МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

К.И. СУХАЧЕВ, Д.В. РОДИН, А.С. ДОРОФЕЕВ

ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ ЦИФРОВОЙ ЭЛЕКТРОНИКИ НА БАЗЕ ПРОГРАММИРУЕМОЙ ЛОГИКИ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для обучающихся по основным образовательным программам высшего образования по направлениям подготовки 11.03.03, 11.04.03 Конструирование и технология электронных средств, 11.03.01 Радиотехника

САМАРА
Издательство Самарского университета
2022

УДК 004.43(075)
ББК 32.973.26я7
С910

Рецензенты: д-р техн. наук, проф. Д. П. Т а б а к о в ,
канд. техн. наук, доц. И. А. К у д р я в ц е в

Сухачев, Кирилл Игоревич
С910 Особенности проектирования цифровой электроники на базе программируемой логики: учебное пособие / *К.И. Сухачев, Д.В. Родин, А.С. Дорофеев.* – Самара: Издательство Самарского университета, 2022. – 200 с.: 72 ил.

ISBN 978-5-7883-1732-8

В данном учебном пособии представлена информация об основных семействах программируемых логических микросхем, дано их сравнение и области применения. Представлен вводный курс в HDL Verilog, проведено сравнение с другими популярными HDL и графическим методом разработки дизайна ПЛИС. Дано описание среды разработки Quartus free, на примере полного цикла разработки реального проекта.

Представлены разработанные авторами исходные коды различных ip-модулей, доступные к повторению и свободному использованию, в том числе некоторые варианты реализации вычислителей, процессорных ядер, модулей интерфейсов и т.д.

Приведены задания для самостоятельного и группового выполнения на практических и лабораторных занятиях.

Изложенные материалы позволят обучающимся получить базовые знания и практические навыки из области программируемой логики. Обучающиеся старших курсов смогут самостоятельно сначала повторить представленные в пособии примеры, а после самостоятельно проводить разработку и верификацию своих проектов.

УДК 004.43(075)
ББК 32.973.26я7

ISBN 978-5-7883-1732-8

ОГЛАВЛЕНИЕ

1 ВВЕДЕНИЕ В ПРОГРАММИРУЕМУЮ ЛОГИКУ 5

2 ОБЛАСТИ ПРИМЕНЕНИЯ ПЛИС И ВЫБОР ИМС 20

3 ОСНОВЫ VERILOG HDL 28

3.1 Примеры комбинаторных модулей на Verilog_HDL 36

4 АЛГОРИТМЫ МАШИННОЙ АРИФМЕТИКИ 38

4.1 Алгоритм умножения 38

4.2 Алгоритм извлечения квадратного корня 38

4.3 Алгоритмы деления 42

5 ОСНОВЫ РАБОТЫ В СРЕДЕ QUARTUS II 47

5.1 Разработка описания в графическом редакторе 50

5.2 Разработка описания на структурном языке AHDL 60

5.3 Разработка описания на поведенческом языке VHDL 63

5.4 Разработка описания на поведенческом языке Verilog 65

5.5 Полный цикл разработки проекта для ПЛИС 67

6 ОСНОВНЫЕ ПОДХОДЫ ПРИ РАЗРАБОТКЕ ОПИСАНИЯ ДЛЯ ПЛИС 86

6.1 Пример разработки учебного софт-процессора для ПЛИС 87

6.2 Пример разработки модулей периферии учебного софт-процессора 100

6.3 Пример разработки учебного процессора для синтезируемой SoC 119

7 ПРАКТИЧЕСКИЕ ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ 133

7.1 Практическое занятие №1. Работа со счетчиками и разработка ШИМ контроллера 133

7.2 Практическое занятие №2.
Разработка многофазного контроллера блока питания 138
7.4 Практическое занятие №4.
Разработка простого учебного процессорного ядра 142
7.7 Практическое занятие №7. Разработка сопроцессора 152
7.8 Практическое занятие №8. Разработка СнК
на базе ПЛИС 153

СПИСОК ЛИТЕРАТУРЫ 155

Приложение А. Описание периферийных модулей
на Verilog 157
Приложение Б. Описание процессорного ядра NMR
на Verilog 170
Приложение В. Система команд «NMR» 194
Приложение Г. Описание отладочной платы
и ИМС MAXII 197
Приложение Д. Описание отладочной платы DE0-Nano 198
Ресурсы СБИС Cyclone IV E 199

1 ВВЕДЕНИЕ В ПРОГРАММИРУЕМУЮ ЛОГИКУ

По мере развития науки и техники изменились структура и принципы построения электрических схем, изменилась и элементная база, используемая при разработке электронной аппаратуры ЭА. Если тридцать лет назад большинство схем были аналоговыми и только индикаторные компоненты были логическими или цифровыми, то в последнее время аналоговые компоненты используются только для обеспечения первичного преобразования физических величин (различные датчики), в цепях питания и пропорциональных исполнительных устройствах, остальная часть схемы – цифровая. В большинстве случаев цифровая схема состоит из центрального процессора ЦП или микроконтроллера МК, а также схемы коммутатора, позволяющего подключать к ЦП или МК различные периферийные устройства. Типовая структура современного цифрового устройства показана на рисунке 1.1.

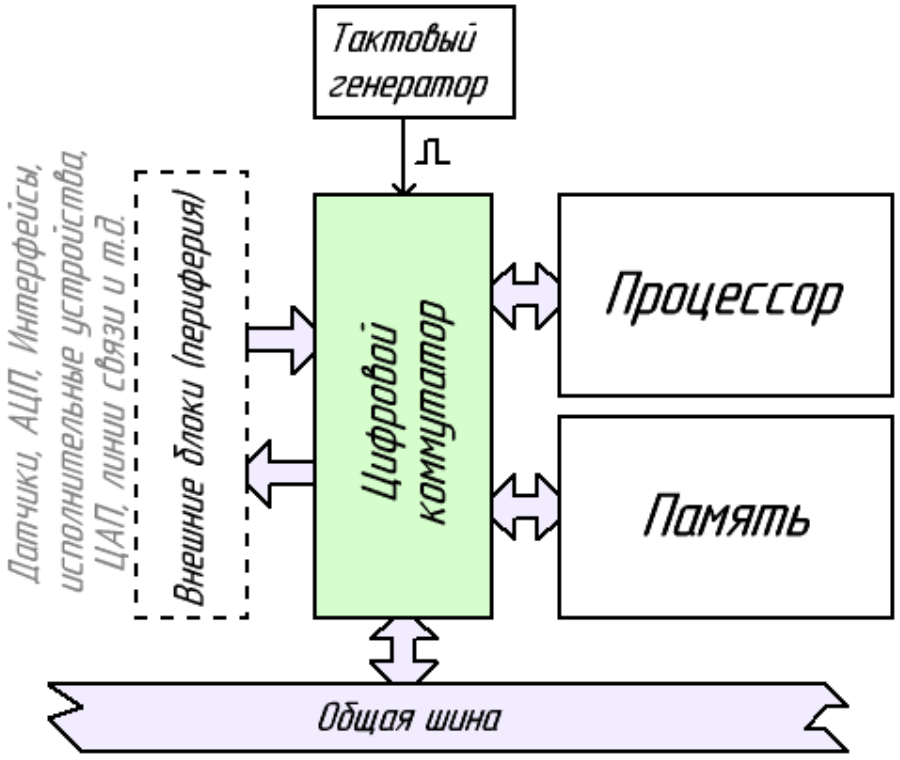


Рисунок 1.1 – Структурная схема цифровых устройств

Использование универсального процессора делает удобным написание программного обеспечения (ПО) для него и последующую отладку, а цифровой коммутатор разрабатывается специально для каждой схемы, в зависимости от структуры периферии, выполняемых задач и функций устройства. Часто коммутатор может состоять из большого числа дискретных логических элементов, таких как мультиплексоры, сдвиговые регистры, триггеры, логические вентили и другие.

Использование дискретной логики существенно усложняет печатную плату (рисунок 1.2), не позволяет снизить массу и габариты устройства, а также не позволяет без изменения топологии платы и элементной базы вносить изменения в логику работы схемы, проводить модернизацию или оперативно исправлять ошибки, например, выявленные при испытаниях устройства.

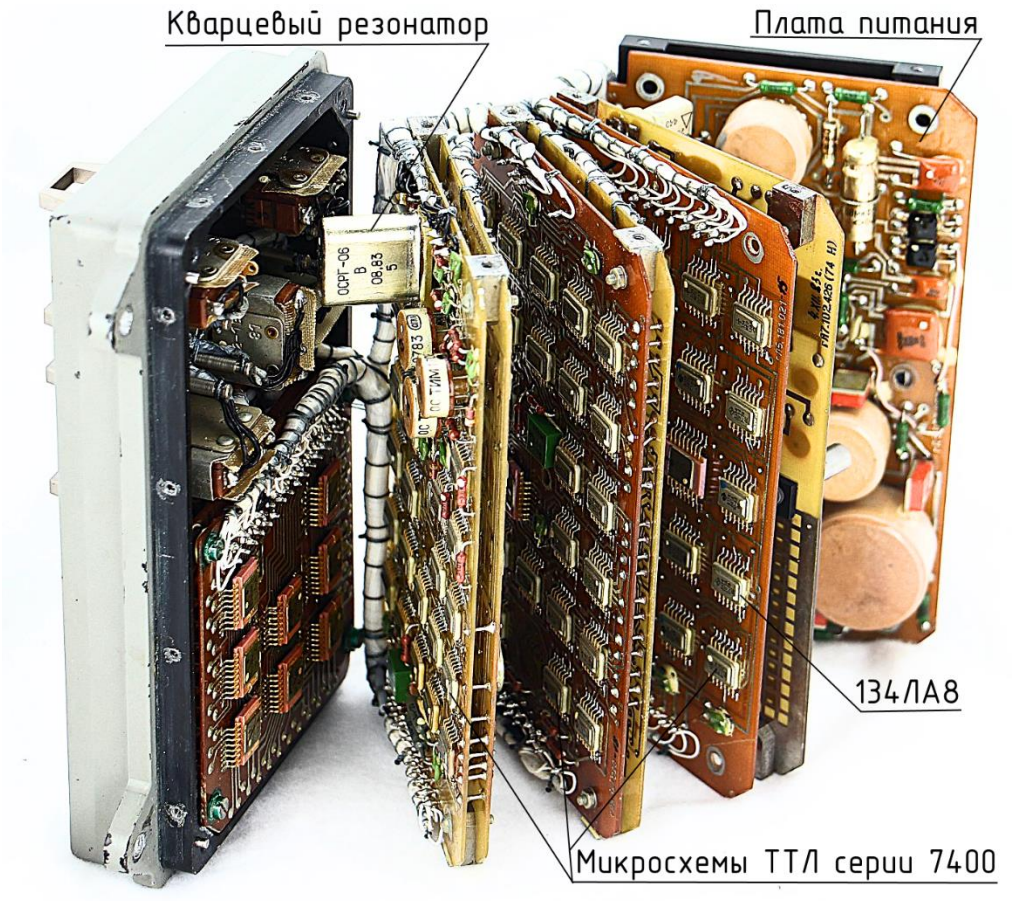


Рисунок 1.2 – Бортовой модуль «часов» с корабля «Союз»

Стремление разработчиков собрать все или большинство ИМС в единый корпус понятно, но из-за того, что для каждого устройства набор компонентов и соединения между ними уникальны, унифицировать коммутатор и сделать его универсальным невозможно. Возникает необходимость того, что коммутатор должен быть программируемым и состоять из набора стандартных компонентов и программируемых ключей, позволяющих создавать необходимые соединения между компонентами. Такие микросхемы были созданы и называются «программируемые логические интегральные микросхемы – ПЛИС».

Простейшие ПЛИС – это программируемые логические матрицы или ПЛМ. Исторически они появились первыми и состоят из элементов «НЕ», «ИЛИ», «И». В зарубежной литературе ПЛИС данного типа имеют обозначение FPLA (Field Programmable Logic Array) или FPLS (Field Programmable Logic Sequensers). Данная структура морально устарела и применяется для замещения относительно не сложных логических схем, данные микросхемы относятся к ИМС средней степени интеграции. Принцип действия ПЛМ основан на том, что любая логическая операция может быть представлена как комбинация упомянутых выше логических элементов. Для алгоритмов цифровой обработки сигналов (ЦОС) или конечных автоматов ПЛМ не пригодны. Примеры ПЛМ представлены в таблице 1.1.

Таблица 1.1 – Примеры и характеристики ИМС типа ПЛМ

Модель микро-схемы	Кол-во интегральных элементов	Кол-во «И»	Кол-во «ИЛИ»	Кол-во входов	Тв	Особенности
КР556РТ1 аналог: N82S101	5600	48	8	16	50 нс	Открытый коллектор, потребление 1Вт
КР556РТ2 аналог: PLS100	6150	48	8	16	50 нс	Три состояния выводов, потребление 1Вт

Окончание табл. 1.1

Модель микро-схемы	Кол-во интегральных элементов	Кол-во «И»	Кол-во «ИЛИ»	Кол-во входов	Тв	Особенности
КР556РТ22 аналог: 82S167AC	6050	48	8	14	20 нс	Три состояния выводов, потребление 900мВт
КР556РТ21 аналог: N82S105A	6100	48	8	16	30 нс	Открытый коллектор, потребление 900мВт
583РТ1	–	20	18	20	50 нс	Потребление 880мВт

В попытке избавиться от недостатка микросхем ПЛМ, связанного со слабым использованием матрицы элементов «ИЛИ», появился подкласс ПЛИС типа ПМЛ (программируемая матричная логика) или PAL (Programmable Array Logic) в зарубежной литературе. Однако они тоже не получили широкого распространения, как следующие более сложные ПЛИС, названные программируемыми коммутируемыми матричными блоками, или ПКМБ и CPLD в зарубежной литературе соответственно. ПКМБ – это ПЛИС, содержащая несколько логических матричных блоков, представляющих собой структуру, схожую с ПМЛ, которые соединены общей коммутационной матрицей. Структура ПЛИС типа ПКМБ представлена на рисунке 1.3. ПКМБ обычно имеют высокую степень интеграции и содержат до сотен макроячеек и десятков тысяч эквивалентных логических вентилей. Макроячейки, входящие в состав ПКМБ или CPLD (Complex Programmable Logic Device), соединены с блоками ввода-вывода и связаны между собой внутренними коммутационными шинами. Представителями микросхем CPLD являются семейства MAX5000, MAX7000, MAX, и MAXII фирмы Altera (Intel), а также XC7000 и XC9500 фирмы Xilinx.

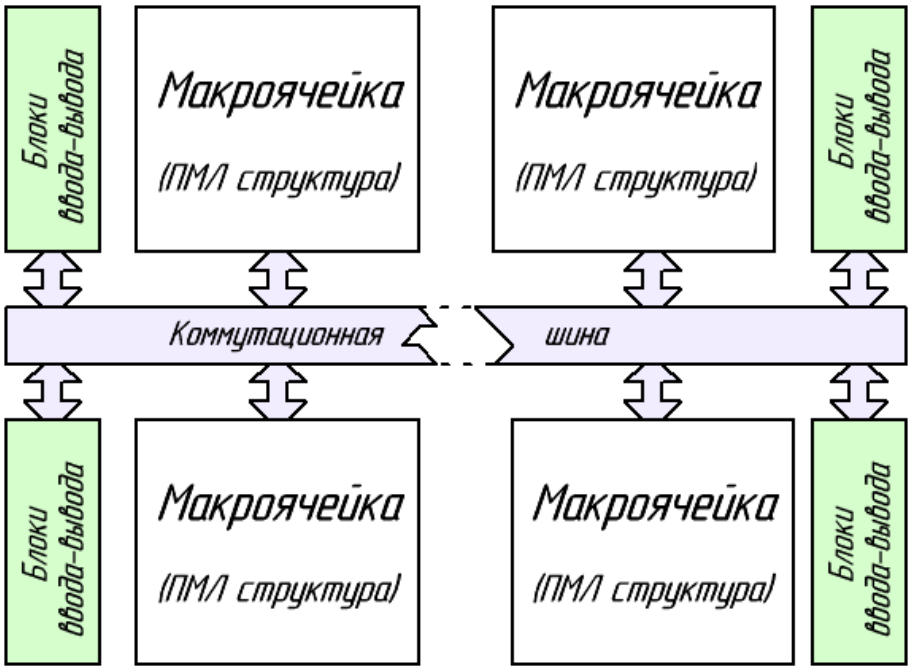


Рисунок 1.3 – Структура ПЛИС типа ПКМБ (CPLD)

ПЛИС типа ПКМБ оказались удобны и часто применяются для организации несложных цифровых автоматов. Однако степень интеграции все еще не достаточна высока для алгоритмов ЦОС и других сложных задач.

Следующим еще более сложным типом ПЛИС являются программируемые вентильные матрицы ПВМ, в зарубежной литературе называются FPGA (Field-Programmable Gate Array). ПВМ состоят из логических блоков ЛБ, программируемых соединительных матриц МС и коммутаторов К, как показано на рисунке 1.4.

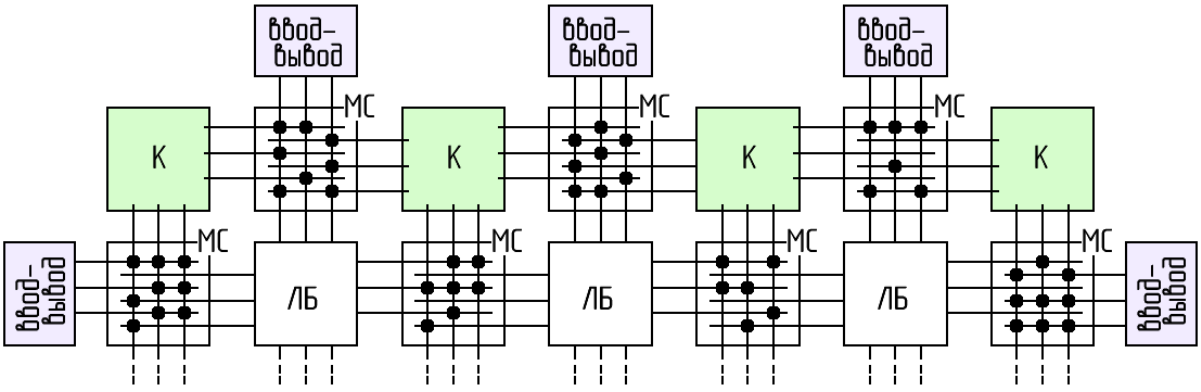


Рисунок 1.4 – Структура ПЛИС типа ПВМ (FPGA)

Логические блоки ПВМ состоят из относительно простых логических элементов, в основе типового логического блока лежит таблица перекодировки LUT (Look-Up Table), программируемый мультиплексор, Д-триггер и управляющие цепи. Обобщенная структура логического блока ПВМ показана на рисунке 1.5.

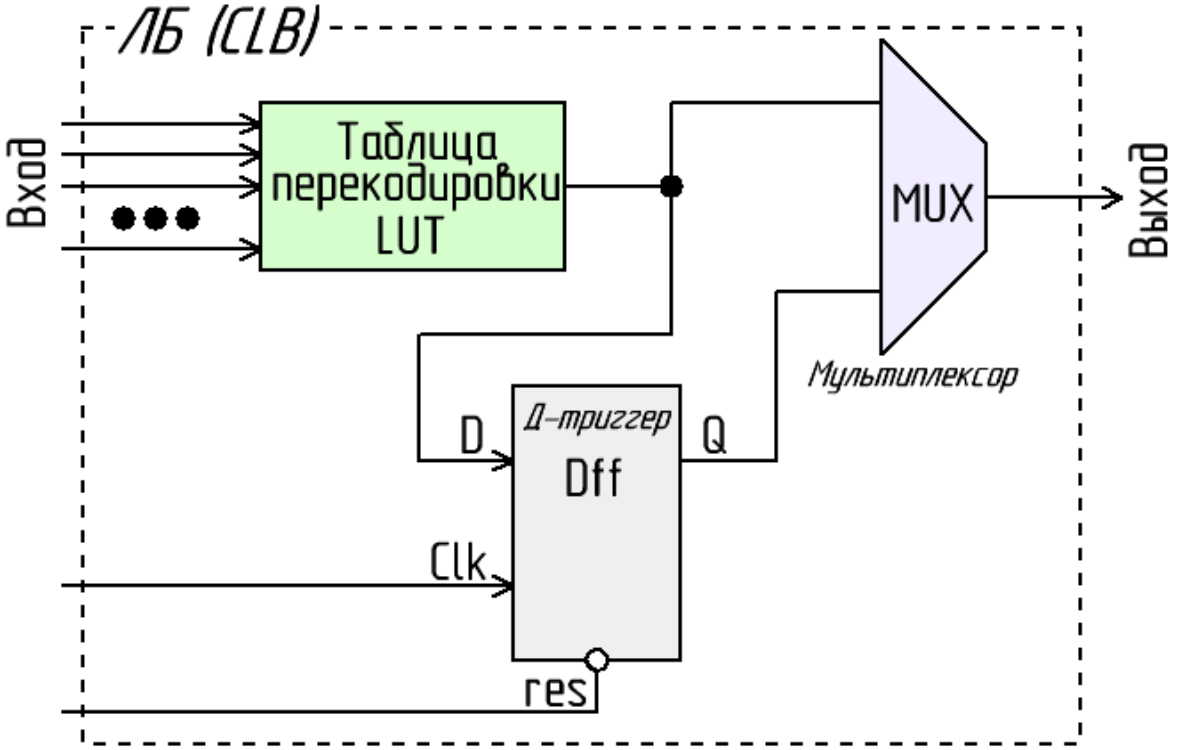


Рисунок 1.5 – Типовая структура
логического блока ПВМ ПЛИС (CLB FPGA)

В современных ПЛИС типа ПВМ количество таких ЛБ может достигать десятков и сотен тысяч. В сочетании со встроенными аппаратными умножителями, синтезаторами частоты и блоками встроенной конфигурируемой памяти ПВМ удобны для задач ЦОС и формирования сложных систем управления вплоть до синтеза софт-процессоров и вычислительных систем на их основе. Блоки ввода-вывода в ПВМ позволяют настроить отдельно взятый вывод микросхемы как вход, выход или как двунаправленный порт, определить подтяжку вывода к земле или к питанию, а также

переключить его в третье состояние. Все современные ПЛИС поддерживают тестирование узлов с помощью интерфейса JTAG. Данная архитектура ПЛИС универсальна и находит самые разнообразные применения в РЭС. Существует множество моделей ПВМ (FPGA) от различных производителей, таких как Intel (Altera), Xilinx, lattice, Actel (Microsemi), ВЗПП-С, ПКК Миландр и другие.

Дальнейшее развитие технологии ПЛИС идет по пути объединения в одной микросхеме архитектур CPLD и FPGA, а также по пути введения аппаратных ядер процессоров, интерфейсных блоков, таких как Ethernet, PCI, LVDS, контроллеры DDR и других. В состав ПЛИС вводятся различные типы памяти и прочая периферия, например блоки АЦП и ЦАП. Такие системы принято называть системами на кристалле или SOC в зарубежной литературе. Более подробно про архитектуру и выбор ПЛИС под конкретную задачу будет рассказано далее. На начальном этапе можно использовать ПЛИС, независимо от того, FPGA это или CPLD, не задумываясь, как она работает. Будем считать, что с её помощью можно (с разной степенью эффективности) организовать любую логическую функцию, если на это хватает количества логических элементов ПЛИС.

Для хранения информации о конфигурации ПЛИС в настоящее время используется несколько основных технологий, такие как статическое ОЗУ (SRAM), электрически перепрограммируемое ПЗУ (EPROM или Flash), в особо ответственных областях применения используется однократно программируемая память на основе технологии «Antifuse». Независимо от технологии хранения, создание файла конфигурации современных ПЛИС невозможно без автоматизированной среды разработки, которая существует у всех ведущих производителей ПЛИС. Такие системы, как QuartusII от Intel, Vivado от Xilinx, Actel Libero IDE от Actel или

Synopsys Synplify + компилятор iCEcube2 для работы с Lattice, существуют и в платных, и в бесплатных версиях, с различным уровнем функциональных возможностей. При работе в большинстве сред разработки описание схемы, которую необходимо реализовать внутри ПЛИС, может быть получено двумя различными способами, такими как: описание на специальных HDL языках (Verilog, System Verilog, VHDL, AHDL, System C), либо создание схемы в графическом редакторе, в котором разрабатывается именно электрическая схема с использованием стандартных логических элементов или ранее созданных ip-блоков. Возможно совместное использование всех способов описания, например, блоки нижнего уровня описываются на HDL, а файл верхнего оформляется в графическом представлении в виде структурной схемы блоков. Тестовые блоки удобно создавать на System Verilog. Комбинированный подход позволяет наиболее наглядно представить работу системы, не вдаваясь в частности работы каждого отдельного блока, рассматривая их как «черный ящик». Реже используют описание в виде графиков входных и выходных сигналов. Работу по трассировке соединений внутри ПЛИС с проведением оптимизации топологии для обеспечения минимальной задержки прохождения сигнала и минимального занимаемого объема внутри обеспечивает уже автоматизированная среда разработки, которая так же обеспечивает создание в необходимом формате файла прошивки и осуществляет программирование. Часто в среду так же встроены системы моделирования и отладки проекта.

Конкурирующими по отношению к ПЛИС являются другие технологии, позволяющие получить пользователю микросхемы с уникальными параметрами и функционалом. Такими технологиями является ИМС на основе базовых матричных кристаллов БМК, либо заказная интегральная микросхема специального назначения ASIC (Application-specific integrated circuit).

В отличие от ПЛИС, БМК программируется технологически, путём нанесения маски соединений одного или нескольких последних слоев металлизации. Заказчиком разрабатывается описание целевой схемы, которая на предприятии изготовителя БМК преобразуется в схему соединений, и далее – в маски слоев. Они переносятся в качестве последних слоев на базовый матричный кристалл, и разобщенные элементарные элементы, входящие в состав БМК складываются в одну большую схему, соответствующую дизайну заказчика.

В отличие от обычных интегральных схем общего назначения, специализированные интегральные схемы ASIC применяются в конкретном устройстве и выполняют строго ограниченные функции, характерные только для него. Технология ASIC предлагает решения, обладающие более высокой скоростью при сниженном энергопотреблении, чем у FPGA. Разница в скорости между двумя методами проектирования отличается на порядок и более. Исторически разработка ASIC включала трассировку и размещение в проекте биполярных транзисторов или N-FET. Поскольку процесс у каждого производителя был разным, описания структуры также должны были меняться от одного производителя к другому. Существовал инструментарий (ПО САПР), чтобы помочь в разработке, но эти инструменты были несовместимы между предприятиями. В результате проекты, которые работали с техпроцессом одного производителя, не могли быть переданы другому, без существенной, иногда полной переработки. Ситуация изменилась со стандартизацией используемых элементов и введением стандартных ячеек (StandardCell). Стандартные ячейки аналогичны логическим блокам в ПЛИС. Таким же образом, как произвольная логика может быть создана посредством коммутационной матрицы логических блоков в FPGA, произвольная логика может быть создана с использованием стандартных ячеек и внутри ASIC.

Для обеспечения непрерывного цикла проектирования перспективной радиоэлектронной аппаратуры с использованием функциональных узлов на основе БМК или ASIC флагманами радиоэлектронной промышленности, такими как фирмы Intel (Altera) и Xilinx, было предложено создавать свои семейства БМК на базе прототипов FPGA проектов. Данная методология проектирования устойчива к позднему обнаружению ошибок и позволяет снизить стоимость проектов ASIC на 25–80 %, одновременно сократив сроки проектирования.

Существует возможность прототипирования и отладки проекта с использованием FPGA, а после полной верификации дизайна – заказа у производителя БМК с функционалом, идентичным и отлаженным на ПЛИС. Причем в большинстве случаев для изготовления БМК производителю достаточно полученного описания RTL схемы или описания на HDL. На рисунке 1.6 показан пример, когда для экономии времени и средств на изготовление одна и та же печатная плата применяется и для отладки на ПЛИС, и для финальной версии проекта.

При разработке электроники в настоящее время наибольшую популярность приобрели так называемые системы на кристалле СНК или SoC, причем данные системы могут быть как с аппаратными ip-модулями, так и с синтезируемыми, т.е. любая ИМС ПЛИС, если позволяет количество логических элементов, при соответствующей прошивке может выполнять задачи целой системы и будет являться SoC. Устройства, в которых применены комплексные технические решения на базе SoC, т.е. электронная схема, выполняющая функции целого устройства и размещённая на одной интегральной схеме, будут обладать конкурентными преимуществами перед аналогами, в частности, обладать более высокой надежностью и стойкостью к внешним факторам.

