезируемых Verilog-описаний

Отличия в использовании блокирующего и неблокирующего процедурных присваиваний. Как отмечалось ранее, блокирующие присваивания выполняются в порядке их записи в последовательном блоке **begin-end**, а неблокирующие присваивания выполняются параллельно. На рис.1.6 приведены результаты синтеза двух Verilog-описаний, отличающихся только типом процедурного присваивания. Видно, что в случае неблокирующего присваивания все три присваивания выполняются параллельно и, следовательно, синтезируются три триггера, а в случае блокирующего присваивания происходит переприсваивание переменных и синтезируется только один триггер:

```
module ShiftReg_NBlock ( input clk, input d, output reg q );
reg a, b;
always @ (posedge clk)
begin
        a <= d; b <= a; q <= b;
    end
endmodule
```

```
module ShiftReg_Block ( input clk, input d, output reg q );
reg a, b;
always @ (posedge clk)
begin
        a = d; b = a; q = b;
    end
endmodule
```

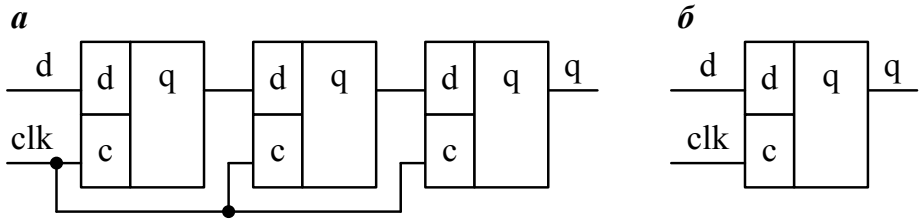


Рис.1.6. Результаты синтеза ShiftReg_NBlock (*а*) и ShiftReg_Block (*б*)

D-триггер по переднему фронту с синхронными сбросом и установкой (активный уровень «1»).

```
module DFF ( input clk, input set, input reset, input d, output reg q );
always @ (posedge clk)
    if (set) q <= 1 'b1;
        else if (reset) q <= 1 'b0;
            else q <= d;
endmodule
```

D-триггер по переднему фронту с асинхронными сбросом и установкой (активный уровень «0»).

```
module DFF ( input clk, input set, input reset, input d, output reg q );
always @ (posedge clk or negedge set or negedge reset)
    if (!set) q <= 1 'b1;
        else if (!reset) q <= 1 'b0;
            else q <= d;
endmodule
```

Триггер-защелка.

```
module Latch ( input enable, input data, output reg q );
always @ (enable)
```

```
if (enable) q <= data;
endmodule
```

4-разрядный двоичный счетчик с асинхронным сбросом (активный уровень «1»).

```
module Counter (  input clk, input reset, output reg [3:0] q );
always @ (posedge clk or posedge reset)
        if (reset) q <= 4 'b0000; else q <= q + 1‘b1;
endmodule
```

Комбинационный сумматор, описанный с помощью непрерывного присваивания assign и конструкции always.

```
module Adder1 ( input [3:0] a, input [3:0] b, output [4:0] sum );
    assign  sum = a + b;
endmodule
module Adder2 ( input [3:0] a, input [3:0] b, output reg [4:0] sum );
    always  @ (a or b) sum = a + b;
endmodule
```

Сумматор с накоплением и синхронным сбросом (аккумулятор).

```
module Acc (input clk, input reset, input [3:0] a, output reg [7:0] sum);
    always  @ (posedge clk)
        if (reset) sum <= 8 'b0000_0000; else sum = sum + a;
endmodule
```

Синхронное 8-разрядное ОЗУ на 256 элементов.

```
module RAM ( input clk, input wr, input [7:0] Address,
    input [7:0] InputData, output [7:0] OutputData );
    reg [7:0] RAM_data [0:255];
    always  @ (posedge clk)
        if (wr) RAM_data[Address] <= InputData;
            else OutputData <= RAM_data[Address];
endmodule
```

Синхронное 8-разрядное ПЗУ на 256 элементов. Основным отличием от предыдущего описания является наличие конструкции **initial,** которая используется для инициализации памяти. В этой конструкции применена системная функция **$readmemh**. В качестве ар-

гументов передаются имя текстового файла с данными в определенном формате и имя ва, который необходимо инициализировать при компиляции проекта.

В приведенном примере конфигурационный файл должен содержать 256 строк, каждая из них содержит 8-разрядное число, записанное в шестнадцатеричном формате.

```
module ROM ( input clk, input [7:0] Address, input [7:0] InputData,
output [7:0] OutputData );
reg [7:0] ROM_data [0:255];
initial $readmemh("ROM.hex", ROM_data);
always  @ (posedge clk)
OutputData <= ROM_data[Address];
endmodule
```

Комбинационная логика. Для описания комбинационной логики можно использовать как непрерывные присваивания, так и блочную конструкцию always. В случае применения блочных конструкций в качестве события указывается либо символ *, либо перечень сигналов, изменение уровня которых влияет на результат работы схемы.

```
module LogicAssign ( input a, input b, input c, output out );
        assign out = (a^b) | c;
endmodule
module LogicAlways ( input a, input b, input c, output reg out );
        always @ (a or b)               // или always @ *
                out = (a^b) | c;
endmodule
```

Мультиплексоры. Verilog-описание мультиплексоров может быть основано как на условных конструкциях, так и на конструкциях ветвления. Отличие заключается в том, что сложные условные конструкции (**if-else**) синтезируются в приоритетно-дешифрированную логику, а **case**-конструкции - в параллельную. На рис.1.7 приведены результаты синтеза обоих вариантов описания мультиплексора.

```
module MuxIfElse ( input a, input b, input c, input [2:0] select,
        output reg out );
        always @ *
                if (select[0]) out = a;
                        else if (select[1]) out = b;
                                else if (select[2]) out = c;
                                        else out = 1 'b0;
endmodule
```

```
module MuxCase ( input a, input b, input c, input [2:0] select,
        output reg out );
        always @ *
                case (select)
                        3 'b001: out = a;
                        3 'b010: out = b;
                        3 'b100: out = c;
                        default: out = 1 'b0;
                endcase
endmodule
```

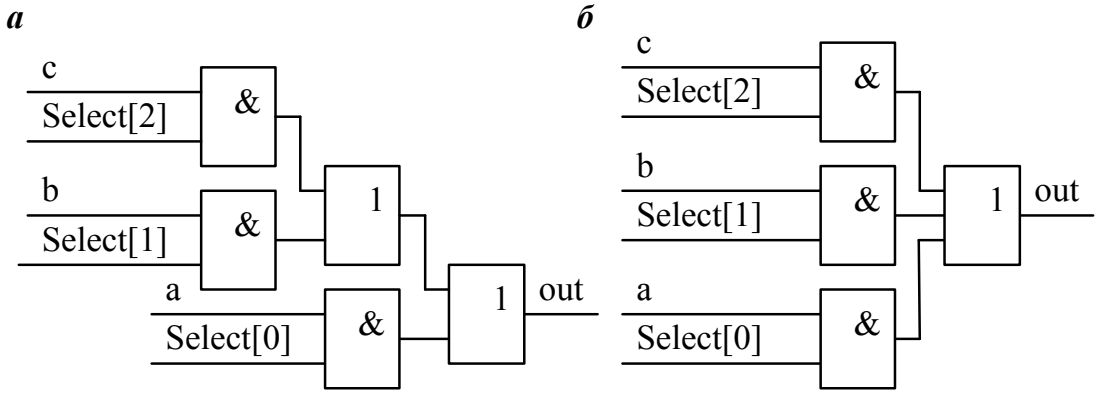


Рис.1.7. Результаты синтеза сложных условных конструкций **if-else** (*а*) и конструкции ветвления **case** (*б*)

1.5. Основы разработки несинтезируемых описаний на языке Verilog

Часто при разработке цифровой аппаратуры на ПЛИС проект достигает такого размера, что использование графических средств задания тестовых воздействий весьма затруднено. В этом случае целесообразно создать файл, который будет содержать тестовые воздействия в текстовом виде. Такой файл носит название *файла тестовых воздействий* (*testbench*).

В основе создания файла тестовых воздействий лежит разработка HDL-описания модели тестовых воздействий, представляющих виртуальное периферийное оборудование, осуществляющее взаимодействие с разработанным в рамках проекта устройством. При этом по сложности проект и модель тестовых воздействий могут быть сопоставимы.

Процессы разработки файла тестовых воздействий и проекта с использованием Verilog HDL практически полностью совпадают. Модель тестовых воздействий может

включать в себя те же конструкции, описанные с помощью тех же синтаксических правил. Однако для расширения функциональных возможностей были введены несинтезируемые конструкции, значительно упрощающие процесс моделирования. Именно об этих особенностях разработки файлов тестовых воздействий пойдет речь в этом разделе.

Несинтезируемые типы данных. В языке Verilog предусмотрены специальные возможности для интерпретации регистров как целых или действительных чисел, а также для хранения информации о временных отсчетах моделирования (модельном времени).

Любую информацию можно хранить в переменных типа **reg**, но для большего удобства хранения целых чисел и информации о временных отсчетах моделирования, а также организации самодокументирования кода применяют предусмотренные в языке Verilog HDL типы переменных **integer** (32 разряда) и **time** (64 разряда) соответственно.

Пример 16. Объявление переменных типов integer и time.

integer A;

time M1, M2;

В отличие от типа **reg,** тип **integer** является знаковым, т.е. подчиняющимся соответствующим правилам арифметики. Переменные типа **time** не являются знаковыми.

Допускается также использование массивов чисел типов **integer** и **time**, которые объявляются аналогично массивам регистрового типа.

Пример 17. Объявление массивов чисел типов integer и time.

integer R [1:64];

time M3 [1:100];

Для упрощения и ускорения разработки HDL-описаний моделей тестовых воздействий в стандарте языка Verilog предусмотрено применение действительных чисел и средств их обработки. Регистры действительного типа и действительные константы создаются с помощью ключевого слова **real**.

Действительные числа могут быть представлены как в формате с фиксированной запятой, так и с плавающей.

Пример 18. Объявление переменных типа real.

real A = 83.375; *//формат с фиксированной запятой*

real B = 1.2e-5; *//формат с плавающей запятой*

real C = 4.13e10; *//формат с плавающей запятой*

Явное указание выделяемого для действительного сигнала количества разрядов в языке Verilog не поддерживается. Присваивание целочисленному регистру действительного значения приводит к округлению последнего до ближайшего целого числа. Таким

образом, язык Verilog, в отличие от таких строго типизированных языков, как VHDL или Ada, допускает неявное преобразование типов данных.

Необходимо отметить, что не все операторы могут быть применены к вещественным числам. Следующие операции не могут быть выполнены переменными типа **real**:

• описание положительного или отрицательного фронтов (**posedge**, **negedge**);

• выбор отдельного бита или группы битов;

• применение в качестве индексов массивов, битов или групп битов;

• создание массива вещественных чисел.

Основные структурные конструкции моделирования. Структурные конструкции **always** и **initial** являются базовыми в поведенческом моделировании электронных устройств.

Каждый из этих операторов представляет собой отдельный поток обработки данных. Все потоки выполняются в модели параллельно, а начало выполнения потоков совпадает с началом модельного времени, однако не допускается использование конструкций с вложенными операторами **always** и **initial**.

В языке Verilog операторы **always** и **initial** представляют собой последовательные или параллельные блоки поведенческих операторов, сгруппированные по функциональному признаку. Поскольку применение оператора **always** было описано ранее, здесь остановимся на операторе **initial**.

Оператор **initial** предназначен для формирования начальных условий моделирования, запускается один раз в начале процесса и не повторяется после его остановки.

Синтаксис оператора **initial**:

initial begin

<процедурные присваивания/операторы>;

end

Если в модуле более одного блока **initial**, то все они будут запущены параллельно в нулевой отсчет времени моделирования. Остановка выполнения одного из блоков **initial** не зависит от других блоков этого же типа. Если в блоке **initial** более одного поведенческого оператора, то необходимо заключить их в операторные скобки **begin**..**end**. В противном случае операторные скобки можно опустить.

Пример 19. Использование структурного оператора initial.

initial x = 4'b1010; //в блоке один поведенческий оператор

initial begin //в блоке два поведенческих оператора

Line000 = 1'b1;

Line001 = 1'b0;

end

При разработке файла тестовых воздействий часто возникает необходимость в задании в конструкции **initial** задержек выполнения поведенческих операторов, например, при формировании временных диаграмм для простых моделей тестовых воздействий. В этом случае задержка формируется следующим образом:

initial begin

<процедурное присваивание/оператор_1>*;*

<задержка>

<процедурное присваивание/оператор_2>*;*

end

где <задержка> - константа, указывающая число шагов моделирования, начиная от текущего, на которое будет задержано выполнение блока <процедурного присваивания/оператора_2>.

Для иллюстрации функционирования механизма задержек рассмотрим работу модуля, представленного в следующем примере.

Пример 20. Использование задержек в конструкции initial.

`timescale 1ns/ 1ps

module Delay_Demo;

reg a;

initial begin

#50 a = 1'b1;

#40 a = 1'b0;

end

endmodule

Первая строка модуля представляет собой директиву компилятора <**`timescale>**, определяющую длительность одного шага моделирования (1 нс) и дискретность временного интервала (1 пс).

До значения модельного времени 50 нс сигнал **a** на временной диаграмме имеет значение x, на 50 нс произойдет присваивание **a** = 1. Задержка вычисляется от текущего значения модельного времени. Таким образом, значение сигнала **a** в результате выполнения второго оператора присваивания изменится с 1 на 0 в момент времени 50 нс + 40 нс = 90 нс.

Поскольку все операторы в поведенческих блоках **initial** выполняются последовательно, то задержку можно интерпретировать как интервал времени выполнения оператора, причем изменение значений управляемых оператором сигналов происходит в конце

указанного интервала времени. Сумма всех задержек в последовательном блоке определяет общее время выполнения блока операторов.

Общая характеристика подпрограмм в языке Verilog. Одним из наиболее распространенных методов повышения эффективности программ моделирования и процессов их разработки, а также читаемости кода является процедурная декомпозиция программ. Задача декомпозиции состоит в выделении отдельных функционально независимых участков кода, решающих определенные задачи. Это позволяет распределить задачи между отдельными разработчиками, участвующими в создании проекта, и существенно сократить время его написания за счет возможности повторного использования фрагментов кода.

Язык Verilog включает два типа независимых фрагментов кода (подпрограмм), обеспечивающих возможность реализовать декомпозицию программы: функции (function) и задачи (task).

Функции в языке Verilog мало отличаются от функций в традиционных языках программирования, например Pascal или C. Задачи в языке Verilog больше напоминают процедуры в языке Pascal. В то же время различия между обоими типами подпрограмм в языке Verilog являются более глубокими, чем в Pascal, и не ограничиваются различиями в способах обмена информацией с основной программой.

К ограничениям, соблюдающимся при создании функций в языке Verilog, относятся следующие:

• изнутри функций допускается вызов других функций, но не допускается вызов задач;

• функции всегда должны выполняться мгновенно, за нулевой промежуток модельного времени;

• функции не могут содержать никаких операторов задержки или контроля событий;

• функции всегда возвращают в качестве результата только одно значение, а список их формальных параметров не может содержать аргументов типа **output** или **inout**.

Задачи лишены перечисленных ограничений и позволяют вызывать как функции, так и другие задачи, управлять задержками и осуществлять контроль событий, а также могут иметь произвольное число входных, выходных и двунаправленных параметров.

Общие для задач и функций свойства состоят в следующем:

• могут использоваться локальные переменные и регистры целого и действительного типов, а также типа **time**;

• нельзя применять локальные цепи (**wires**);

• могут использоваться только поведенческие операторы и вызываться из других блоков поведенческих операторов (**always**, **initial** и пр.). Функции также могут быть вызваны из операторов непрерывного присваивания **assign**.

Задачи (**task**) предназначены для выполнения каких-либо общих процедур в нескольких различных местах описания и разбиения больших участков кода на малые. Из приведенного далее примера можно видеть, что по синтаксису и способу описания интерфейса задачи подобны модулям:

```
task <идентификатор задачи>;
        <описание интерфейса>;
        begin
                <процедурные присваивания/операторы>;
        end
endtask
```

Описание интерфейса определяет направленность и порядок следования портов (список формальных параметров задачи). Типы портов должны быть определены внутри задачи.

Пример 21. Использование конструкции task.

```
task example;
//описание интерфейса
input I1, I2, I3;
output O1, O2, O3;
//описание задачи
begin
        if (I1 && !I2 && !I3) {O1, O2, O3} = 3'b100;
        else if (!I1 && I2 && !I3) {O1, O2, O3} = 3'b010;
        else if (!I1 && !I2 && I3) {O1, O2, O3} = 3'b001;
        else {O1, O2, O3} = 3'b000;
end
endtask
```

Вызов задач осуществляется изнутри поведенческих блоков путем указания их идентификаторов и перечня (в круглых скобках) фактических параметров. При этом фактические параметры должны быть перечислены в том же порядке, что и соответствующие им формальные параметры в разделе описания интерфейса. Вызов задачи в программе не обязательно должен следовать ниже текста описания самой задачи.

Так как Verilog не имеет конструкций, аналогичных модулям в языке Pascal, библиотекам в языках C/C++ или пакетам в VHDL, то для подключения к программе внешних задач следует пользоваться системной функцией **$include**.

Например, если существует необходимость включения в текущий модуль задачи, исходный текст которой размещен в файле scenario.v, то следует в произвольном месте модуля поместить команду:

$include scenario.v

Функции (function). Для создания подпрограммы-функции необходимо воспользоваться парой ключевых слов **function** и **endfunction**. Описание функции может размещаться в произвольном месте модуля, вызывающего данную функцию. Обязательным условием объявления функции в языке Verilog является наличие хотя бы одного входного параметра (аргумента). Для определения возвращаемого функцией результата следует разместить в ее теле оператор присваивания, в котором имя изменяемого сигнала совпадает с именем текущей функции:

function <тип результата> <идентификатор функции>;
<перечень входных параметров>;
//должен присутствовать как минимум один входной параметр
begin
    <процедурные присваивания/операторы>;
    //определение возвращаемого функцией результата
    <идентификатор функции> = <возвращаемый результат>;
end
endfunction

Приведем в качестве примера функцию, определяющую большее из двух беззнаковых целых чисел.

Пример 22. Использование конструкции function.

`timescale 10 ns/1ps
module Function_Demo (Res, X1, X2);
input X1, X2;
output Res;
wire [15:0] Res;
wire X1, X2;
//заголовок функции:
function min;
input [15:0] A,B; //два входных параметра

```
//тело функции
begin
    if (A<B) min = A; //в качестве результата возвращается сигнал A,
    else min = B; // в противном случае – сигнал B
    end
endfunction
//вызов функции в операторе непрерывного присваивания:
assign Res = min (X1,X2);
endmodule
```

Подключение библиотечных функций, как и библиотечных задач, осуществляется с применением системной функции **$include**.

Часть 2. Интерфейс и основные настройки САПР Quartus II

САПР Quartus II имеет богатый графический пользовательский интерфейс, который может быть настроен в зависимости от желаний и потребностей разработчика.

При стандартных настройках главное окно САПР Quartus II (рис.2.1) включает несколько областей:

1 - заголовок окна (отображает название проекта и его рабочую папку);

2 - главное меню (предоставляет доступ ко всем функциям САПР);

3 - панель инструментов (осуществляет доступ к наиболее используемым функциям САПР);

4 - навигатор проекта (Project Navigator) (отображает иерархию проекта, файлы проекта и команды быстрого запуска);

5 - окно состояния процедуры компиляции проекта (Tasks) (содержит информацию о стадиях и этапах компилирования проекта);

6 - окно сообщений (Messages) (выводит информацию о выполнении операций, ошибках и предупреждениях);

7 - рабочая область (Workspace) (отображает файлы, отчеты и другие элементы проекта).