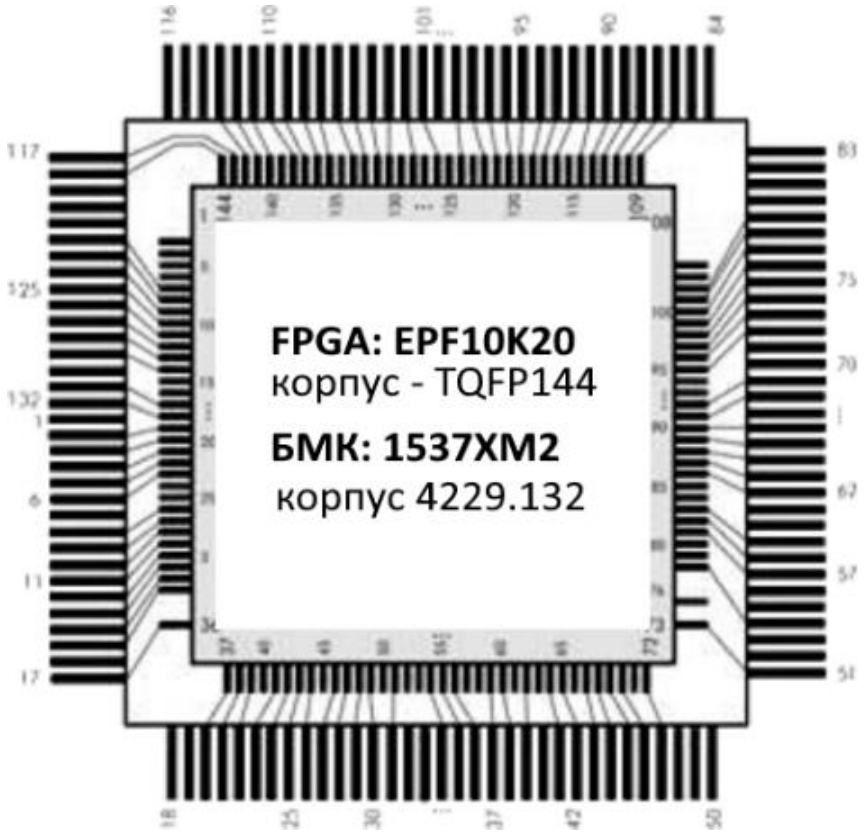


Рисунок 1.6 – Вариант топологии проекта
с возможностью установки и ПЛИС, и БМК

Типичная SoC содержит:

– один или несколько процессоров, дополнительно может содержать DSP сопроцессоры; SoC, содержащую несколько процессоров, называют *многопроцессорной системой на кристалле*;

– блок памяти, состоящий из модулей ПЗУ, ОЗУ, ППЗУ и Flash, либо сочетание типов памяти;

– источники опорной частоты, кварцевые или RC генераторы и схемы ФАПЧ (фазовой автоподстройки частоты) или PLL;

– таймеры, счётчики, цепи задержки;

– блоки, реализующие стандартные интерфейсы для подключения внешних устройств: USB, FireWire, Ethernet, USART, SPI и другие;

– блоки цифро-аналоговых и аналого-цифровых преобразователей;

– программируемые порты ввода-вывода;

– блоки программируемой логики.

Существуют три варианта построения уникальной SoC:

– в виде заказной СБИС (ASIC);

– в виде полузаказной СБИС на базе БМК;

– на базе ПЛИС высокой интеграции (FPGA).

Все варианты реализации имеют свои достоинства и недостатки, которые целесообразно оценить в сравнении с традиционным способом монтажа систем на печатной плате из отдельных микросхем – системами на плате или системами на модуле СнМ или SoM.

Преимущества систем на плате или SoM:

– использование хорошо проверенных серийных компонентов;

– более простой процесс тестирования и отладки при относительно простых системах;

– возможность замены неисправных компонентов;

– низкая стоимость создания опытных образцов и малых серий.

Пример типичной системы на плате, содержащей несколько корпусов ИМС, представлен на рисунке 1.7.

Данная СнМ является многоканальным блоком высокоскоростных и синфазных измерений с последующей обработкой данных. Конструктивно СнМ выполняется в виде небольших модулей с возможностью монтажа на коммутационных платах. Представленная СнМ выполнена на четырехслойной печатной плате и имеет размеры 68мм × 68мм.

Пример, приведенный на рисунке 1.7, содержит относительно большую по емкости ПЛИС EP3C120F780, которая при определенных случаях и соответствующем описании может считаться СнК сама по себе, однако в рамках данного примера из-за

наличная внешних ИМС ОЗУ: IS61WV1024, АЦП: AD9288 и AD7606 и ИМС Flash-памяти 45db161, система является модулем СнМ.

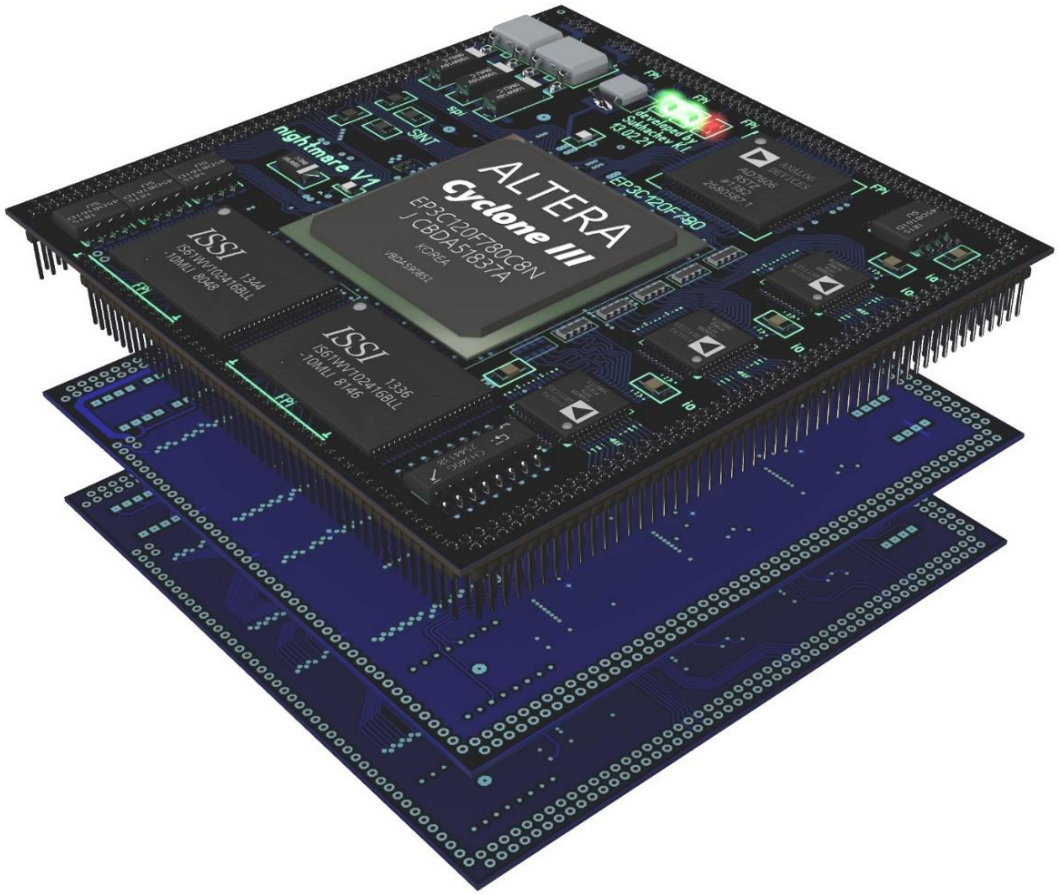


Рисунок 1.7 – Пример системы на плате, разработки ИКП, содержащей несколько ИМС высокой степени интеграции (плата изображена послойно)

Преимущества СнК ASIC:

– возможность получения максимально высоких технических показателей (производительность, энергопотребление, массогабаритные характеристики);

– более низкая стоимость при крупносерийном выпуске.

В настоящее время реализация СнК в виде ASIC является приемлемой только для ограниченного числа высокобюджетных

проектов. Во всех случаях, когда можно достичь заданных характеристик, реализуя системы на плате, этот вариант является более предпочтительным ввиду названных преимуществ. Примером ИМС ASIC может являться промышленный контроллер ETHERNET до 100Мбит/с с процессором ARM 946, коммутатором на четыре порта и контроллером PCI, показанный на рисунке 1.8.

Рисунок 1.8 – Пример СнК ASIC, Siemens Ertec 400

Альтернативой может быть реализация СнК на базе FPGA, содержащих миллионы эквивалентных логических вентилей или на базе БМК, позволяющих проводить разработку и отладку с использованием комплементарных ПЛИС.

Преимущества реализации СнК на FPGA или БМК:

– малые затраты на разработку и создание опытных образцов;

– возможность многократной коррекции проекта;

– использование хорошо проверенных серийных изделий;

– более простой процесс тестирования и отладки (возможность реализации и отладки «по частям»).

Таким образом, СнК на базе FPGA имеют практически те же достоинства, что и системы на плате или СнМ, но отличаются лучшими техническими характеристиками – меньшими габаритами и массой. При необходимости существует возможность совершить переход на БМК и тем самым еще больше приблизиться к характеристикам ASIC. При этом все равно сохранить существенно более низкую стоимость разработки и внедрения (не для крупносерийных массовых производств). Однако по таким параметрам, как производительность, энергопотребление, СнК на базе FPGA и даже БМК уступают СнК, реализованным в виде ASIC. Исходя из сказанного, можно сделать вывод, что СнК на базе FPGA будут конкурировать и постепенно вытеснять системы на плате. При этом вместо микропроцессоров и микроконтроллеров в этих СнК будут использоваться различные варианты процессорных блоков аппаратных или синтезируемых [1–3].

2 ОБЛАСТИ ПРИМЕНЕНИЯ ПЛИС И ВЫБОР ИМС

Для получения наилучших характеристик, ИМС ПЛИС для каждого проекта необходимо грамотно выбирать, так же тщательно подходить к выбору ПЛИС следует из экономических соображений, особенно если это проект, подразумевающий массовое тиражирование. В случае если речь идет о штучном проекте без рекордных показателей быстродействия, то рациональнее пользоваться проверенными решениями. Выбор следует осуществлять из специфики проекта и задач, возлагаемых на ПЛИС. У ПЛИС существует несколько основных областей применения.

1. Коммутатор шин и соединений на плате

Если через ИМС ПЛИС проброшено большинство соединений на печатной плате между другими ИМС, такими как: контроллер, память, АЦП и другие, появляется возможность исправления ошибок в схеме и в трассировке ПП, а также возможность внесения изменений и модернизации дизайна всего устройства, только путем изменения описания структуры ПЛИС. Кроме того, печатная плата с ПЛИС значительно проще трассируется, благодаря широким возможностям конфигурации выводов, в том числе благодаря возможности питания разных портов ввода-вывода разным напряжением, что делает ПЛИС одновременно ИМС согласования уровней, если это необходимо.

2. Повышение надежности устройства

ПЛИС – достаточно надежные устройства. Многим сериям ИМС не свойственны сбои при неблагоприятных внешних воздействиях. Большинство FPGA быстро загружаются из внешней памяти и позволяют проводить постоянную реконфигурацию, как один из механизмов повышения отказоустойчивости [4]. При необходимости дополнительного повышения надежности и стойкости устройства, можно осуществить переход на однократно программируемые

ПЛИС, в том числе в металлокерамических корпусах со структурами на сапфире [3]. Однократно программируемая ПЛИС производства ВЗПП в металлокерамическом корпусе представлена на рисунке 2.1. Такие ПЛИС позволяют вести отладку по JTAG, а после завершения отладки осуществить однократное программирование ПЛИС, после которого изменить структуру связей уже будет невозможно. Перечисленные особенности ПЛИС позволяют использовать их для управления ответственными или быстропротекающими процессорами и периферией, а также могут реализовывать вспомогательные функции коммутации, задержек и многое другое, что, например, позволит разгрузить контроллер или процессор, который будет заниматься только основным алгоритмом.

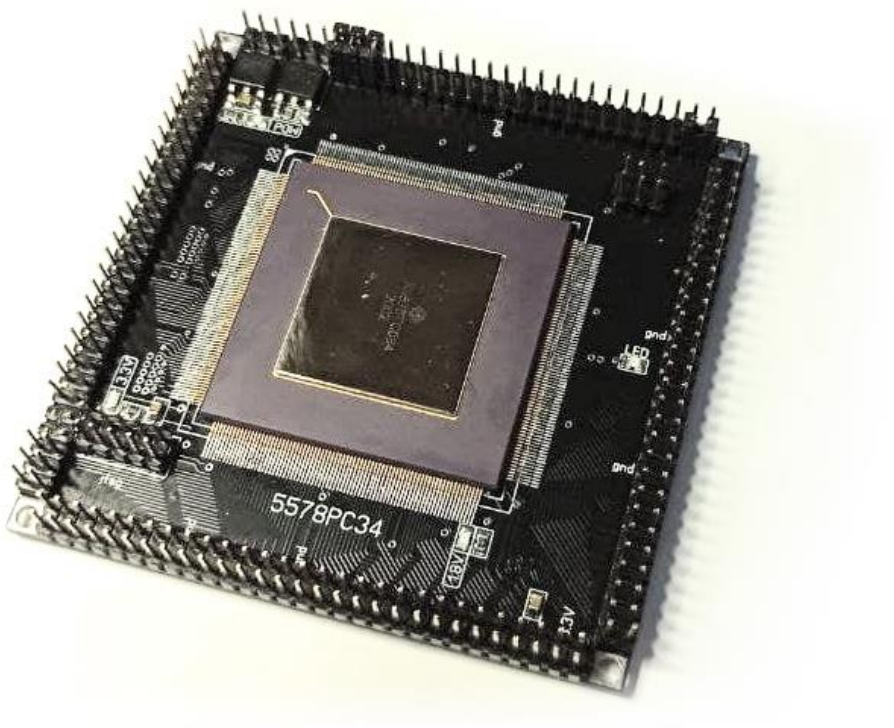


Рисунок 2.1 – Отладочная плата, разработки ИКП
с использованием ИМС 5578РС34

Надежность ПЛИС, если она уже есть в системе, позволяет нагрузить её функциями «восстанавливающего устройства»,

схемы запуска, сторожевого таймера или устройства переключения полукомплектов в системах с резервированием.

3. Автоматы состояний или аппаратное программирование

При работе с процессорными системами сначала создается «исполнитель команд», то есть процессор, а потом в него загружается последовательность команд, которые он будет выполнять. В общем случае берется готовый ЦП, и под него разрабатывается программа. На FPGA подход другой, так как можно писать программу с командами, как бы «вшитыми» в структуру ПЛИС. При этом отсутствует избыточность процессора, контроллера или архитектуры в целом, следовательно, снижается потребление при той же скорости и функциональности, появляется гарантированное время выполнения и высокая надежность.

4. Создание процессора внутри FPGA

Разработка процессорного ядра и синтез его внутри ПЛИС позволяет оптимизировать под свои задачи как систему команд, так и архитектуру ЦП. Синтезированное ядро возможно окружить блоками периферии, не отнимающими от ЦП ресурсов производительности и работающими параллельно с ним. Объем современных ПЛИС позволяет встраивать в одну ИМС большое количество одновременно работающих процессоров и получать настоящую многозадачность с низким энергопотреблением. Разработка процессора под ПЛИС проста и легко реализуема на FPGA. Недостатком синтезированных процессоров является отсутствие готовых компиляторов и отладчиков. Для создания проекта с синтезируемым внутри ПЛИС процессором целесообразно использовать FPGA с большой внутренней ОЗУ, которую можно использовать как КЕШ ОЗУ ЦП или для памяти программ с загрузкой программы из конфигурационной Flash-памяти при старте. Из общедоступных FPGA фирмы Intel/Altera подходят ИМС FLEX, Cyclone I-IV и более поздние модели FPGA, ПЛИС ВЗПП-С, которые позволяют создать полноценный soft-процессор, это 5576ХС4Т и все ПЛИС серии 5578ТС [5].

5. Использование готовых библиотек процессоров для FPGA

Библиотеки готовых процессоров (от 8086 до ARM) есть у любого производителя FPGA, что позволяют быстро создать проект с определенным набором периферии и использовать его в проекте FPGA. К готовым решениям от производителя всегда прилагается компилятор и отладчик. Такой подход удобен в использовании, но зачастую избыточен и потому ограничен по быстродействию. Наиболее распространенные soft-процессоры представлены в таблице 2.1.

Таблица 2.1 – Примеры soft-процессоров для синтеза в FPGA

Название	Кол-во занимаемых логических элементов	Производитель	Открытый код	Особенности
NIOS II	800–2500	Intel	нет	32-битный RISC-процессор гарвардской архитектуры Три состояния выводов
Microblaze	до 3000	Xilinx	нет	
PicoBlaze	200	Xilinx	да	8-битный RISC-процессора; скорость работы на ПЛИС семейства Virtex 4 может достигать 100 MIPS
LEON3	до 20 000	ESA	да	совместим со SPARC
Mipsfpga	до 15 000	–	да	набор инструкций MIPS32r3
CORTEX-M0/M3	–	ARM	нет	ARMv6
RISC-V (RV32I)	2500	–	да	Instruction Set Architecture
Mico32	–	Lattice	да	32-битный RISC-процессор гарвардской архитектуры, Шина периферии WISHBONE

Для синтеза готового процессорного ядра от производителя необходимо уточнять, какие семейства поддерживаются средой разработки. А для использования процессора с открытым кодом

желательно иметь 50% запас емкости по логическим элементам и памяти, для возможности размещения периферии и прочих блоков, необходимых для проектов и создания soft SoC. Поэтому для данных целей лучше выбирать FPGA, например, семейства Cyclone 4 или Cyclone 10, с логической емкостью не менее 25 000 элементов, либо аналогичные ИМС других фирм.

6. Объединять аппаратный процессор, периферию FPGA в одной микросхеме – SoC

Пример «младшей» системы на базе SoC от Intel (Altera) [6] приведен на рисунке 2.2. Другие семейства SoC от Intel – это Arria и Stratix (рисунок 2.3), но они на порядок мощнее, имеют множество встроенных аппаратных функций, емкость до нескольких миллионов логических элементов, поэтому они существенно дороже.

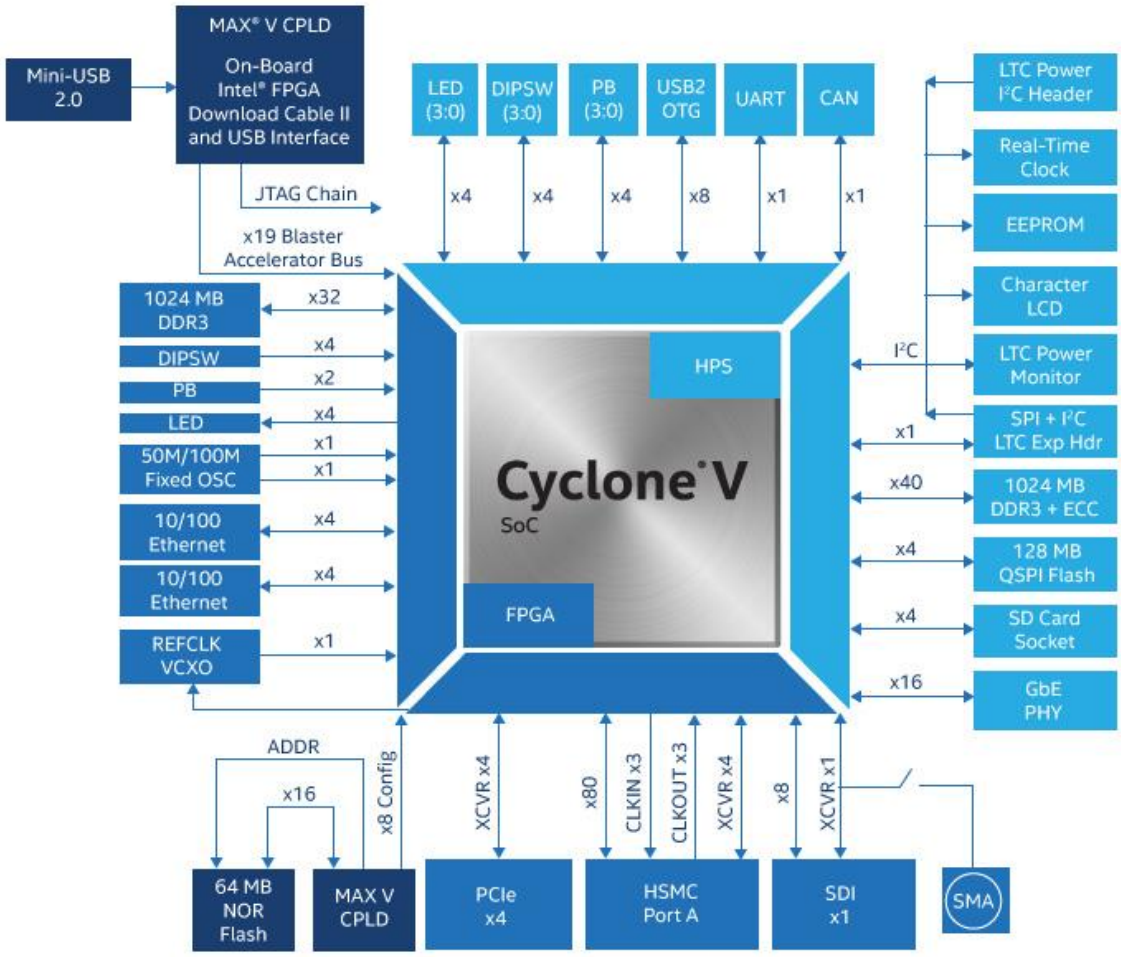


Рисунок 2.2 – Структура SoC CycloneV Intel

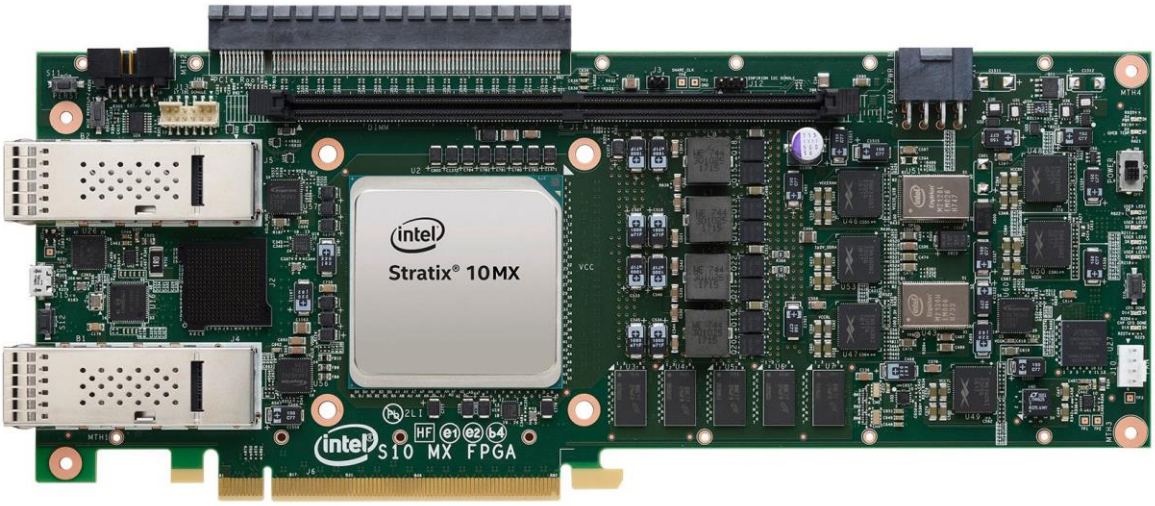


Рисунок 2.3 – PCI ускоритель на SoC Stratix

Технология SoC позволяет в одной микросхеме иметь полноценный центральный процессор (поддерживающий операционную систему, например Linux), микроконтроллер и большую FPGA, соединенные шинами с общей внутренней памятью и интерфейсами к внешней памяти и устройствам. То есть проблема эффективной, простой и быстрой передачи информации между FPGA, процессором и памятью уже решена производителем. FPGA и HPS (Host Processor System) процессор находятся внутри одной микросхемы и окружены программируемыми ножками ввода-вывода, что делает данное решение действительно многофункциональной системой на одном кристалле.

7. Ускорители вычислений

FPGA, являясь матрицей программируемых логических ячеек и триггеров, работающих параллельно, позволяет одновременно проводить много параллельных операций. Данная особенность в корне отличает FPGA от процессора, «параллельность»

которого ограничена количеством ядер и потоков. FPGA в данном плане становятся ближе к видеопроцессорам, выполняющим множество несложных однотипных операций параллельно. Поэтому FPGA можно использовать как сопроцессор к центральному процессору, перенося на FPGA все требовательные к вычислительной мощности операции. Например, центральный процессор занимается логической обработкой задачи, а FPGA производит параллельные вычисления, такие как контрольные суммы, хэши, поиск совпадения, перебор вариантов и так далее. Быстродействие FPGA ограничено только количеством параллельных блоков и временем выполнения одной операции. Существуют готовые платы с быстрыми интерфейсами (рисунок 2.3) и средства отладки.

8. Реализация нейронных сетей на FPGA

Нейронные сети и глубокие нейронные сети в настоящий момент используются в разных областях, и популярность данных решений постоянно возрастает. Однако реализация нейронных сетей на микропроцессоре оказывается крайне неэффективной, так как необходимо реализовать много вычислений, которые могут работать полностью параллельно (нейроны одного слоя, например, вычисляются независимо). Поэтому переход, при проектировании нейронных сетей на ИМС FPGA, позволяет значительно (много порядков) увеличить скорость работы нейронной сети. При выборе ПЛИС для построения нейронной сети остается обеспечить высокоскоростной интерфейс, необходимый для загрузки исходных данных и получения результата работы. В качестве примера можно рассмотреть реализацию системы распознавания лиц: так на процессоре Intel i7 9-го поколения система распознает до 20 лиц за секунду с одной видеокамеры HD, а реализация на FPGA распознает более 1000 лиц с нескольких камер. Пример реализации сверточной нейронной сети на ПЛИС

для распознавания рукописных символов можно увидеть в статье [7]. Общую информацию по нейронным сетям можно изучить в [8]. Типовой процесс разработки нейронной сети для ПЛИС состоит из следующих этапов:

1) создание модели и разработка алгоритма;
2) подготовка к аппаратной реализации;
3) автоматическая генерация кода;
4) подключение кода в САПР для ПЛИС.

Довольно часто в качестве основы разработки ложится ПО MATLAB/Simulink, что дает большое количество готовых библиотек и возможность подключения внешнего кода, а также устраняет разрыв между алгоритмическим описанием и HDL-описанием схемы, что, в совокупности, существенно сокращает время разработки.

3 ОСНОВЫ VERILOG HDL

Комбинационные логические цепи используются как в линиях данных, так и в линиях управления более сложных систем. Они могут быть созданы различным образом. Например, с использованием операторов непрерывного присваивания, которые включают выражения с логикой, арифметическими операторами и операторами сравнения. Отличительной особенностью непрерывного присвоения является немедленное изменение состояния присваиваемой цепи «**wire**», при изменении состояния входных цепей, влияющих на нее. Все операторы непрерывного присвоения всегда начинаются с ключевого слова «**assign**». Цепи «**wire**» являются средствами коммутации между регистрами, поэтому код, начинающийся с ключевого слова «**assign**», является элементом структурного описания разрабатываемой схемы.

Для функционального или поведенческого описания в языке Verilog существует блок «**always**», для чисто функционального описания существуют блок задач «**task**» и функций «**function**». Отличительной особенностью процедурного присвоения является то, что изменение состояния присваиваемой регистровой переменной «**reg**» происходит только под управлением некоторой процедуры, которую в Verilog олицетворяет блок «**always**». Подробно синтаксис и правила Verilog HDL описаны в [9–12].

Все типы операторов: «**assign**», «**always**», «**function**» и «**task**» должны быть помещены в теле модуля так, как показано в листинге 3.1.

Листинг 3.1. Структура модуля в Verilog HDL

```
module module_name (ports);
[assign operator]       // структурное описание
    ....
    [always blocks]// поведенческое описание
    ....
```

```
        [assign operator]
        ....
        [always_blocks]
        ....
endmodule
```

Логические операторы Verilog HDL могут использоваться чтобы синтезировать комбинационные схемы. В листинге 3.2 приведен пример описания комбинаторной логики, а сгенерированная компилятором схема RTL (register transfer level), ему соответствующая, представлена на рисунке 3.1.

Листинг 3.2. Пример комбинационной схемы на Verilog HDL

```
module combinatorial_logic
(input a, b, c, d,        // объявление входных портов или входов
output y);                // объявление выходных портов или выходов
wire e;                           // объявление внутренней цепи
assign y = (a & b) | e;
assign e = (c | d);
endmodule
```

Схема RTL (register transfer level), соответствующая этому примеру, показана на рисунке 3.1. Данная схема автоматически сгенерирована компилятором из описания в листинге 3.2. Отметим, что компилятор удалил при синтезе RTL внутренний сигнал «**wire** e» вместе с ее формирующей схемой «2ИЛИ», заменив последнюю на общую схему «3ИЛИ».

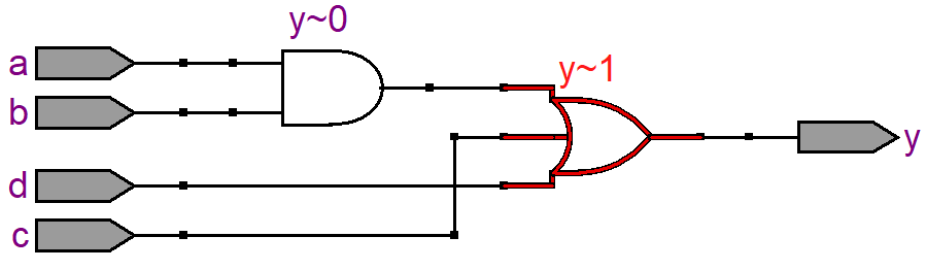


Рисунок 3.1 – RTL схема, соответствующая листингу 2.2

Описание схемы, содержащей простые логические элементы, но с четырехразрядными шинами в качестве входов, представлено в листинге 3.3.

Листинг 3.3. Пример комбинаторной схемы на Verilog HDL

```
module combinatorial_logic2
(input [3:0] a, b,      // объявление входных векторных порта
output [3:0] y,         // объявление выходного векторного порта
output z);
assign y = a & b;       // bit-wise AND
assign z = & y;         // reduction AND
endmodule
```

Важно, что по отношению к шинам операторы AND и OR могут быть как бинарными операторами (побитовыми), так и унарными (редукционными). Их различие видно по RTL схеме, представленной на рисунке 3.2. Схема соответствует описанию с листинга 3.3.

Пример синтезируемой логики операторов сравнения на Verilog HDL представлен на листинге 3.4.