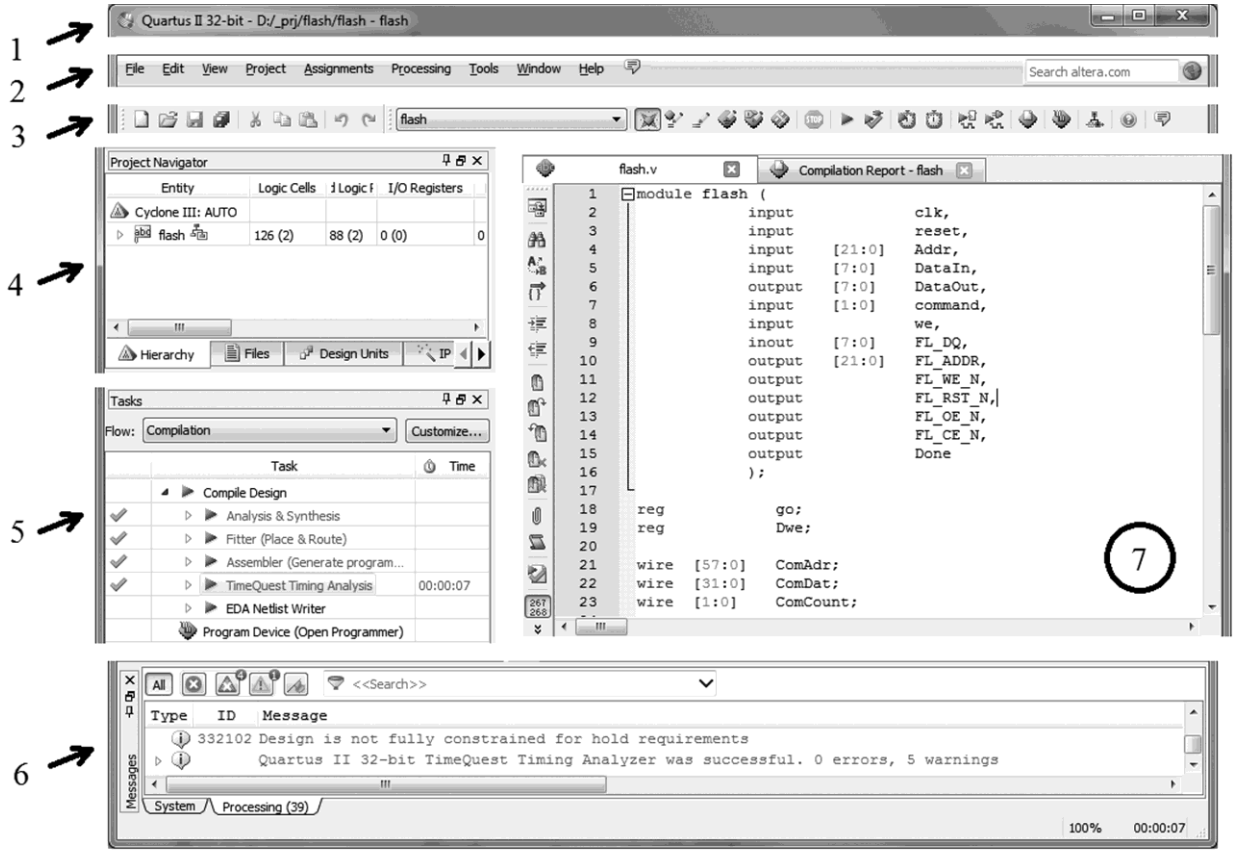


Рис.2.1. Главное окно САПР Quartus II

2.1. Навигация по проекту

Под проектом в САПР Quartus II понимается набор файлов, связанных с конкретным проектом, которые делят на две группы:

1) логические файлы (Design Files), описывающие алгоритм функционирования устройства;

2) вспомогательные файлы (Ancillary Files).

Проект может содержать один либо несколько логических файлов, образующих иерархическое описание создаваемого модуля. При иерархическом описании среди множества логических файлов различают:

• файл верхнего уровня (Top-level Design File);

• файлы нижних (одного или нескольких) уровней (Low-level Design Files).

В файле верхнего уровня иерархии задается структура проекта и определяются набор входящих в его состав модулей и их взаимосвязи. Описания этих модулей содержатся в логических файлах более низкого уровня иерархии. В их состав, в свою очередь, в виде компонентов также могут входить модули, описания которых приведены в логических файлах еще более низкого уровня иерархии, и т.д.

Имя проекта должно совпадать с именем модуля верхнего уровня, а следовательно, с именем логического файла, в котором хранится его описание. Имена модулей нижних уровней иерархии, в свою очередь, должны совпадать с именами файлов, в которых они описаны.

Навигатор проекта (Project Navigator) обеспечивает прямой визуальный доступ к ключевой информации и содержит представление иерархии проекта, файлы, связанные с текущим проектом, IP-ядра для быстрого доступа к различным командам меню и функциям, которые недоступны из других меню.

Окно Project Navigator имеет несколько вкладок.

Вкладка Hierarchy (рис.2.2) отражает иерархическую структуру разрабатываемого проекта в виде таблицы, в крайней левой колонке которой находятся имена используемых модулей, а в остальных дополнительная информация о каждом из них, включающая, например, такие поля, как Logic Cells (Количество занимаемых на кристалле логических ячеек), Pins (Количество используемых портов ввода/вывода). Слева от имени модуля верхнего уровня расположен элемент управления - ▷, по нажатию на который открывается список подключенных модулей следующих уровней иерархии.

Вкладка Files содержит перечень всех подключенных к проекту файлов, таких как: Verilog и VHDL-описания, файлы временных и блочных диаграмм и т.п., что упрощает

навигацию по проекту. Вызов контекстного меню в пустом месте вкладки Files обеспечивает возможность добавлять или удалять необходимые файлы в проект.

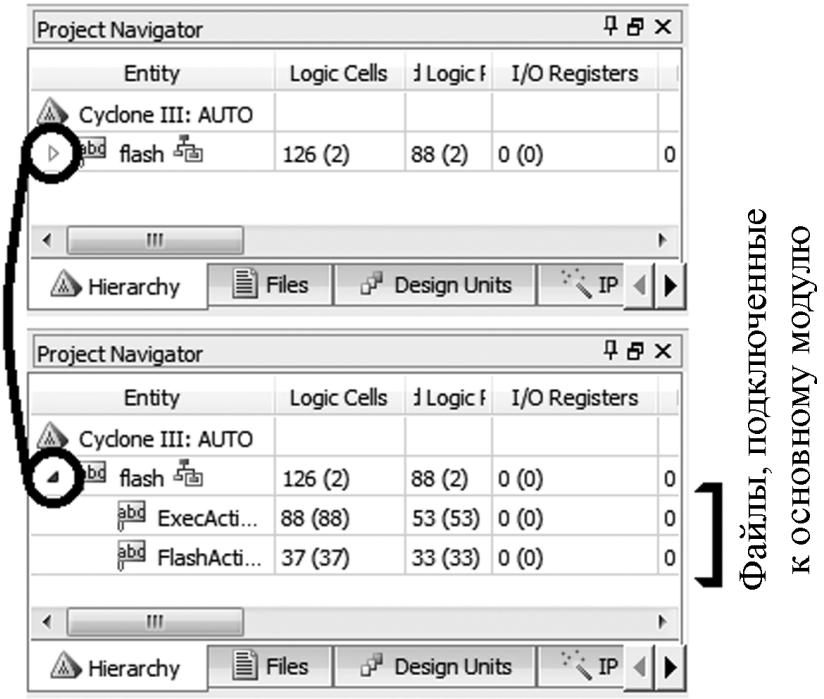


Рис.2.2. Вкладка Hierarchy окна Project Navigator

Доступ к настройкам проекта можно получить с помощью комбинации клавиш Ctrl+Shift+E, либо через меню Assignments -> Settings…, либо нажатием правой кнопки мыши на заглавном модуле и выбором пункта Settings во вкладке Hierarchy.

Все настройки разбиты по категориям и представлены в иерархическом виде в окне настроек слева. Прокомментируем некоторые из них.

Категория Compilation process settings включает настройки процесса компиляции.

Настройка Use Smart Compilation, например, позволяет компилятору осуществить так называемую «разумную» компиляцию, после чего следующие компиляции могут выполняться быстрее. Во время «разумной» компиляции компилятор определяет, какие модули требуются для текущей обработки проекта. При этом он основывается на том, какие изменения были внесены в проект после предыдущей «разумной» компиляции, и пропускает неизмененные модули. Если внести любые изменения в логику проекта, компилятор использует все модули. По умолчанию эта опция выключена.

Настройка Incremental Synthesis Only включает инкрементальный синтез, при котором заново синтезируются только заданные разделы проекта. При этом уменьшаются время синтеза и использование памяти. Эту опцию можно включить следующим образом:

• открыть Project Navigator;

• нажать правой кнопкой мыши на имя блока;

• выбрать Set as Design Partition;

• в появившемся диалоге выбрать Incremental synthesis only.

Категория Analysis & Synthesis settings содержит настройки, касающиеся анализа и синтеза проекта, в том числе параметры моделирования временных диаграмм.

2.2. Компиляция проекта

Компилятор САПР Quartus II состоит из подпрограмм и утилит, выполняющих ряд функций и охватывающих несколько ступеней маршрута проектирования:

• синтез;

• проверка проекта на наличие ошибок (верификация);

• размещение и трассировка проекта в кристалле;

• генерация выходных файлов для моделирования проекта и анализ временных характеристик;

• программирование ПЛИС.

Компиляцию проекта, т.е. ход выполнения текущей задачи, можно наблюдать в окне Tasks.

Сначала из проекта извлекается информация об иерархических связях между составляющими его файлами, описание проекта проверяется на наличие ошибок. Затем создается организационная карта проекта, все файлы преобразуются в единую базу данных, с которой впоследствии работает система.

Компиляция может выполняться с учетом заданных требований, к которым относятся:

• обеспечение необходимых временных характеристик проекта;

• увеличение быстродействия;

• оптимизация используемых ресурсов ПЛИС.

Компилятор создает файлы для программирования и конфигурирования ПЛИС фирмы Altera. Промежуточные и окончательные результаты компиляции в системе Quartus II можно посмотреть в окне отчета о компиляции Compilation Report.

При создании нового проекта система Quartus II по умолчанию устанавливает значения всех необходимых параметров. Параметры, заданные по умолчанию, можно переопределить в соответствии с поставленными требованиями. Кроме того, есть возможность выбора различных параметров настройки непосредственно при выполнении компиляции.

Подпрограмма анализа и синтеза (Analysis & Synthesis) проекта проверяет файлы на ошибки и затем строит базу данных, которая интегрирует все файлы в иерархию. Кроме того, подпрограмма синтезирует и оптимизирует проект. В конце разработка приводится в технологическое соответствие устройству, в котором она должна быть запрограммирована.

Подпрограмма размещения и трассировки (Fitter (Place & Route)), или так называемый «сборщик», осуществляет монтаж проекта в структуру выбранного кристалла программируемой логики. Другими словами, полученная на этапе синтеза модель полного представления проекта в техническом базисе кристалла отображается на внутренние ресурсы ПЛИС (которыми являются конфигурируемые логические блоки, блоки встроенной памяти, встроенные умножители) и устанавливаются соответствующие соединения с помощью ресурсов трассировки кристалла. Подпрограмма размещения и трассировки подбирает для каждой логической функции подходящее место на кристалле с точки зрения уменьшения времени распространения сигнала, выполняет соответствующие соединения и назначения контактов ввода/вывода.

На этапе размещения и трассировки пользователь может задать свои собственные назначения, после чего подпрограмма реализует их, а затем выполнит оптимизацию оставшейся части проекта.

После монтажа пользователь может просмотреть результаты размещения и трассировки с помощью специального средства Chip Planner, доступного в меню Tools, и в случае необходимости изменить некоторые назначения.

Ассемблер (Assembler (Generate Programming Files)) завершает обработку проекта, превращая то, что сгенерировал «сборщик», в образ для программирования устройства в форме одного или нескольких файлов.

Подпрограмма временного анализа (TimeQuest Timing Analysis) анализирует, отлаживает и утверждает временную производительность всей логики в дизайне. Прежде чем осуществить временной анализ, необходимо выполнить анализ и синтез, а также запустить «сборщик».

2.3. Отчет о результатах компиляции

Информация о проведенной компиляции проекта выводится в окне отчета о компиляции (Compilation Report) (рис.2.3).

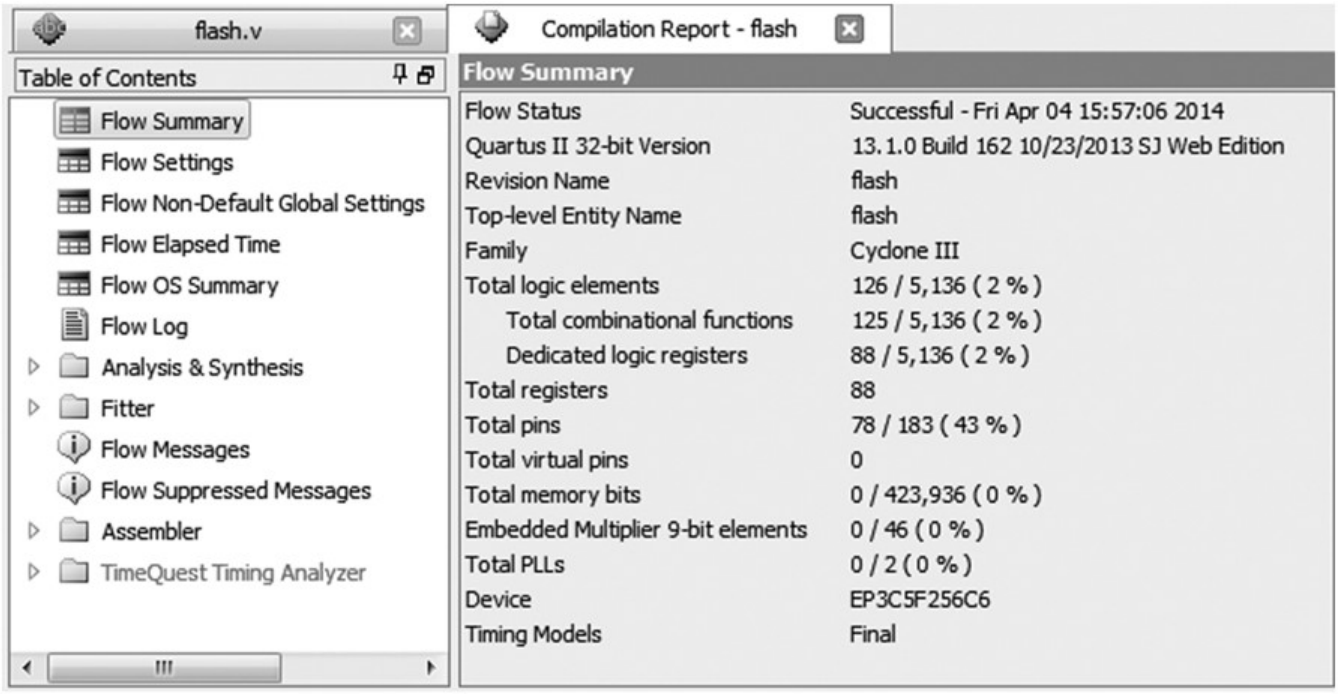


Рис. 2.3. Окно Compilation Report

В правой части окна непосредственно после завершения компиляции появляется сводный отчет (Flow Summary), который содержит информацию о дате и времени проведения компиляции, версии системы Quartus II, имени файла верхнего уровня проекта, семействе и типе используемой ПЛИС, типе компиляции (промежуточная или окончательная), степени удовлетворения проекта заданным временным параметрам и количестве использованных логических блоков, выводов и ячеек памяти.

В левой части окна размещается организованный по иерархическому принципу каталог сообщений отдельных блоков компилятора. Здесь собрана подробная информация обо всех этапах компиляции, включая предварительно сделанные установки и результаты работы конкретных блоков компилятора. Каждая отдельная утилита, входящая в состав компилятора, обновляет информацию в своей директории.

Отчет Flow Summary содержит информацию о количестве используемых логических элементов (Total logic elements), объеме используемой памяти (Total memory bits), количестве применяемых встроенных умножителей (Embedded multiplier 9-bit elements), количестве задействованных портов ввода/вывода (Total pins) и др.

В разделе Resource Section (в папке Fitter) в таблице Resource Utilization by Entity содержится информация об используемых ресурсах, таких как блоки умножения (mult) и блоки памяти (ram).

В разделе Control Signals находятся данные об управляющих сигналах и их коэффициентах разветвления.

2.4. Сообщения

Во время компиляции в окне процессора сообщений (Messages) появляются информационные сообщения, сообщения об ошибках, предупреждения, сделанные компилятором в ходе анализа проекта. Эти записи относятся к определенному месту в файле проекта или в другом исходном файле.

Для того чтобы локализовать часть проекта, являющуюся источником ошибки, необходимо раскрыть соответствующее сообщение, щелкнув по значку «+», далее два раза щелкнуть по интересующему вас сообщению, после чего в окне редактора появится файл, к которому относится данное сообщение. Кроме того, цветом будет выделен непосредственный источник сообщения.

2.5. Дополнительные инструменты разработки

Дополнительные инструменты разработки находятся в меню Tools.

Для просмотра внутреннего представления проекта используется средство Netlist Viewer. С его помощью можно просматривать реализацию проекта на программируемом кристалле на разных уровнях абстракции.

Technology Map Viewer позволяет просматривать технологическую карту проекта, т.е. его реализацию с использованием таких ресурсов кристалла, как функциональные преобразователи (LUT) и триггеры.

Для просмотра результатов компиляции модулей проекта, описанных в виде конечного автомата, предназначено средство State Machine Viewer.

Для просмотра и анализа результатов компиляции проекта служит средство Chip Planner, для внесения изменений в результаты компиляции - средство Resource Property Editor.

С помощью RTL Viewer (рис.2.4) можно просматривать результаты компиляции проекта на уровне регистровых передач. Утилита RTL Viewer позволяет представить логическую реализацию проекта в графическом виде. Это очень полезный инструмент для анализа результатов синтеза HDL-проектов.

В левой части окна RTL Viewer находится панель навигации по списку соединений (Netlist Navigator), которая отображает иерархию проекта, включающую модули, примитивы, порты ввода/вывода, конечные автоматы и провода. По нажатию на экземпляр в списке иерархии он подсвечивается в схематичном изображении в правой части окна RTL Viewer. По двойному щелчку открывается его внутренняя логика, например, в качестве вкладки.

Контекстное меню всех элементов схемы содержит поле свойств (Properties). Его выбор в левой части окна делает доступной вкладку Properties, которая содержит информацию о свойствах данного элемента: названиях портов ввода/вывода и их параметрах.

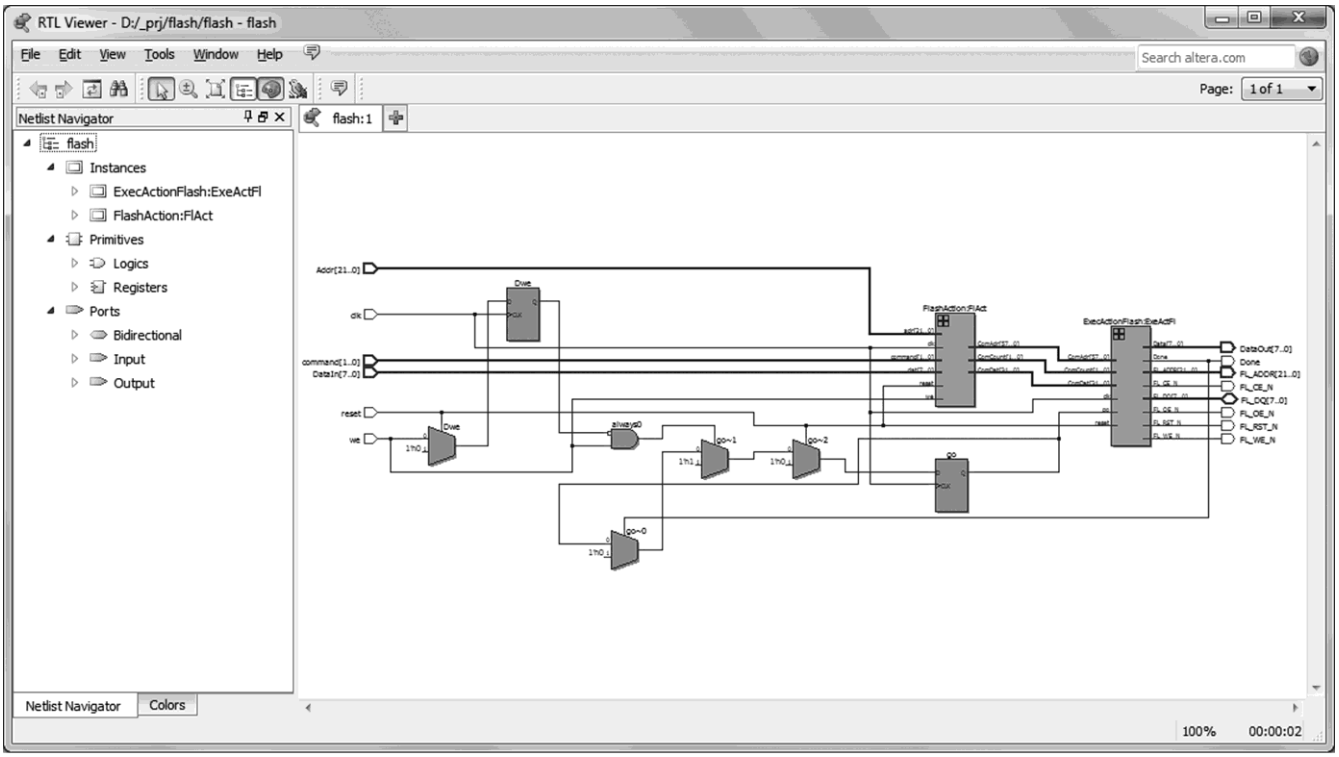


Рис.2.4. Окно RTL Viewer

2.6. Моделирование проекта

Современные ПЛИС становятся все более сложными и производительными, выдвигая перед разработчиками задачу моделирования и проверки работоспособности (верификации) проектов. Моделирование позволяет проверять правильность работы всего проекта без физической отладки, требующей много времени.

Программное обеспечение Quartus II содержит в своем составе симулятор, который может решать следующие задачи:

• функциональное моделирование проекта;

• временное моделирование;

• отладку проекта.

Функциональное моделирование проекта позволяет проверить правильность работы схемы с точки зрения логики и схемотехники. Функциональное моделирование возможно на любой стадии работы, если проект логически завершен и не содержит ошибок.

Входными файлами моделирования являются рабочие файлы проекта (Verilog- либо VHDL-описания или графические схемы) и файлы тестовых воздействий (input stimulus).

Тестовые воздействия содержат входные сигналы, которые, будучи поданными на разрабатываемое устройство, порождают ожидаемую однозначную реакцию со стороны схемы. Сравнение записанной реакции с теоретическими расчетами позволяет делать выводы о работоспособности схемы и локализовывать возможные ошибки.

Для запуска функционального моделирования необходимо создать файл тестовых воздействий и привязать его к проекту. Для привязки файла надо войти в настройки проекта (пункт меню File → Settings). Далее в списке категорий выбрать Simulator Settings. Затем в списке режимов (Simulation mode) выбрать режим функционального моделирования (Functional). В поле Simulation Input выбрать файл тестовых воздействий. После чего нажать кнопку OK и запустить моделирование (меню Processing → Start Simulation).

Временное моделирование отличается от функционального тем, что учитывает задержки в распространении сигналов при работе реальной ПЛИС. Оно позволяет выявить наиболее проблемные места схемы с точки зрения быстродействия и показывает, на какую частоту работы можно рассчитывать при физическом запуске схемы в ПЛИС. Для проведения временного моделирования проект должен быть скомпилирован (пройдены этапы синтеза и размещения и трассировки).

Временное моделирование запускается так же, как и функциональное, за исключением того, что в настройках проекта необходимо выбрать режим временного моделирования (Timing).

Встроенный режим отладки позволяет устанавливать точки останова (breakpoints), тем самым приостанавливая моделирование в нужное время или при наступлении требуемого условия на сигнале или шине. Для создания точек останова необходимо выбрать пункт меню Processing → Simulation Debug → Breakpoints. В появившемся диалоговом окне следует ввести логическое условие (Equation → Condition) и действие (Action), предпринимаемое симулятором при наступлении этого условия. Среди возможных условий есть остановка (Stop), предупреждение (Warning Message), сообщение об ошибке (Error Message) и информационное сообщение (Information Message).

2.7. Редактор тестовых воздействий

Для создания файла тестовых воздействий в виде временной диаграммы необходимо выбрать пункт меню File → New. Далее в появившемся списке выбрать Vector Waveform File (.vwf-файл). При этом откроется окно редактора тестовых воздействий (рис.2.5). Основными элементами этого окна являются:

1) панель инструментов;

2) шкала времени;

3) маркеры времени (основной и дополнительный);

4) поле названий сигналов;

5) графики сигналов.

В пустую временную диаграмму необходимо вставить сигналы и шины. Для этого следует выбрать пункт меню Edit → Insert и нажать Insert Node or Bus. Откроется окно Insert Node or Bus. Можно задать имя и тип вставляемого сигнала вручную (поля Name и Type) либо воспользоваться автоматизированным поиском (кнопка Node Finder).

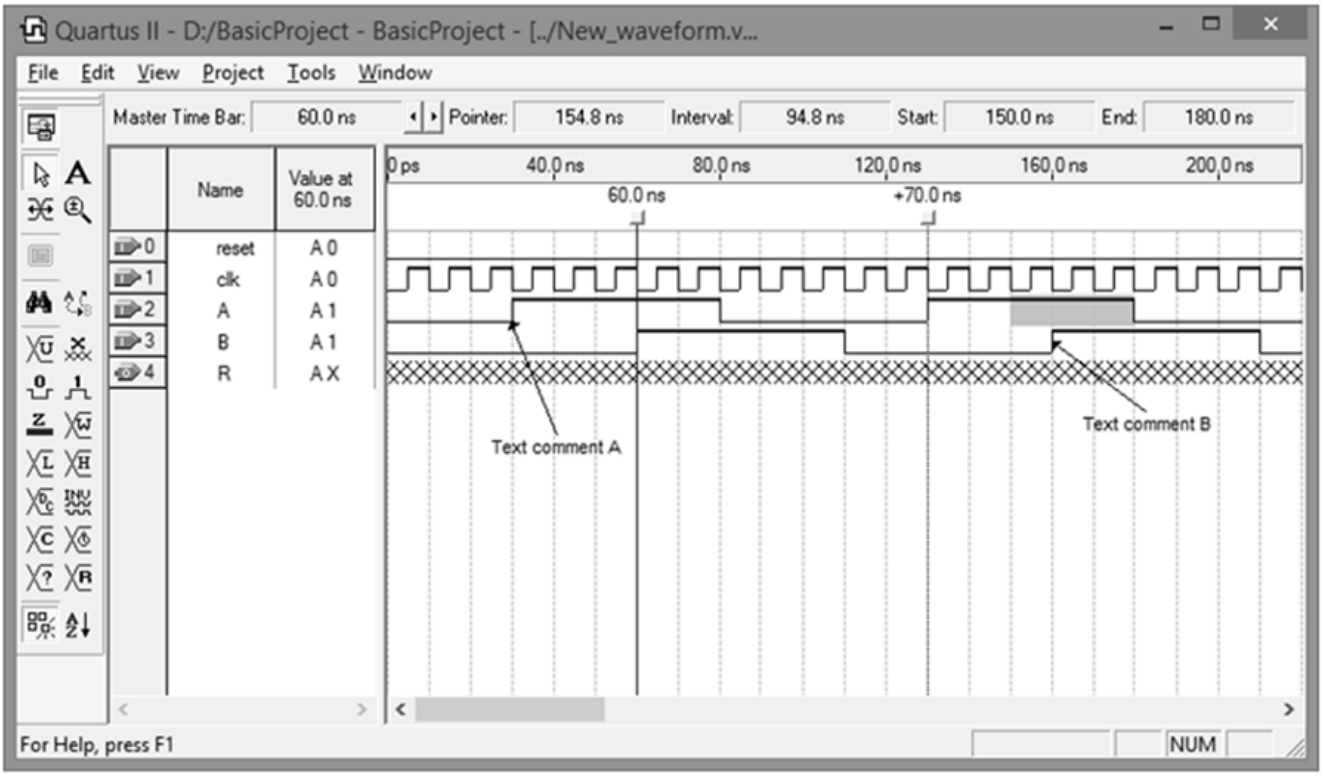


Рис.2.5. Редактор тестовых воздействий

Окно Node Finder (рис.2.6) позволяет найти сигнал или шину в проекте с помощью трех фильтров поиска:

• Named (имя сигнала; символ * обозначает любую комбинацию символов);

• Filter (задание типа цепи: входы/выходы блока (Pins), внутренние регистры блока (Registers), любые цепи, имеющиеся в проекте);

• Look in (поиск цепи в конкретном модуле проекта).

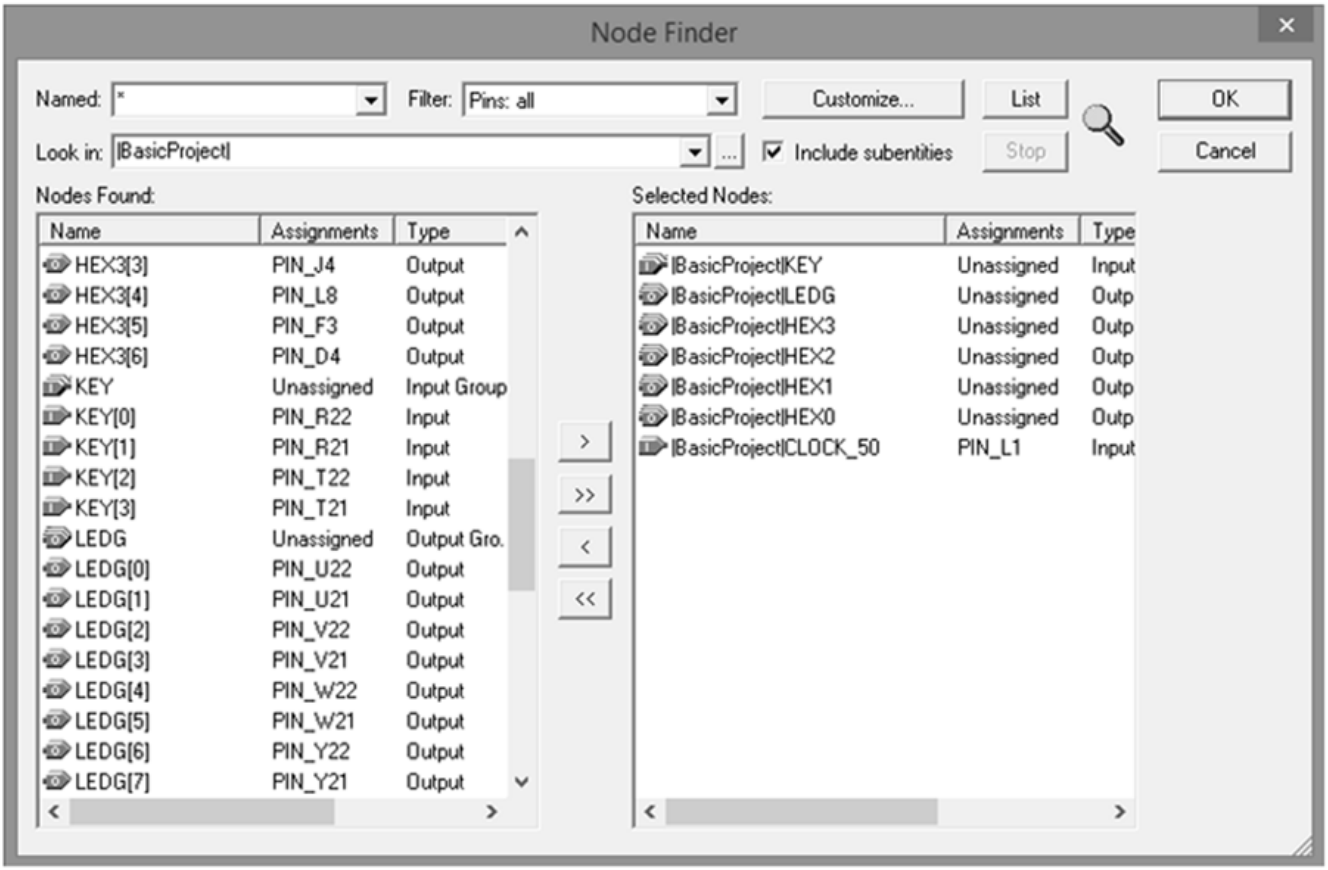


Рис.2.6. Окно Node Finder

После ввода ключевых слов фильтра необходимо нажать на кнопку List для вывода результатов поиска. Затем следует выбрать интересующие цепи в поле Nodes Found и нажатием кнопки «>» перенести их в правую часть окна. Кнопка «>>» переносит все найденные цепи в правую часть. После нажатия кнопки OK все цепи, перенесенные в правую часть окна, будут добавлены на временную диаграмму.

Добавленные цепи отобразятся в левой части окна редактирования. Для удобства они могут быть сгруппированы в шины. Для этого надо выделить нужные цепи и выбрать пункт меню Edit → Grouping → Group. Возможен и обратный процесс – разгруппировка, для этого необходимо выбрать пункт Edit → Grouping → Ungroup.

Следующим этапом является редактирование временных диаграмм входных сигналов. Для редактирования сигнала следует выделить его часть (или весь сигнал целиком) курсором и установить для нее одно из значений, выбрав соответствующий пункт меню Edit → Value.

Основные пункты этого меню продублированы на панели инструментов редактора и горячими клавишами:

• Forcing Low (Ctrl + Alt + 0) - выставляет логический нуль на выделенном участке сигнала;

• Forcing High (Ctrl + Alt + 1) - выставляет логическую единицу на выделенном участке сигнала;

• Count Value (Ctrl + Alt + V) - создает счетчик на выделенном участке шины, значение которого увеличивается на заданную величину через равные промежутки времени. В настройках счетчика есть возможность задавать стартовое значение и значение, после которого счет должен прекратиться;

• Overwrite Clock (Ctrl + Alt + K) - задает периодический прямоугольный сигнал (меандр) на выделенном участке. В настройках меандра можно указать период сигнала и его скважность;

• Arbitrary Value (Ctrl + Alt + B) - выставляет постоянное значение на выделенном участке сигнала. В настройках можно выбрать желаемую систему счисления для ввода и отображения данных (Binary, Octal, Decimal, Hexadecimal);

• Random Data (Ctrl + Alt + R) - заполняет выделенный участок сигнала случайными данными. В настройках есть возможность указать период изменения данных с привязкой или без привязки к текущей временной сетке.

Для изменения масштаба временной диаграммы на панели инструментов присутствует кнопка масштабирования (Zoom Tool). При выборе этой кнопки левый щелчок мышью по полю сигналов увеличивает масштаб, а правый - уменьшает.

Кнопка вставки текста (Text Tool) позволяет вставлять текстовые комментарии в поле сигналов с привязкой ко времени. Такие комментарии могут быть полезными при анализе временных диаграмм и обсуждении результатов моделирования с другими разработчиками.

В верхней части окна редактирования отображена шкала времени моделирования. Значения Start и Stop - это начальное и конечное время моделирования соответственно. Для изменения конечного времени необходимо выбрать пункт меню Edit → End Time и ввести желаемое время. Поле Pointer показывает время от старта до курсора. Поле Interval показывает время от основного маркера до курсора.

Для удобства работы сигналы и измерения привязываются к временной сетке. Для того чтобы ее изменить, необходимо выбрать пункт меню Edit → Grid Size, где выбрать желаемый период сетки.

Для измерения временных интервалов на диаграмме доступны несколько временных маркеров (Time Bar). Дополнительные маркеры можно создавать с помощью пункта меню Edit → Time Bar → Insert Time Bar.

2.8. Отчет о результатах моделирования

Результаты моделирования проекта выводятся в окне отчета моделирования (рис.2.7). Информация в отчете моделирования позволяет проверить отсутствие ошибок и временные параметры проекта.

Отчет моделирования состоит из отчетов о покрытии моделирования и результирующей временной диаграммы.

Результирующая временная диаграмма включает созданные пользователем тестовые воздействия и реакции схемы на них. При попытке редактирования диаграммы появится окно, в котором будет предложено перейти к редактированию файла тестовых воздействий.

Встроенный симулятор Quartus II имеет возможность сравнения текущего результата моделирования с предыдущим результатом. Для сравнения двух диаграмм необходимо включить параметр Check outputs в настройках моделирования (Simulator Settings). Если эта опция включена, то на экране будут одновременно отображены две диаграммы: оригинальная (черным цветом) и сравниваемая (красным цветом).

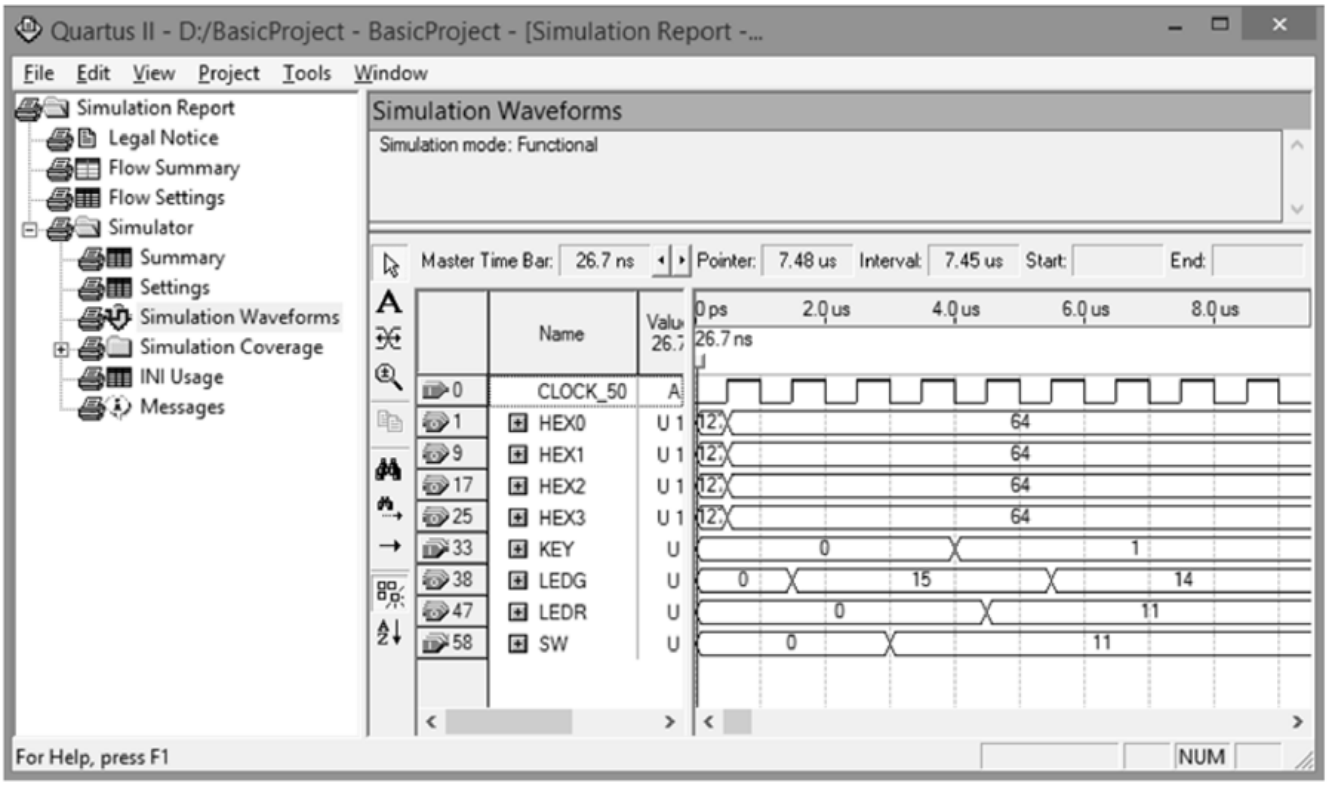


Рис.2.7. Окно отчета моделирования

2.9. Редактор назначения выводов

Редактор назначения выводов (Assigments Editor) - это табличный интерфейс, позволяющий создавать и редактировать соответствия входных/выходных сигналов проекта физическим выводам ПЛИС.

Существует два метода назначения выводов. Первый метод - выбор вывода микросхемы из списка всех возможных выводов для конкретной ПЛИС и назначение ему сигнала проекта. Второй метод - обратный: выбор сигнала проекта из списка и назначение ему вывода микросхемы. Редактор поддерживает оба метода и частично автоматизирует работу с помощью автозаполнения названий выводов и автоматической нумерации сигналов шин.