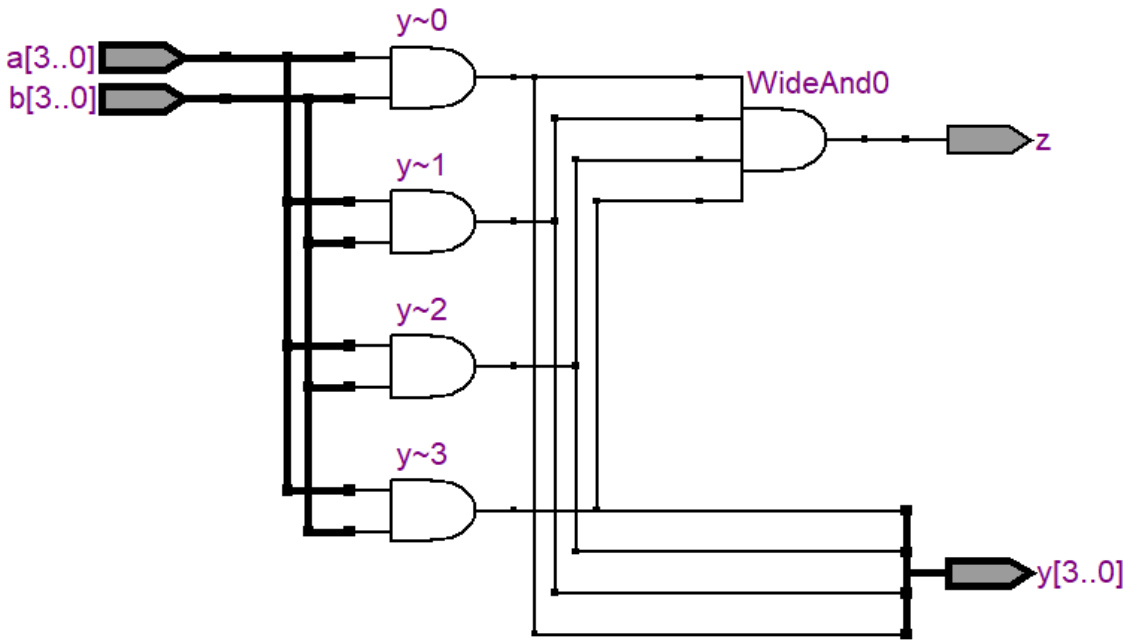


Рисунок 3.2 – RTL схема, соответствующая листингу 3.3

Листинг 3.4. Пример логики сравнения на Verilog HDL

```
module comparison_logic
(input [3:0] a, b,
output y);
assign y = (a == b);

endmodule
```

Схема, соответствующая листингу 3.4, представлена на рисунке 3.3.

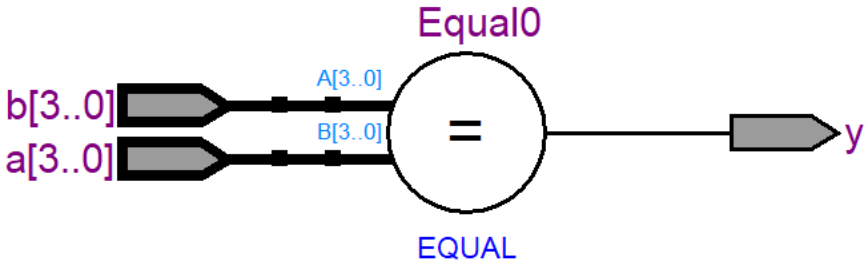


Рисунок 3.3 – RTL схема, соответствующая листингу 3.4

Операторы, описывающие условную логику в Verilog HDL, представлены тремя типами: «**if**», «**case**» и тернарный оператор «**?**». Тернарный оператор работает и в непрерывных, и в процедурных присвоениях. Пример, представленный в листинге 3.5, иллюстрирует использование в Verilog HDL оператора «**if**» для создания условной логики, а в комментариях показан образец использования тернарного оператора.

Листинг 3.5. Пример оператора «if» на Verilog HDL

```
module comparison_logic2
(input [7:0] a, b,
input sel,
output reg [7:0] y);
always @(a or b or sel) begin
```

```
if (sel==1)      y = b;
else             y = a;
/* эквивалентная запись с помощью тернарного оператора:
y = sel? b:a; */
end
endmodule
```

Следующий пример, представленный в листинге 3.6, показывает вариант использования оператора «**case**» для создания описания мультиплексора. Все возможные комбинации на входах должны иметь описание. Программист может убедиться в этом при использовании ветви «**default**» внутри оператора «**case**».

Листинг 3.6. Пример оператора «case» на Verilog HDL

```
module comparison_logic3
(input a, b, c, d,
input [1:0] sel,
output reg y);
always @(sel or a or b or c or d)
case(sel)
    2'd0: y = a;
    2'd1: y = b;
    2'd2: y = c;
    2'd3: y = d;
    default: y = a;
endcase
endmodule
```

Как будут выполняться арифметические операнды, сильно зависит от **технологии на целевой ИМС ПЛИС.** Пример, представленный в листинге 3.7, демонстрирует использование арифметических операторов и круглых скобок для ИМС Altera, которые позволяют в некоторых пределах управлять синтезируемой логической структурой. Так, на рисунках 3.4 и 3.5 показаны соответствующие RTL схемы.

Листинг 3.7. Пример сложения и вычитания на Verilog HDL

```
module arithmetic
(a, b, c, d, y1, y2);
input wire [7:0] a, b, c, d;
output wire [9:0] y1, y2;
assign y1 = a + b - c - d;              // Рисунок 3.4
assign y2 = (a + b) - (c + d);  // Рисунок 3.5
endmodule
```

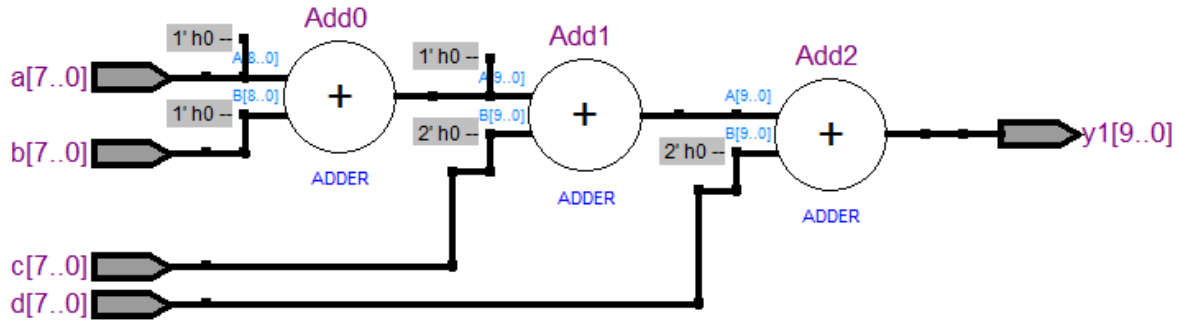


Рисунок 3.4 – RTL схема, без указания приоритета операций

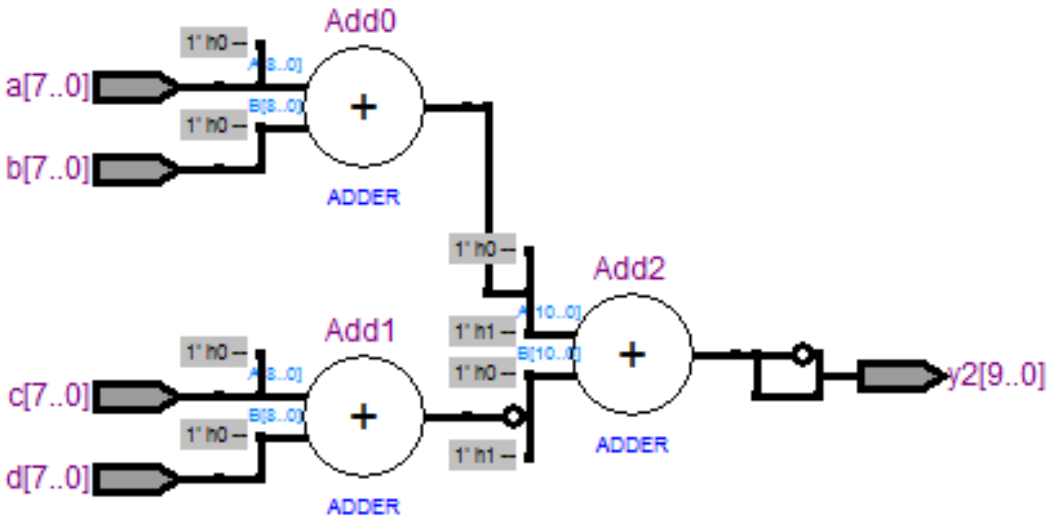


Рисунок 3.5 – RTL схема, с указанием приоритета операций

Можно заметить, что компилятор создал две разные схемы для описания с листинга 3.7 [12]. Технологические схемы и временные диаграммы RTL с последних двух рисунков представлены на рисунке 3.6.

Рисунок 3.6 – Technology map Viewer и временные диаграммы листинга 3.7

Операнды умножения и деления (листинг 3.8.) также синтезируются в RTL представление (рисунок 3.7), только при отсутствии аппаратных блоков (при высокой разрядности операнд) данные операции занимают много ресурсов ИМС. Так, описание, представленное ниже, занимает 280 логических элементов.

Листинг 3.8. Операторы умножения и деления на Verilog HDL

```
module arithmetic
(a, b, c, d, y);
input wire [8:0]a, b, c, d;
output wire [20:0] y;
assign y = a * b + c / d;
endmodule
```

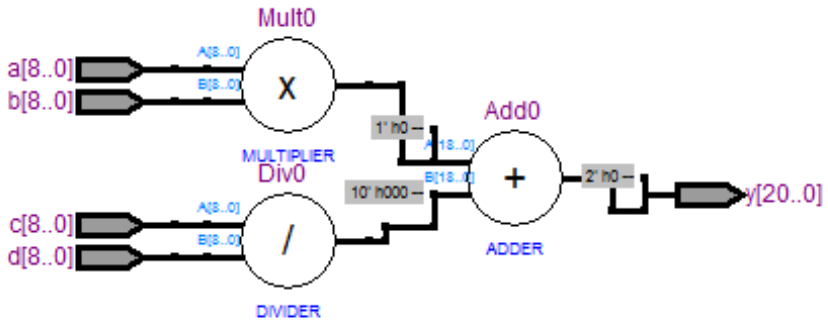


Рисунок 3.7 – RTL представление листинга 3.8

Исходя из особенностей данных операндов, без аппаратных блоков пользоваться ими имеет смысл, когда необходимо максимально быстрое вычисление результата. В противном случае лучше применить правила деления и умножения беззнаковых двоичных чисел и составить собственное описание последовательного выполнения необходимой операции. В цифровой схемотехнике сложные арифметические операции, такие как умножение, деление, извлечение корня и др., сводятся к более простым (сложению, вычитанию и сдвигу) и реализуются с помощью базовых операционных элементов, которые будут синтезироваться в любых ИМС ПЛИС без привязки к аппаратной части. В разделе 4 данного пособия приведены некоторые алгоритмы по работе с двоичными беззнаковыми числами.

Существует еще одна возможность создания асинхронного дизайна для реализации арифметических действий. Она состоит в том, чтобы заключить операторы присваивания значений в блок always со всеми входными сигналами в списке чувствительности. С точки зрения синтеза схемы не будет никакого различия [12]. Однако моделирование (временных задержек) может быть более простым, если блок always используется для описания той же самой цепи. Описанный вариант представлен в листинге 3.9.

Листинг 3.9. Пример описания арифметических операций внутри блока always

```
module arithmetic2
(a, b, c, d, y1, y2);
input wire [7:0] a, b, c, d;
output reg [9:0] y1, y2;
always @ (a or b or c or d) begin
        y1 = a + b + c + d;
        y2 = (a + b) + (c + d);
end
endmodule
```

3.1 Примеры комбинаторных модулей на Verilog_HDL

В этом подразделе представлено несколько стандартных комбинационных блоков и способов их описания на Verilog_HDL. Эти блоки обычно представляют конструкции, которые используются для формирования более сложных описаний. Представленные примеры легко поддаются изменению, и их можно использовать для выполнения практических занятий и самостоятельных работ.

Листинг 3.10. Декодер на Verilog HDL

```
module decoder
(a,y);
input wire [7:0] a;
output reg [2:0] y;
always @(a) begin
        case(a)
        8'b00000001: y = 3'd0;
        8'b00000010: y = 3'd1;
        8'b00000100: y = 3'd2;
        8'b00001000: y = 3'd3;
        8'b00010000: y = 3'd4;
        8'b00100000: y = 3'd5;
        8'b01000000: y = 3'd6;
        8'b10000000: y = 3'd7;
        default: y = 3'bX;
        endcase
        end
endmodule
```

Листинг 3.11. Управляемый переключатель на Verilog HDL

```
module switch_performer
( CLK, reset,
AFS, DFS, WES, NO_ERROR,
out_switch );
parameter ADR = 65200;
```

```
input wire CLK, reset;
// HAVOC:
input wire [15:0] AFS;
input wire [15:0] DFS;
input wire  WES, NO_ERROR;
// out:
output reg [15:0] out_switch;
always @ (posedge CLK or posedge reset) begin
        if (reset)          out_switch<=0;
        else begin
                if (WES&NO_ERROR)begin
                        if (AFS==ADR) begin
                                out_switch[15:0]<=DFS[15:0];
                        end
                end
        end
end // always
endmodule
```

4 АЛГОРИТМЫ МАШИННОЙ АРИФМЕТИКИ

Сложные арифметические операции, такие как умножение, деление, извлечение корня и др., сводятся к более простым и реализуются с помощью базовых операционных элементов цифровой схемотехники.

В данном разделе будут рассмотрены алгоритмы умножения, деления, извлечения квадратного корня. Понимание данных алгоритмов будет необходимо для успешного выполнения практических занятий. Для алгоритма извлечения корня будет приведен пример его реализации на Verilog HDL.

4.1 Алгоритм умножения

Алгоритм умножения «в столбик» и примеры умножения двух 4-разрядных беззнаковых чисел представлены на рисунке 4.1, где «N» – это разрядность операндов соответственно.

В процессе вычисления результата выполнялось умножение каждого разряда второго операнда «b» на первый операнд «a» и сложение частных результатов. Умножение на 0 и 1 выполняется простейшим образом: разряды второго операнда можно рассматривать как условие, определяющее необходимость сложения первого операнда с полученным на предыдущих шагах частным результатом. Каждое сложение выполняется со сдвигом на один разряд относительно предыдущего шага.

4.2 Алгоритм извлечения квадратного корня

В данном разделе представлен целочисленный алгоритм извлечения квадратного корня с округлением до ближайшего меньшего целого числа. Например, при извлечении квадратного корня из 17

получим 4,12, но в качестве результата будет 4. Представленный на рисунке 4.2 алгоритм вычисления квадратного корня использует только операции сложения, вычитания, сравнения и сдвига. Символом «N» обозначена разрядность входного операнда «x».

На листинге 4.1 представлено описание модуля, содержащего структуру, выполняющую алгоритм с рисунка 4.2, и позволяющего вычислять целую часть квадратного корня из 32-битного двоичного числа. Если данный модуль синтезировать в семействе FPGA Cyclone III, то он займет примерно 212 логических элементов, максимальная частота тактирования будет до 220МГц, и вычисление корня из 32-битного числа будет происходить примерно за 360нс при тактовой 50МГц.

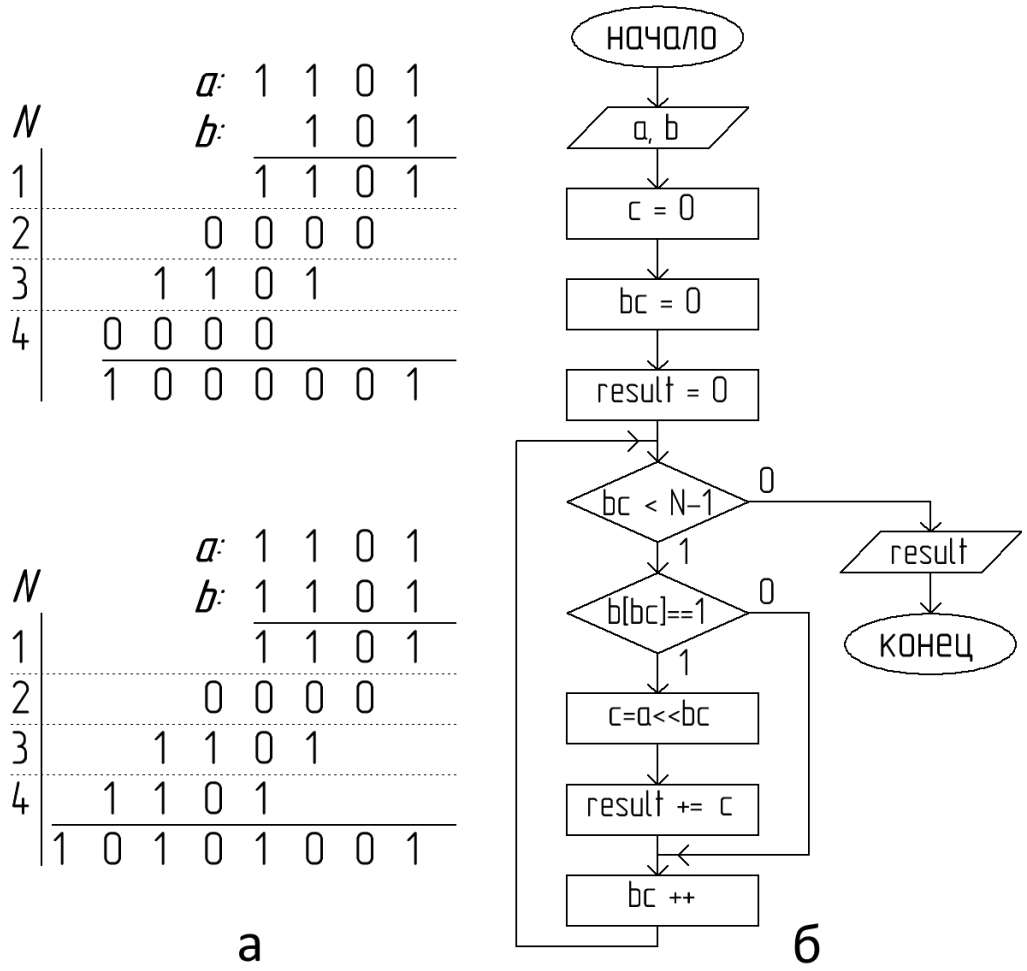


Рисунок 4.1 – Умножение беззнаковых двоичных чисел:
а – пример; б – алгоритм

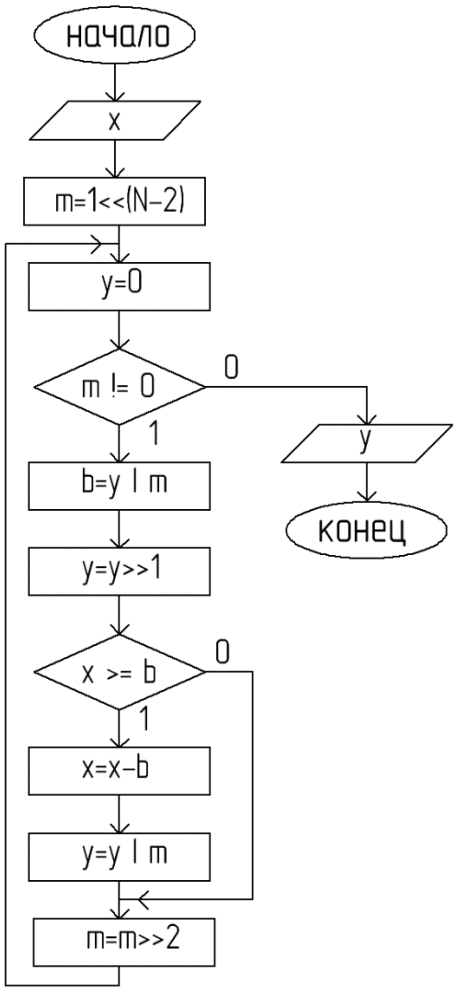


Рисунок 4.2 – Алгоритм извлечения квадратного корня

Листинг 4.1. Модуль вычисления квадратного корня

```
module nirnroot
(CLK,reset, start, // глобальные сигналы и сигнал запуска
argument,result); // аргумент и результат
input wire CLK, reset, start;
input wire [31:0] argument;
output reg [31:0] result;
reg [3:0] FSM;          // автомат состояний
reg [31:0] buffer;
reg [31:0] M;  // см. алгоритм на рис.4.2
reg [31:0] B;  // см. алгоритм на рис.4.2
always @(posedge CLK or posedge reset) begin
        if (reset) begin
                FSM<=0; buffer<=0; M<=0; result<=0; B<=0;
```

```
        end // reset
        else begin
                if (start) begin
                        FSM<=4'd1;
                        buffer[31:0]<=argument[31:0];                   end
                case (FSM)
                4'd1:   FSM<=FSM+1'd1; // не обязательный пропуск такта
                4'd2:   begin
                                FSM<=FSM+1'd1;
                                M<=32'd1<<30;
                                result<=0;
                                B<=0;
                        end
                4'd3:   begin
                                        if (M==0) begin
                                                FSM<=4'd15;             end
                                        else begin
                                                B<=result|M;
                                                FSM<=FSM+1'd1;  end
                        end
                4'd4:   begin
                                FSM<=FSM+1'd1;
                                result<=result>>1;
                                if (buffer>=B) begin
                                        FSM<=FSM+1'd1;  end
                                else begin
                                        FSM<=4'd6;              end
                        end
                4'd5:   begin
                                FSM<=FSM+1'd1;
                                buffer<=buffer-B;
                                result<=result|M;
                        end
                4'd6:   begin
                                FSM<=4'd3;
                                M<=M>>2;
                        end
                default:;
                endcase
        end // CLK
end // always
endmodule // nirnroot
```

Временные диаграммы, полученные в результате моделирования модуля «nirnroot», показаны на рисунке 4.3.

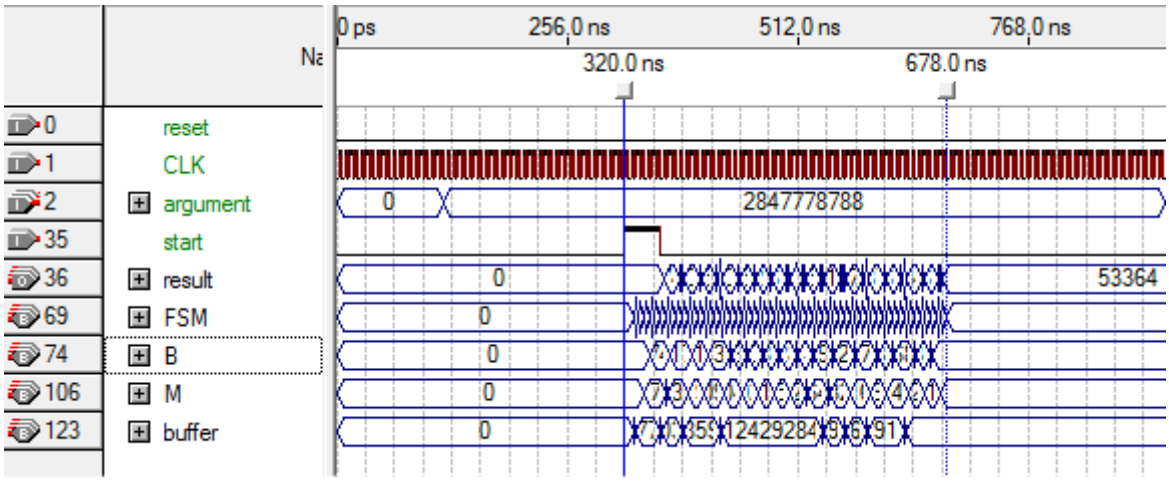


Рисунок 4.3 – Результаты моделирования модуля «nirnroot»

4.3 Алгоритмы деления

Модуль, выполняющий операцию деления 32-битных чисел и вычисление остатка, представлен на листинге 4.2. Данный модуль использует только операции вычитания и сдвига, поэтому может быть синтезирован любой средой. Однако по скорости вычисления реализованный алгоритм «деления столбиком» существенно проигрывает дизайну, синтезируемому Quartus при использовании оператора деления «/», но требует в несколько раз меньше логических элементов. Листинг 4.3 содержит описание примера «аппаратного» деления. По результатам компилирования модуль деления столбиком занял 515 логических элементов и может работать на частоте тактирования до 160МГц. Модуль аппаратного деления занял 1245 логических элементов, частота тактирования в данном модуле не должна превышать 50МГц. Таким образом, время вычисления в первом случае не превышает 400нс, а во втором составляет не более 60нс при частоте тактирования, равной 50МГц в обоих случаях.

Листинг 4.2. Модуль, выполняющий операцию деления с остатком

```
module divider
(CLK, reset,
start, ready,
argument, divider, result, remainder);
input wire CLK, reset, start;
output reg ready;                       // флаг о готовности результата
input wire [31:0] argument;             // делимое
input wire [31:0] divider;              // делитель
output reg [31:0] result;               // результат
output reg [31:0] remainder;            // остаток от деления
reg [31:0] arg;                         // буфер делимого
reg [31:0] div;                         // буфер делителя
reg [1:0] FSM;
reg [4:0] bitcounter;
wire [4:0]capacity_B;                   // разрядность делителя
assign capacity_B=divider[31]?31:divider[30]?30:divider[29]?29:divider[28]?28:
divider[27]?27:divider[26]?26:divider[25]?25:divider[24]?24:divider[23]?23:
divider[22]?22:divider[21]?21:divider[20]?20:divider[19]?19:divider[18]?18:
divider[17]?17:divider[16]?16:divider[15]?15:divider[14]?14:divider[13]?13:
divider[12]?12:divider[11]?11:divider[10]?10:divider[9]?9:divider[8]?8:
divider[7]?7:divider[6]?6:divider[5]?5:divider[4]?4:divider[3]?3:divider[2]?1:di-
vider[1]?1:0;
always @ (posedge CLK or posedge reset) begin
if (reset) begin
        ready<=0; result<=0; arg<=0; div<=0;
        FSM<=0; bitcounter<=0;
end // reset
else begin
        if (start) begin        // захват значений
                div[31:0]<=divider[31:0];
                arg[31:0]<=argument[31:0];
                FSM<=4'd1;   // переход в первое состояние
                ready<=0; result<=0; remainder<=0;
                bitcounter<=5'd31-capacity_B;                   end
        case (FSM)
        2'd1:   begin
                        div[31:0]<=div[31:0]<<(31-capacity_B);
                        FSM<=2'd2;
```

```
            end
        2'd2:   begin           // вычисление
                    if (bitcounter==0) begin
                        FSM<=2'd3;                  end
                    else begin
                        bitcounter<=bitcounter-1'd1; end
                    if (arg<div) begin
                        result[bitcounter]<=0;
                        div[31:0]<=div[31:0]>>1;    end
                    else begin
                        result[bitcounter]<=1'd1;
                        arg[31:0]<=arg[31:0]-div[31:0];
                        div[31:0]<=div[31:0]>>1;    end
            end
        2'd3:   begin           // вывод результата и переход к ожиданию
                    ready<=1'd1; remainder[31:0]<=arg[31:0];
                    FSM<=0; arg<=0; div<=0; bitcounter<=0;
            end
        default:;
        endcase
    end // CLK
    end // always
    endmodule // long_divider
```

Результат моделирования модуля «divider» показан на рисунке 4.4.

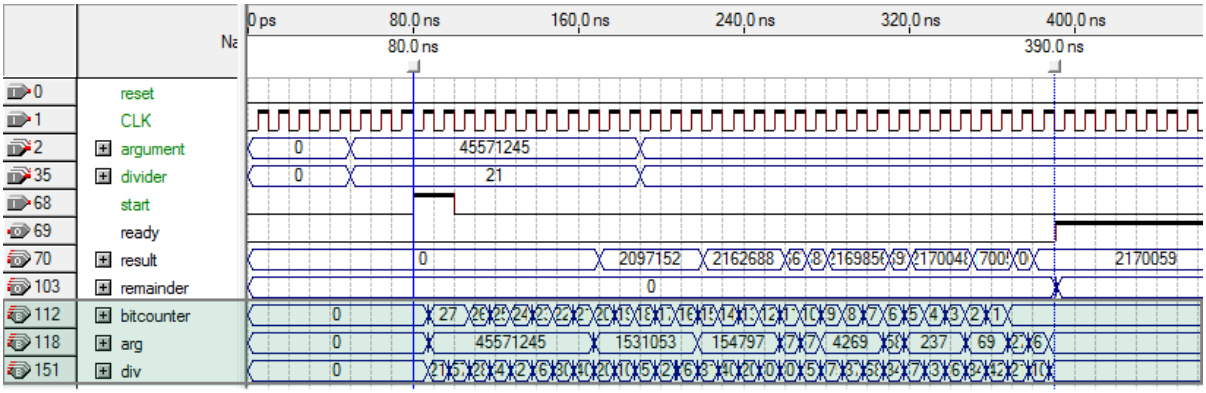


Рисунок 4.4 – Результаты моделирования модуля «divider»

Листинг 4.3. Модуль «аппаратного» деления

```
module fast_divider
(CLK, reset,
start, ready,
argument, divider, result, remainder);
input wire CLK, reset, start;
output reg ready;
input wire [31:0] argument;
input wire [31:0] divider;
output reg [31:0] result;
output reg [31:0] remainder;
reg [31:0] arg;// буфер исходных данных
reg [31:0] div;// буфер исходных данных
reg [1:0] FSM;
always @ (posedge CLK or posedge reset) begin
if (reset) begin
        ready<=0; result<=0; arg<=0; div<=0;
        FSM<=0;
end // reset
else begin
        4 (start) begin // захват данных
                div[31:0]<=divider[31:0];
                arg[31:0]<=argument[31:0];
                FSM<=4'd1; ready<=0; result<=0; remainder<=0;  end
        case (FSM)
        2'd1:   begin
                        result<=arg/div;
                        FSM<=2'd2;
                end
        2'd2:   begin
                        ready<=1'd1; FSM<=0; arg<=0; div<=0;
                remainder[31:0]<=arg[31:0]-result[31:0]*div[31:0];
                end
        default:;
        endcase
end // CLK
end // always
endmodule // fast_divider
```

Результат моделирования модуля «fast_divider» показан на рисунке 4.5.