Для запуска редактора необходимо выбрать пункт меню Assigments → Assigment Editor, в результате откроется окно, представленное на рис.2.8. Основными элементами этого окна являются:

1) панель меню;

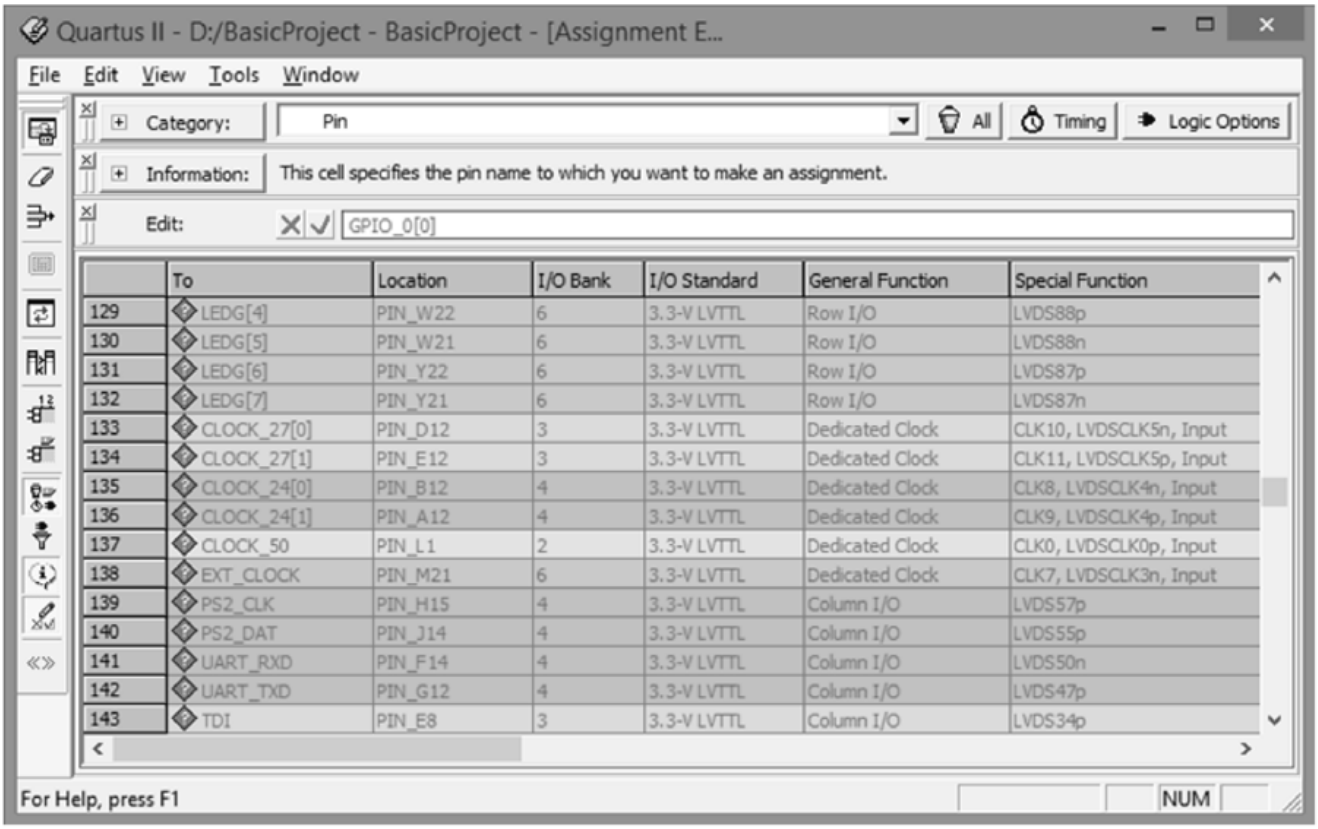
2) панель инструментов;

3) панель выбора и редактирования;

4) таблица выводов и назначений.

Для начала работы следует выбрать вариант Pin из списка Category. В зависимости от используемого метода назначения выводов надо включить отображение всех неназначенных выводов (выбрать пункт меню View → Show All Assignebal Pin Numbers) или всех сигналов проекта (Show All Known Pin Names). В таблице выводов и назначений в колонке To отображаются названия сигналов проекта, а в колонке Location - выводы ПЛИС. Остальные колонки таблицы отвечают за дополнительные настройки выводов, такие как стандарт сигнала (ТТЛ, КМОП и т.п.) и функция вывода (дифференциальная или одиночная линия и т.п.).

Каждому входному и выходному сигналу проекта необходимо поставить в соответствие вывод микросхемы (вида PIN_AA3) в таблице назначений. При вводе редактор назначений будет автоматически предлагать правильные варианты ввода. Другими словами, достаточно ввести AA3, и редактор сам дополнит это название до правильного PIN_AA3 в соответствии с типом корпуса ПЛИС. Точно такое же автодополнение действует и при вводе сигналов проекта.



	To	Location	I/O Bank	I/O Standard	General Function	Special Function
129	LEDG[4]	PIN_W22	6	3.3-V LVTTL	Row I/O	LVDS88p
130	LEDG[5]	PIN_W21	6	3.3-V LVTTL	Row I/O	LVDS88n
131	LEDG[6]	PIN_Y22	6	3.3-V LVTTL	Row I/O	LVDS87p
132	LEDG[7]	PIN_Y21	6	3.3-V LVTTL	Row I/O	LVDS87n
133	CLOCK_27[0]	PIN_D12	3	3.3-V LVTTL	Dedicated Clock	CLK10, LVDSCLK5n, Input
134	CLOCK_27[1]	PIN_E12	3	3.3-V LVTTL	Dedicated Clock	CLK11, LVDSCLK5p, Input
135	CLOCK_24[0]	PIN_B12	4	3.3-V LVTTL	Dedicated Clock	CLK8, LVDSCLK4n, Input
136	CLOCK_24[1]	PIN_A12	4	3.3-V LVTTL	Dedicated Clock	CLK9, LVDSCLK4p, Input
137	CLOCK_50	PIN_L1	2	3.3-V LVTTL	Dedicated Clock	CLK0, LVDSCLK0p, Input
138	EXT_CLOCK	PIN_M21	6	3.3-V LVTTL	Dedicated Clock	CLK7, LVDSCLK3n, Input
139	PS2_CLK	PIN_H15	4	3.3-V LVTTL	Column I/O	LVDS57p
140	PS2_DAT	PIN_J14	4	3.3-V LVTTL	Column I/O	LVDS55p
141	UART_RXD	PIN_F14	4	3.3-V LVTTL	Column I/O	LVDS50n
142	UART_TXD	PIN_G12	4	3.3-V LVTTL	Column I/O	LVDS47p
143	TDI	PIN_E8	3	3.3-V LVTTL	Column I/O	LVDS34p

Рис.2.8. Окно редактора назначения выводов

После заполнения таблицы выводов и назначений и закрытия редактора в проект будет добавлен файл назначений (.qsf), содержащий все записи о выводах ПЛИС. Этот файл будет использоваться в дальнейшем при анализе и компиляции проекта.

2.10. Программирование ПЛИС

В результате успешной компиляции проекта Quartus II создает двоичный файл (.sof) для программирования ПЛИС. Для конфигурирования ПЛИС с помощью этого файла используется встроенный программатор (меню Tools → Programmer), окно которого имеет вид, представленный на рис.2.9.

В верхней части окна располагаются настройки аппаратуры программатора. Quartus II поддерживает программирование с помощью параллельного порта (ByteBlaster), USB (UsbBlaster) и сети Ethernet (EthernetBlaster). Выбор нужного варианта производится в настройках при нажатии кнопки Hardware Setup.

В левой части окна находятся кнопки запуска программирования (Start), остановки (Stop) и управления файлами прошивки: удалить прошивку (Delete), добавить (Add File…), заменить файл (Change File…).

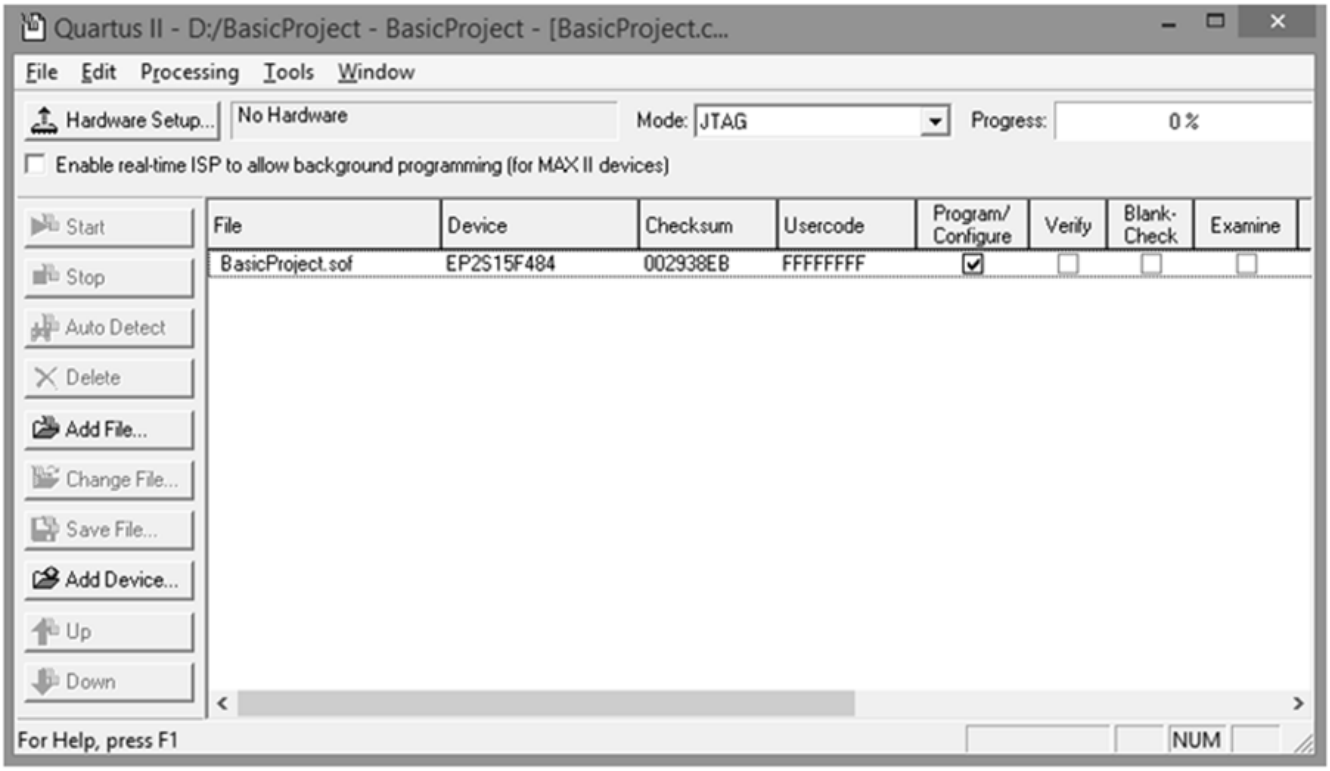


Рис.2.9. Окно программатора

В центральной части окна показана таблица подключенных устройств для программирования с файлами прошивок. Для запуска программирования необходимо добавить в эту таблицу нужную прошивку (файл .sof) и нажать кнопку Start.

Часть 3. Лабораторный практикум

Лабораторная работа № 1

Разработка арифметико-логического устройства

Лабораторное задание

Необходимо разработать и промоделировать работу 4-разрядного АЛУ с последовательной и параллельной организацией арифметического переноса. За схемотехническую основу предлагается взять классическую структуру АЛУ.

Процесс выполнения задания можно разделить на следующие этапы:

1) создание в САПР Quartus II;

2) разработка Verilog-описания конфигурируемого сумматора SM;

3) проверка работы сумматора SM в симуляторе;

4) создание примитива (символа) сумматора SM;

5) сборка схемы АЛУ с последовательным переносом;

6) проверка работы АЛУ с последовательным переносом в симуляторе;

7) разработка HDL-описания схемы ускоренного переноса (СУП) и сборка схемы АЛУ с параллельным переносом;

8) проверка работы АЛУ с параллельным переносом в симуляторе.

Структура АЛУ

На рис.1 представлена схема 4-разрядного АЛУ с последовательной организацией арифметического переноса.

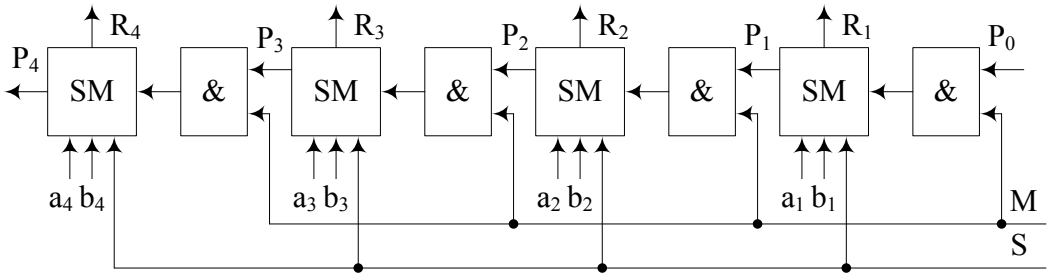


Рис.1. Схема АЛУ с последовательной организацией арифметического переноса на примере 4-разрядной секции

Основу данной схемы составляет одноразрядный комбинационный сумматор SM, который изменяет свою конфигурацию в зависимости от управляющего слова S. Управляющее слово определяет тип операции (сложение, вычитание и т.д.). Сигнал M (модификатор) обеспечивает разделение арифметических и логических операций: при $M=1$ выполняются арифметические операции, при $M=0$ - логические. P_i - бит арифметического переноса; a_i; b_i - разряды операндов A, B; R_i - разряды результата.

За основу работы сумматора SM возьмем выражение

$$R_i = (S_3 a_i b_i + S_2 a_i \bar{b}_i) \oplus (a_i + S_1 \overline{b_i} + S_0 b_i) \oplus M P_{i-1}, \qquad (1)$$

где символы $\oplus$, $+$ означают сложение по модулю 2, логическую операцию ИЛИ соответственно.

Заметим, что 4-разрядное управляющее слово S и модификатор M позволяют закодировать 16 логических и 16 арифметических операций; учитывая, что P_0 может принимать два значения, получаем 32 арифметические операции. Таким образом, АЛУ с описанной выше структурой может выполнять 48 операций.

В табл.1 приведены командные слова и значения модификатора для четырех основных операций: арифметических сложения и вычитания и логических операций И, ИЛИ.

Таблица 1

Командные слова для основных операций АЛУ

Операция	S_3	S_2	S_1	S_0	M	P_0
Вычитание	0	1	1	0	1	1
Сложение	1	0	0	1	1	0
Или	0	0	0	1	0	x
И	0	1	0	0	0	x

Введем следующие обозначения для подготовительных функций нулевого порядка:

$$D_i = S_3 \cdot a_i \cdot b_i + S_2 \cdot a_i \cdot \bar{b}_i;$$

$$F_i = a_i + S_1 \cdot \overline{b_i} + S_0 \cdot b_i.$$

Схема с последовательным переносом используется, когда не требуется высокое быстродействие, но предъявляются жесткие требования к аппаратным затратам. В этом случае каждая секция формирует бит переноса

$$P_i = D_i + F_i P_{i-1}. \qquad (2)$$

Схема с параллельным переносом обеспечивает более высокое быстродействие, для этого используется следующая совокупность выражений (для 4-разрядного АЛУ):

$$\begin{cases} P_1 = D_1 + P_0F_1 \\ P_2 = D_2 + P_1F_2 = D_2 + D_1F_2 + P_0F_1F_2 \\ P_3 = D_3 + P_2F_3 = D_3 + D_2F_3 + D_1F_2F_3 + P_0F_1F_2F_3 \\ P_4 = D_4 + P_3F_4 = D_4 + D_3F_4 + D_2F_3F_4 + D_1F_2F_3F_4 + P_0F_1F_2F_3F_4. \end{cases} \quad (3)$$

Из (3) видно, что каждый из битов переноса зависит только от операндов и нулевого бита переноса, т.е. все они могут вычисляться параллельно, за счет этого и достигается увеличение производительности по сравнению с последовательным переносом.

Методические указания

Создание проекта в САПР Quartus II с помощью Project Wizard.

1. Выберите пункт меню File→New Project Wizard.

2. В появившемся окне помощника укажите путь, где будут располагаться проект, имена проекта и файла верхнего уровня.

3. Дважды нажмите кнопку Next (пропустите второй этап создания проекта), в появившемся окне выберите Device Family: Cyclone II, Package: FBGA, Pin Count: 484, Speed Grade: 7, Available Devices: EP2C20F484C7.

4. Нажмите кнопку Finish.

Разработка Verilog-описания конфигурируемого сумматора SM.

1. Выберите пункт меню File→New.

2. В появившемся окне выберите Verilog HDL File и нажмите кнопку OK.

3. В основном окне Quartus появилось окно текстового редактора с пустым неименованным файлом. Создайте в нем Verilog-описание конфигурируемого одноразрядного сумматора SM в соответствии с (1) и (2).

На рис.2 приведено объявление модуля SM, где a, b - биты операндов; S, M - управляющее слово и модификатор; R, D, F - результат и подготовительные функции; Pin, Pout - входной и выходной бит переноса соответственно. Поскольку SM - комбинационная схема, данное описание может состоять как из четырех непрерывных присваиваний (assign), так и из одной или нескольких блочных конструкций always, в которых определяются выходы модуля.

4. Выберите пункт меню File→Save, указав имя файла SM.

```
1   module SM (a, b, S, M, Pin, R, D, F, Pout);
2
3     input            a, b, M, Pin;
4     input    [3:0]   S;
5     output           R, D, F, Pout;
6
7     // Описание комбинационной схемы сумматора
8
9     endmodule
10
```

Рис.2. Заготовка модуля SM

Проверка работы сумматора SM в симуляторе.

1. В окне Project Navigator выберите закладку Files и пункт контекстного меню (правая кнопка мыши) для файла SM.v Set as Top-Level Entity (если окно Project Navigator закрыто, то выберите пункт меню View→Utility Windows→ Project Navigator).

2. Выполните компиляцию проекта, для этого выберите Processing→Start Compilation.

3. Дождитесь сообщения об успешной компиляции или сообщения об ошибках, в случае появления ошибок откорректируйте HDL-описание и повторите компиляцию.

4. Для проверки устройства в симуляторе необходимо задать входные воздействия и указать, какие выходные сигналы будут контролироваться. Для этого выберите пункт меню File→New, в появившемся окне выберите Vector Waveform File и нажмите кнопку OK.

5. Добавьте необходимые сигналы для построения временных диаграмм. Это делается с помощью диалога Insert Node or Bus, который появляется после двойного щелчка мышью в любой строке временной диаграммы. Далее нажмите кнопку Node Finder. В окне Node Finder установите Filter: Pins: all и нажмите кнопку List. После этого в списке Node Found отобразятся названия всех портов модуля SM.v, скопируйте их в список Selected Nodes и нажмите кнопку OK. Затем нажмите кнопку OK в диалоге Insert Node or Bus.

6. Установите время окончания симуляции 30 мкс с помощью команды Edit→End Time, шаг сетки 1 мкс с помощью команды Edit→Grid Size. Чтобы отредактировать временную диаграмму требуемого узла, выделите мышью необходимую ее часть (всю диаграмму узла можно выделить, щелкнув мышью в поле Value нужного узла). Далее с помощью панели инструментов, расположенной слева в редакторе временных диаграмм,

сформируйте необходимые временные диаграммы. Сохраните файл входных воздействий с именем SM.vwf командой File →Save.

7. Привяжите созданный файл входных воздействий к модулю SM.v. Для этого в окне Project Navigator выберите закладку Hierarchy и пункт контекстного меню (правая кнопка мыши) для файла SM.v Settings. В появившемся окне выберите категорию Simulator Settings, укажите SM.vwf в поле Simulation Input и нажмите кнопку OK.

8. Запустите симулятор командой Processing→Start Simulation.

9. Проанализируйте полученные временные диаграммы, в случае необходимости откорректируйте HDL-описание и файл входных воздействий и повторите компиляцию проекта и моделирование.

На рис.3 приведены результаты моделирования данной схемы.

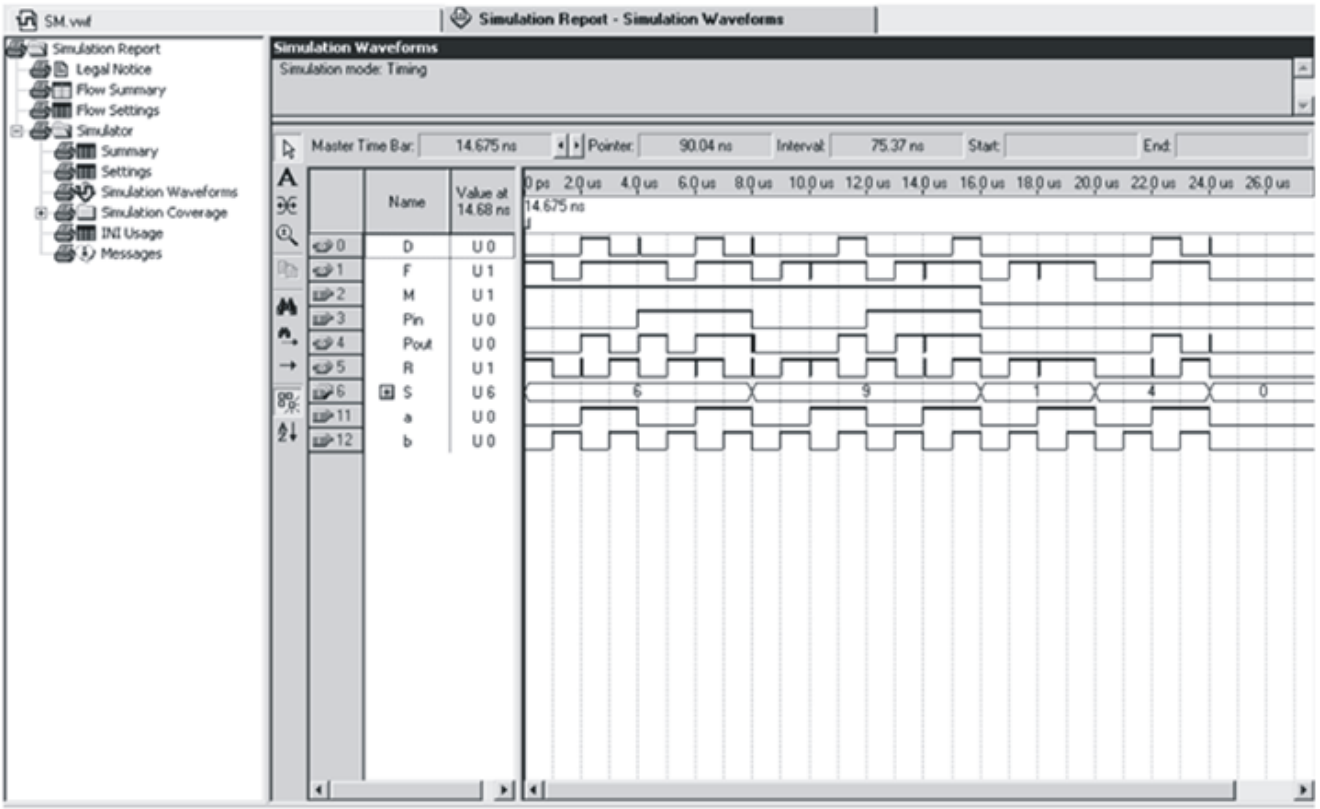


Рис.3. Результаты моделирования модуля SM

Создание примитива (символа) сумматора SM.

Для использования Verilog-модулей в схемотехническом редакторе необходимо создать их схемотехнический символ. Рассмотрим этот процесс на примере модуля SM.

1. Откройте файл SM.v.

2. Выполните команду File→Create/Update→Create Symbol Files for Current File.

3. Дождитесь сообщения об успешном создании символа.

Сборка схемы АЛУ с последовательным переносом.

1. Выберите пункт меню File→New.

2. В появившемся окне выберите Block Diagram/Schematic File и нажмите кнопку OK.

3. В основном окне Quartus появилось окно схемотехнического редактора с пустым неименованным файлом, в котором нужно собрать схему АЛУ с последовательной организацией переноса (рис.4).

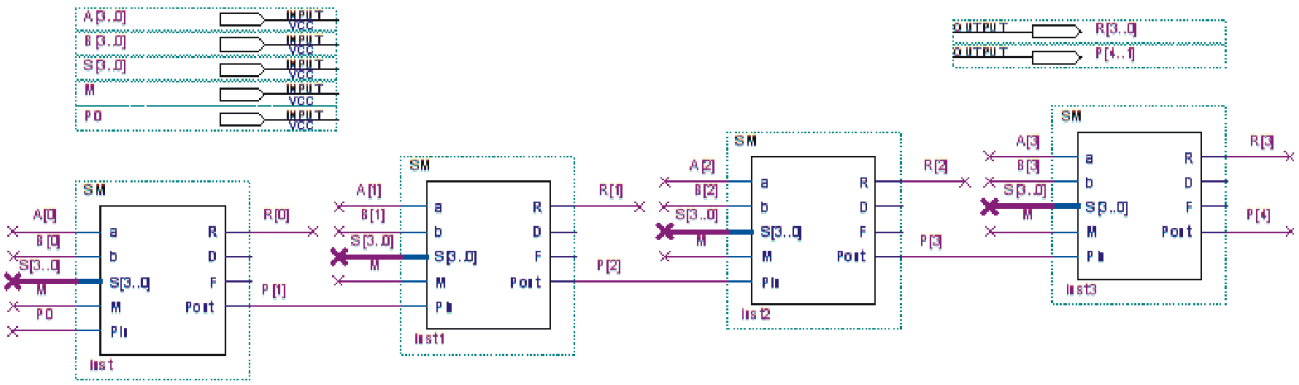


Рис.4. Схема АЛУ с последовательной организацией переноса

4. Введите элемент SM. Для этого щелкните два раза левой кнопкой мыши в пустом пространстве схемотехнического редактора. Появившийся диалог Symbol позволяет добавлять примитивы в файл графического редактора. Поле Name содержит имя добавляемого примитива, в списке Libraries перечислены каталоги, содержащие библиотеки примитивов Quartus. Выберите примитив SM из папки Project и нажмите кнопку OK.

5. Введите остальные три элемента SM (см. предыдущий пункт).

6. Введите входные и выходные контакты, которые обозначают входы и выходы схемы. Входные и выходные контакты вводятся так же, как и остальные примитивы. Имя примитива для входного контакта - input, для выходного - output. Для того чтобы добавить контакт, щелкните два раза левой кнопкой мыши в пустом пространстве схемотехнического редактора и введите в поле Name имя входного (input) или выходного (output) контакта. Всем контактам назначается имя по умолчанию - pin_name.

7. Разместите элементы на экране в удобной для соединения последовательности.

8. Назначьте имена входным и выходным контактам.

9. Соедините элементы в соответствии со схемой. Для того чтобы проложить соединение, подведите курсор мыши к выходу или входу элемента. При этом курсор изме-

нит свой вид на перекрестие. Далее нажмите левую кнопку мыши, переместите мышь к концу предполагаемого соединения и отпустите мышь.

10. Сохраните схему с именем ALU_SerialCarry.bdf командой File→Save.

11. В окне Project Navigator выберите закладку Files и пункт контекстного меню (правая кнопка мыши) для файла ALU_SerialCarry.bdf Set as Top-Level Entity.

12. Скомпилируйте проект командой Processing→Start Compilation.

13. Дождитесь сообщения об успешной компиляции или сообщения об ошибках. В случае появления ошибок откорректируйте схему и повторите компиляцию.

14. Создайте примитив ALU_SerialCarry.

Проверка работы АЛУ с последовательным переносом в симуляторе.

Для моделирования АЛУ необходимо создать файл входных воздействий и связать его с файлом ALU_SerialCarry.bdf. Аналогичные действия выполнялись при моделировании модуля SM.

На рис.5 приведены результаты моделирования, иллюстрирующие работу АЛУ

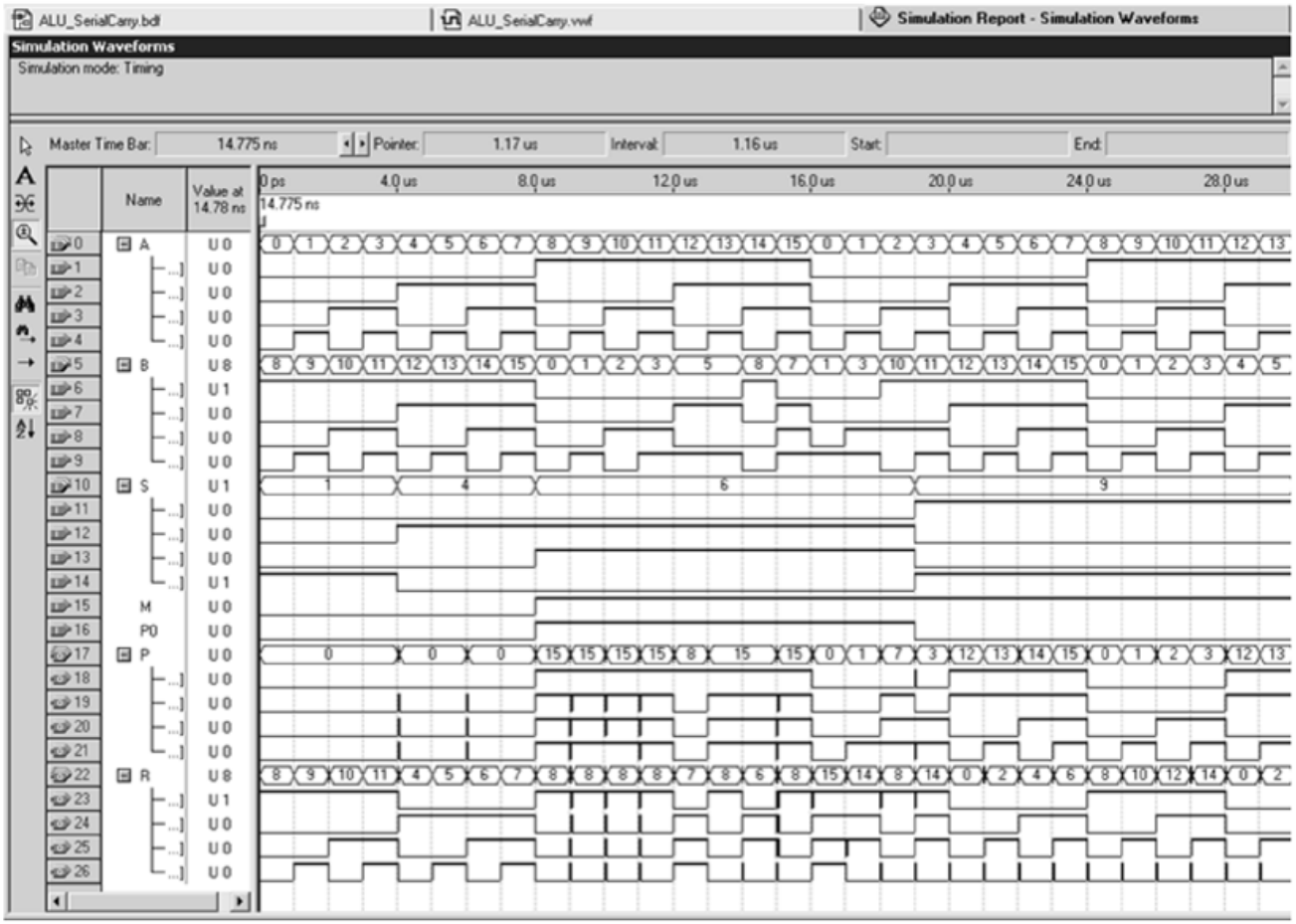


Рис.5. Результаты моделирования АЛУ с последовательной организацией переноса

при выполнении четырех основных операций. Проанализируйте полученные временные диаграммы, в случае необходимости откорректируйте схему АЛУ и файл входных воздействий и повторите компиляцию проекта и моделирование.

Разработка HDL-описания СУП, сборка схемы АЛУ с параллельным переносом.

1. Создайте новый проект ALU_ParallelCarry.qpf.

2. Добавьте в проект Verilog-описание СУП FCU.v, объявление модуля приведено на рис.6.

```
1  module FCU (PO, D, F, P);
2
3    input            PO;
4    input   [4:1]    D, F;
5    output  [4:1]    P;
6
7    // Описание схемы ускоренного переноса
8
9    endmodule
```

Рис.6. Заготовка Verilog-описания СУП

3. Скомпилируйте проект и создайте примитив FCU.

4. Добавьте в проект новый файл схемотехнического редактора, соберите в нем схему АЛУ с параллельным арифметическим переносом и сохраните под именем ALU_ParallelCarry.bdf (рис.7).

5. Создайте файл ALU_ParallelCarry.bdf файлом верхнего уровня в проекте, скомпилируйте и при необходимости откорректируйте модуль FCU и схему АЛУ.

6. Выполните моделирование схемы аналогично моделированию АЛУ с последовательным переносом.

7. Создайте примитив ALU_ParallelCarry.

8. Сравните быстродействие схем с последовательным и параллельным переносом.

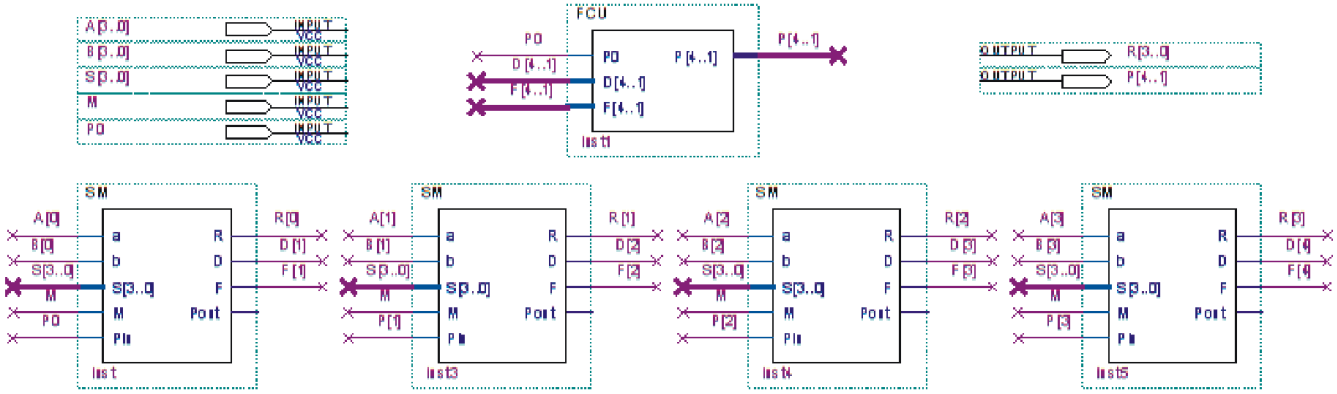


Рис. 7. Схема АЛУ с параллельной организацией переноса

Требования к отчету

В качестве отчета о выполнении лабораторной работы вы должны представить проект Quartus, содержащий схемы, Verilog-описания, тестовые воздействия и результаты моделирования схем АЛУ с последовательной и параллельной организацией арифметического переноса. По результатам моделирования должны быть сделаны выводы о быстродействии схем с различной организацией арифметического переноса.

Лабораторная работа № 2

Разработка регистрового арифметико-логического устройства

Лабораторное задание

Необходимо разработать Verilog-описание и промоделировать работу 4-битного разрядно-модульного регистрового АЛУ (РАЛУ).

Процесс выполнения задания можно разделить на следующие этапы:

1) создание проекта в САПР Quartus II;

2) разработка Verilog-описания РАЛУ с использованием созданной ранее схемы АЛУ с последовательной организацией арифметического переноса;

3) проверка работы РАЛУ в симуляторе;

4) создание примитива (символа) РАЛУ;

5) выполнение индивидуального задания.

Структура разрядно-модульного РАЛУ

Основу структуры разрядно-модульного РАЛУ (рис.1) составляет комбинационная схема АЛУ, на вход которой подаются два операнда, представляющие собой содержимое регистра A (RgA) и регистра B (RgB).

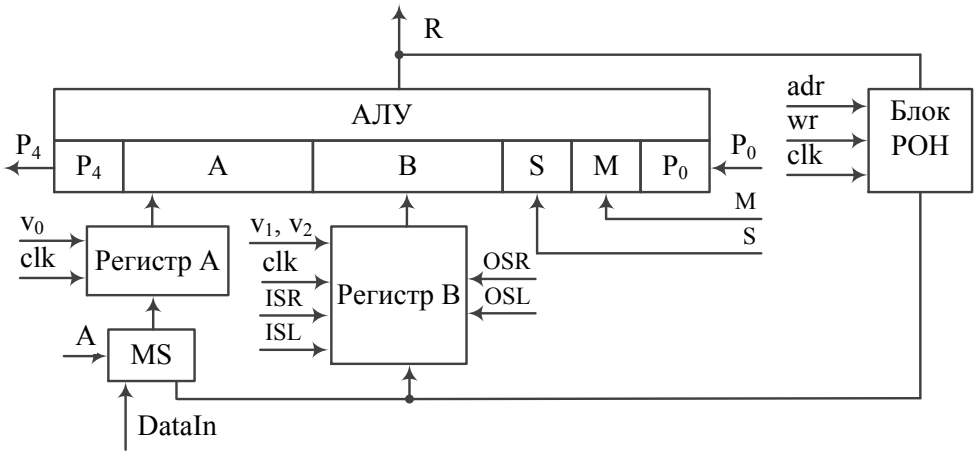


Рис.1. Схема разрядно-модульного РАЛУ

Регистр A имеет два режима (записи и чтения) и способен принимать данные как из блока РОН, так и от внешних устройств (ВУ), что реализуется с помощью мультиплексора.

Регистр B имеет четыре режима (записи, чтения, сдвига влево и сдвига вправо) и принимает данные только из блока РОН. Результат операции может быть записан в блок РОН, информация с выхода АЛУ также попадает к ВУ.

Определим состав и назначение управляющих сигналов для данной структуры РАЛУ.

Сигналы wr, v_0 предназначены для управления режимами чтения/записи блока РОН и регистра A: 0 - чтение информации; 1 - запись.

Мультиплексор управляется сигналом A: при A = 0 на вход регистра A подаются данные из блока РОН, при A = 1 - от ВУ (шина DataIn). Регистр B в данной схеме принимает данные только из блока РОН.

Блок РОН представляет собой синхронное ОЗУ статического типа, содержащее набор регистров. Для выбора нужного регистра используются линии адреса adr.

Для выбора режима работы регистра B используется пара управляющих сигналов:

$$\{v_2, v_1\} = \begin{cases} 00 - \text{чтение}; \\ 01 - \text{сдвиг влево}; \\ 10 - \text{сдвиг вправо}; \\ 11 - \text{запись}. \end{cases}$$

При сдвигах применяются две пары сигналов:

ISR - значение разряда, подаваемого на вход регистра при сдвиге вправо;

ISL - значение разряда, подаваемого на вход регистра при сдвиге влево;

OSR - значение разряда, появляющееся на выходе регистра при сдвиге вправо;

OSL - значение разряда, появляющееся на выходе регистра при сдвиге влево.

При выполнении арифметических операций используются бит переноса из предыдущей секции РАЛУ P_0, бит переноса в следующую секцию P_4.

Все описанные сигналы можно разделить на три категории: информационные, управляющие и сигналы синхронизации.

Информационные сигналы: шина от ВУ, P_0, ISL, ISR (входные), шина к ВУ, P_4, OSR, OSL (выходные).

Сигналы управления: v_0, v_1, v_2, wr, adr, M, S.

Сигнал синхронизации: clk.

Рассмотрим примеры формирования микрокоманд и микропрограмм для приведенной ранее структуры РАЛУ (табл.1). Состояние сигналов, не влияющих на выполнение текущей операции, будем обозначать символом «X».

Таблица 1

Примеры формирования микрокоманд и микропрограмм для разрядно-модульного РАЛУ

Пример	Операция	Обозначение	adr	A	wr	S_3	S_2	S_1	S_0	M	v_0	v_1	v_2	P_0
1	Передача содержимого регистра A в блок РОН с адресом $A_к$	$RgA \rightarrow РОН(A_к)$	$A_к$	X	1	0	0	0	0	0	0	0	0	X
2	Выполнение арифметических или логических операций над содержимым регистров A и B с размещением результата в блоке РОН с адресом $A_к$	$RgA\ [S;M;P_0]$ $RgB \rightarrow РОН(A_к)$	$A_к$	X	1	S_3	S_2	S_1	S_0	M	0	0	0	P_0
3	Передача данных от ВУ в регистр A	$ВУ \rightarrow RgA$	X	1	0	X	X	X	X	X	1	0	0	X
4	Запись данных из блока РОН с адресом $A_к$ в регистры A и B	$РОН(A_к) \rightarrow RgA$, RgB	$A_к$	0	0	X	X	X	X	X	1	1	1	X
5	Сложение содержимого блока РОН с адресами A_n и A_m с размещением результата в блоке РОН с адресом $A_к$	$РОН(A_n) \rightarrow RgA$ $РОН(A_m) \rightarrow RgB$ $RgA + RgB \rightarrow$ $РОН(A_к)$	A_n A_m $A_к$	0 X X	0 0 1	X X 1	X X 0	X X 0	X X 1	X X 1	1 0 0	0 1 1	0 1 0	X X 0

Пример 1. Передача содержимого регистра A в блок РОН с адресом $A_к$. В этом случае АЛУ не выполняет преобразований данных, вместо этого происходит трансляция операнда A на выход комбинационной схемы. Это действие обеспечивается выполнением логической операции, задаваемой управляющим кодом $S_3 = 0$, $S_2 = 0$, $S_1 = 0$, $S_0 = 0$. Значение бита переноса из предыдущей секции не влияет на результат. Все регистры, не задействованные в данной операции, должны хранить свое текущее значение, поэтому $v_0 = v_1 = v_2 = 0$. На управляющие входы блока РОН подаем адрес $A_к$ и сигнал записи wr = 1. Состояние сигнала, управляющего мультиплексором, не оказывает влияния на результат, поскольку для выполнения данной операции неважно входное значение регистра 1.

Пример 2. Выполнение арифметических или логических операций над содержимым регистров A и B с размещением результата в блоке РОН с адресом $A_к$. Регистры A и B находятся в режиме чтения $v_0 = v_1 = v_2 = 0$. Блок РОН находится в режиме записи

wr = 1. Запись в блок РОН осуществляется по адресу $A_к$, поэтому adr = $A_к$. Сигналы S, M, P_0 задают тип выполняемой операции. Состояние сигнала, управляющего мультиплексором, не оказывает влияния на результат.