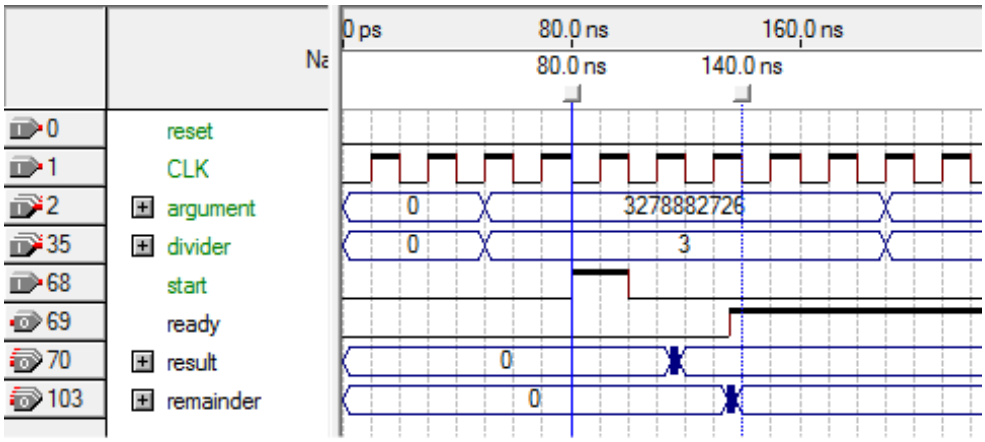


Рисунок 4.5 – Результаты моделирования модуля «fast_divider»

Рассмотрим алгоритм еще более простого для понимания способа деления, без применения соответствующего оператора, и даже без сдвигов: метод – повторяющегося уменьшения, алгоритм метода приведен на рисунке 4.6.

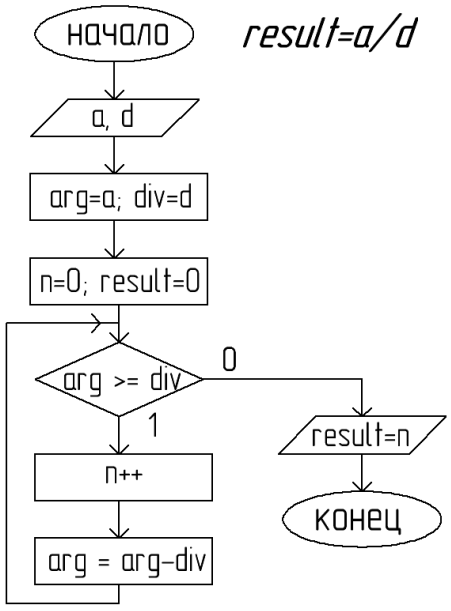


Рисунок 4.6 – Алгоритм деления повторяющимся уменьшением

5 ОСНОВЫ РАБОТЫ В СРЕДЕ QUARTUS II

В данном разделе описан принцип работы в системе автоматизированного проектирования (САПР) QuartusII (далее Quartus), разработки фирмы Intel. Версия Prime является бесплатной и доступна для скачивания с официального сайта Intel (https://fpgasoftware.intel.com). Quartus предназначен для проектирования программного обеспечения микросхем программируемой логики (ПЛИС) фирмы Intel. В разделе описан пример проектирования в Quartus, даны рекомендации по использованию различных модулей программы. Интерфейс Quartus почти не меняется от версии к версии, что позволяет осваивать САПР с версии 8.0 и более поздние. Версии до 9.1 интересны тем, что они поддерживают семейства Flex и CycloneIII, у данных микросхем существуют их отечественные аналоги. Версии QuartusII до 9.1 включительно имели в составе симулятор с возможностью проводить моделирование работы разработанного описания с учетом временных задержек прохождения сигнала и особенностей выбранной целевой микросхемы ПЛИС. Дополнительно о среде QuartusII можно узнать из [13–14].

Знакомство со средой Quartus будет проведено на простом примере классического «Hello World» в электронике: необходимо будет помигать светодиодом на отладочной плате с частотой 1Гц. Данная задача будет решена как графическим методом описания, так и на нескольких HDL языках. В качестве отладочной платы может быть использована любая с ПЛИС от Altera. Необходимо будет иметь на руках схему выбранной отладочный платы, чтобы знать, куда подключен тактовый генератор и как разведены выводы ИМС.

После установки и запуска Quartus при начале работы с новым проектом необходимо открыть меню «new project» во вкладке «File» и выбрать директорию, где будут храниться файлы нового

проекта, далее необходимо назначить имя проекта, как показано на рисунке 5.1, и нажать кнопку «Next>». В открывшемся окне предлагается добавить, если это необходимо, в иерархию создаваемого проекта ранее разработанные файлы. В случае первого запуска данное окно просто пропускается «Next>». Далее необходимо выбрать целевую ИМС из доступных в данной версии программы семейств. На рисунке 3.1 выбрана EPM240T100C3 – это CPLD MAXII в корпусе TQFP 100. Также видно, что ИМС содержит 240 логических блоков и один UFM блок доступной пользователю Flash-памяти, внутренняя логика питается от 3,3В. Индекс «C3» в названии микросхемы говорит о низких уровнях задержки распространения сигнала (чем меньше индекс, тем быстрее ИМС). Далее можно сразу нажать кнопку «Finish».

Чтобы начать разработку описания, необходимо создать файл того типа, какой метод описания будет использован. Для этого необходимо во вкладке «File» открыть меню «new».

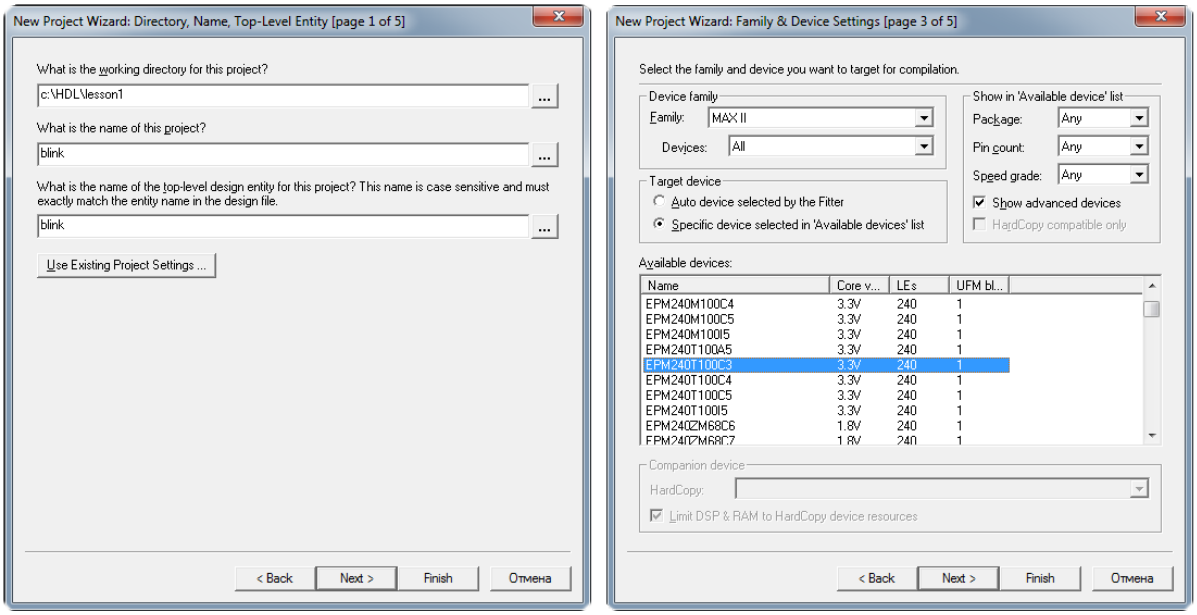


Рисунок 5.1 – Создание проекта в Quartus

В появившемся окне в разделе «Design File» выбрать «Block Diagram/Schematic File» и нажать кнопку «OK». Откроется пустое

поле графического редактора. Созданный графический файл следует сразу сохранить в директории проекта и присвоить ему корректное имя – для этого во вкладке «File» выбрать «Save As..». В рабочей области графического файла теперь можно создавать схему, используя панель инструментов и доступные примитивы, библиотеку которых можно вызвать двойным нажатием ЛКМ на любом сете рабочей области (рисунок 5.2).

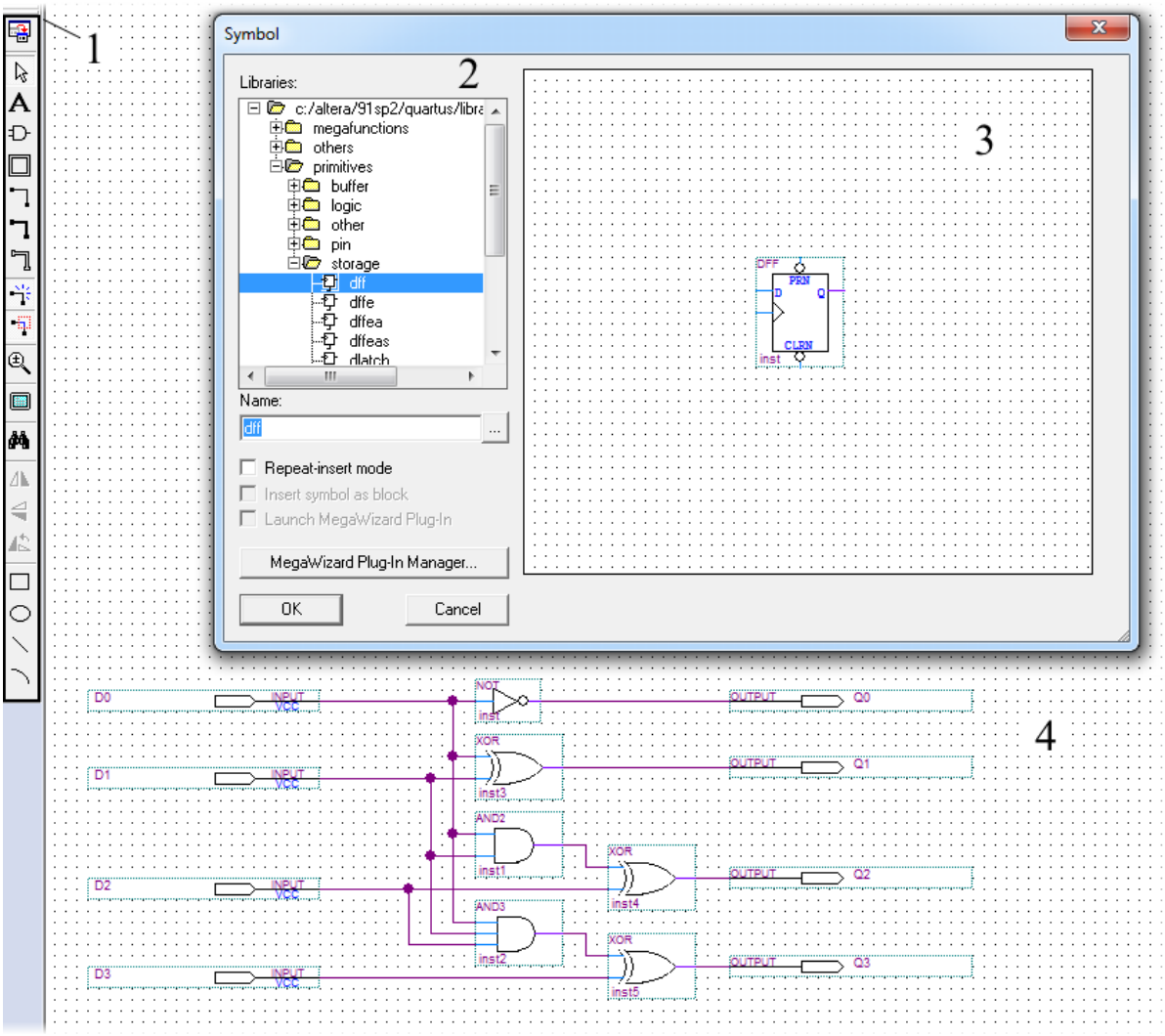


Рисунок 5.2 – Графический редактор Quartus:
1 – панель инструментов графического редактора;
2 – библиотека стандартных элементов и примитивов;
3 – графическое изображение библиотечного элемента (Д-триггера);
4 – рабочая область графического редактора
(изображена схема 4-битного инкремента)

5.1 Разработка описания в графическом редакторе

Для реализации проекта «blink», т.е. модуля, который сможет зажигать светодиод с частотой 1Гц, необходимо реализовать в ПЛИС схему делителя частоты, т.е. сделать счетчик. Данную задачу и в графическом редакторе можно решить разными методами. Рассмотрим самый базовый, будем использовать только примитивы (триггеры и логические элементы). Для реализации счетчика необходимо сделать модуль инкремента, который будет прибавлять к значению на входе единицу и выводить на выводы. Схема 4-битного инкремента также представлена на рисунке 5.2. По аналогии схема может быть расширена до необходимой разрядности. Для организации модульной структуры создадим новый графический файл с названием «increment», нарисуем там схему инкремента, пользуясь библиотекой. Для быстрого поиска элементов в библиотеке можно вводить имя необходимого элемента. Так, вход – «input», выход – «output», исключающее или – «xor», логическое «И» – «and2», логическое «И» с тремя входами – «and3». После файл надо сохранить и создать для него символ, для этого во вкладке «File» необходимо выбрать «Create/_Update» и «Create Symbol Files for Current Files».

Далее создаем новый графический файл с именем «counter4». В нем двойным нажатием ЛКМ в рабочей области вызываем библиотеку элементов, в которой, перейдя в корневую директорию проекта, можно найти созданный ранее элемент с именем «increment». Добавляем его, а также добавляем четыре Д-триггера «dff» и входы – выходы, чтобы можно было составить схему, показанную на рисунке 5.3. Данная схема является 4-битным счетчиком. Сохраняем файл «counter4» и временно назначаем его файлом верхнего уровня, для этого во вкладке «Project» выбираем «Set as Top-Level Entity».

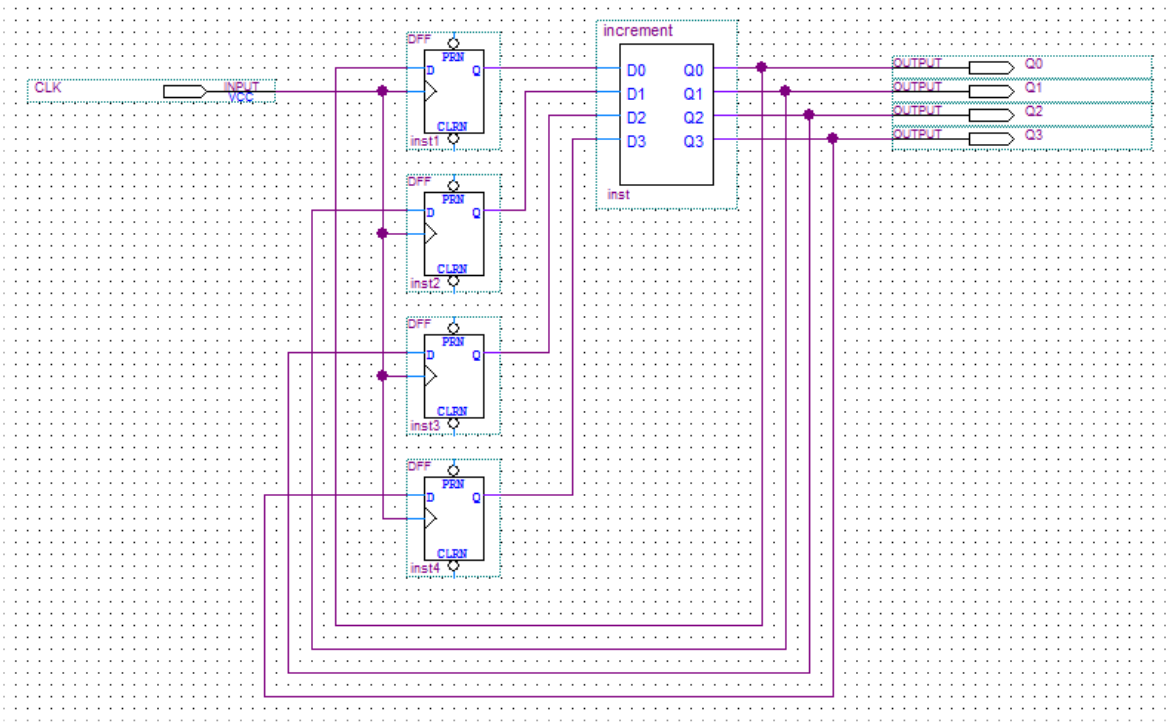


Рисунок 5.3 – Схема 4-битного счетчика
в графическом редакторе Quartus

Когда текущий файл является файлом верхнего уровня, с ним можно проводить ряд операций, таких как проверка, компиляция и трассировка в ПЛИС, после чего можно провести моделирование. Чтобы посмотреть какой файл в данный момент является файлом верхнего уровня, какие вообще файлы входят в проект и какая у него иерархия, можно обратиться к окну «Project Navigator» (рисунок 5.4).

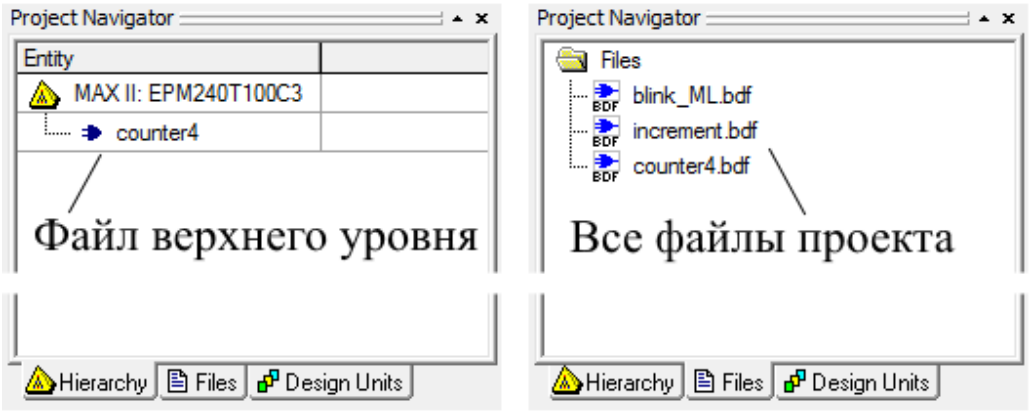


Рисунок 5.4 – Окно «Project Navigator» в Quartus

После успешной компиляции «counter4» появится окно с кратким отчетом, как показано на рисунке 5.5. Из отчета видно, что проект свободно размещается в целевой микросхеме EPM240 и занимает менее 2% доступных логических элементов.

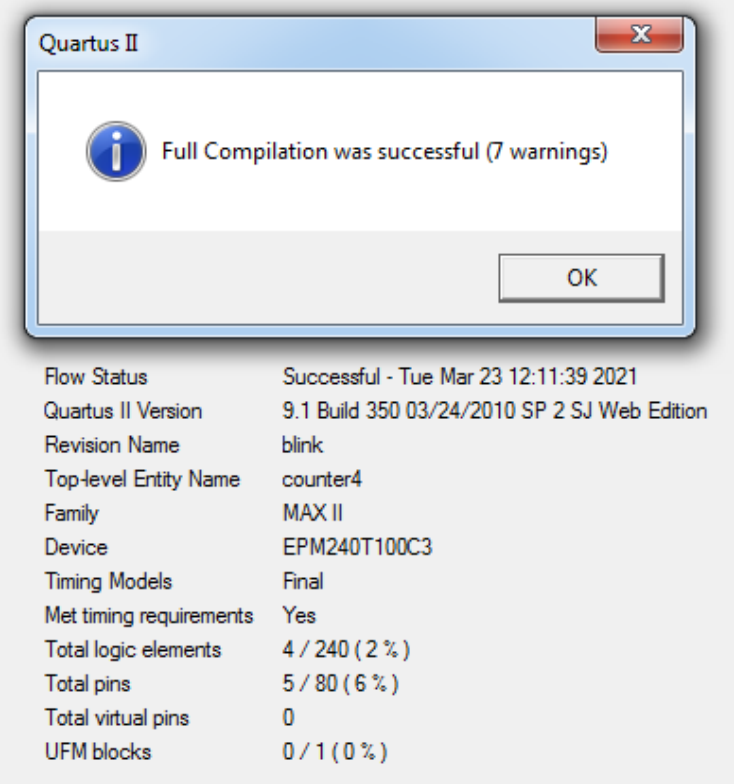


Рисунок 5.5 – Отчет о компиляции файла «counter4» проекта «blink» в Quartus

После успешной компиляции можно провести моделирование работы модуля «counter4». Для этого создаем новый файл, только теперь выбираем тип файла «Vector Waveform File». Далее необходимо сохранить его с именем файла «counter4.vwf» и добавить сигналы в пустую рабочую область редактора диаграмм, как показано на рисунке 5.6.

На рисунке 5.6 показан уже результат моделирования, но изначально поля выводов и диаграмм пустые. Чтобы добавить наполнение, необходимо двойным нажатием ЛКМ на поле имен сигналов «Name» вызвать окно «Insert Node or Bus», далее нажать

«Node Finder…», для перехода в окно поиска и определения узлов, где, нажав «List» из появившегося списка, выбрать необходимые сигналы. В данном случае понадобятся все выводы и вводы «Pins:all». На тактовый ввод «CLK» необходимо подать тактовый сигнал с частотой, равной частоте тактового генератора, подключенного к ИМС, или любой другой при необходимости. Для этого нужно выделить сигнал «CLK» и нажать «Count value», далее выбрать параметры сигнала и сохранить файл диаграмм. Чтобы запустить моделирование, необходимо во вкладке «Processing» выбрать «Simulator Tool», в появившемся окне настроек параметров моделирования (рисунок 5.7) указать в качестве входа файл диаграмм «counter4.vwf».

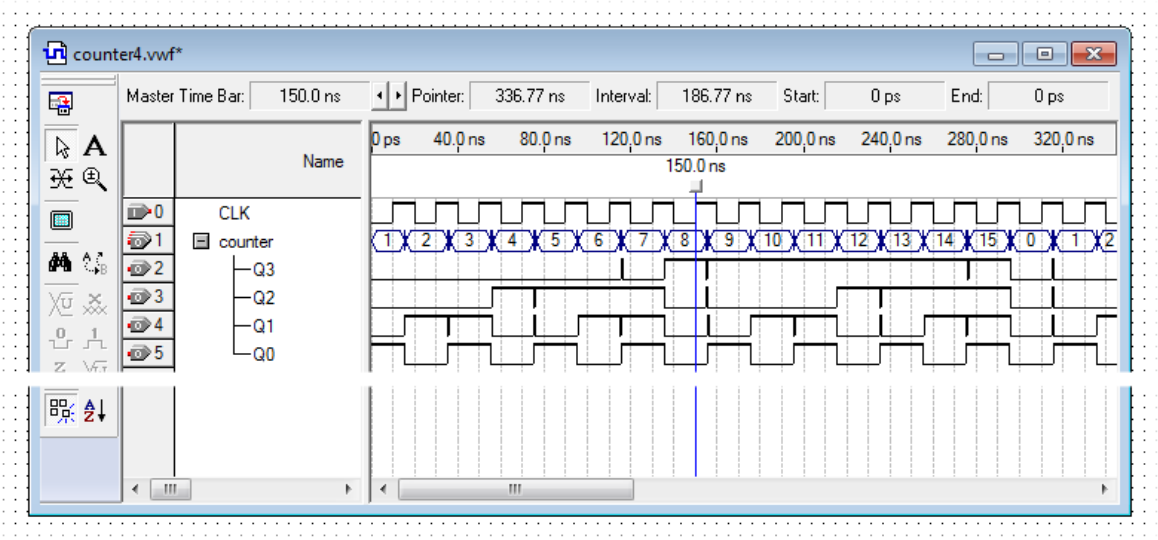


Рисунок 5.6 – Моделирование файла «counter4» проекта «blink» в Quartus

Тип моделирования можно оставить временной «Timing». При таком моделировании будут учтены задержки на распространение сигнала в реальной микросхеме, данный тип является самым точным. При работе с асинхронными схемами необходимо использовать именно такой тип моделирования, при синхронном дизайне допускается проводить моделирование в режиме «Functional». Однако если временные ресурсы позволяют, все же лучше проводить

именно временное моделирование. Далее необходимо нажать кнопку запуска «Start» и перейти к файлу отчета, который теперь совпадает с рисунком 5.6. Из рисунка 5.6. видно, что счетчик по фронту тактового сигнала увеличивает свое значение на единицу, также видна разница в скорости прохождения сигналов по каждому разряду (такое подключение является нежелательным и приведено как наглядный пример). Так как счетчик всего 4-разрядный, то счет идет с 0 до 15, после чего повторяется вновь.

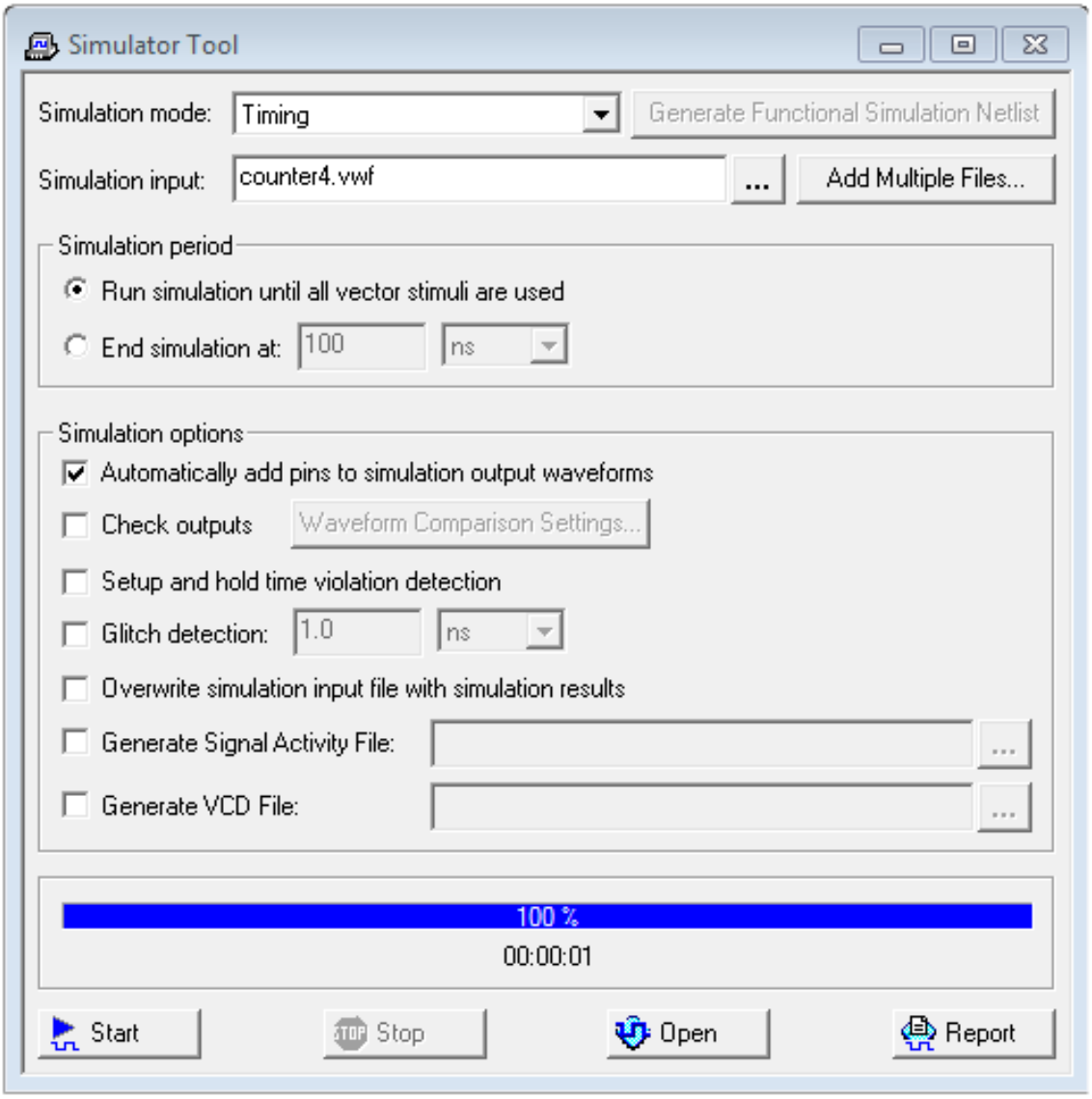


Рисунок 5.7 – Окно настроек параметров моделирования в Quartus

Для реализации проекта «blink» необходимо, чтобы при достижении строго определенного значения счетчик обнулялся, и при этом происходило переключение, т.е. инвертирование предыдущего состояния вывода, который будет подключен к светодиоду. Для этого создадим новый графический файл «comparator» и составим в нем схему, представленную на рисунке 5.8.

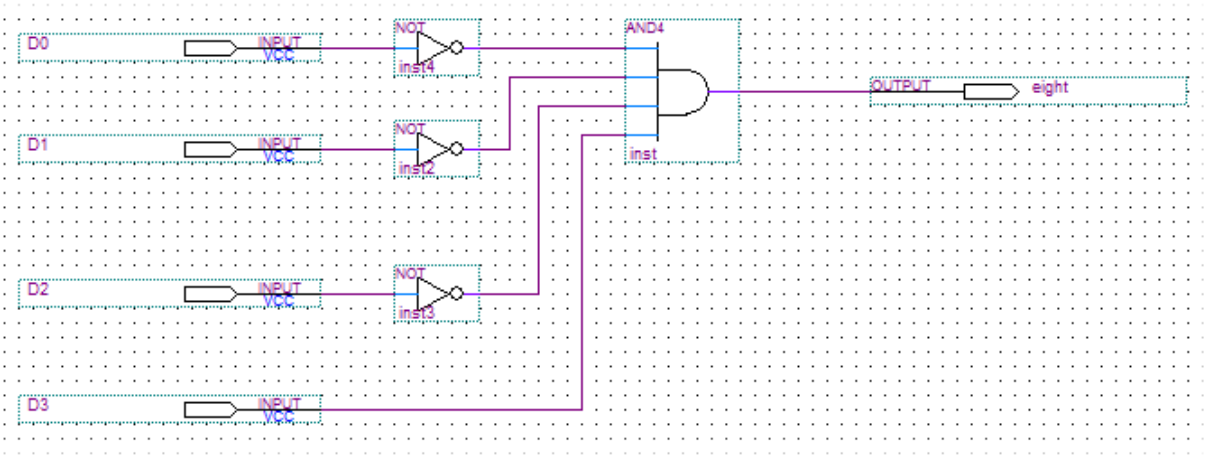


Рисунок 5.8 – Схема, детектирующая значение «8» на 4-битном входе

Схему четырехбитного счетчика необходимо модернизировать, добавив к нему вывод синхронного сброса, и обновить его графическое отображение. После чего необходимо перейти в самый первый графический файл «blink_ML» или создать новый, назначить его файлом верхнего уровня и собрать схему, состоящую из собственных макромодулей и примитивов, как показанная на рисунке 5.9. Из схемы видно, что добавлены два D-триггера. Первый для синхронизации вывода компаратора и тактового сигнала, чтобы исключить ложные переключения второго триггера, вызванные цифровым дребезгом на выходе счетчика. Второй D-триггер как раз выполняет переключение вывода «LED», к которому будет подключен светодиод. Создадим новый файл диаграмм для схемы на рисунке 5.9 и проведем временное моделирование, результаты которого представлены на рисунке 5.10.

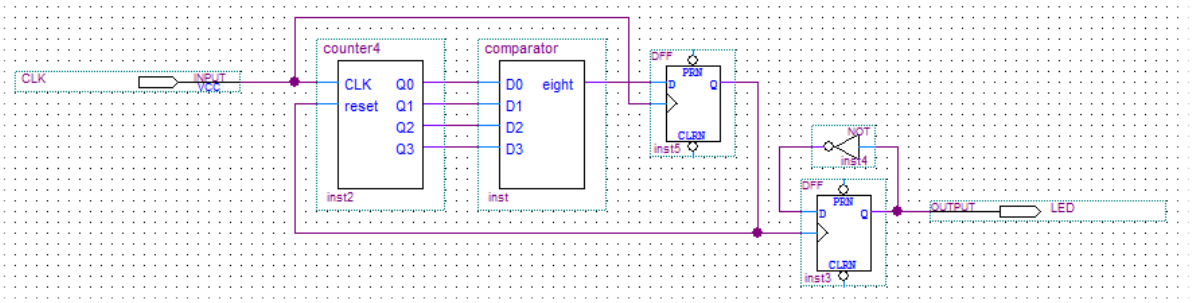


Рисунок 5.9 – Схема модуля верхнего уровня проекта «blink»

При выводе результатов моделирования сигнал можно смотреть не только на тех линиях, что являются терминалами, но и на внутренних регистрах, как показано на рисунке 5.10, где значения счетчика и линии сброса являются внутренними относительно модуля сигналами.

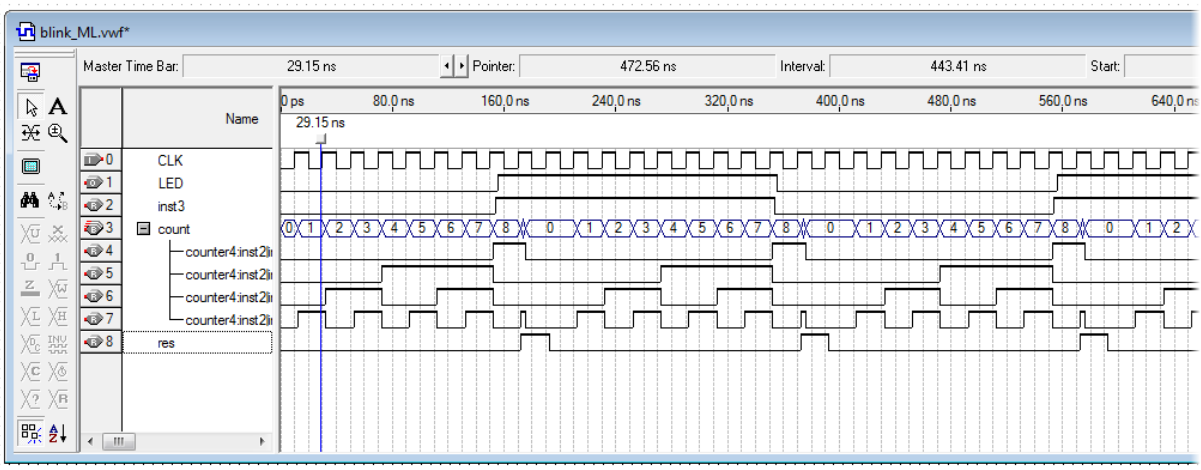


Рисунок 5.10 – Моделирование файла «blink_ML»
проекта «blink» в Quartus

Из графиков диаграмм видно, что светодиод будет переключаться на очень высокой частоте, так как частота тактового сигнала равна 50МГц, и чтобы разработанная схема позволяла переключать светодиод раз в секунду, счетчик должен считать до 50_000_000, после чего модуль «comparator» должен генерировать сигнал сброса. Данный модуль сейчас настроен на число «8», а должен, соответственно, на «50_000_000», для этого его разрядность и разрядность счетчика необходимо увеличить до 26 бит.

Более простой способ выполнить проект «blink» – использовать мегафункцию «LPM_Counter» из стандартной библиотеки. Данное решение также будет относиться к графическому методу создания описания. Для этого нужно вызвать мегафункцию «LPM_Counter», указать количество разрядов, равное 26, и указать «LMP_Modulus» равным «50_000_000», как это показано на рисунке 5.11.

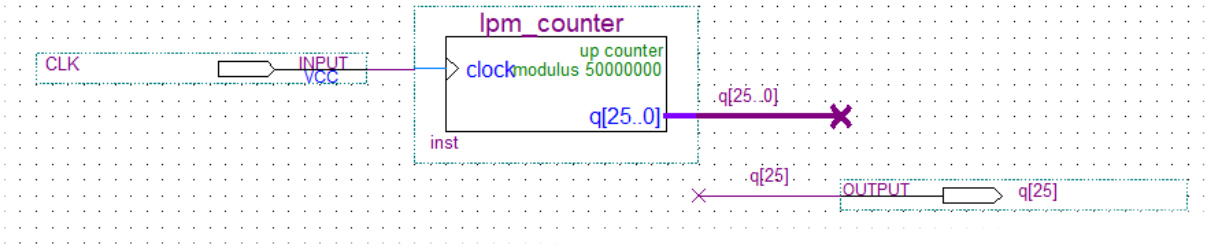


Рисунок 5.11 – Схема проекта «blink» в Quartus
с использованием «LPM_Counter»

После компиляции данного варианта проекта «blink» в MAXII EPM240T100C3 будет занято 37 логических элементов, что составляет 15% от всего объема ИМС, минимум 26 триггеров, из которых занимают регистры счетчика. В этом варианте светодиод, подключенный к выводу q[25] (старшего разряда счетчика), уже будет мигать с частотой 1Гц. Недостатком будет являться то, что время свечения и паузы будет неодинаковым. На данном этапе уже можно осуществить настройку выводом ИМС ПЛИС, для этого необходимо во вкладке «Assignments» выбрать раздел «Pins», в результате чего откроется окно, показанное на рисунке 5.12.

В открывшемся окне планировщика выводов «Pin Planner» можно назначать выводы, указывая их номер непосредственно в графе «Location» или перетаскивая мышкой название вывода на обозначения выводов на ИМС. В таблице внизу окна можно производить дополнительные настройки выводов. После проведения необходимых манипуляций в «Pin Planner» необходимо еще раз произвести компиляцию проекта, после которой в графическом файле

верхнего уровня, рядом с выводами, появится привязка к номерам ножек ИМС. Через раздел «Task» в подразделе «Time Analyzer», выбрав «View Report» или запустив «TimeQuest Timing Analyzer» для расширенного анализа, можно посмотреть временные параметры разработанного описания, например максимальную частоту тактового сигнала, которая в данном проекте равна 151МГц.

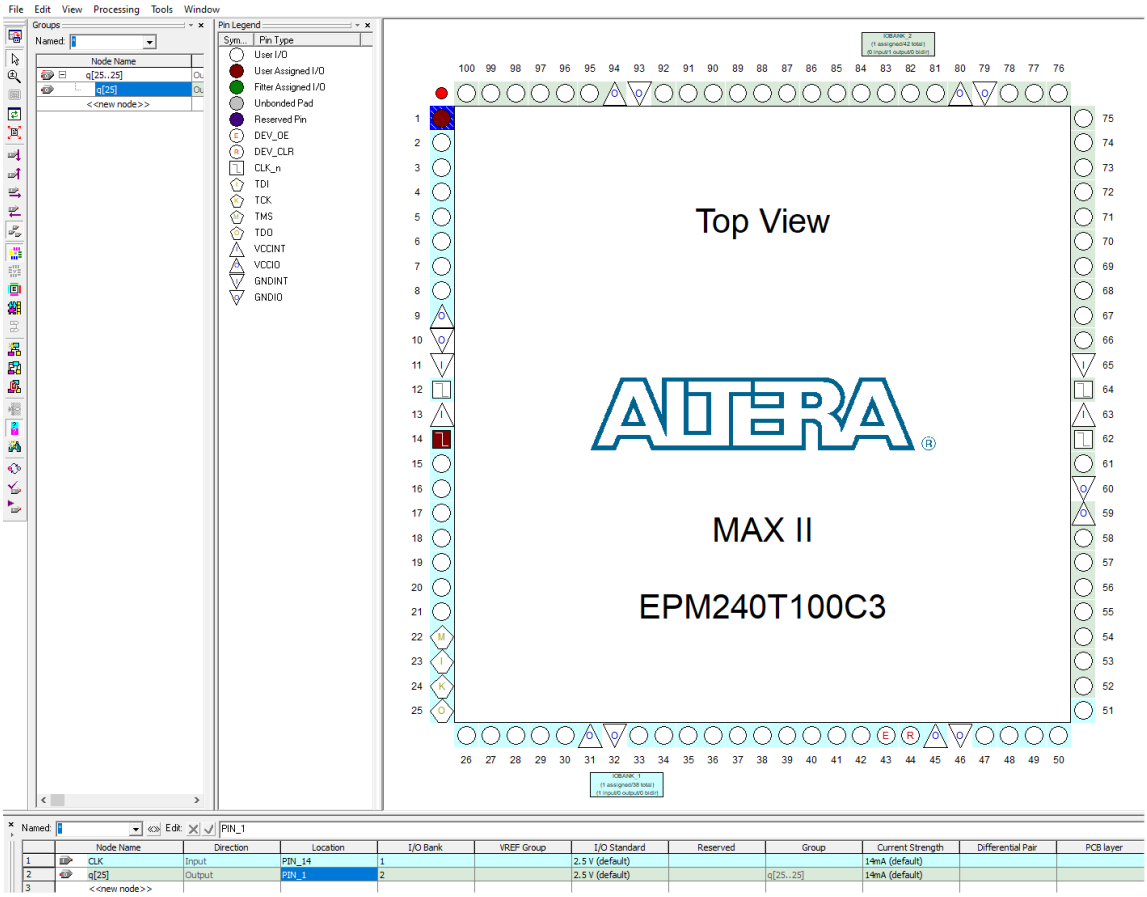


Рисунок 5.12 – Окно «Pin Planner» в Quartus

После полной компиляции проекта можно посмотреть, как компилятор построил структуру по указанному описанию. Для этого во вкладке «Tools» в разделе «Netlist_Viewers» выбрать «RTL_Viewer». Откроется иерархическая структура, внизу которой будет схема, отражающая реальную структуру, организованную внутри ПЛИС. В данном проекте структура будет, как показано на рисунке 5.13.

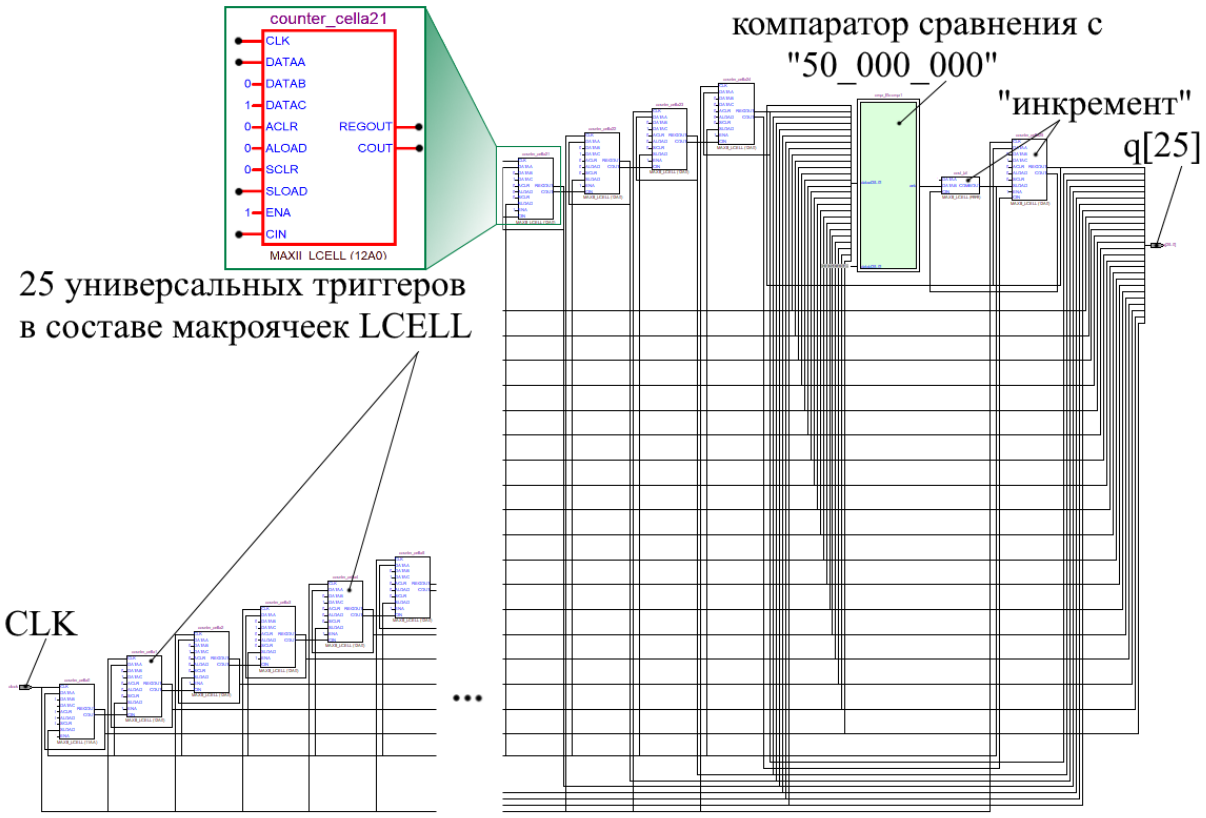


Рисунок 5.13 – RTL файла «blink_ML» в Quartus

Таким образом, проект «blink», позволяющий «мигать» светодиодом с частой 1Гц, полностью готов и его можно записать в конфигурационную память микросхемы, которая для MAXII – ПЛИС типа CPLD является встроенной flash-памятью. Для конфигурации микросхемы необходимо выбрать «Programmer» во вкладке «Tools» и, выбрав программатор и исходный файл прошивки, нажать «Start».

* Задание:

1. Переработать схему, изображенную на рисунке 5.3, чтобы исключить явления неодновременного переключения разрядов счетчика, показанного на рисунке 5.6.
2. Объяснить функцию триггера inst15 на рисунке 5.9.
3. Провести помодульное моделирование проекта «blink» в разных режимах: «Functional» и «Timing», сравнить результаты.

5.2 Разработка описания на структурном языке AHDL

Для разработки структуры на данном языке описания необходимо создать новый файл с типом «AHDL File». Учебник по AHDL приведен в [15]. После чего отроется окно текстового редактора, которое нужно сохранить, присвоить ему имя, например «blink_AHDL», и назначить файлом верхнего уровня. В окне текстового редактора необходимо записать код, представленный в листинге 5.1.

Листинг 5.1. Описание проекта «blink_AHDL»

```
subdesign blinck_AHDL
(CLK: input; LED: output)
variable
        counter [25..0] : DFF;
begin
        counter[].clk=CLK;
        if counter[]==50000000 then
                counter[]=0;
        else
                counter[]=counter+1;
        end if;
        LED = counter[25];
end;
```

Важно, чтобы имя файла совпадало с заголовком модуля, или как принято в AHDL – субдизайном. Описание, представленное выше, работает следующим образом.

В круглых скобках обозначаются имена и назначение внешних вводов данного модуля. Вывод «CLK» является входом, а «LED» – выводом. Ниже обозначена внутренняя переменная «counter» с разрядностью 26 бит. Тип «counter» DFF, т.е. триггеры Д типа. После идет указание на то, что у каждого из 26 D-

триггеров тактовый вход соединен непосредственно с выводом «CLK», о чем говорит строчка «counter[].clk=CLK;», после чего задано условие, описывающее, какие сигналы подаются на «D» входы триггерам, образующим переменную или регистр «counter». Согласно этому условию, на «D» вход триггеров переменной «counter» подается ноль, если она равна числу 50 000 000, а если не равна, то подается предыдущее значение переменной «counter», увеличенное на единицу. Другими словами, если значение счетчика достигло пятидесяти миллионов, то все триггеры сбрасываются (счетчик обнуляется), если нет, то идет нормальный счет по тактовому сигналу. Строчка «LED = counter[25];» говорит о том, что вывод модуля «LED» связан со старшим разрядом переменной «counter».

Согласно отчету о компиляции файла «blink_AHDL», в проекте использовано 35 логических элементов, что составляет 15% объема ИМС, а полученная RTL схема может работать на частоте 167МГц. Сама RTL схема представлена на рисунке 5.14. Схема относительно предыдущего вида описания изменилась.

Язык AHDL является структурным, с его помощью описывается именно структуры проектируемой схемы, отсюда и такое сходство RTL и описания на AHDL. Модуль, описанный на языке AHDL, может быть сразу использован для назначения выводов ИМС и создания файла конфигурации для последующей «прошивки» ПЛИС, а может использоваться для создания графического символа или файла вложения для использования в более сложном проекте.

На листинге 5.2 представлен еще один пример описания на AHDL. Данный модуль является декодером двоичного формата числа в формат, подходящий для 7-сегментного индикатора.

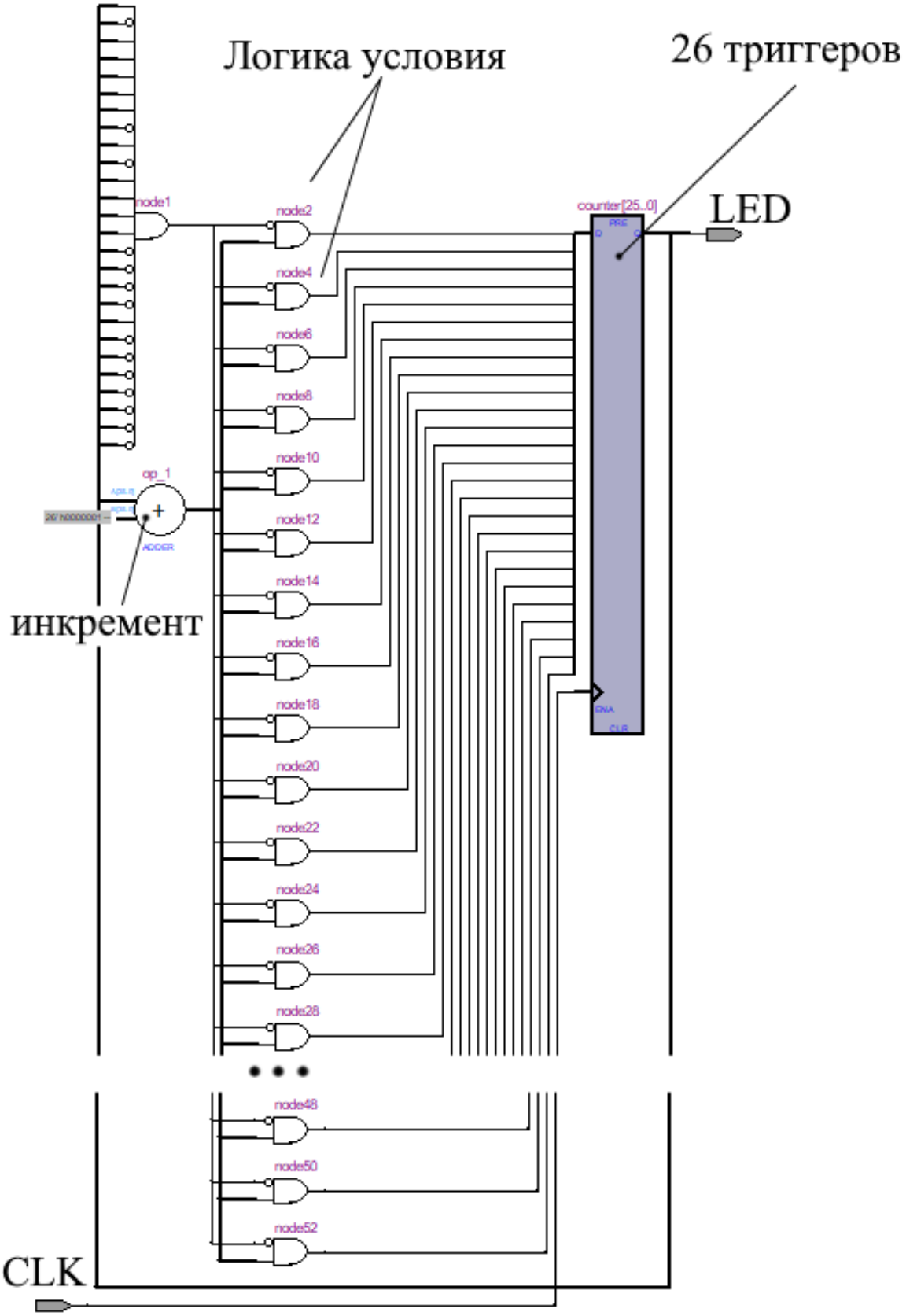


Рисунок 5.14 – RTL схема проекта «blink_AHDL» в Quartus

Листинг 5.2. Описание модуля «Segment» на AHDL

```
subdesign Segment (data[3..0]: INPUT; seg7[6..0]:OUTPUT)
begin
        case data[] is
                when 0=> seg7[]=b"1000000";
                when 1=> seg7[]=b"1111001";
                when 2=> seg7[]=b"0100100";
                when 3=> seg7[]=b"0110000";
                when 4=> seg7[]=b"0011001";
                when 5=> seg7[]=b"0010010";
                when 6=> seg7[]=b"0000010";
                when 7=> seg7[]=b"1111000";
                when 8=> seg7[]=b"0000000";
                when 9=> seg7[]=b"0010000";
                when others => seg7 = VCC;
        end case;
end;
```

5.3 Разработка описания на поведенческом языке VHDL

Языки описания аппаратуры, такие как VHDL и Verilog, отличаются от структурных тем, что позволяет описывать не только конкретные ячейки в их соединениях, но и как должна действовать разрабатываемая структура.

Для создания на языке VHDL необходимо создать новый соответствующий файл и назначить его файлом верхнего уровня, предварительно сохранив его и присвоив уникальное имя, например, blink_VHDL. Код, который необходимо ввести в окне текстового редактора, представлен в листинге 5.3.

Листинг 5.3. Описание проекта «blink_VHDL»

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity blink_VHDL is
```

```
port (CLK: in std_logic; LED: out std_logic);
end blink_VHDL;

architecture blink of  blink_VHDL is
signal counter: unsigned(25 downto 0);
begin
process (CLK)
        begin
                if rising_edge (CLK) then
                        if counter = to_unsigned(50000000,26) then
                                counter<=(others => '0');
                        else
                                counter<=counter+1;
                        end if;
                end if;
end process;
LED<=counter(25);
end blink;
```

Описание, представленное выше, работает следующим образом.

Сначала (первые три строки) описываются используемые библиотеки. Строка «use ieee.numeric_std.all;» необходима для корректной работы с беззнаковым представлением бинарных чисел. В следующем блоке обозначается имя модуля «blink_VHDL» и дается описание выводов «CLK» и «LED». Далее идет описание переменной или в VHDL сигнала «signal counter: unsigned (25 downto 0);», в данной строке говорится о том, что разрядность сигнала «counter» составляет 26 бит, и дано описание положения старшего и младшего бита.

Следующий блок описывает логику работы, согласно которой на вывод «LED» всегда подается старший разряд сигнала «counter». Внутри конструкции «process» описаны действия, которые будут всегда происходить при выполнении условия «if rising_edge (CLK) then», т.е. действия будут происходить синхронно с фронтом сигнала со входа «CLK». Дале описаны сами

действия. Согласно описанию, если значение переменной «counter» не равно числу 50 000 000, то её значение увеличивается на единицу, в противном случае переменная «counter» обнуляется.

Согласно отчету о компиляции файла «blink_VHDL», использовано 53 логических элемента, что составляет 22% объема, а полученная RTL схема, представленная на рисунке 5.15, может работать на частоте 180МГц.

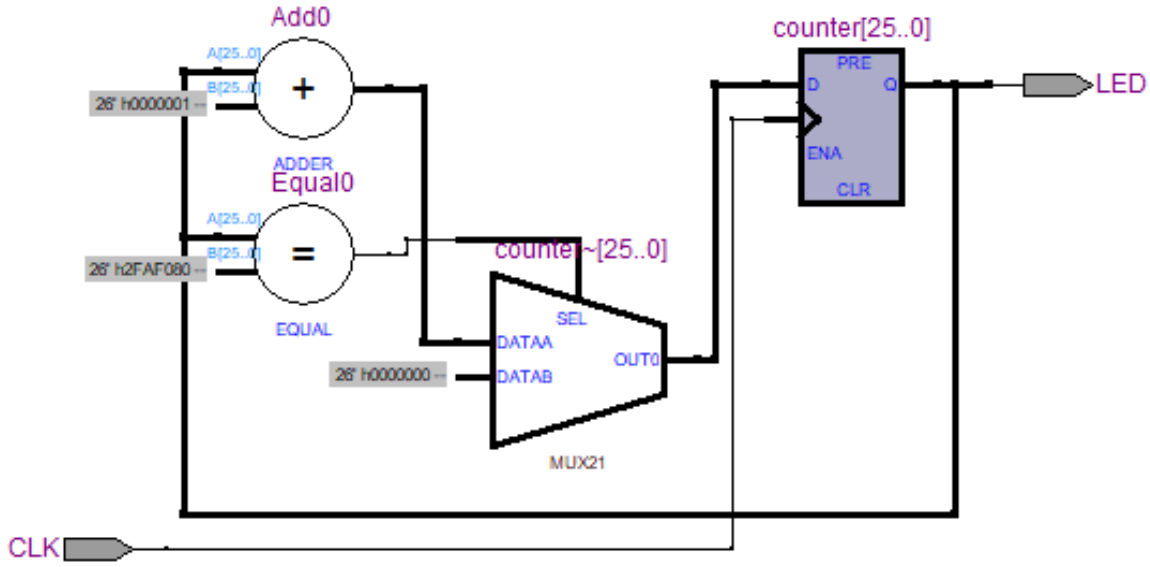


Рисунок 5.15 – RTL схема проекта «blink_VHDL» в Quartus

5.4 Разработка описания на поведенческом языке Verilog

Создадим новый файл верхнего уровня с именем «blink_verilog» и запишем в него код, представленный в листинге 5.4.

Листинг 5.4. Описание проекта «blink_verilog»

```
module blink_verilog
(CLK, LED);
input wire CLK;
output wire LED;
reg [25:0] counter;
```