Пример 3. Передача информации от ВУ в регистр А. В этой операции активными являются регистр А и мультиплексор: $v_0 = 1$, A = 1. Сигналы управления АЛУ не влияют на результат. Все элементы памяти, кроме регистра А, находятся в режиме чтения w/r = $v_1 = v_2 = 0$.

Пример 4. Запись информации из блока РОН с адресом $A_к$ в регистры А и В. Эта операция аналогична предыдущей, за исключением режима мультиплексора и регистра В: A = 0, $v_1 = v_2 = 1$.

Пример 5. Сложение содержимого блока РОН с адресами A_n и A_m с размещением результата в блоке РОН с адресом $A_к$. Для выполнения этой операции необходимо составить микропрограмму из трех микрокоманд, выполняющих следующие действия: размещение первого операнда в регистре А, размещение второго операнда в регистре В, выполнение операции сложения с сохранением результата в блоке РОН.

Анализ приведенных примеров показывает, что при составлении микрокоманд и микропрограмм необходимо следовать основному правилу: содержимое регистров, не участвующих в выполнении микрокоманды, должно быть сохранено или восстановлено без искажения данных.

Пример проектирования

Рассмотрим процесс создания Verilog-описания РАЛУ. Из схемы РАЛУ (см. рис.1) видно, что ее Verilog-описание должно состоять из подключения ранее разработанного комбинационного модуля АЛУ, описания блока РОН, двух регистров и мультиплексора.

Для реализации регистров используйте конструкцию **always**. Пример ее применения приведен при описании D-триггера по переднему фронту с синхронным сбросом и установкой (см. раздел 1.3, пример 14).

Описание мультиплексора можно совместить с описанием регистра А в одной конструкции **always** с помощью условной конструкции (**? :**). (см. раздел 1.3, пример 12).

Для реализации блока РОН используйте массив регистров и конструкцию **always**. В качестве образца можно использовать описание синхронного 8-разрядного ОЗУ на 256 элементов (см. раздел 1.4).

На рис.2 представлена заготовка Verilog-описания с подключенным модулем АЛУ и рекомендуемым описанием портов модуля РАЛУ, где:

• clk - тактовый сигнал;

• reset - сигнал сброса регистров схемы в нулевое состояние;

• DataIn, R - шины от ВУ и к ВУ;

• S, M - управляющее слово и модификатор;

• A - сигнал управления мультиплексором;

• v - сигнал управления режимами чтения/записи регистров A и B;

• wr - сигнал управления режимами чтения/записи блока РОН;

• adr - 3-разрядная шина адреса блока РОН, т.е. блок содержит восемь 4-разрядных регистров;

• ISL, ISR - значение бита, подаваемого на вход второго регистра при сдвиге влево и вправо соответственно;

• OSL, OSR - значение бита, снимаемого с выхода второго регистра при сдвиге влево и вправо соответственно;

• P_4 - выходной бит переноса.

```
 1  module RALU (   input           clk,
 2                  input           reset,
 3                  input   [3:0]   DataIn,
 4                  input   [3:0]   S,
 5                  input           M,
 6                  input           PO,
 7                  input           A,
 8                  input   [3:0]   v,
 9                  input           wr,
10                  input   [2:0]   adr,
11                  output          OSR,
12                  output          OSL,
13                  input           ISR,
14                  input           ISL,
15                  output          P4,
16                  output  [3:0]   R );
17
18
19  reg     [3:0]   RgA, RgB;
20  wire    [4:1]   P;
21
22  ALU_ParallelCarry ALU ( .A(RgA),
23                          .B(RgB),
24                          .S(S),
25                          .M(M),
26                          .PO(PO),
27                          .R(R),
28                          .P(P) );
29  endmodule
```

Рис.2. Заготовка Verilog-описания модуля РАЛУ

На рис.3 представлены результаты моделирования работы РАЛУ. В данном примере последовательно производится запись в блок РОН чисел 4, 6, 3, 2 по адресам 0, 1, 2, 3 соответственно. Далее выполняются операции сложения содержимого блока РОН с адресами 0, 2 и 1, 3. Результаты записываются в блок РОН по адресам 0 и 1. Все регистры схемы фиксируют данные и сигналы управления по переднему фронту тактового сигнала clk.

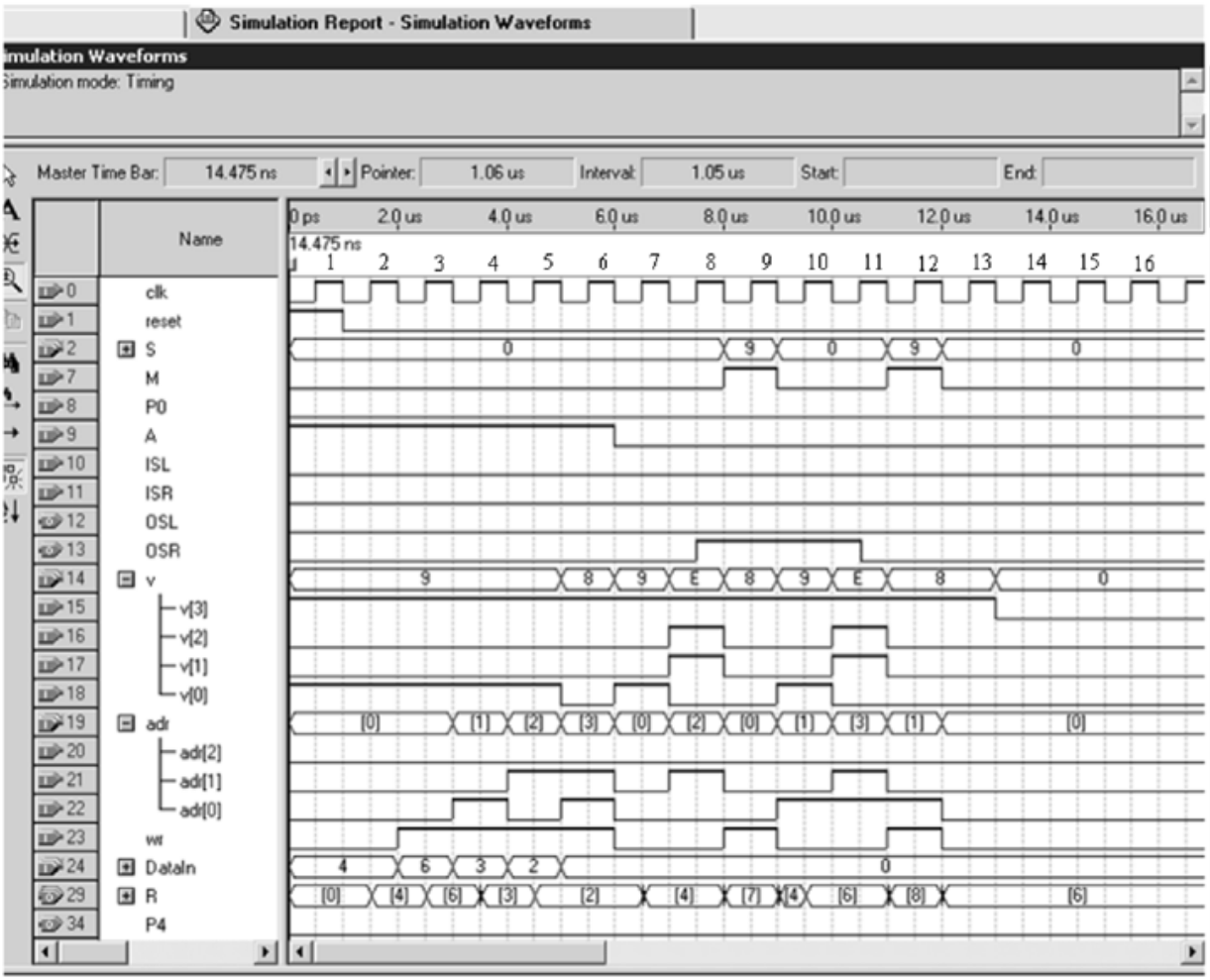


Рис.3. Результаты моделирования работы РАЛУ

В табл.2 приведено потактовое описание временной диаграммы, представленной на рис.3. Видно, что данный пример требует последовательного выполнения 12-ти микрокоманд. Обратите внимание, что действия, выполняемые некоторыми блоками схемы АЛУ, можно совмещать в одной микрокоманде. Так, на тактах 3 - 5 совмещаются записи данных в регистр А и блок РОН или на тактах 9, 12 совмещаются арифметическая операция сложения и запись результата в блок РОН.

Таблица 2

Описание временной диаграммы

Номер такта	Описание
1	Сигнал reset находится в активном состоянии, все регистры обнуляются
2	Данные с шины DataIn (4) записываются в регистр A ($v_0 = 1$) и появляются на выходе АЛУ, что обеспечивается управляющим словом $S_3 = S_2 = S_1 = S_0 = 0$
3	Данные с шины DataIn (6) и выхода АЛУ записываются в регистр A (A = 1, $v_0 = 1$) и блок РОН по адресу 0 (wr = 1, adr = 0) соответственно
4	Данные с шины DataIn (3) и выхода АЛУ записываются в регистр A (A = 1, $v_0 = 1$) и блок РОН по адресу 1 (wr = 1, adr = 1) соответственно
5	Данные с шины DataIn (2) и выхода АЛУ записываются в регистр A (A = 1, $v_0 = 1$) и блок РОН по адресу 2 (wr = 1, adr = 2) соответственно
6	Данные с выхода АЛУ записываются в ячейку блока РОН по адресу 3 (wr = 1, adr = 3)
7	Данные из блока РОН с адресом 0 записываются в регистр A (A = 0, $v_0 = 1$, adr = 0, wr = 0)
8	Данные из блока РОН с адресом 2 записываются в регистр B (A = 0, $v_1 = v_2 = 1$, adr = 2, wr = 0)
9	Выполняется операция арифметического сложения содержимого регистров A и B с записью результата в блок РОН по адресу 0 ($S_3 = 1$, $S_2 = 0$, $S_1 = 0$, $S_0 = 1$, M = 1, $P_0 = 1$, adr = 0, wr = 1)
10	Данные из блока РОН с адресом 0 записываются в регистр A (A = 0, $v_0 = 1$, adr = 1, wr = 0)
11	Данные из блока РОН с адресом 2 записываются в регистр B (A = 0, $v_1 = v_2 = 1$, adr = 3, wr = 0)
12	Выполняется операция арифметического сложения содержимого регистров A и B с записью результата в блок РОН по адресу 1 ($S_3 = 1$, $S_2 = 0$, $S_1 = 0$, $S_0 = 1$, M = 1, $P_0 = 1$, adr = 1, wr = 1)

Рассмотренный пример функционирования РАЛУ показывает, что выполнение команд в операционных устройствах процессора можно представить в виде следующей последовательности действий:

1) размещение операндов на входах операционного устройства;

2) выполнение операции в операционном устройстве;

3) запись результата операции в элемент памяти.

Каждое действие в приведенной последовательности невозможно разделить на более простые подоперации. Такие элементарные преобразования, не разложимые на более простые и выполняющиеся в течение одного такта сигнала синхронизации, называют микрооперациями, или микрокомандами.

Для выполнения пункта 3 лабораторного задания (проверка работы РАЛУ в симуляторе) повторите результаты моделирования, приведенные на рис.3, считайте результаты сложений из блока РОН и запишите их в регистры A и B соответственно. Таким образом, ваша временная диаграмма должна дополниться еще двумя полезными тактами (общее количество тактов равно 14).

Индивидуальное задание

В качестве индивидуального задания предлагается промоделировать работу РАЛУ при выполнении конкретной микропрограммы. В табл.3 приведены варианты операций, которые необходимо представить в виде микропрограммы. Операнды A и B последовательно считываются с шины DataInput.

Требования к отчету

В качестве отчета о выполнении лабораторной работы вы должны представить проект Quartus, содержащий схемы, Verilog-описания, тестовые воздействия и результаты моделирования схемы РАЛУ.

Полученные временные диаграммы должны отражать действия в соответствии с примером (см. рис.3, табл.2), дополненные операциями записи результатов в регистры A и B. Помимо этого вы должны построить временные диаграммы, иллюстрирующие выполнение одного из вариантов индивидуального задания, представленных в табл.3.

Таблица 3

Варианты операций для индивидуального задания

№	Операция	№	Операция
1	$(A+B)\cdot A$	16	$(A+A\cdot B)\vee B$
2	$(\overline{A}+B)\cdot A$	17	$(A-1)\cdot B$
3	$(A-\overline{B})\cdot A$	18	$(A+B)\vee(A-B)$
4	$(A+B)\oplus B$	19	$(A\cdot B+A\vee\overline{B})\cdot A$
5	$(A\oplus B)+B$	20	$(A-1)\vee B$
6	$(A+B+1)\vee B$	21	$(A-(B>>1))\vee A$
7	$(A\vee\overline{B})+B$	22	$(A+(B<<2)+1)\vee B$
8	$(A+B)\cdot(A-B)$	23	$((A<<1)-\overline{B})\cdot A$
9	$(A\cdot B+A\vee\overline{B})\vee A$	24	$(A+B)\vee(A-(B>>1))$
10	$(A-1)\oplus B$	25	$(A\vee\overline{B})-(A\oplus B)$
11	$(A+(B<<1))\cdot A$	26	$(A+A\cdot B)\oplus B$
12	$((A<<1)+B)\cdot(A-B)$	27	$(A-1)\cdot(B>>1)$
13	$(A-\overline{B})\cdot(A>>2)$	28	$(A+B)\oplus(A-B)$
14	$(A+B)\cdot(A<<3)$	29	$(A\cdot B+A\vee\overline{B})\oplus A$
15	$(A\vee\overline{B})-(A\cdot B)$	30	$(A\vee\overline{B})-B$

Лабораторная работа № 3

Разработка вычислительного устройства с микропрограммным управлением

Лабораторное задание

Необходимо реализовать вычислительное устройство с микропрограммным управлением на основе созданного ранее РАЛУ с разрядно-модульной организацией и вновь разработанного устройства микропрограммного управления (УМУ).

УМУ должно обеспечивать последовательное выполнение микрокоманд, в том числе условного и безусловного перехода. Микрокоманды условного перехода должны выполняться по результату анализа флагов переполнения и нулевого результата от РАЛУ. Описание схемы устройства управления необходимо выполнить на Verilog HDL, структуру всего устройства с микропрограммным управлением можно представить в виде схемы.

Следует разработать две микропрограммы, отличающиеся типами используемых переходов (безусловный и условный). Обе программы должны вычислять выражение из табл.3 лабораторной работы № 2, соответствующее вашему варианту.

Первая микропрограмма:

- по шине DataInput считывается пара операндов A и B;
- вычисляется выражение из табл.3 лабораторной работы № 2;
- с помощью команды безусловного перехода начинается исполнение микропрограммы сначала.

Вторая микропрограмма:

- по шине DataInput считывается константа C, а также пара операндов A и B;
- вычисляется выражение из табл.3 лабораторной работы № 2;
- сравниваются результат вычислений R и константа C, для чего из результата R вычитается константа C;
- выполняется микрокоманда условного перехода, устройство либо вновь переходит к считыванию константы C в случае выполняемого перехода, либо переходит к повторному вводу операндов в случае невыполняемого перехода.

Условный переход считается выполняемым, если заданное в табл.1 отношение R и C выполняется, например, R = C для вариантов 21 - 25. Обратите внимание, что схема устройства с микропрограммным управлением должна обеспечивать выполнение любого условия или комбинации условий из табл.1.

Таблица 1

Варианты индивидуального задания

Варианты	1 - 5	6 - 10	11 - 15	16 - 20	21 - 25	26 - 30
Условие	Не равно	Больше	Меньше	Равно	Больше либо равно	Меньше либо равно

Структура устройства микропрограммного управления

Обобщенная структура УМУ представлена на рис.1 и состоит из устройства формирования адреса (УФА), управляющей памяти, регистра микрокоманд и дешифратора.

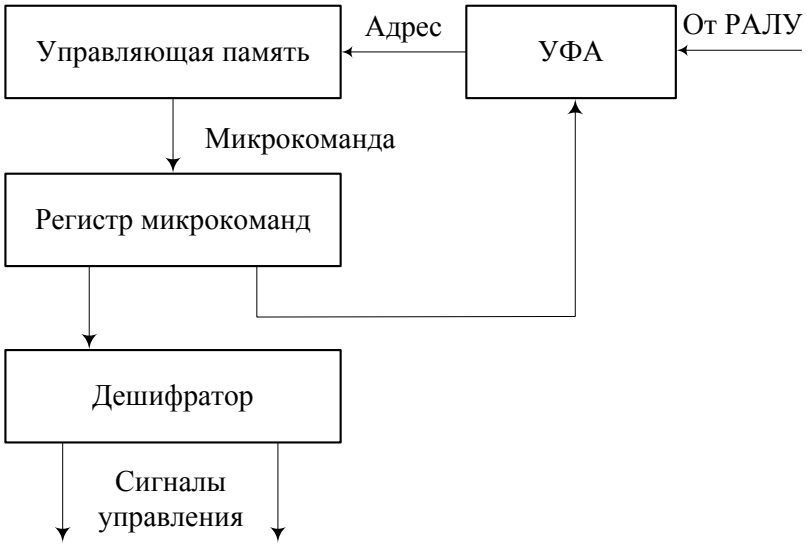


Рис.1. Обобщенная структура УМУ

УФА вырабатывает адрес следующей микрокоманды в зависимости от текущей микрокоманды и флагов, поступающих от РАЛУ. Коды микрокоманд хранятся в управляющей памяти и на каждом такте синхронизации подаются в регистр микрокоманд. Микрокоманда состоит из двух частей - операционной, в ней содержатся управляющие сигналы, и адресной, которая поступает в УФА для управления загрузкой следующей микрокоманды (рис.2). Дешифратор преобразует операционную часть микрокоманды в управляющие сигналы.

Отметим, что в простейшем случае УМУ может состоять только из УФА и управляющей памяти. При этом управляющие сигналы содержатся в микрокоманде в явном виде и подаются на все блоки непосредственно с выхода управляющей памяти.

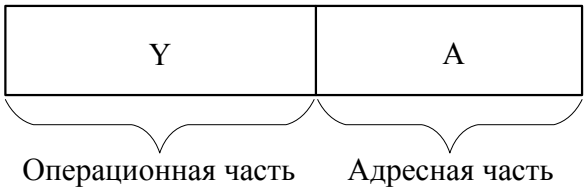


Рис.2. Структура микрокоманды

УМУ можно классифицировать по нескольким признакам. Один из них - по способу представления операционной части микрокоманды. По этому признаку существует разделение на три типа микропрограммирования: горизонтальное, вертикальное и смешанное.

При горизонтальном микропрограммировании (рис.3) каждому разряду операционной части соответствует один определенный управляющий сигнал из набора Y. Если N - количество управляющих сигналов, которые необходимо реализовать, то разрядность операционной части регистра будет также равна N. Для этого типа микропрограммирования характерна очень большая разрядность управляющей памяти и, соответственно, большой объем микрокода. В то же время такая реализация проще со схемотехнической точки зрения.

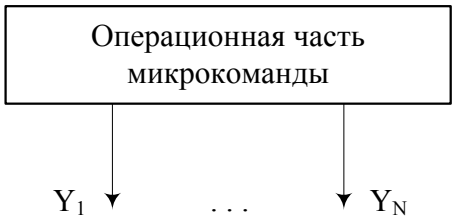


Рис.3. Горизонтальное микропрограммирование

При вертикальном микропрограммировании (рис.4) операционная часть управляющего слова хранится в памяти в закодированном виде и подвергается дешифрации перед подачей в операционные устройства микропроцессора. Достоинством такого подхода является уменьшение требований к объему памяти для микрокода. При этом сложность микропроцессора увеличивается, а быстродействие уменьшается в связи с появлением дополнительных комбинационных схем - дешифраторов.

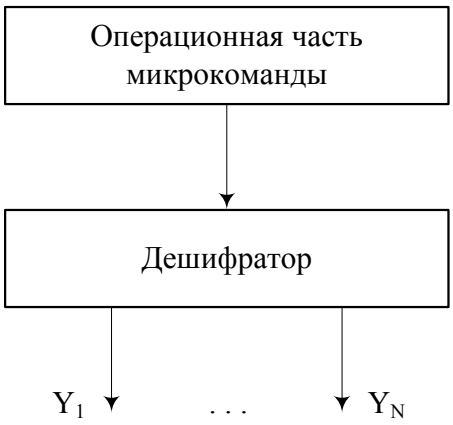


Рис.4. Вертикальное микропрограммирование

Смешанное микропрограммирование (рис.5) сочетает достоинства горизонтального и вертикального. При этом набор управляющих сигналов Y разбивается на некоторое количество поднаборов. Каждый из этих поднаборов управляет отдельным операционным устройством либо редко использующимися совместно устройствами. Далее на каждый поднабор устанавливается свой дешифратор. Как результат, уменьшается объем микрокода вследствие некоторого допустимого усложнения схемы УМУ.

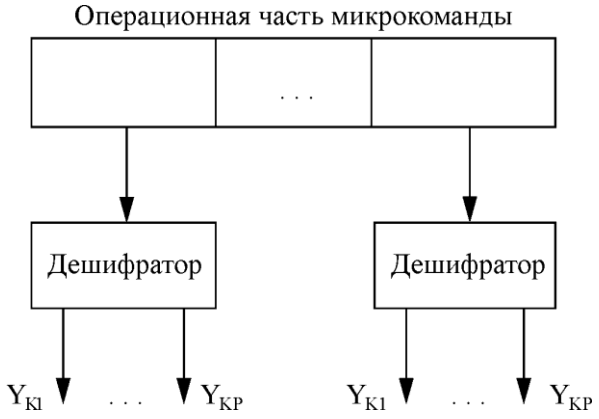


Рис.5. Смешанное микропрограммирование

Развитие технологий производства микросхем, удешевление изготовления внутренней памяти, а также необходимость увеличения частот микропроцессора привели к тому, что в настоящее время вертикальное микропрограммирование практически не используется и вытеснено горизонтальным или смешанным.

Другой распространенный признак классификации УМУ - по способу формирования адреса следующей микрокоманды. Различают УМУ с принудительной адресацией и с естественной.

В УМУ с *принудительной адресацией* (рис.6) каждая микрокоманда содержит адресную часть, которая управляет порядком следования микрокоманд. Адресная часть содержит поле логических условий U', которое является маской для набора внешних логических условий U (флаги от РАЛУ), и два поля адреса A_0 и A_1 следующей микрокоманды. Поле U' совместно с флагами от РАЛУ управляет мультиплексором выбора адреса следующей микрокоманды. На мультиплексор будет подана единица, только если в позиции соответствующих коду условий будет единица и при этом условие было выполнено. В таком случае выполняется ветвление, и в регистр адреса будет загружен A_1. Если условие не выполнено, то ветвления нет, и используется поле A_0.

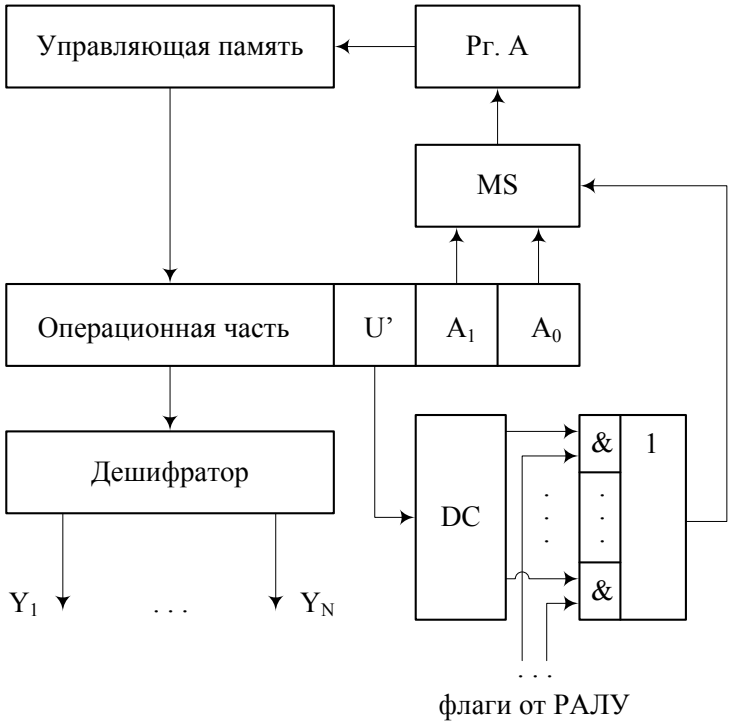


Рис.6. Схема УМУ с принудительной адресацией

В УМУ *с естественной адресацией* (рис.7) в УФА используется инкрементирующий счетчик адреса следующей микрокоманды. Если признак Р, содержащийся в микрокоманде, равен нулю, то линейный порядок следования микрокоманд не изменяется, и адрес просто инкрементируется в счетчике. Если Р равен единице, то операционная часть микрокоманды выступает в роли нового адреса, который загружается в счетчик. Таким образом реализуется механизм ветвлений.

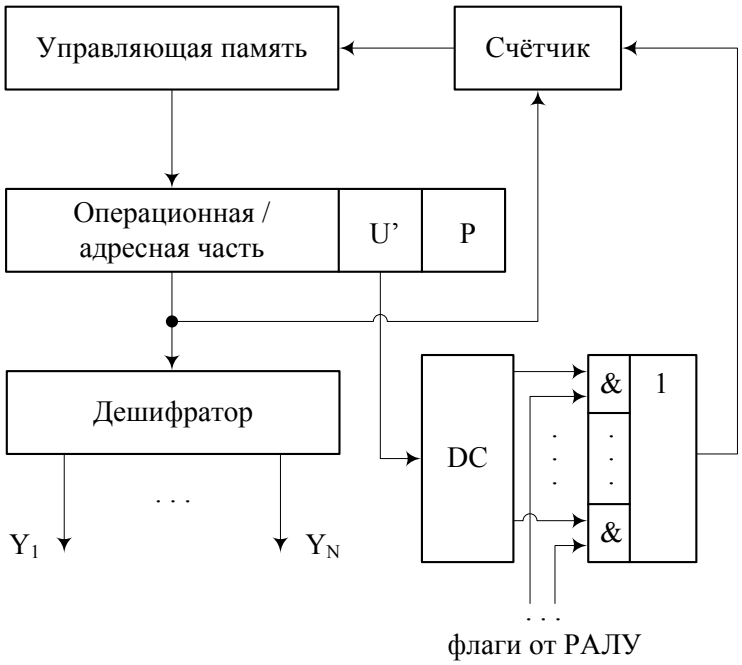


Рис.7. Схема УМУ с естественной адресацией

Пример проектирования

На рис.8 приведен пример структуры вычислительного устройства с микропрограммным управлением, которая состоит из известной схемы РАЛУ, устройства

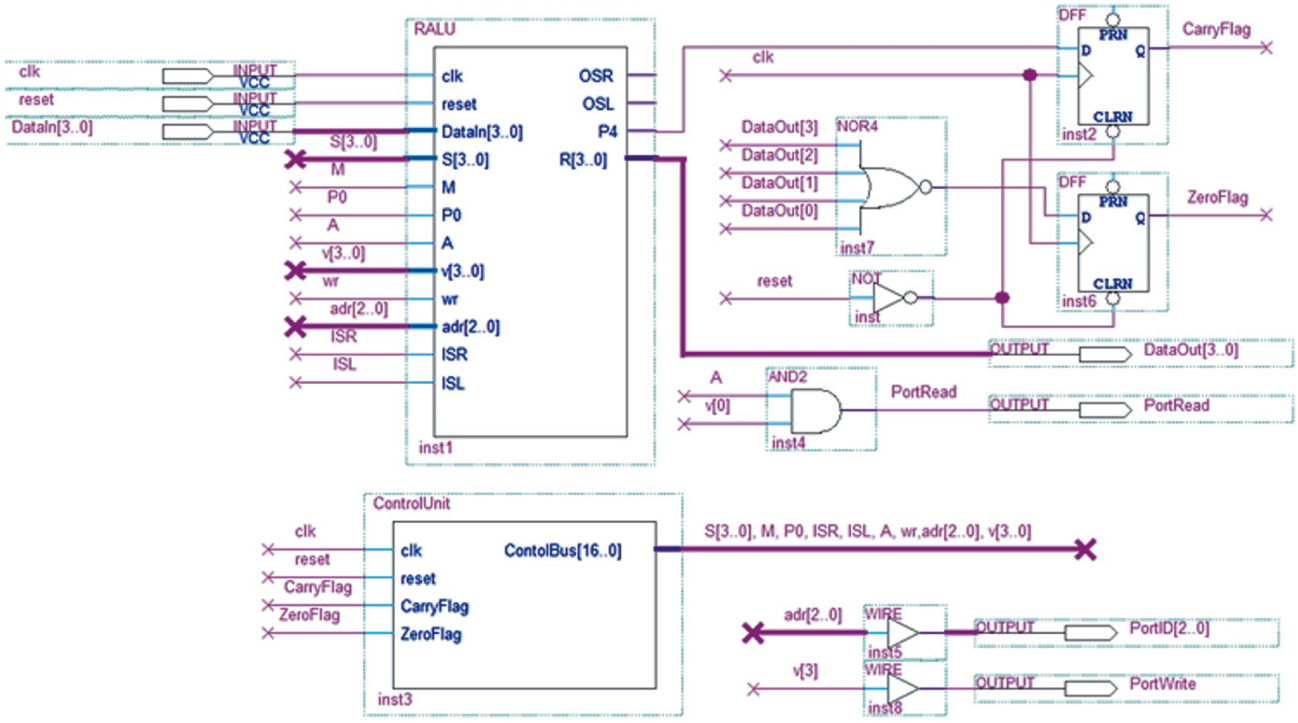


Рис.8. Структура вычислительного устройства с микропрограммным управлением

микропрограммного управления, двух триггеров и логических элементов. Все элементы памяти (регистры и триггеры) тактируются и сбрасываются сигналами clk и reset соответственно.

УМУ построено с использованием горизонтального микропрограммирования и естественной адресации микрокоманд. УМУ формирует все необходимые для РАЛУ сигналы управления в зависимости от текущей микрокоманды и флагов. Микропрограмма хранится в ПЗУ, входящем в состав УМУ.

Логический элемент ИЛИ-НЕ на четыре входа и два триггера используются для формирования флагов переполнения (CarryFlag) и нулевого результата (ZeroFlag). Триггеры необходимы для синхронизации работы РАЛУ и УМУ таким образом, чтобы при выполнении команды условного перехода УМУ анализировало флаги, соответствующие предыдущей операции.

Для реализации команд условных и безусловных переходов используются комбинации флагов, приведенные в табл.2. Указанные комбинации передаются в микрокоманде с помощью кода перехода. Нулевой код перехода соответствует обычной команде без функции управления программой. Адрес выполняемого перехода содержится в младших разрядах микрокоманды, т.е. в битах adr[2:0] и v[3:0]. При этом на все управляющие сигналы УМУ выдает неактивные уровни.

Таблица 2

Реализация команд переходов

Условие перехода	Состояние флагов	Код перехода
Не равно	ZeroFlag = 0	001
Больше	CarryFlag = 1 и ZeroFlag = 0	010
Меньше	CarryFlag = 0 и ZeroFlag = 0	011
Равно	ZeroFlag = 1	100
Больше либо равно	CarryFlag = 1 или ZeroFlag = 1	101
Меньше либо равно	CarryFlag = 0 или ZeroFlag = 1	110
Безусловный переход	Любое	111

В данном устройстве реализованы команды обращения к внешним портам на чтение и запись. Для этого формируются сигналы PortRead, PortWrite, PortID.

Сигнал PortRead - это результат логического умножения сигнала записи в первый регистр РАЛУ и сигнала управления мультиплексором. Таким образом, логическая «1» в

сигнале PortRead появляется, когда в микрокоманде установлены в «1» биты v[0] и A, т.е. производится запись данных из внешнего устройства в первый регистр РАЛУ.

Сигналы PortWrite и PortID содержатся непосредственно в микрокоманде, причем шина PortID совмещена с шиной адреса блока РОН.

На рис.9 представлены результаты моделирования вычислительного устройства с микропрограммным управлением, выполняющего микропрограмму, приведенную в табл.3. Данная программа состоит из семи микрокоманд. Выполняются последовательное считывание двух операндов с шины DataIn, операция сложения этих операндов и безусловный переход на нулевой адрес, т.е. выполняется бесконечный цикл. Описание части временной диаграммы, соответствующей второй итерации цикла, представлено в табл.4.

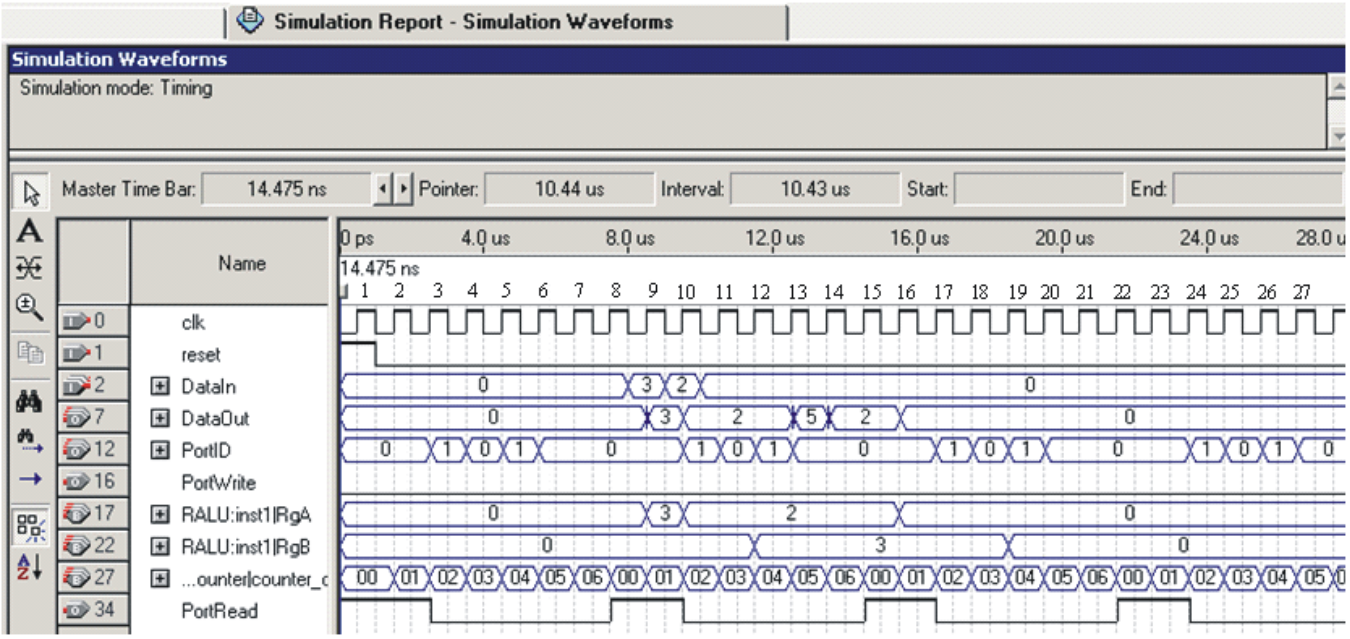


Рис.9. Результаты моделирования вычислительного устройства с микропрограммным управлением

Вы можете предложить и реализовать свою структуру устройства управления.

Требования к отчету

В качестве отчета о выполнении лабораторной работы вы должны представить проект Quartus, содержащий схемы, Verilog-описания, тестовые воздействия и результаты моделирования схемы устройства с микропрограммным управлением, выполняющей один из вариантов индивидуального задания. Ваше устройство должно выполнять все команды переходов, заданные в табл.2.

Таблица 3

Пример микропрограммы

№	Действие	Микрокоманда								
		Код перехода	S[3:0]	M	P_0	ISR, ISL	A	wr	adr[2:0]	v[3:0]
0	DataIn→РгА	000	0000	0	0	00	1	0	000	0001
1	РгА→БРОН-0 DataIn→РгА	000	0000	0	0	00	1	1	000	0001
2	РгА→БРОН-1	000	0000	0	0	00	0	1	001	0000
3	БРОН-0→РгВ	000	0000	0	0	00	0	0	000	0110
4	БРОН-1→РгА	000	0000	0	0	00	0	0	001	0001
5	РгАплюс РгВ	000	1001	1	0	00	0	0	000	0000
6	PC=0	111	0000	0	0	00	0	0	000	0000

Таблица 4

Описание временной диаграммы

Номер такта	Значение программного счетчика	Описание
9	0	Данные с шины DataIn (3) записываются в регистр A ($v_0 = 1$) и появляются на выходе АЛУ, что обеспечивается управляющим словом $S_3 = S_2 = S_1 = S_0 = 0$
10	1	Данные с шины DataIn (2) и выхода АЛУ (3) записываются в регистр A (A = 1, $v_0 = 1$) и блок РОН по адресу 0 (wr = 1, adr = 0) соответственно
11	2	Данные с выхода АЛУ (2) записываются в регистр A (A = 1, $v_0 = 1$) и блок РОН по адресу 1 (wr = 1, adr = 1) соответственно
12	3	Данные из блока РОН с адресом 0 записываются в регистр B (A = 0, $v_1 = v_2 = 1$, adr = 0, wr = 0)
13	4	Данные из блока РОН с адресом 1 записываются в регистр A (A = 0, $v_0 = 1$, adr = 0, wr = 0)
14	5	Выполняется операция арифметического сложения содержимого регистров A и B без сохранения результата ($S_3 = 1$, $S_2 = 0$, $S_1 = 0$, $S_0 = 1$, M = 1, $P_0 = 1$, wr = 0)
15	6	Выполняется команда безусловного перехода по адресу 0

Лабораторная работа № 4

Отладка проекта вычислительного устройства с микропрограммным управлением на учебном стенде

Лабораторное задание

Предлагается интегрировать разработанное ранее вычислительное устройство с микропрограммным управлением в базовый проект для учебного стенда. Для этого необходимо организовать ввод информации в ваше устройство с помощью компьютерной клавиатуры с интерфейсом PS/2, кнопок или переключателей, а также отображение результатов на семисегментных индикаторах и линейках светодиодов, что потребует как аппаратной, так и программной доработки.

Конечный вариант устройства должен работать следующим образом:

- после включения ожидается ввод с помощью клавиатуры пары 4-битных аргументов в шестнадцатеричном формате;
- аргументы отображаются на двух семисегментных индикаторах;
- выполняется операция из табл.2 лабораторной работы № 2;
- результат отображается на семисегментном индикаторе;
- ожидается ввод новой пары аргументов.

Описание учебного стенда и базового проекта

Учебный стенд представляет собой плату, на которой размещены интегральная схема программируемой логики и устройства ввода/вывода (рис.1). На задней и боковой сторонах платы установлены разъемы для подключения источника питания, инструментального компьютера, монитора, дополнительных устройств ввода/вывода и т.д. Слева расположен переключатель для выбора одного из двух режимов программирования, который для удачной загрузки в ОЗУ ПЛИС должен быть в положении **RUN**.

На рис.2 приведена упрощенная структура стенда. Центральное место на плате занимает программируемая логическая интегральная схема фирмы Altera Cyclone II EP2C20F484C7N. Загрузка ОЗУ ПЛИС производится с помощью инструментального компьютера, кабеля программирования и САПР Quartus II. Для подключения стенда к инструментальному компьютеру достаточно соединить соответствующим кабелем порт USB Blaster платы и USB-вход компьютера.