```
assign LED = counter[25];

always @(posedge CLK) begin
        if (counter == 26'd50_000_000) begin
                counter<=0;
        end
        else begin
                counter<=counter+1'd1;
        end
end
endmodule
```

Verilog оперирует понятием модуля, и все описание соответственно разбивается по отдельным модулям. Вначале необходимо указать название модуля (рекомендуется, чтобы оно совпадало с названием файла). Далее описываются внешние относительно модуля вводы-выводы, и дается описание их типа. Строчка «reg [25:0] counter;» дает описание внутреннему 26-битному регистру с именем «counter» и описание, какой разряд является страшим.

Строка «assign LED = counter[25];» является асинхронной частью модуля и описывает постоянное подключение вывода «LED» к старшему биту 26-битного регистра «counter».

Внутри блока «always» все происходит параллельно, синхронно с сигналами из «списка чувствительности», в данном случае синхронно с фронтом тактового сигнала «CLK». Внутри блока «always» заключено условие, согласно которому, если значение регистра «counter» не равно числу 50 000 000, то его значение увеличивается на единицу, в противном случае «counter» обнуляется.

Результаты компиляции файла «blink_verilog» идентичны по всем параметрам результатам, полученным при компиляции «blink_VHDL», включая параметры быстродействия. Полученная RTL схема также идентична рисунку 5.15.

* Задание:

1. Объяснить различие в описаниях с листингов 5.1, 5.3 и 5.4.
2. Объяснить их принцип работы.
3. Объяснить принцип работы описания с листинга 5.2, создать графическое представление модуля, проанализировать RTL.

5.5 Полный цикл разработки проекта для ПЛИС

Ранее был разобран простой пример создания описания первого проекта на ПЛИС. В данном разделе предлагается рассмотреть пример полного цикла разработки описания, которое будет также вестись на трех разных HDL, а верхний уровень будет представлять собой графический файл. В качестве примера будет разработан проект цифровых часов «di_watch» с выводом на семисегментный индикатор.

В качестве исходных данных будет выступать схема электрическая принципиальная, представленная на рисунке 5.16.

Небольшие сегментные индикаторы можно подключать напрямую к ИМС ПЛИС, однако для наглядности в схему включены транзисторные каскады, полностью разгружающие выводы ПЛИС и позволяющие подключать крупные и яркие индикаторы с большим током потребления на сегмент. Цепи питания ПЛИС показаны условно, так как конкретная модель ИМС будет выбрана после разработки описания. По этой же причине не назначены выводы, так как их целесообразно назначать после трассировки платы. Будет использована максимально компактная CPLD, поэтому внешней конфигурационной памяти нет. Программирование осуществляется через стандартный разъем, совместимый со стандартным программатором Altera USB-blaster. Тактовый генератор подключен к одному из доступных входов тактирования и имеет частоту 50Мгц.

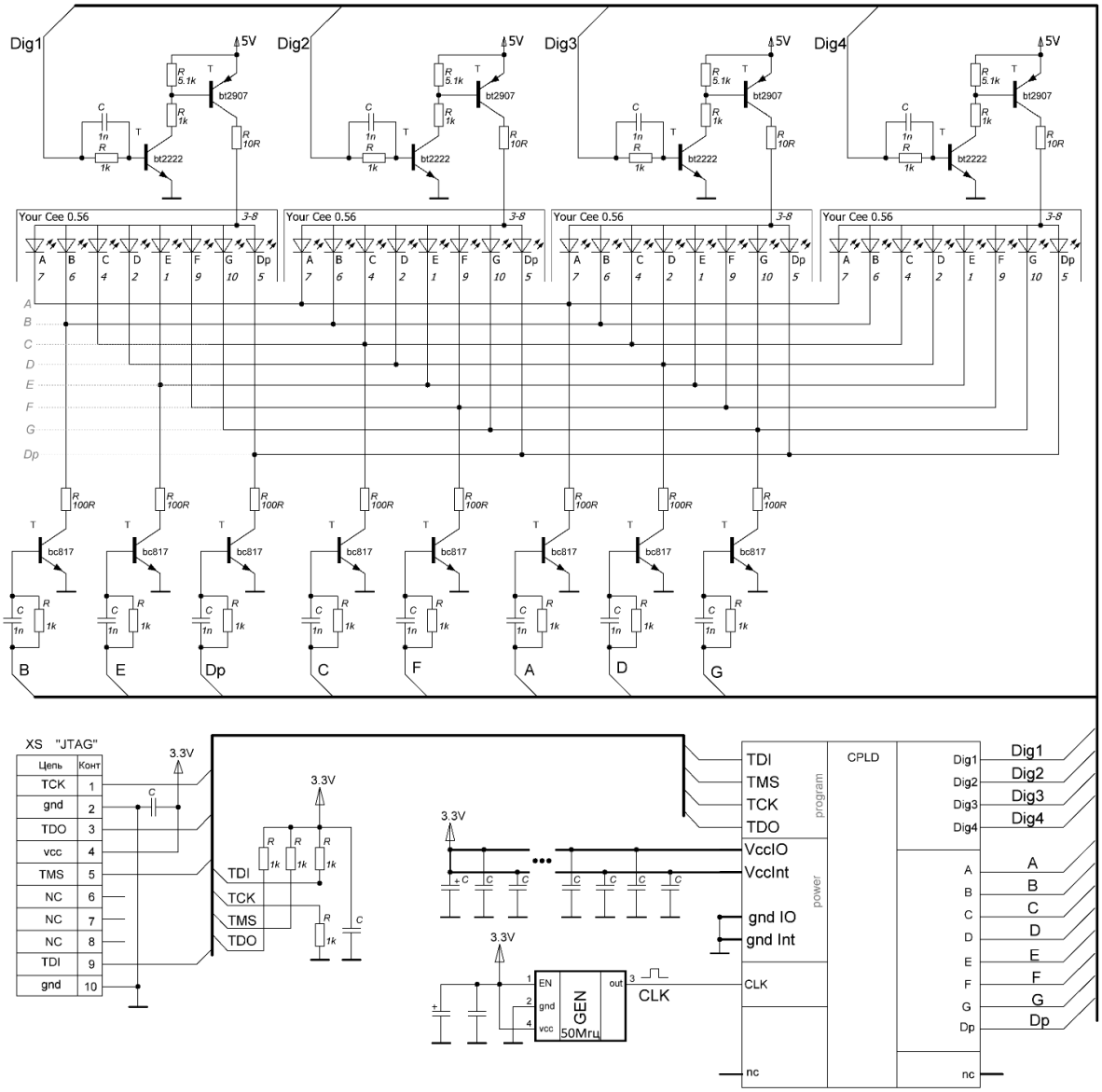


Рисунок 5.16 – Принципиальная схема проекта цифровых часов

Перед началом разработки описания необходимо составить структурную схему, или алгоритм, если планируется использовать поведенческое описание. Структурная схема на начальном этапе может отличаться от финального результата, однако начинать разработку сложных проектов вообще без структуры, хотя бы в первом приближении недопустимо. Структурная схема цифровых часов представлена на рисунке 5.17.

Создадим в Quartus новый проект с именем «di_watch». В качестве целевой микросхемы можно выбрать заведомо большую,

чем необходимо по объему. После отладки проекта, перед назначением выводов, её можно поменять на оптимальную. При создании проекта можно добавить счетчик, разработанный ранее. Далее создадим графический файл для верхнего уровня с именем «di_watch_ML». Ранее разработанный 26-разрядный счетчик необходимо модернизировать и добавить в него линию сброса. При разработке синхронного дизайна все блоки должны тактироваться от глобального тактового сигнала и, желательно, иметь линию полного сброса всех триггеров, входящих в дизайн. Сначала будет применен «модульный» подход, полностью повторяющий структурную схему. Описание отдельных модулей будет выполнено на Verilog. При написании кода необходимо понимать, какие цели преследуются: это достижение минимально занимаемого объема или же достижение максимальной скорости работы, выбор, как правило, зависит от целевой микросхемы. Так, например, CPLD имеет относительно мало логических элементов, и их следует экономить, однако сами элементы весьма быстрые и вносят небольшие задержки в распространение сигнала. Банальный счетчик, с небольшими дополнительными функциями для проекта цифровых часов, для CPLD лучше выполнить, как показано в листинге 5.5.

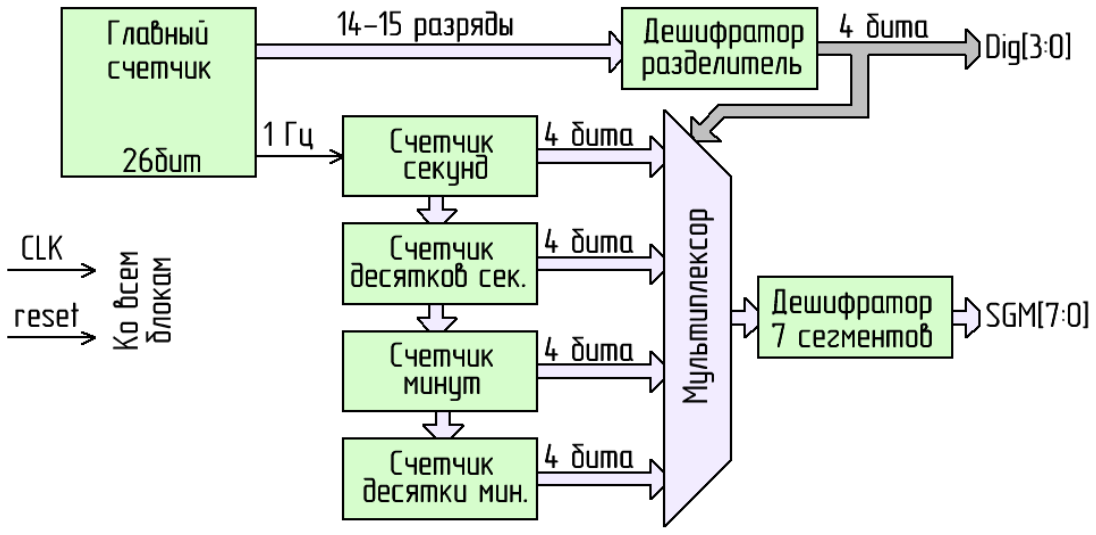


Рисунок 5.17 – Структурная схема проекта цифровых часов

Листинг 5.5. Описание счетчика для CPLD

```
module M_count
(CLK, reset, Q, Hz);
input wire CLK, reset;
output wire [1:0]Q; assign Q[1:0] = counter [15:14];
output reg Hz;
reg [25:0] counter;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                Hz<=0; counter<=0;
        end // reset
        else begin
                if (counter==26'd50_000_000) begin
                        counter<=0;
                        Hz<=1'd1;
                end
                else begin
                        counter<=counter+1'd1;
                        if (Hz) begin
                                Hz<=0;
                        end
                end
        end // CLK
end // always
endmodule
```

Такое описание может работать на частоте 130Мгц (для MAXII C3) и занимает 62 логических элемента. Проблема данного типа описания – это длинные цепочки условий, каждое условие может порождать создание логики перед входом D-триггеров, что вносит потенциальную задержку. Для ряда ИМС длинные цепочки вложенных условий могут являться причиной необходимости снижения частоты тактирования конкретного домена или ИМС целиком. Другой вариант описания представлен в листинге 5.6.

Листинг 5.6. Вариант описания счетчика

```
module M_count
(CLK, reset, Q, Hz);
input wire CLK, reset;
output wire [1:0]Q; assign Q[1:0] = counter [15:14];
output reg Hz;

reg [25:0] counter;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                Hz<=0; counter<=0;
        end // reset
        else begin
                case (counter)

                26'd0:  begin
                                counter<=counter+1'd1;
                                Hz<=1'd1;
                        end
                26'd1:  begin
                                counter<=counter+1'd1;
                                Hz<=0;
                        end
                26'd50_000_000:     begin
                                        counter<=0;
                                end
                default: counter<=counter+1'd1;
                endcase
        end // CLK
end // always
endmodule
```

Второй вариант использует уже 66 логических элементов, но максимальная частота при прочих равных равна 153Мгц. Тут происходит потактовое разделение действий в поведении схемы. А компилятором создается дизайн, аналогичный описанию счетчика на AHDL, RTL которого представлена на рисунке 5.14. Результат моделирования второго варианта показан на рисунке

5.18. Частота тактового сигнала составляла 100Мгц, поэтому через каждые 500мс генерируется короткий, равный одному тактовому периоду, импульс. Выводы «Q» будут использоваться для организации сканирования разрядов сегментных индикаторов при динамической индикации.

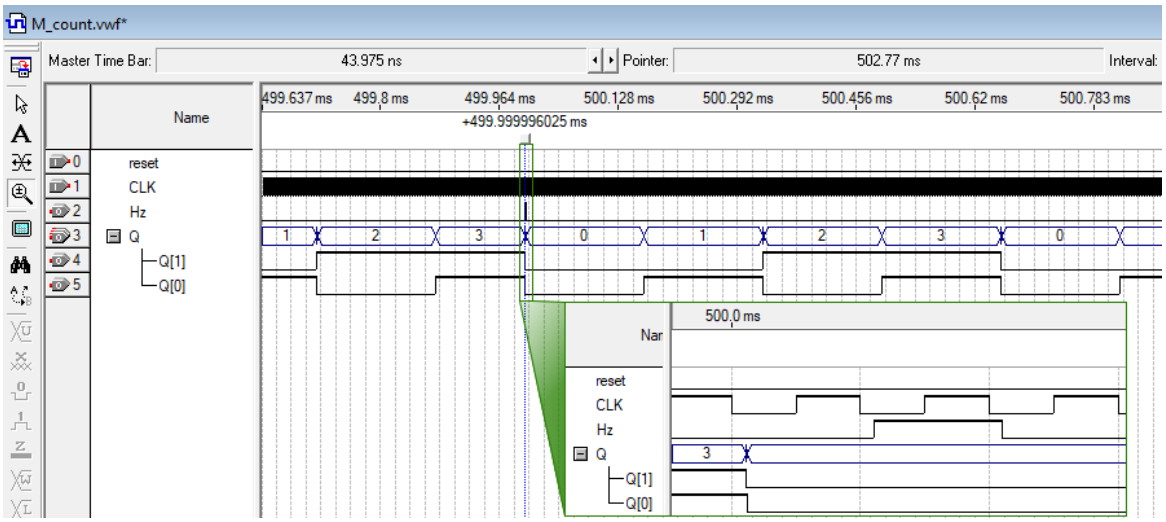


Рисунок 5.18 – Временные диаграммы,
полученные при моделировании файла «M_count»

Создадим универсальный четырехбитный счетчик «time_ counter», описание его модуля «time_counter» представлено на листинге 5.7.

Листинг 5.7. Описание модуля «time counter»

```
module time_counter
(CLK, reset, strob, Q_time, transfer);
parameter mod_count=9;
input wire CLK, reset, strob;
output reg [3:0] Q_time;
output reg transfer;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                transfer<=0; Q_time<=0;
        end // reset
        else begin
                if (strob) begin
```

```
                if (Q_time==mod_count) begin
                        Q_time<=0;
                        transfer<=1'd1;
                end
                else begin
                        Q_time<=Q_time+1'd1;
                end
        end // strob
        if (transfer) begin
                transfer<=0;
        end
    end // CLK
end // always
endmodule
```

Основание счета модуля «time_counter» задается параметром «**parameter** mod_count=9». Временные диаграммы показаны на рисунке 5.19.

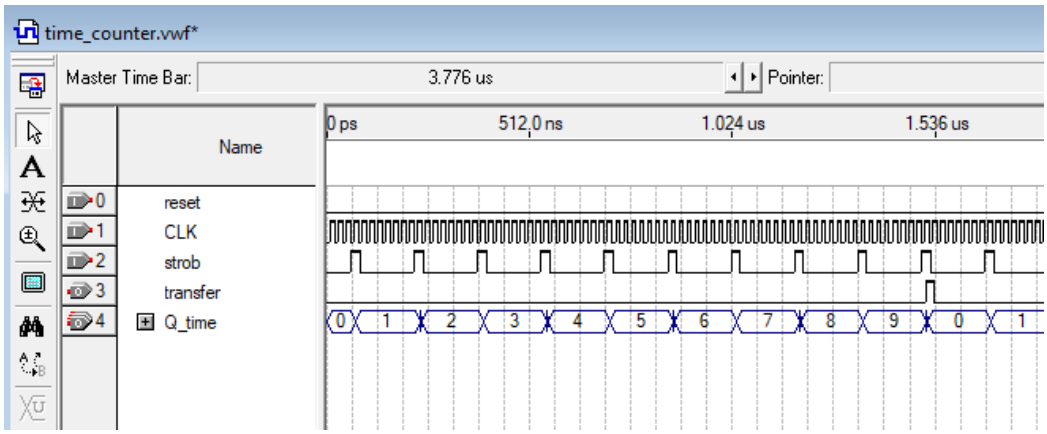


Рисунок 5.19 – Временные диаграммы,
полученные при моделировании файла «time_counter»

Следующий модуль, необходимый для построения схемы часов по структурной схеме, представленной на рисунке 5.17, – дешифратор «separator», который будет из двух разрядов счетчика генерировать последовательные импульсы для переключения мультиплексора и разрядов сегментных индикаторов. Его описание приведено на листинге 5.8, а временные диаграммы представлены на рисунке 5.20.

Листинг 5.8. Описание модуля «separator»

```
module separator
(CLK, reset, Q, Dig);
input wire CLK, reset;
input wire [1:0] Q;
output reg [3:0] Dig;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                Dig<=0;
        end // reset
        else begin
                case (Q)
                2'd0:   Dig[3:0]<=4'b0001;
                2'd1:   Dig[3:0]<=4'b0010;
                2'd2:   Dig[3:0]<=4'b0100;
                2'd3:   Dig[3:0]<=4'b1000;
                endcase
        end // CLK
end// always
endmodule
```

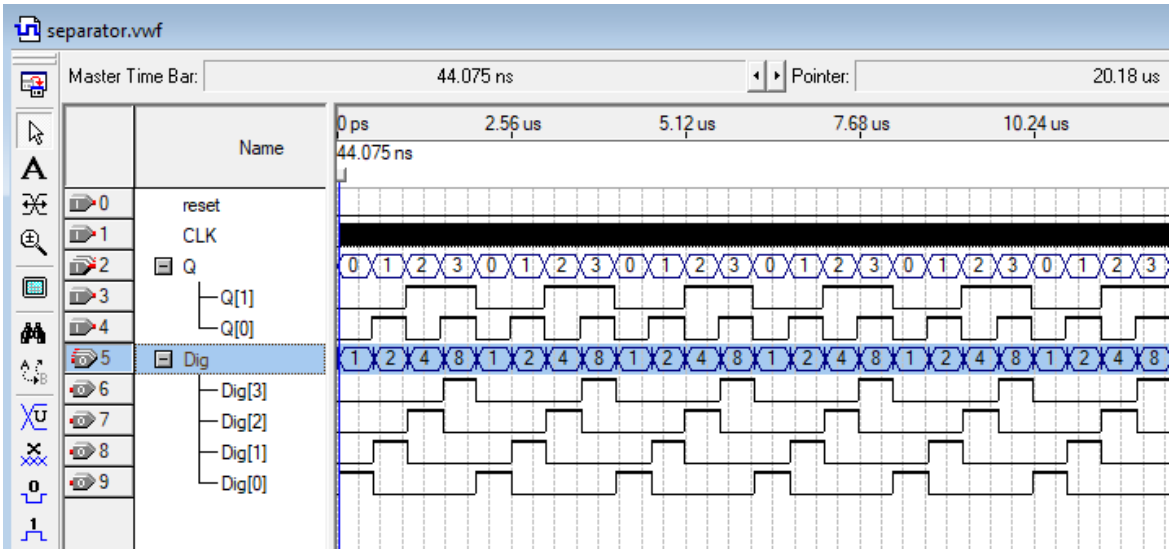


Рисунок 5.20 – Временные диаграммы,
полученные при моделировании файла «separator»

Модуль мультиплексора «MUX_4X4» обеспечивает переключение четырех входов на один выход по сигналам управления от модуля «separator». Описание «MUX_4X4» представлено в листинге 5.9.

Листинг 5.9. Описание модуля «MUX 4X4»

```
module MUX_4X4
(Dig,
Q_time1,
Q_time2,
Q_time3,
Q_time4,
mux_out);
input wire[3:0] Dig;
input wire [3:0] Q_time1;
input wire [3:0] Q_time2;
input wire [3:0] Q_time3;
input wire [3:0] Q_time4;
output wire [3:0] mux_out;
assign mux_out[3:0]= Dig[0]?Q_time1:Dig[1]?Q_time2:
                     Dig[2]?Q_time3:Dig[3]?Q_time4:0;
endmodule
```

Модуль «MUX_4X4», в силу своей простоты, построен по асинхронному дизайну и состоит только из логических блоков мультиплексоров, а его временные диаграммы представлены на рисунке 5.21.

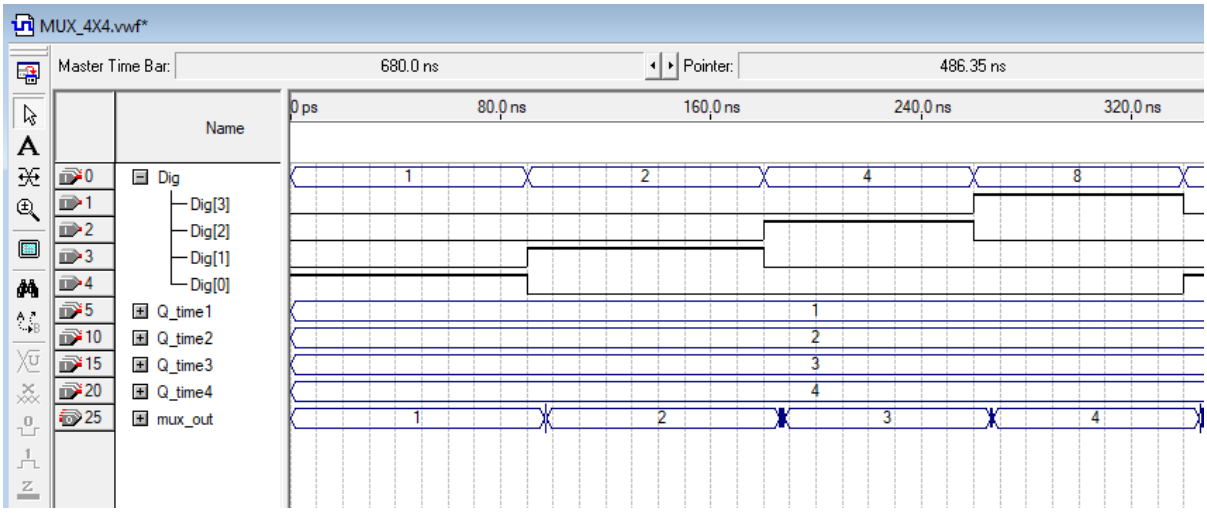


Рисунок 5.21 – Временные диаграммы,
полученные при моделировании файла «MUX_4X4»

Последний необходимый для проекта часов модуль – это модуль «MUX_7sig», являющийся дешифратором двоичных чисел в код сегментных индикаторов. Его описание представлено на листинге 5.10.

Листинг 5.10. Описание модуля «MUX 7sig»

```
module MUX_7sig
(CLK, reset, mux_out, SGM);
input wire CLK, reset;
input wire [3:0] mux_out;
output reg [6:0] SGM;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                SGM<=0;
        end // reset
        else begin
                case (mux_out)
                4'd0:   SGM[6:0]<=7'b1000000;
                4'd1:   SGM[6:0]<=7'b1111001;
                4'd2:   SGM[6:0]<=7'b0100100;
                4'd3:   SGM[6:0]<=7'b0110000;
                4'd4:   SGM[6:0]<=7'b0011001;
                4'd5:   SGM[6:0]<=7'b0010010;
                4'd6:   SGM[6:0]<=7'b0000010;
                4'd7:   SGM[6:0]<=7'b1111000;
                4'd8:   SGM[6:0]<=7'b0000000;
                4'd9:   SGM[6:0]<=7'b0010000;
                default:SGM[6:0]<=7'b1111_1111;
                endcase
        end // CLK
end // always
endmodule
```

Теперь в графическом файле верхнего уровня собирается схема, показанная на рисунке 5.22.

После компиляции проект часов занял всего 111 логических элементов, что позволяет использовать младшую ИМС из семейства MAXII, содержащую 240 логических элементов, кроме того, можно выбрать ИМС с индексом скорости «C5» EPM240T100C5, что все

равно позволяет использовать частоту тактирования более 100Мгц, а для данного проекта часов высокая частота не требуется вовсе.

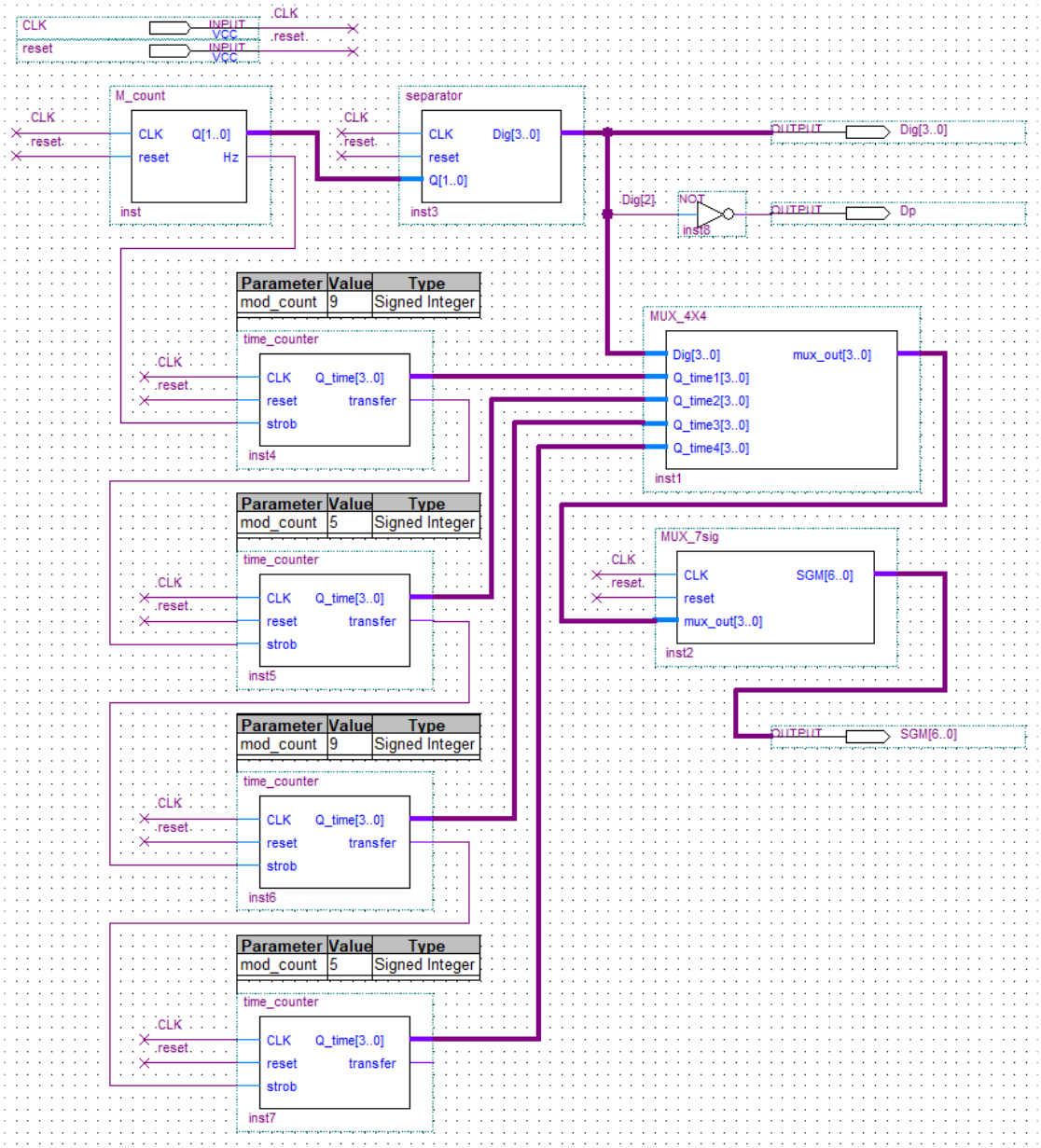


Рисунок 5.22 – Схема модуля верхнего уровня «di_watch_ML» проекта «di_watch»

После назначения выводов и повторной компиляции проекта можно загрузить прошивку в ПЛИС. Для этого необходимо перейти из вкладки «Tools» в раздел «Programmer», при этом откроется окно, показанное на рисунке 5.23.

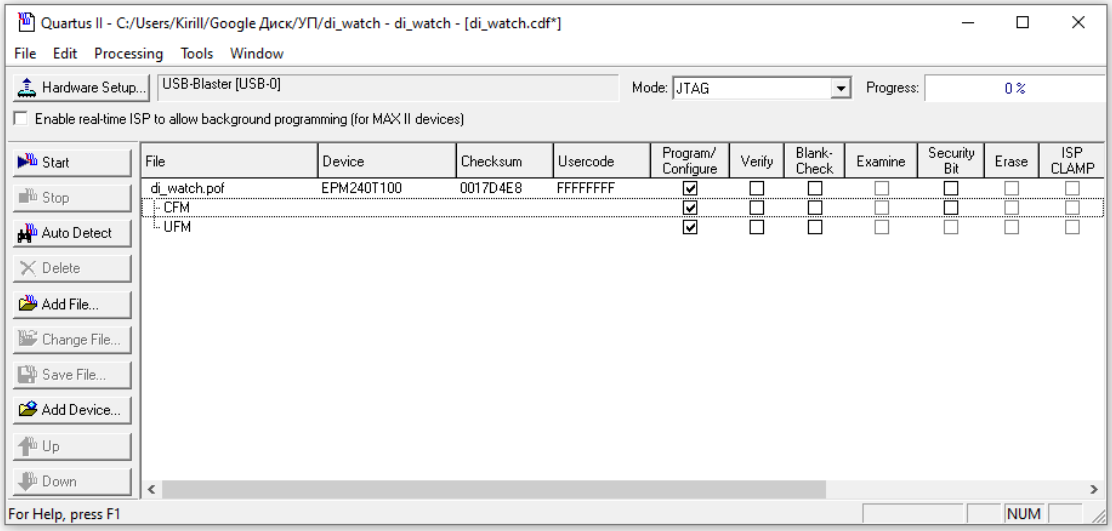


Рисунок 5.23 – Окно «Programmer»

Для программирования необходимо подать питание на плату с ИМС, подключить программатор и выбрать его через меню «Hardware Setup..», указать задачу «Program/Configure» и нажать кнопку «Start».

Схема, представленная на рисунке 5.22, требует дополнительной проверки на метастабильные состояния. Минимальный временной анализ можно провести с помощью Classic Timing Analyzer, который проводит анализ всех триггеров схемы по параметру Setup/Hold относительно CLK. Отчет о проверке «di_watch_ML» приведен на рисунке 5.24. Подробнее про (STA) – Static timing analysis и применение TimeQuest Timing Analyzer можно прочитать в [16–17].

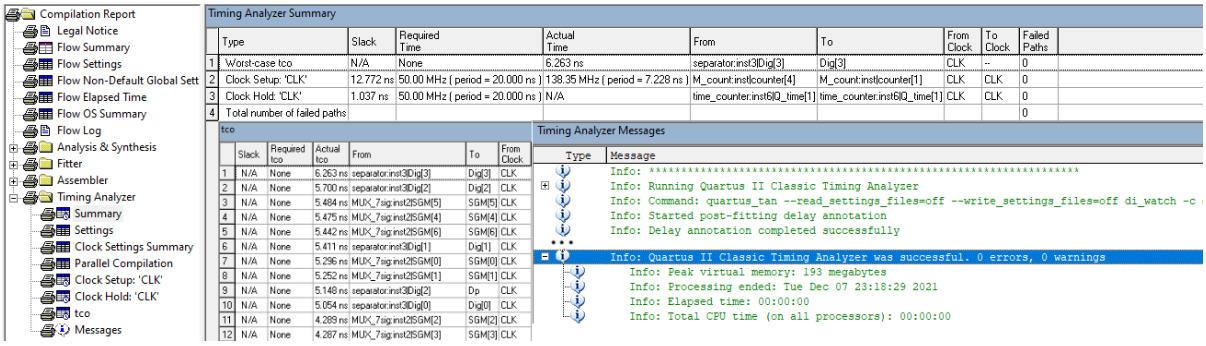


Рисунок 5.24 – Report Timing Analyzer Summary

В рассмотренном примере полного цикла создания описания ПЛИС можно отметить, что излишнее разбиение проекта на субмодули вызывает необходимость создавать однообразное описание обязательной части модуля, чего можно избежать при описании всего дизайна на HDL. Для примера в листинге 5.11 представлено описание проекта часов, но в одном текстовом файле.

Листинг 5.11. Описание модуля «di watch Verilog»

```
module di_watch_verilog
(CLK, reset, Dig, SGM, Dp);
parameter FRQ=50_000_000; // generator frequency
input wire CLK, reset;
output reg [6:0] SGM;
output reg [3:0] Dig;
output wire Dp;
assign Dp=!counter[15]&!counter[14];
reg [25:0] counter;
reg [3:0] sec_9;
reg [3:0] sec_59;
reg [3:0] min_9;
reg [3:0] min_59;
reg [2:0] FSM;
// Seven segment decoder:
        task Segment;
        input [3:0] bin;
        output [6:0] seg;
        begin
                case (bin)
                4'd0:   seg[6:0]<=7'b1000000;
                4'd1:   seg[6:0]<=7'b1111001;
                4'd2:   seg[6:0]<=7'b0100100;
                4'd3:   seg[6:0]<=7'b0110000;
                4'd4:   seg[6:0]<=7'b0011001;
                4'd5:   seg[6:0]<=7'b0010010;
                4'd6:   seg[6:0]<=7'b0000010;
                4'd7:   seg[6:0]<=7'b1111000;
                4'd8:   seg[6:0]<=7'b0000000;
                4'd9:   seg[6:0]<=7'b0010000;
                default:seg[6:0]<=7'b1111_1111;
```

```
            endcase
        end
        endtask
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                Dig<=0;  counter<=0;
                sec_9<=0; sec_59<=0;
                min_9<=0; min_59<=0;
                FSM<=0;
        end // reset
        else begin
                if (counter==FRQ) begin
                        FSM<=3'd1;
                        counter<=0;
                end
                else begin
                        counter<=counter+1'd1;
                end
                case (FSM)
                3'd1:           begin
                                        if (sec_9==4'd9) begin
                                                sec_9<=0;
                                                FSM<=FSM+1'd1;
                                        end
                                        else begin
                                                sec_9<=sec_9+1'd1;
                                                FSM<=0;
                                        end
                                end // FSM=1
                3'd2:           begin
                                        if (sec_59==4'd5) begin
                                                sec_59<=0;
                                                FSM<=FSM+1'd1;
                                        end
                                        else begin
                                                sec_59<=sec_59+1'd1;
                                                FSM<=0;
                                        end
                                end // FSM=2
                3'd3:           begin
                                        if (min_9==4'd9) begin
                                                min_9<=0;
```

```
                    FSM<=FSM+1'd1;
                end
                else begin
                    min_9<=min_9+1'd1;
                    FSM<=0;
                end
            end // FSM=3
        3'd4:   begin
                if (min_59==4'd5) begin
                    min_59<=0;
                end
                else begin
                    min_59<=min_59+1'd1;
                end
                FSM<=0;
            end // FSM=4
        default:;
        endcase
        case (counter[15:14])
        2'd0:   begin
                Dig[3:0]<=4'b0001;
                Segment(sec_9,SGM);
            end
        2'd1:   begin
                Dig[3:0]<=4'b0010;
                Segment(sec_59,SGM);
            end
        2'd2:  begin
                Dig[3:0]<=4'b0100;
                Segment(min_9,SGM);
            end
        2'd3:   begin
                Dig[3:0]<=4'b1000;
                Segment(min_59,SGM);
            end
        endcase
    end // CLK
end // always
endmodule
```

Представленный дизайн «di_watch_verilog» содержит отдельную задачу «task» для описания декодера сегментного индикатора, а десятичные счетчики описаны универсальным способом с временным разделением посредством встроенного регистра состояний «FSM». Дизайн занял 130 логических элементов и может работать на частотах свыше 100МГц.

Для сравнения вариантов описания на листинге 5.12 представлено описание на VHDL, а на листинге 5.13 – на AHDL.

Листинг 5.12. Описание модуля «di watch VHDL»

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
entity di_watch_VHDL is port(
        CLK : in STD_LOGIC;
        reset : in STD_LOGIC;
        Dig : out STD_LOGIC_VECTOR (3 downto 0);
        SGM   : out STD_LOGIC_VECTOR (7 downto 0));
end di_watch_VHDL;
architecture di_watch of di_watch_VHDL is
signal counter : unsigned (25 downto 0);
signal sec_9 : unsigned (3 downto 0);
signal sec_59 : unsigned (3 downto 0);
signal min_9 : unsigned (3 downto 0);
signal min_59 : unsigned (3 downto 0);
function Segment(
        bin: in unsigned (3 downto 0))
        return std_logic_vector is
        variable seg : std_logic_vector (6 downto 0);
        begin
        if bin = X"0" then seg:=b"1000000";
        elsif bin = X"1" then seg:=b"1111001";
        elsif bin = X"2" then seg:=b"0100100";
        elsif bin = X"3" then seg:=b"0110000";
        elsif bin = X"4" then seg:=b"0011001";
        elsif bin = X"5" then seg:=b"0010010";
        elsif bin = X"6" then seg:=b"0000010";
        elsif bin = X"7" then seg:=b"1111000";
```

```
        elsif bin = X"8" then seg:=b"0000000";
        elsif bin = X"9" then seg:=b"0010000";
        else seg:=(others=> '1');
        end if;
        return std_logic_vector (seg);
end;
begin
        process (CLK, reset) begin
        if reset='0' then
                counter <=(others=>'0');
                sec_9 <=(others=>'0');
                sec_59 <=(others=>'0');
                min_9 <=(others=>'0');
                min_59 <=(others=>'0');
        elsif rising_edge(CLK) then
                if counter=to_unsigned(50000000,26) then
                        counter<=(others => '0');
                                if sec_9=9 then
                                sec_9<=x"0";
                                        if sec_59=5 then
                                        sec_59<=x"0";
                                                if min_9=9 then
                                                min_9<=x"0";
                                                        if min_59=5 then
                                                        min_59<=x"0";
                                                        else
                                                        min_59<=min_59+1;
                                                        end if;
                                                else
                                                min_9<=min_9+1;
                                                end if;
                                        else
                                        sec_59<=sec_59+1;
                                        end if;
                                else
                                sec_9<=sec_9+1;
                                end if;
                        else
                        counter<=counter+1;
                        end if;
        case to_integer(counter(15 downto 14)) is
        when 0=> Dig<=b"1000"; SGM<=('1' & Segment(sec_9));
```

```
        when 1=> Dig<=b"0100"; SGM<=('1' & Segment(sec_59));
        when 2=> Dig<=b"0010"; SGM<=(counter(25) & Segment(min_9));
        when 3=> Dig<=b"0001"; SGM<=('1' & Segment(min_59));
        when others => Dig <= b"0000"; SGM<=(others=>'1');
        end case;
end if;
end process;
end di_watch;
```

Листинг 5.13. Описание модуля «di watch AHDL»

```
include "lpm_counter";
include "Segment ";
subdesign di_watch_AHDL (CLK  ,reset: INPUT;
        led[4..0], Dig[3..0], seg[7..0]: OUTPUT)
variable
cnt:    lpm_counter WITH (LPM_WIDTH=26, LPM_MODULUS=5000000);
sec_9:  lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
sec_59:         lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=6);
min_9:lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=10);
min_59:         lpm_counter WITH (LPM_WIDTH=4, LPM_MODULUS=6);
begin
cnt. clock=CLK;
led[]=cnt.q[25..21];
sec_9. clock =CLK;
sec_59. clock =CLK;
min_9. clock =CLK;
min_59. clock =CLK;
sec_9. clk_en=cnt. cout;
sec_59. clk_en=cnt. cout&sec_9. cout;
min_9. clk_en=cnt. cout&sec_9. cout&sec_59. cout;
min_59. clk_en=cnt. cout&sec_9. cout&sec_59. cout&min_9. cout;
case cnt.q [15..14] is
        when 0 =>Dig[]=b"0001"; seg=(VCC, Segment (min_59.q[]));
        when 0 =>Dig[]=b"0010"; seg=(cnt.q[25], Segment (min_9.q[]));
        when 0 =>Dig[]=b"0100"; seg=(VCC, Segment (sec_59.q[]));
        when 0 =>Dig[]=b"1000"; seg=(VCC, Segment (sec_9.q[]));
end case;
end;
```

Дизайн, описанный на AHDL, занял 127 логических элементов и допускает тактирование на частоте до 136МГц. Результаты отчета о компиляции проекта на VHDL совпадают с результатами проекта на Verilog.

* Задание:

1. Объяснить, в чем разница представленных описаний на Verilog, VHDL и AHDL в листингах 5.11–5.13 соответственно.

2. Объяснить, почему при компиляции описания с листинга 5.13 возникает ошибка, что необходимо добавить в данный проект.

6 ОСНОВНЫЕ ПОДХОДЫ ПРИ РАЗРАБОТКЕ ОПИСАНИЯ ДЛЯ ПЛИС

Существует несколько основных подходов к реализации сложного проекта или алгоритма на ПЛИС. Все они позволяют решить поставленные задачи, но обладают спецификой применения, которую необходимо брать во внимание при выборе подхода к разработке. В действительно сложном проекте, скорее всего, целесообразно применить компиляцию различных подходов для реализации отдельных функций и задач. В любом случае можно выделить следующие парадигмы:

– описание посредством комбинаторной логики;

– создание конечного автомата;

– реализация софт-процессора на ПЛИС и разработка программы к нему;

– использование аппаратных возможностей систем на кристалле SoC, если они заложены в используемую ИМС, или создание синтезируемой SoC.

Первый вариант состоит в разработке логической схемы с частым использованием асинхронного дизайна. Такой подход применим для создания несложных или специфических проектов, с использованием нетипичных для ПЛИС приемов описания, например, измеритель субтактовых задержек для TDC (time-to-digital converter) [18].

Наиболее распространённый прием описания – это создание конечного автомата. Конечный автомат будет так или иначе использован всегда, когда необходимо преодолеть «прирожденную параллельность» ПЛИС и выполнить какую-то последовательность действий. Можно сказать, что данный подход является основным в создании синхронных управляющих дизайнов, так

как даже процессор, создание которого является отдельным вариантом выполнения сложного проекта на ПЛИС, также является конечным автоматом. Конечный автомат, или по-другому – автомат состояний FSM в англоязычной литературе, представляет из себя устройство, находящееся в каждый момент времени в одном из множества возможных состояний, число которых конечно, переходы между состояниями вызываются внешними воздействиями. Введение в теорию конечных автоматов представлено в [19].

6.1 Пример разработки учебного софт-процессора для ПЛИС

Разработка процессора внутри ПЛИС может быть удобным и выгодным решением, когда необходимо реализовать сложный алгоритм с разветвлённой логикой и большим количеством последовательных операций. Синтезированный (софт)-процессор удобен для управления периферийными модулями, выполняющими свою определенную функцию. Модули периферии лишь при необходимости обращаются к процессору, например, через прерывания, а непосредственно ЦП следит только за общим выполнением алгоритма. Другие варианты FSM описания учебных процессоров приведены в [2].

Далее представлено описание максимально упрощенного учебного процессора, вполне пригодного для учебных и ознакомительных целей или для применения в максимально простых проектах в качестве сопроцессора, для выполнения линейных алгоритмов не требовательных к скорости исполнения. Структурная схема простого процессора (simple processor), далее «SIP», представлена на рисунке 6.1. Описание ядра процессора «SIP» на Verilog_HDL приведено в листинге 6.1.

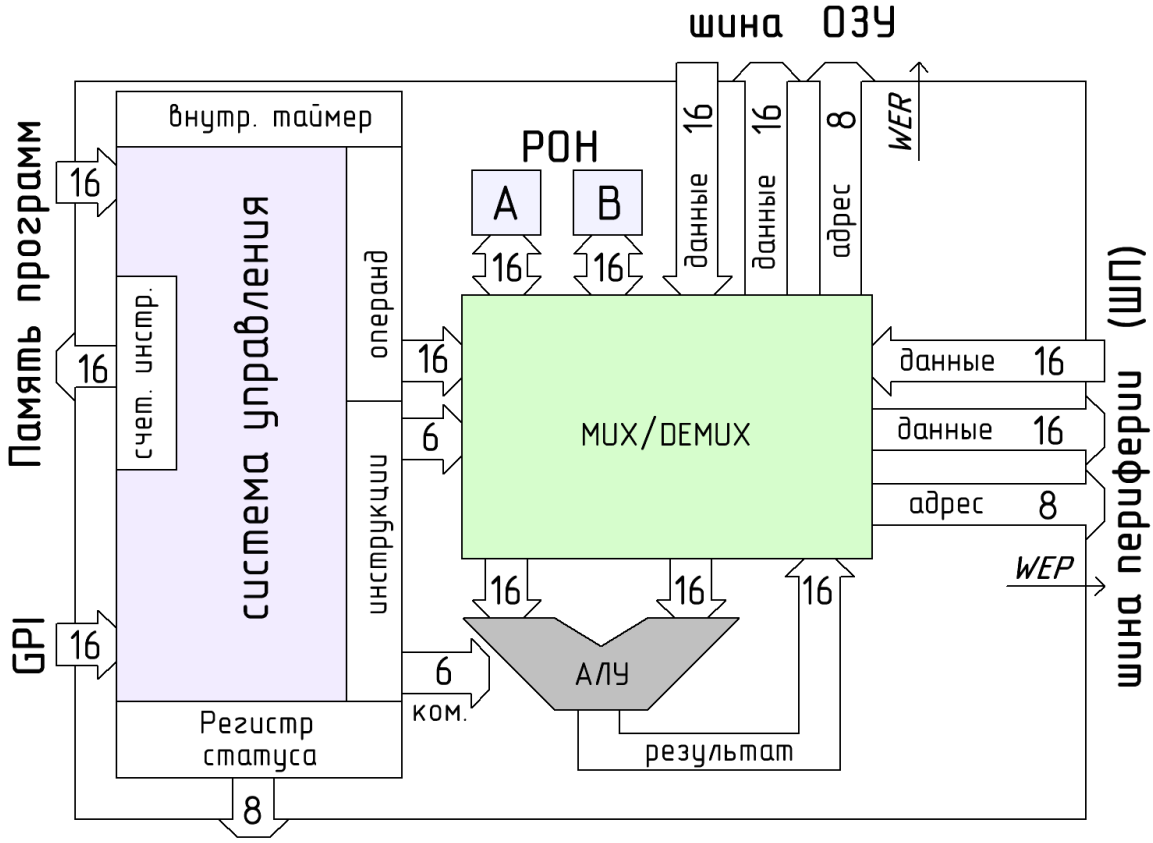


Рисунок 6.1 – Структура Simple processor модуль «SIP»

Процессор «SIP» является 16-битным процессором и имеет в своем составе: два регистра общего назначения (РОН); 16-битное АЛУ; систему управления и коммутатор (MUX/DEMUX). Все операции и вычисления происходят с РОН. Процессор имеет отдельную шину для инструкций, шину для подключения к оперативной памяти ОЗУ и отдельную шину для периферийных устройств PRH. Шина инструкций состоит из 16-битной шины адреса (выход счетчика инструкций) и также 16-битной шины самих инструкций. После выполнения обычных команд счетчик инструкций увеличивает значение на единицу. На вход ЦП поступает новая инструкция, и начинается её исполнение. Счетчик инструкций может менять свое значение «скачком», если выполняется команда «перехода». Инструкция для SIP состоит из двух байт: байт с командой и байт операнда. Структура одной инструкции (командного слова) приведена на рисунке 6.2.

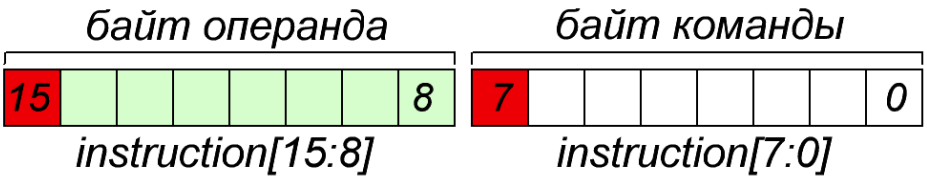


Рисунок 6.2 – Структура инструкции для модуля «SIP»

Основная работа любого процессора заключается в модификации содержимого памяти. Процессор «SIP» позволяет подключить 512 байт ОЗУ (256×16) через шину ОЗУ. Шина ОЗУ по внутреннему составу и принципу работы аналогична шине периферии (ШП или PRH). Логика данных шин следующая:

– Для записи необходимо установить 8 бит адреса и 16 бит данных, причем порядок установки не имеет значения. На следующий такт необходимо перевести триггер разрешения записи на время достаточное, чтобы уловить фронт CLK. В результате данные будут записаны по указанному адресу.

– Для чтения необходимо установить 8 бит адреса на шине, а на следующий такт считать данные со входа данной шины. Если адрес не будет изменяться, то на входе будут удерживаться соответствующие данные.

Листинг 6.1. Описание модуля «SIP»

```
module SIP // nano instruction control system. CPU (simple processor)
( CLK,reset,
ATR, DTR, DFR, WER,              // ОЗУ (RAM)
ATP, DTP, DFP, WEP,              // Периферия (PRH)
status,   GPI,                   // Регистр статуса и входной порт
instruction, inst_counter,     // Память программ (memory instruction)
FSM);                          // Главный конечный автомат ЦП
input wire CLK, reset;           // Тактовый сигнал и сигнал сброса
// cache+RAM:
output reg [7:0] ATR;          // Регистр адреса ОЗУ (Address To RAM)
output reg [15:0] DTR;         // Данные в ОЗУ (Data To RAM)
input wire [15:0] DFR;         // Данные из ОЗУ (Data From RAM)
output reg WER;                // Запись ОЗУ (Write Enable RAM)
```

```
// Peripherie:
output reg [7:0] ATP;          // Регистр адреса периферии (Address To PRH)
output reg [15:0] DTP;         // Данные в периферию (Data To PRH)
input wire [15:0] DFP;         // Данные из периферии (Data From PRH)
output reg WEP;                // Запись периферии (Write Enable PRH)
// External REG or input port:
input wire [15:0] GPI;         // Порт входа
output reg [7:0] status;       // Регистр статуса
// Memory instruction:
input wire [15:0] instruction;          // Инструкции из памяти программ
output reg [15:0] inst_counter;         // Адрес инструкции (счетчик команд)
reg [15:0] counter_buf;                 // * зачем это нужно?
// Maine FSM:
output reg [2:0] FSM;          // Главный конечный автомат процессора
reg [15:0] A;  // РОН (основной)
reg [15:0] B;  // РОН (вторичный)
// additional registers:
reg [15:0] delay;       // Дополнительный регистр для инструкции «ожидания»
initial begin
        ATR<=0;     DTR<=0;     WER<=0;     // Задание начального состояния
        ATP<=0;     DTP<=0;     WEP<=0;     // Задание начального состояния
        status<=0;  inst_counter<=0;        // Задание начального состояния
        FSM<=0;     delay<=0;               // Задание начального состояния
end // initial
always @ (posedge CLK or posedge reset) begin // Список чувствительности
        if (reset) begin        // когда фронт reset:
                ATR<=0;     DTR<=0;     WER<=0;
                ATP<=0;     DTP<=0;     WEP<=0;
                inst_counter<=0;    FSM<=0; delay<=0;
        end //reset
        else begin              // когда фронт CLK:
                case (FSM)      // главный конечный автомат ЦП:
                3'd0:   begin           // обнуление триггеров разрешения записи:
                                WER<=0;     WEP<=0;     FSM<=FSM+1'd1;
                        end             // FSM = 0 конец первой стадии.
                3'd1:   begin           // FSM=1 начало второй стадии:
                                inst_counter<=inst_counter+1'd1; // memory instruction
// декодирование и выполнение однотактовых инструкций:
                                case (instruction[7:0])         // анализ байта команд
                                8'd0:   FSM<=0;
// запись числа из инструкции в системный регистр адреса ОЗУ:
```

```
                8'd1:   begin
                            ATR[7:0]<=instruction[15:8];
                            FSM<=0;       end
// запись числа из инструкции в системный регистр адреса шины периферии:
                8'd2:   begin
                            ATP[7:0]<=instruction[15:8];
                            FSM<=0;       end
// запись в РОН А из ячейки ОЗУ с заранее установленным адресом:
                8'd3:   begin
                            A[15:0]<=DFR[15:0];
                            FSM<=0;       end
// запись в РОН А из шины периферии с заранее установленным адресом:
                8'd4:   begin
                            A[15:0]<=DFP[15:0];
                            FSM<=0;       end
// запись в РОН В из ячейки ОЗУ с заранее установленным адресом:
                8'd5:   begin
                            B[15:0]<=DFR[15:0];
                            FSM<=0;       end
// запись в РОН В из шины периферии с заранее установленным адресом:
                8'd6:   begin
                            B[15:0]<=DFP[15:0];
                            FSM<=0;       end
// запись в ОЗУ из блока РОН А по предустановленным адресам:
                8'd7:   begin
                            DTR[15:0]<=A[15:0];
                            WER<=1'd1;
                            FSM<=0;       end
// запись в ОЗУ из РОН А с предустановкой + инкремент адреса ОЗУ:
                8'd8:   begin
                            DTR[15:0]<=A[15:0];
                            WER<=1'd1;
                            FSM<=3'd3;   end // след. стадия
// запись в ОЗУ из Периферии по предустановленным адресам:
                8'd9:   begin
                            DTR[15:0]<=DFP[15:0];
                            WER<=1'd1;
                            FSM<=0;              end
// запись в ОЗУ из Периферии с предустановкой + инкремент адреса ОЗУ:
                8'd10:  begin
                            DTR[15:0]<=DFR[15:0];
```

```
                    WER<=1'd1;
                    FSM<=3'd3;   end // след. стадия
    // запись в Периферию из РОН A с предустановленным адресом:
                8'd11:  begin
                    DTP[15:0]<=A[15:0];
                    WEP<=1'd1;
                    FSM<=0;              end
    // запись в Периферию из РОН A с предустановкой + инкремент адреса PRH:
                8'd12:  begin
                    DTP[15:0]<=A[15:0];
                    WEP<=1'd1;
                    FSM<=3'd3;   end // след. стадия
    // запись в Периферию из РОН B с предустановленным адресом:
                8'd13:  begin
                    DTP[15:0]<=B[15:0];
                    WEP<=1'd1;
                    FSM<=0;              end
    // запись в Периферию из РОН B с предустановкой + инкремент адреса PRH:
                8'd14:  begin
                    DTP[15:0]<=B[15:0];
                    WEP<=1'd1;
                    FSM<=3'd3;   end // след. стадия
    // запись в Периферию из ОЗУ с предустановленными адресами:
                8'd15:  begin
                    DTP[15:0]<=DFR[15:0];
                    WEP<=1'd1;
                    FSM<=0;              end
    // запись в Периферию из ОЗУ с предустановленными адресами ++ инкремент адреса Периферии:
                8'd16:  begin
                    DTP[15:0]<=DFR[15:0];
                    WEP<=1'd1;
                    WER<=1'd1;
                    FSM<=2'd3;   end // след. стадия
    // запись в системный регистр адреса ОЗУ содержимого РОН A:
                8'd17:  begin
                    ATR[7:0]<=A[7:0];
                    FSM<=0;              end
    // запись в системный регистр адреса Периферии содержимого РОН A:
                8'd18:  begin
                        ATP[7:0]<=A[7:0];
```

```
                                        FSM<=0;       end
    // запись в РОН B содержимого из РОН A:
                                8'd19:  begin
                                        B[15:0]<=A[15:0];
                                        FSM<=0;       end
    // запись в РОН A содержимого из РОН B:
                                8'd20:  begin
                                        A[15:0]<=B[15:0];
                                        FSM<=0;       end
    // запись в младшие 8 бит РОН A числа из инструкции:
                                8'd21:  begin
                                        A[7:0]<=instruction[15:8];
                                        FSM<=0;
                                    end
    // запись в старшие 8 бит РОН A числа из инструкции:
                                8'd22:  begin
                                        A[15:8]<=instruction[15:8];
                                        FSM<=0;       end
    // обмен содержимого РОН A и РОН B:
                                8'd23:  begin
                                        A[15:0]<=B[15:0];
                                        B[15:0]<=A[15:0];
                                        FSM<=0;       end
    // безусловный переход на адрес инструкции из РОН A:
                                8'd24:  begin
                                        inst_counter[15:0]<=A[15:0];
                                        FSM<=3'd2;    end // след. стадия
    /* Инструкции с 25 по 27 – команды условного перехода, по которым, если условие выполняется, происходит переход на N инструкций вперед по ПП, иначе выполняется следующая по порядку инструкция. Величина прыжка N задается операндом из инструкции*/
    8'd25:  begin
                if (A>B) begin
                    inst_counter[15:0]<=inst_counter[15:0]+instruction[15:8];
                    FSM<=3'd2;    end // след. стадия
                else begin
                    FSM<=0;
                end
            end
    8'd26:  begin
                if (A==B) begin
```

```
                inst_counter[15:0]<=inst_counter[15:0]+instruction[15:8];
                FSM<=3'd2;   end // след. стадия
            else begin
                FSM<=0;
            end
        end
    8'd27:  begin
            if (A==0) begin
                inst_counter[15:0]<=inst_counter[15:0]+instruction[15:8];
                FSM<=3'd2;   end // след. стадия
            else begin
                FSM<=0;
            end
        end
    // инкремент содержимого РОН A:
                8'd28:  begin
                        A[15:0]<=A[15:0]+1'd1;
                        FSM<=0;       end
    // декремент содержимого РОН A:
                8'd29:  begin
                        A[15:0]<=A[15:0]-1'd1;
                        FSM<=0;       end
    /* В инструкциях с 30 по 35 выполняются математические и логические операции между РОН A и РОН B, результат операции всегда записывается в РОН A*/
    // логическое побитовое «И»:
                8'd30:  begin
                        A[15:0]<=A[15:0]&B[15:0];
                        FSM<=0;       end
    // логическое побитовое «ИЛИ»:
                8'd31:  begin
                        A[15:0]<=A[15:0]|B[15:0];
                        FSM<=0;       end
    // логическое побитовое «исключающее или»:
                8'd32:  begin
                        A[15:0]<=A[15:0]^B[15:0];
                        FSM<=0;       end
    // операция сложения РОН A и РОН B:
                8'd33:  begin
                        A[15:0]<=A[15:0]+B[15:0];
                        FSM<=0;       end
    // операция вычитания из РОН A содержимого РОН B:
```

```
                    8'd34:  begin
                            A[15:0]<=A[15:0]-B[15:0];
                            FSM<=0;       end
// операция вычитания из РОН В содержимого РОН А:
                    8'd35:  begin
                            A[15:0]<=B[15:0]-A[15:0];
                            FSM<=0;       end
// логическое побитовое «НЕ» к содержимому РОН А:
                    8'd36:  begin
                            A[15:0]<=~A[15:0];
                            FSM<=0;       end
// операция изменения порядка значения бит в РОН А:
                    8'd37:  begin
                    A[15]<=A[0]; A[14]<=A[1]; A[13]<=A[2]; A[12]<=A[3];
                    A[11]<=A[4]; A[10]<=A[5]; A[9]<=A[6]; A[8]<=A[7];
                    A[7]<=A[8]; A[6]<=A[9]; A[5]<=A[10]; A[4]<=A[11];
                    A[3]<=A[12]; A[2]<=A[13]; A[1]<=A[14]; A[0]<=A[15];
                            FSM<=0;       end
// сдвиг содержимого РОН А влево на значение из инструкции:
                    8'd38:  begin
                            A[15:0]<=A[15:0]<<instruction[15:8];
                            FSM<=0;       end
// сдвиг содержимого РОН А вправо на значение из инструкции:
                    8'd39:  begin
                            A[15:0]<=A[15:0]>>instruction[15:8];
                            FSM<=0;       end
// запись в дополнительный регистр статуса содержимого РОН А:
                    8'd40:  begin
                            status[7:0]<=A[7:0]; FSM<=0;                end
// запись в РОН А данных с порта ввода:
                    8'd41:  begin
                            A[15:0]<=GPI[15:0]; FSM<=0;       end
// переход к выполнению операции «ожидания»:
                    8'd42:          FSM<=3'd4;
        default: FSM<=FSM+1'd1;
        endcase  //     case (instruction[7:0]) завершение анализа байта команд
        end // FSM =1 конец второй стадии
        3'd2:   FSM<=0; // пропуск такта для установки данных на входе
        3'd3:   begin  // выполнение двух тактовых инструкций:
                    FSM<=0;
                    if (WEP) begin
```

```
                    ATR[7:0]<=ATR[7:0]+1'd1; // инкремент адреса ОЗУ
                    WEP<=0;       end
                if (WER) begin
                    ATP[7:0]<=ATP[7:0]+1'd1;   // инкремент адреса PRH
                    WER<=0;       end
            end // FSM = 3 конец четвертой стадии
        3'd4:   begin // выполнение операции «ожидания»:
                if (delay==B)begin
                    delay<=0;
                    FSM<=0;       end
                else begin
                    delay<=delay+1'd1;
                end
            end // FSM = 4 конец пятой стадии

        default:;
        endcase // FSM
        end // CLK
end // always
endmodule // core SIP
```

После компиляции для целевой ИМС типа Cyclone III модуль «SIP» занимает примерно 700 логических элементов и может работать на частоте до 150МГц. При учете того, что большинство операций выполняются за 2 такта, теоретическая производительность около 75 MIPS. После подключения к «SIP» ОЗУ и организации выводов получается схема, показанная на рисунке 6.3.

Для хранения программы можно использовать как внешнюю, так и внутреннюю память. Для учебных целей и небольших программ отлично подходит ROM память, которая будет записываться при конфигурации ПЛИС из конфигурационной flash-памяти.

Для примера рассмотрим программу, позволяющую «мигать» светодиодом на частоте 1КГц, подключенным к младшему биту регистра «status». Программа для наглядности записана в таблице 6.1.