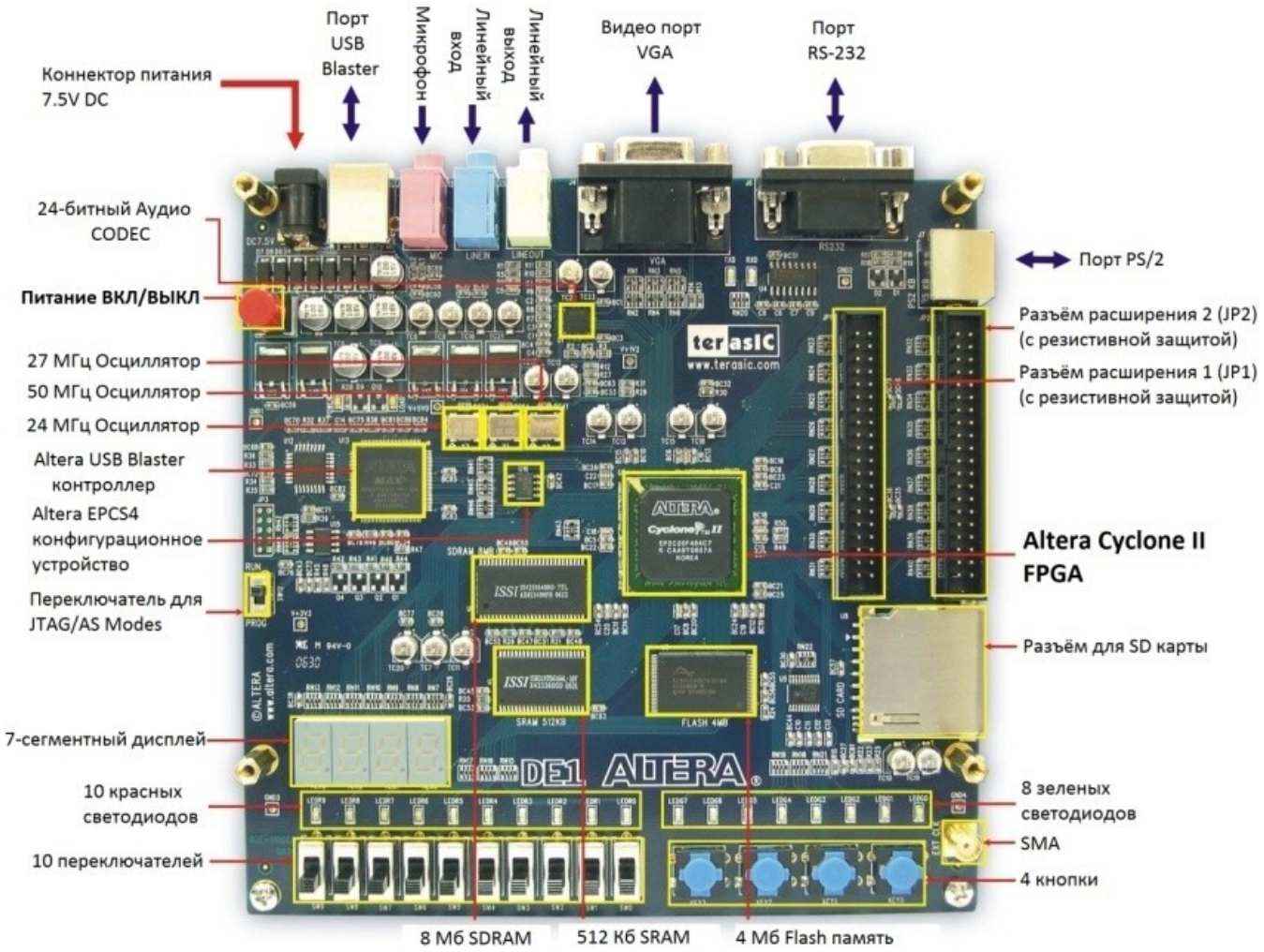


Рис.1. Внешний вид учебного стенда

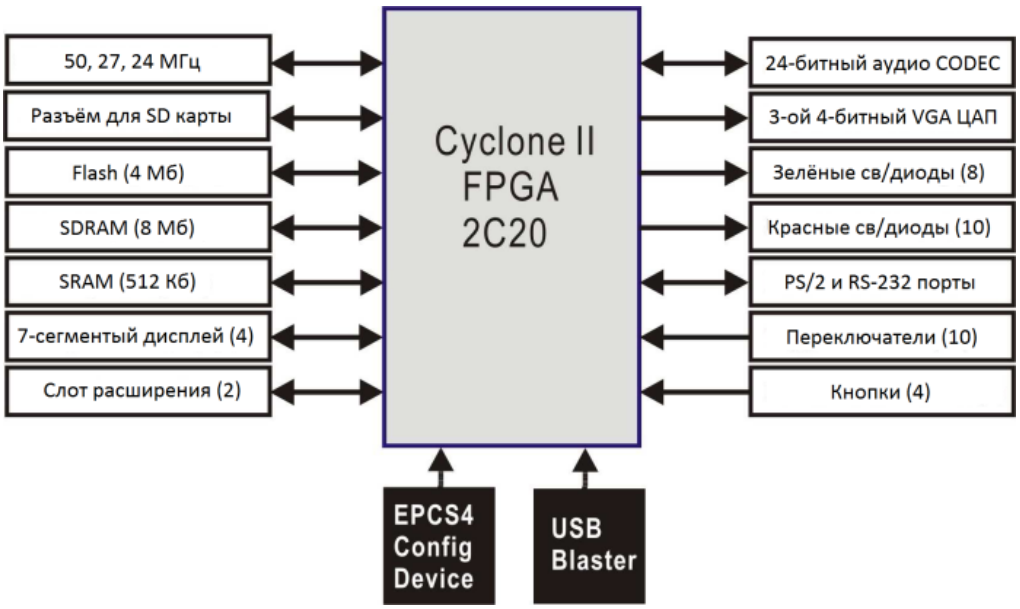


Рис.2. Структура учебного стенда

Базовый проект для ПЛИС реализован в графическом виде (BasicProject.bdf) и состоит из двух основных блоков: InputModule и OutputModule (рис.3), которые предназначены для ввода данных от кнопок, переключателей и клавиатуры и вывода данных на светодиоды и семисегментные индикаторы соответственно. *Для обеспечения работоспособности базового проекта необходимо сохранять без изменений все входные подключения модуля InputModule и все выходные подключения, а также вход clk модуля OutputModule.*

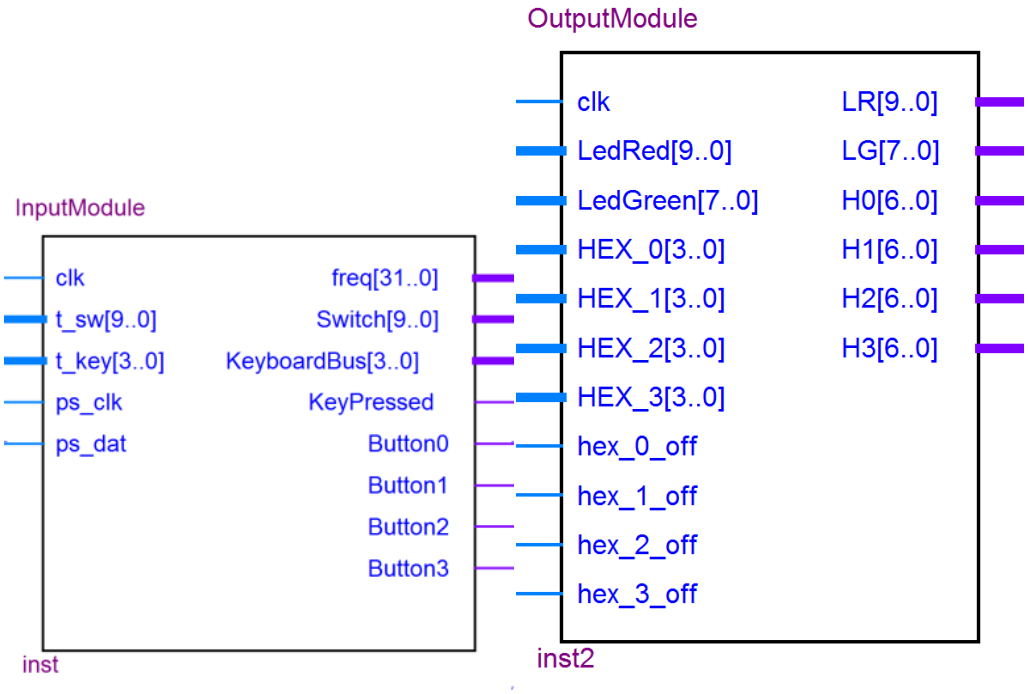


Рис.3. Модули ввода/вывода

На стенде установлен генератор базовой тактовой частоты 50 МГц. С выхода Freq[0..31] можно снимать тактовые частоты с делителя базовой частоты (Freq[0] - 25 МГц, Freq[1] - 12,5 МГц и т.д.).

По шине Switch[0..9] передается состояние десяти переключателей, расположенных на плате снизу. Логическая «1» соответствует состоянию «Включено», логический «0» - «Выключено».

По шине KeyBoardBus[3..0] передается код нажатой клавиши с клавиатуры (используются шестнадцатеричные цифры от 0 - 9 и A - F). Каждое нажатие клавиши сопровождается логической «1» на сигнале KeyPressed.

Сигналы Buton0, Button1 и Buton2, Button3 отображают состояние двух пар кнопок, расположенных на плате.

Шины HEX_0[3..0], HEX_1[3..0], HEX_2[3..0], HEX_3[3..0] предназначены для передачи информации на четыре семисегментных индикатора, расположенных на плате. HEX_0 соответствуют крайнему правому индикатору, HEX_1 - второму справа и т.д. Управление отдельными сегментами реализовано в модуле OutputModule и ориентировано на отображение шестнадцатеричных цифр, т.е. при HEX_0[3..0] = 0000 на крайнем правом индикаторе будет отображен символ нуля, а при HEX_0[3..0] = 1111 - символ F. Сигналы hex_0_off, hex_1_off, hex_2_off и hex_3_off отвечают за засветку семисегментных индикаторов. Например, если hex_0_off выставить в единицу, индикатор HEX_0 погаснет.

С помощью шин LedRed[9..0] и LedGreen[7..0] можно управлять линейками светодиодов, расположенных на плате. Например, подача «1» на LedGreen[0] активирует крайний справа светодиод.

Помимо описанных выше модулей ввода/вывода в состав базового проекта входит модуль test (рис.4). Функцией данного модуля является формирование тестовых воздействий на внешние устройства, чтобы продемонстрировать работоспособность учебного стенда.

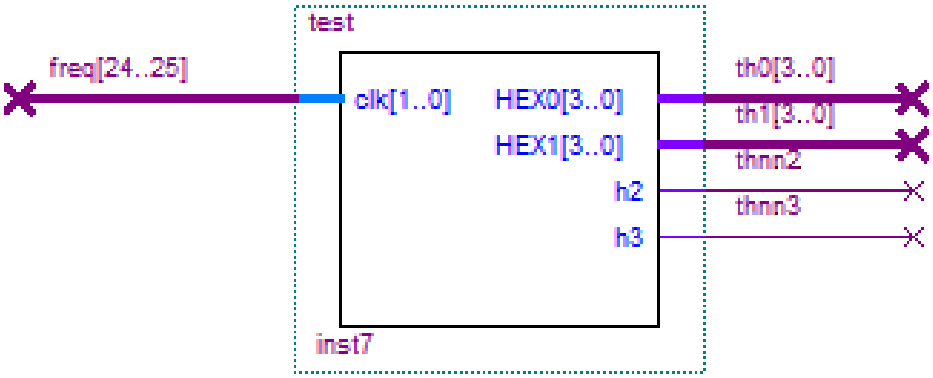


Рис.4. Модуль организации тестовых воздействий

Пример проектирования

На рис.5 приведен пример схемы вычислительного устройства с микропрограммным управлением, дополненной модулями ввода/вывода. Данная схема имеет четыре входных порта (clk - тактовый сигнал, reset - сигнал начальной установки, KB - код нажатой клавиши, KP - индикатор нажатой клавиши) и один выходной порт (Indicator - коды для семисегментных индикаторов).

Модули ввода/вывода описаны на Verilog HDL, их листинги представлены на рис.6 и 7 соответственно. Из приведенных описаний видно, что оба модуля имеют регистровую память, адресуемую с помощью шины PortID.

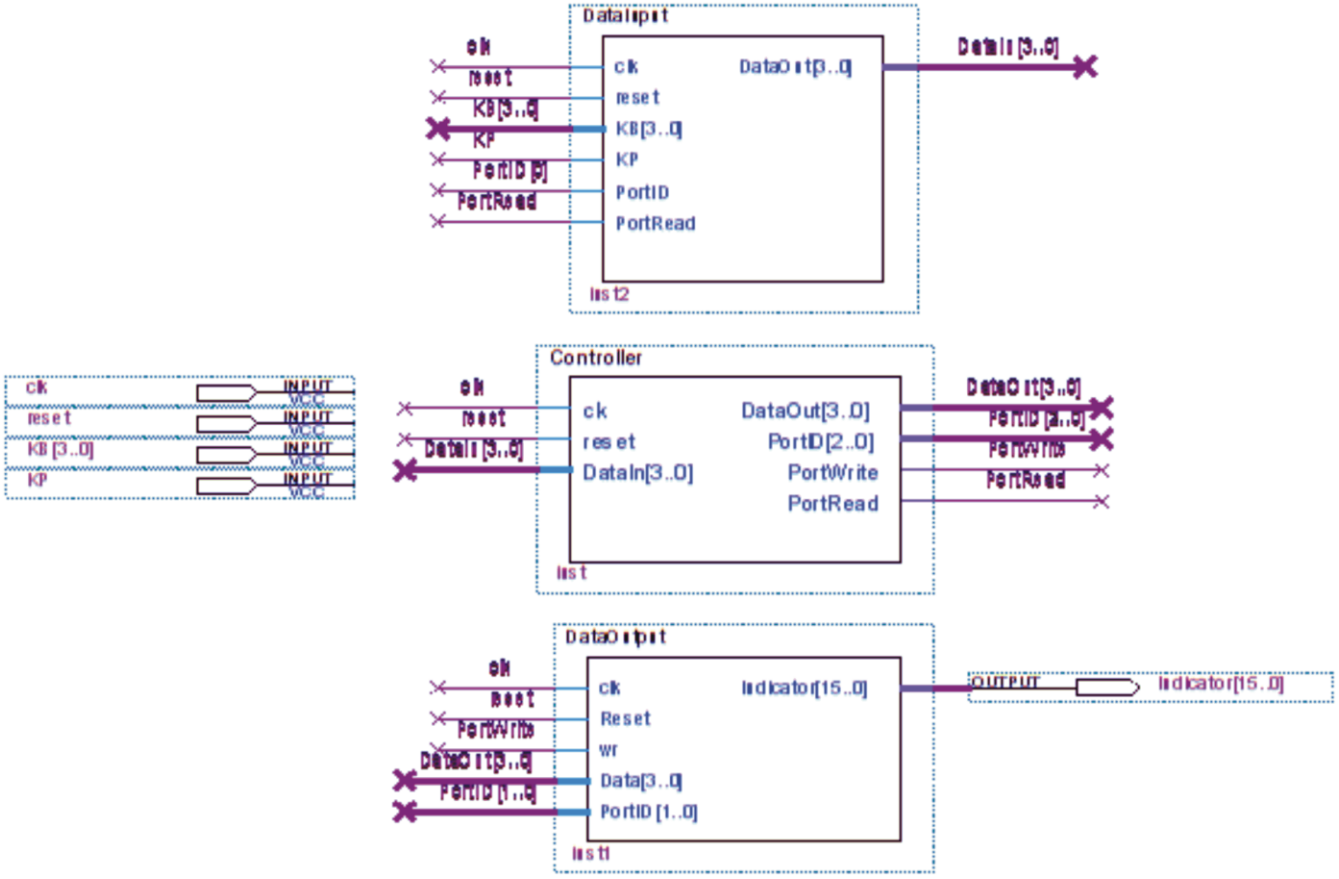


Рис.5. Схема вычислительного устройства с микропрограммным управлением, дополненная модулями ввода/вывода

```
1   module DataInput (  input               clk,
2                       input               reset,
3                       input       [3:0]   KB,
4                       input               KP,
5                       input               PortID,
6                       input               PortRead,
7                       output      [3:0]   DataOut );
8
9   reg         KPD, Ready;
10  reg [3:0]   Data;
11
12  always @ (posedge clk)
13      if (reset) begin
14          KPD <= 1'b0;
15          Data <= 4'b0000;
16      end else begin
17          KPD <= KP;
18          if (KP & ~KPD) Data <= KB;
19      end
20
21  always @ (posedge clk)
22      if (reset | (PortRead & Ready & ~PortID)) Ready <= 1'b0;
23          else if (KP & ~KPD) Ready <= 1'b1;
24
25  assign DataOut = PortID ? Data : {3'b000, Ready};
26
27  endmodule
```

Рис.6. Модуль ввода данных

```
1   module DataOutput (clk, Reset, wr, Data, PortID, Indicator);
2
3   input           clk, Reset, wr;
4   input   [3:0]   Data;
5   input   [1:0]   PortID;
6   output  [15:0]  Indicator;
7
8   reg     [15:0]  Indicator;
9
10  always @ (posedge clk)
11      if (Reset) Indicator <= 16'b0;
12          else if (wr)
13              case (PortID)
14                  0: Indicator[3:0] <= Data;
15                  1: Indicator[7:4] <= Data;
16                  2: Indicator[11:8] <= Data;
17                  3: Indicator[15:12] <= Data;
18              endcase
19  endmodule
```

Рис.7. Модуль вывода данных на семисегментный индикатор

Модуль ввода состоит из двух регистров, доступных для чтения. По нулевому адресу находится регистр готовности данных от клавиатуры, а по первому - сами данные. Таким образом, УМУ должно опрашивать регистр готовности и при наличии в нем «1» считывать данные из первого регистра. Отметим, что регистр готовности сбрасывается при его чтении.

Модуль вывода состоит из четырех адресуемых регистров, каждый из которых соответствует определенному семисегментному индикатору. Так, запись данных в нулевой регистр приводит к отображению этих данных на крайнем правом индикаторе.

В табл.1 представлен пример микропрограммы, реализующей лабораторное задание на примере сложения двух 4-разрядных чисел.

Требования к отчету

В качестве отчета о выполнении лабораторной работы вы должны представить проект Quartus, содержащий схемы, Verilog-описания, тестовые воздействия и результаты моделирования схемы устройства, выполняющей один из вариантов индивидуального задания, а также продемонстрировать его работу на учебном стенде.

Таблица 1

Пример микропрограммы

№	Действие	Микрокоманда								
		Код перехода	S[3:0]	M	P_0	ISR, ISL	A	wr	adr[2:0]	v[3:0]
0	1→PrB	000	0000	0	0	01	0	0	000	0010
1	DataIn-0→PrA	000	0000	0	0	00	1	0	000	0001
2	PrA–PrB	000	0110	1	1	00	0	0	000	0000
3	If R==0 PC=5	100	0000	0	0	00	0	0	000	0101
4	PC=1	111	0000	0	0	00	0	0	000	0001
5	DataIn-1→PrA	000	0000	0	0	00	1	0	000	0001
6	PrA→DataOut-0	000	0000	0	0	00	0	0	000	1000
7	PrA→БРОН-0	000	0000	0	0	00	0	1	000	0000
8	DataInput→PrA	000	0000	0	0	00	1	0	000	0001
9	PrA–PrB	000	0110	1	1	00	0	0	000	0000

Продолжение табл.1

№	Действие	Микрокоманда								
		Код перехода	S[3:0]	M	P_0	ISR, ISL	A	wr	adr[2:0]	v[3:0]
10	If R==0 PC=12	100	0000	0	0	00	0	0	000	1101
11	PC=8	111	0000	0	0	00	0	0	000	1000
12	DataIn-1→PrA	000	0000	0	0	00	1	0	001	0001
13	PrA→DataOut-0	000	0000	0	0	00	0	0	000	1000
14	PrA→БРОН-1	000	0000	0	0	00	0	1	001	0000
15	DataIn-0→PrA	000	0000	0	0	00	1	0	000	0001
16	PrA–PrB	000	0110	1	1	00	0	0	000	0000
17	If R==0 PC=19	100	0000	0	0	00	0	0	001	0011
18	PC=15	111	0000	0	0	00	0	0	000	1111
19	15→PrB	000	0000	0	0	01	0	0	000	0010

Продолжение табл.1

№	Действие	Микрокоманда								
		Код перехода	S[3:0]	M	P_0	ISR, ISL	A	wr	adr[2:0]	v[3:0]
20	15→РгВ	000	0000	0	0	01	0	0	000	0010
21		000	0000	0	0	01	0	0	000	0010
22	DataIn-1→РгА	000	0000	0	0	00	1	0	001	0001
23	РгА–РгВ	000	0110	1	1	00	0	0	000	0000
24	If R=0 PC=29	100	0000	0	0	00	0	0	001	1101
25	1→РгВ	000	0000	0	0	00	0	0	000	0100
26		000	0000	0	0	00	0	0	000	0100
27		00	0000	0	0	00	0	0	000	0100
28	PC=15	111	0000	0	0	00	0	0	000	1111
29	БРОН-0→РгВ	000	0000	0	0	00	0	0	000	0110

Окончание табл.1

№	Действие	Микрокоманда								
		Код перехода	S[3:0]	M	P_0	ISR, ISL	A	wr	adr[2:0]	v[3:0]
30	БРОН-1→РгА	000	0000	0	0	00	0	0	001	0001
31	РгА+ РгВ→DataOut-0	000	1001	1	0	00	0	0	000	1000
32	0→РгВ	000	0000	0	0	00	0	0	000	0010
33		000	0000	0	0	00	0	0	000	0010
34		000	0000	0	0	00	0	0	000	0010
35		000	0000	0	0	00	0	0	000	0010
36	PC=0	111	0000	0	0	00	0	0	000	0000

Литература

1. ***Угрюмов А.Л.*** Цифровая схемотехника. - СПб.: БХВ-Петербург, 2002. - 528 с.

2. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std. 1364-1995).

3. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std. 1364-2001).

4. IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std. 1364-2005).

5. IEEE Standard for Verilog Register Transfer Level Synthesis (IEEE Std. 1364.1-2002).

6. ***Стемпковский А.Л., Семенов М.Ю.*** Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: учеб. пособие. - М.: МИЭТ, 2005. - 140 с.

7. Микропроцессорные средства и системы: курс лекций / ***Д.Н. Беклемишев, А.Н. Орлов, А.Л. Переверзев и др.; под ред. Ю.В. Савченко.*** - М.: МИЭТ, 2013. - 288 с.