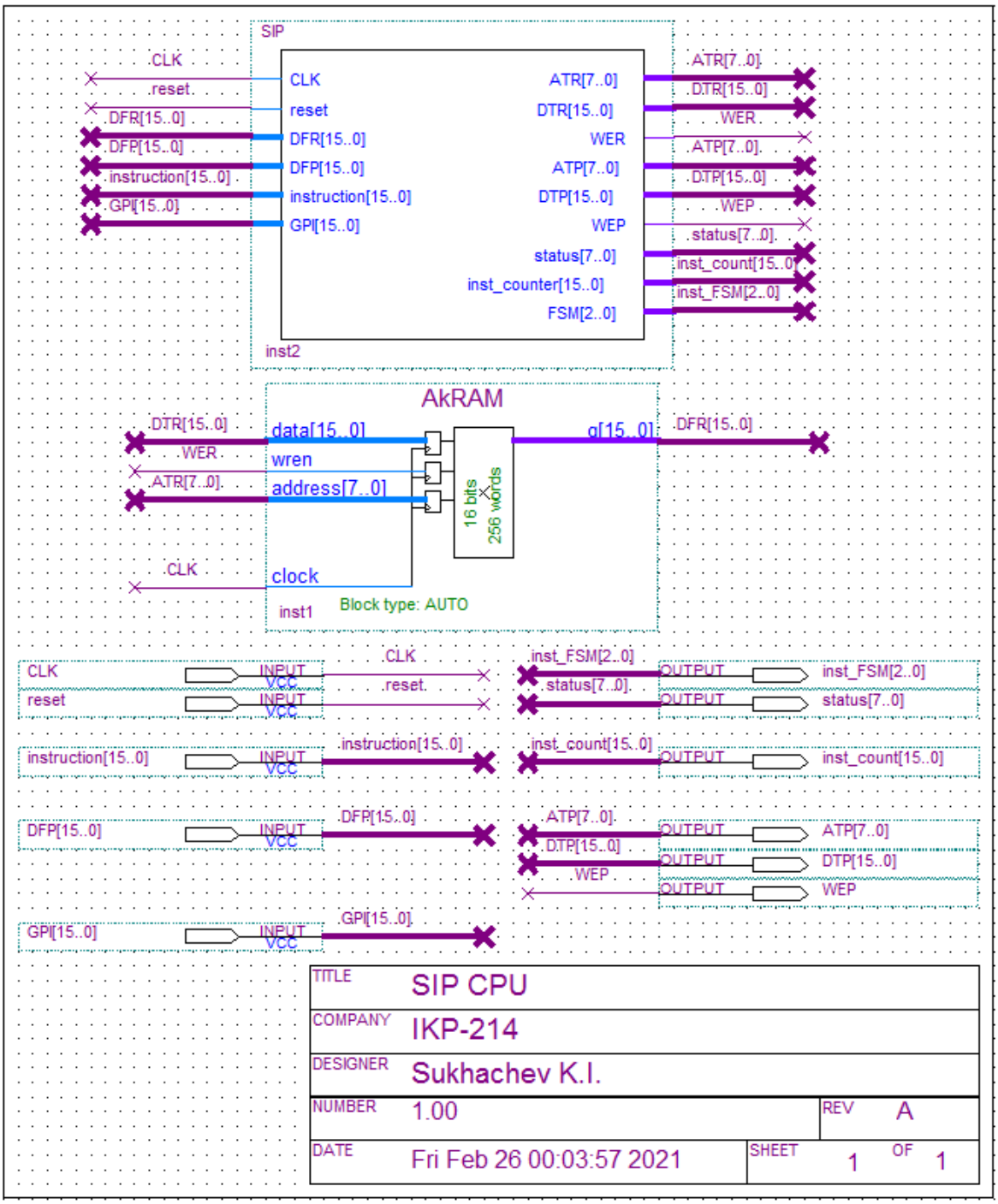


Рисунок 6.3 – Схема подключения ядра процессора «SIP»

Программа, приведённая в таблице 6.1, может быть переведена в формат Hex или файл инициализации памяти MIF, который будет источником для загрузки значений в ROM. Создать MIF можно как сторонними средствами, так и средствами среды Quartus. Для этого необходимо создать новый файл, как показано

на рисунке 6.4, выбрать «Memory Initialization File», задать разрядность одного слова (ячейки памяти) и количество ячеек. Данные параметры должны совпадать с блоком ROM памяти, который необходимо будет загрузить при конфигурации. После можно будет заполнив таблицу (рисунок 6.5), создать MIF файл. Данный файл допускается создавать и не через среду Quartus, а в автоматическом режиме, например, используя компилятор. Формат MIF файла, если его открыть «блокнотом», также представлен на рисунке 6.5.

Таблица 6.1 – Пример программы для процессора «SIP»

№	Ассемблер	instr[];[]	Описание действия
0	AL<=8	8;21	РОН А = 8
1	RAM<=A	0;7	ячейка №0 ОЗУ = 8
2	AL<=1	1;21	РОН А = 1
3	B<=A	0;19	РОН В = 1
4	AH<=195	195;22	РОН А = 49 920
5	AL<=80	80;21	РОН А = 50 000
6	ATR<=1	1;1	регистр адреса ОЗУ = 1
7	RAM<=A	0;7	ячейка №1 ОЗУ = 50 000
8	ATR<=2	2;1	регистр адреса ОЗУ = 2
9	A<=RAM	0;3	РОН А = содержимому ячейки №2 ОЗУ
10	A++	0;28	увеличение РОН А на 1
11	RAM<=A	0;7	в ячейку №2 ОЗУ сохраняется значение РОН А
12	A<=A&B	0;30	A<=…xxxx&…0001
13	status<=A	0;40	запись в регистр значение РОН А
14	ATR<=1	1;1	регистр адреса ОЗУ = 1
15	A<=RAM	0;3	РОН А = 50 000
16	delayA	0;42	ожидание 50 000
17	ATR<=0	0;1	регистр адреса ОЗУ = 0
18	A<=RAM	0;3	РОН А = 8
19	jampA	0;24	прыжок на строчку 8

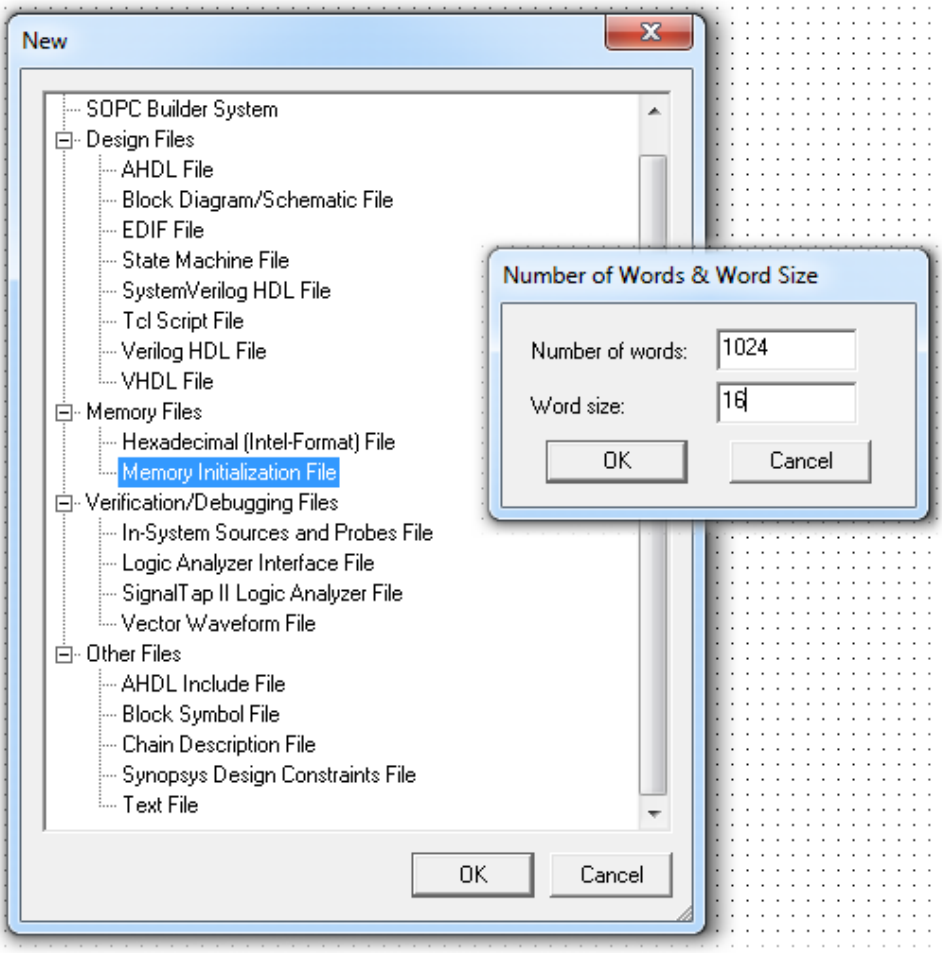


Рисунок 6.4 – Создание файла инициализации памяти: MIF

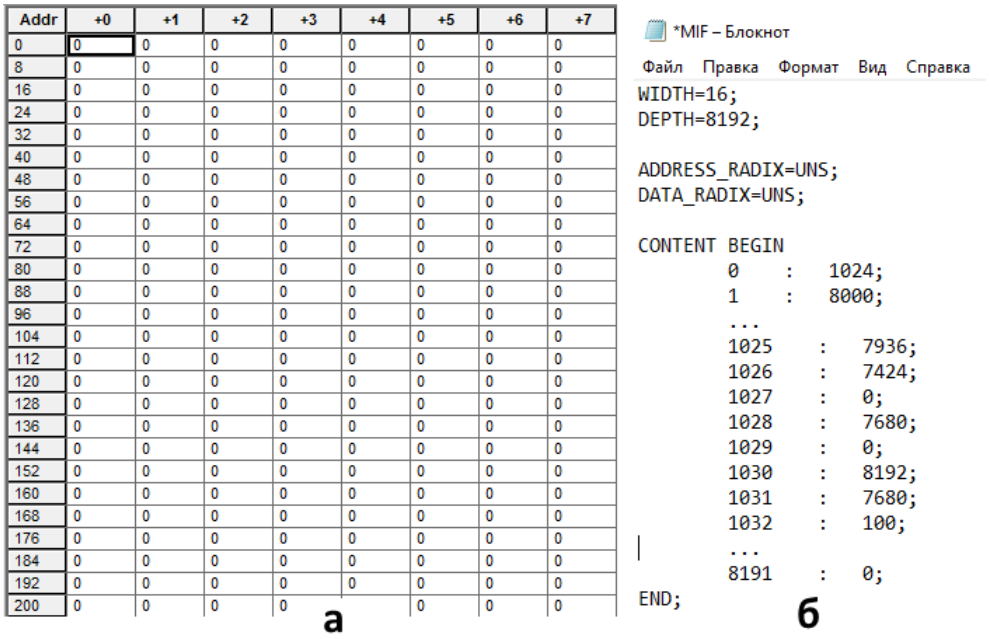


Рисунок 6.5 – Создание файла инициализации памяти,
а – заполнение ячеек в Quartus; б – пример формата MIF файла

После сохранения MIF файла необходимо создать модуль ROM памяти. Например, это можно сделать через вкладку «Tools», далее перейти в «MegaWizard Plug-In Manager» и из меню доступных элементов выбрать «Memory Compiler», «ROM: 1-PORT», как показано на рисунке 6.6. После выполнить настройки создаваемого блока памяти, указать количество ячеек и их разрядность (такие же параметры должны быть и у MIF файла), убрать лишнее тактирование и указать путь до файла инициализации. После описанных действий в корневом каталоге проекта появится графическое изображение блока ROM памяти, который можно использовать в редакторе схемы и подключить его к модулю ядра процессора.

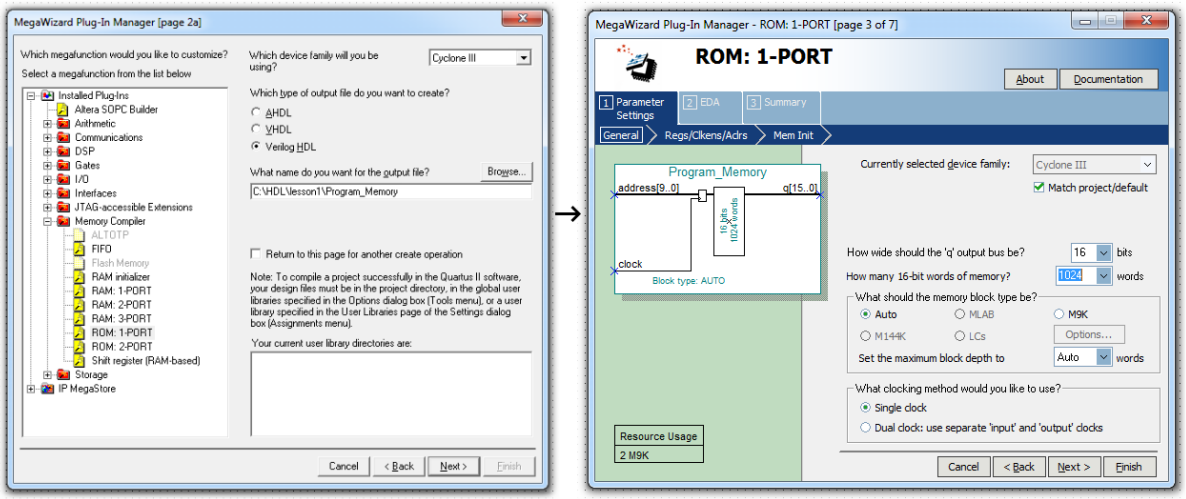


Рисунок 6.6 – Создание модуля ROM в «MegaWizard Plug-In Manager»

Если теперь скомпилировать проект и загрузить его в ПЛИС, то сразу после конфигурации ИМС начнется выполнение программы.

6.2 Пример разработки модулей периферии учебного софт-процессора

Для получения эффективной системы управления на базе ПЛИС с синтезированным внутри нее модулем ЦП последний должен быть окружен функционально законченными модулями с

определенными задачами, например: таймеры-счетчики, модули стандартных и специальных интерфейсов, сетевые контроллеры и контроллеры портов ввода-вывода и т.д.

В данном разделе рассмотрим пример создания модуля, который можно подключить напрямую к шине периферии представленного ранее процессорного ядра. В качестве учебного примера рассмотрим упрощенные модули передатчика и приемника по протоколу UART, позволяющие работать на низких скоростях (до 115 200) и не содержащие мажоритирования и буферизации. Описание на Verilog_HDL модулей передатчика «UART_TR» и приемника «UART_RS» представлено на листингах 6.2 и 6.3 соответственно.

Листинг 6.2. Описание модуля «UART_TR»

```
module UART_TX
(CLK, reset,
ATP, DTP, WEP, // Шина периферии (PRH) от ЦП «SIP»
busy, ready,
TXout);
parameter add_set = 8'd20;  // адрес настройки устройства на ШП
parameter add_out = 8'd24; // адрес устройства на ШП для передачи
input wire CLK, reset;
input wire [7:0] ATP; input wire [15:0] DTP; input wire WEP; // ШП
output reg ready, busy, TXout;// Флаги состояния и выход URTR TX
reg [13:0] URT;                    // Устанавливаемый параметр скорости
reg [7:0] bufdata;                 // Буфер данных
reg [13:0] maincounter;            // Главный счетчик
reg [3:0] bitcounter;          // Счетчик переданных бит
reg [3:0] max;                 // Параметр (кол-во бит в посылке)
reg sum, even;                     // Бит четности
always @(posedge CLK or posedge reset) begin
if (reset) begin
        TXout<=1'd1; bitcounter<=0; maincounter<=0;
        busy<=0;ready<=0; max<=0; sum<=0; even<=0;
end // reset
else begin
        if (WEP) begin          // протокол связи модуля с ШП
                case (ATP)     //  проверка адреса на ШП
                add_set:        begin  // настройка:
```

```
                                    URT[13:0]<=DTP[13:0];
                                    if (DTP[15]) begin
                                            max<=4'd10;
                                            even<=DTP[15];      end
                                    else begin
                                            max<=4'd9;          end
                                    end
                        add_out:    begin  // адрес для передачи
                                    bufdata[7:0]<=DTP[7:0];
                                    busy<=1'd1;                 end
                        default:;
                        endcase
                end // WEP
        if (busy|ready) begin
                        case (maincounter)
                        14'd0:  begin
                                maincounter<=maincounter+1'd1;
                                if (bitcounter==0) begin
                                        TXout<=0;     end
                                else begin
                                        if (bitcounter==max) begin
                                                TXout<=1'd1;
                                                busy<=0;
                                                ready<=1'd1;  end
                                        else begin
                                                if (max==4'd10) begin
                                                        if (bitcounter==4'd9) begin
                                                                TXout<=even? sum: !sum;
                                                                sum<=0;
                                                        end
                                                        else begin
                                                        TXout<=bufdata[bitcounter-1'd1];
                                                        sum<=sum+bufdata[bitcounter-
1'd1];

                                                        end
                                                end
                                                else begin
                                                TXout<=bufdata[bitcounter-1'd1];
                                                sum<=sum+bufdata[bitcounter-1'd1];
                                        end
                                end
                        end // start bit +
```

```
                end
                URT:  begin
                            bitcounter<=bitcounter+1'd1;
                            maincounter<=0;
                            if (bitcounter==max) begin
                                    bitcounter<=0;
                                    maincounter<=0;
                                    ready<=0;
                            end
                        end
                default:        maincounter<=maincounter+1'd1;
                        endcase
                end // busy
        end // CLK
        end // always
        endmodule // UART_TX
```

Модуль передатчика подключается напрямую к ШП процессора «SIP». Для работы с модулем необходимо настроить его, для этого обратиться по адресу настроек (add_set = 8'd20 в примере выше) и передать 16-битный пакет настроек, где 15 бит – это флаг бита четности, 14 бит – свободный и с 13-го по нулевой – это параметры скорости. Для передачи пакета по UART, ЦП необходимо по ШП отправить посылку с адресом передатчика (add_out = 8'd24). Передатчик захватит данные с параллельной ШД и начнет отправку по UART, выставит флаг «busy». После отправки всего пакета передатчик уберет флаг «busy» и на время даст сигнал «ready», который может быть использован как источник прерываний. Стоит заметить, что учебный процессор «SIP» не содержит аппаратного способа обрабатывать прерывания.

Модулю приемника также в начале работы необходимо передать настройки, по структуре пакет настроек аналогичен пакету для модуля передатчика. При получении сообщения по последовательному каналу приемник выставляет флаг «full», а если канал UART настроен с битом четности и при приеме обнаружена ошибка, то приемник дополнительно выставит флаг «error».

Листинг 6.3. Описание модуля «UART_RS»

```
        module UART_RX
        (CLK,reset,
ATP, DTP, WEP,      // Шина периферии (PRH) от ЦП «SIP»
        RXin,
        Dataout, full, error);
        parameter addres = 16'd20;  // адрес настройки устройства на ШП
        input wire CLK, reset, WEP;          // глобальные сигналы
        input wire [15:0] ATP; input wire [15:0] DTP;      //ШП
        input wire RXin;                     // последовательный вход приемника
        output reg [7:0] Dataout;     // параллельные данные с приемника
output reg full, error;                 // Флаги состояния и выход URTR TX
        reg busy, sum, even;
        reg [13:0] URT;                      // Устанавливаемый параметр скорости
        reg [13:0] maincounter;
        reg [3:0] bitcounter;
        reg bhc;
        wire [3:0] max; assign max[3:0] = bhc? 4'd10:4'd9;
        wire [12:0] t1; assign t1[12:0] = URT[13:1];
        always @(posedge CLK or posedge reset) begin
        if (reset) begin
                Dataout<=0; full<=0; error<=0;
                URT<=0;  even<=0; bhc<=0;
                maincounter<=0; bitcounter<=0;
                busy<=0; sum<=0;
        end // reset
        else begin
                if (WEP) begin
                        if (ATP==addres)      begin
                                URT[13:0]<=DTP[13:0];
                                if (DTP[15]) begin
                                        bhc<=1'd1;
                                        even<=DTP[15];                end
                                else begin
                                        bhc<=0;
                                end
                        end     //ATP=addres
                end     // WEP
                if (busy==0) begin
                        if (!RXin) begin busy<=1'd1; end // start
                end
```

```
        else begin
                case (maincounter)
                t1:                     begin
                                        maincounter<=maincounter+1'd1;
                                                if (bitcounter==0) begin

bitcounter<=bitcounter+1'd1;
                                                        maincounter<=0;
                                                        error<=0;
                                                        full<=0;        end
                                        end     // maincounter=t1
                URT:            begin
                                        maincounter<=maincounter+1'd1;
                                        if (bhc) begin
                                                if (bitcounter==4'd9) begin
                                                if ((even? sum: !sum)^RXin) begin
                                                        error<=1'd1;            end
                                                end     // bitcounter==4'd9
                                        end     // bhc
                                        if (bitcounter==max) begin
                                                        if (RXin) begin
                                                                full<=1'd1;     end
                                        end
                                        else begin
                                                if (bitcounter!=4'd9) begin
                                                Dataout[bitcounter-1'd1]<=RXin;
                                                        sum<=sum+RXin;
end
                                                bitcounter<=bitcounter+1'd1;
                                                maincounter<=0;
                                        end     // bitcounter!=max
                                end // maincounter=URT
        URT+1'd1:       begin
                                        if (bitcounter==max) begin
                                                maincounter<=0;
                                                bitcounter<=0;
                                                busy<=0;
                                                sum<=0;                 end
                                end     // maincounter=URT+1
        default:        maincounter<=maincounter+1'd1;
        endcase
end // reception
```

```
        end // CLK
    end // always
    endmodule // UART_RX
```

Согласно отчету о компиляции, представленные модули для реализации UART интерфейса, занимают менее 150 логических элементов каждый. Временные диаграммы, иллюстрирующие работу пары, передатчик-приемник (UART_TX и UART_RX) представлены на рисунке 6.7.

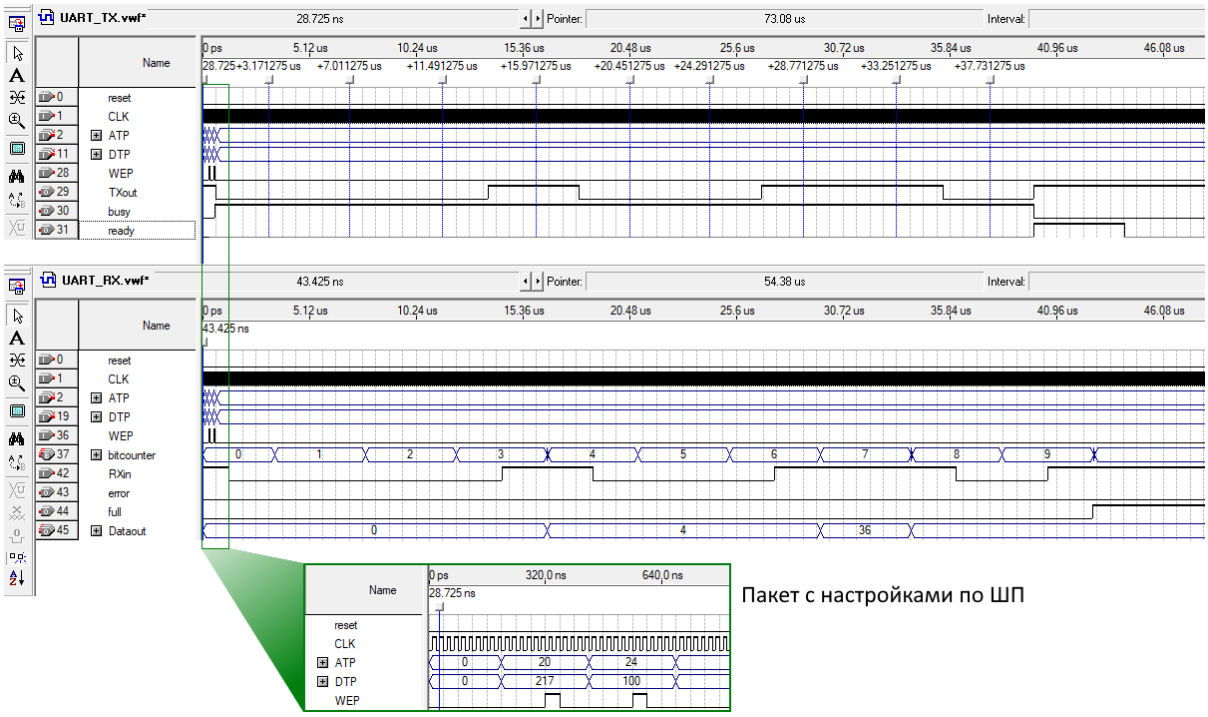


Рисунок 6.7 – Временные диаграммы, полученные при моделировании файлов «UART_TX» и «UART_RX»

Рассмотрим еще один модуль периферии ЦП, который может быть подключен по ШП и использован в реальных проектах на ПЛИС, а также пригодится для выполнения практических заданий. Модуль «PWM_controller», описание которого представлено в листинге 6.4, может быть применен как для управления схемами импульсных преобразователей, так и просто для регулирования мощности посредством ШИМ модуляции.

Листинг 6.4. Описание модуля «PWM controller»

```
module PWM_controller // for NMR, ATOM, QARK or SIP
(CLK, reset,
ATP, DTP, WEP,
OUTA,OUTB,Dead_time);
parameter add_set1 = 16'd100;
parameter add_set2 = 16'd101;
parameter add_out = 16'd102;
input wire CLK, reset, WEP;
input wire [15:0] ATP; input wire [15:0] DTP;
output reg OUTA, OUTB, Dead_time;
reg mode;
reg [9:0] dt;
reg [15:0] value;
reg [15:0] max;
reg [15:0] PWM_counter;
reg [9:0] dt_counter;
reg ff;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                value<=0;       dt<=0; mode<=0; max<=0;   //settings
                PWM_counter<=0; dt_counter<=0;                  //counters
                ff<=0; OUTA<=0; OUTB<=0; Dead_time<=0;      //outs
        end // reset
        else begin
                if (WEP) begin
                        case (ATP)
                        add_set1:       begin
                                                dt [9:0]<=DTP[9:0];
                                                mode<=DTP[15];
end
                        add_set2:       begin
                                                max [15:0]<=DTP[15:0];      end
                        add_out:        value[15:0]<=DTP[15:0];
                        default:;
                        endcase
                end // WEP
                if (value==0) begin
                        PWM_counter<=0; OUTA<=0; OUTB<=0;           end
                else begin
                        case (PWM_counter)
```

```
                0:      begin
                            Dead_time<=0;
                            dt_counter<=0;
                            PWM_counter<=PWM_counter+1'd1;
                            ff<=ff+1'd1;
                        end
                value:  begin
                            OUTA<=0;
                            OUTB<=0;
                            PWM_counter<=PWM_coun-
ter+1'd1;
                        end
                max:    begin
                            Dead_time<=1'd1;
                            if (dt_counter!=dt) begin
                                dt_counter<=dt_coun-
ter+1'd1;

                                OUTA<=0;
                                OUTB<=0;            end
                            else begin
                                PWM_counter<=0;
                                dt_counter<=0;
                                if (mode) begin
                                    if(ff) begin
                                    OUTA<=1'd1;
                                    OUTB<=0;
        end

                                    else begin
                                    OUTA<=0;
                                    OUTB<=1'd1;end
                                end     // парафазный ре-
жим

                                else begin
                                    OUTA<=1'd1;
                                    OUTB<=0;
                                end // одиночный режим
                            end
                        end
                default: PWM_counter<=PWM_counter+1'd1;
                endcase
            end // worck!
        end // CLK
```

```
        end     // always
endmodule // PWM_controller
```

Результаты моделирования модуля «PWM_controller» представлены на рисунке 6.8. Для задания параметров мёртвого времени периода и режима формирования ШИМ необходимо настроить модуль по адресу настроек, а для задания заполнения передать параметр по адресу «add_out = 16'd102».

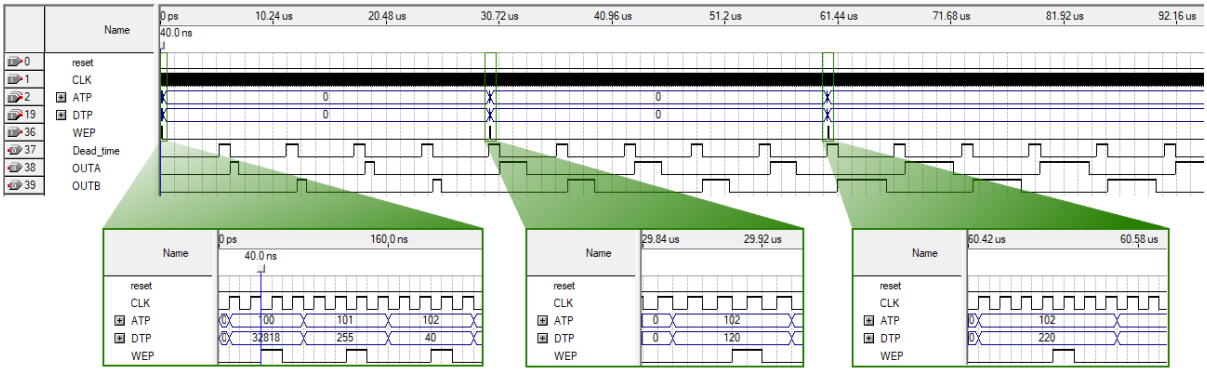


Рисунок 6.8 – Временные диаграммы, полученные при моделировании файлов «PWM_controller»

Множество устройств требуют подключения внешних органов управления для реализации интерфейса между оператором и блоком электроники. Это могут быть различные элементы и системы, например, кнопки, переключатели, матричные клавиатуры и другие механические или сенсорные органы управления, но всем им в той или иной степени свойственно явление «дребезга». Для борьбы с ним, кроме аппаратных решений, можно применить метод «программного» подавления дребезга. Одним из вариантов может является модуль «контроллера кнопок» с функцией подавления дребезга, представленный на листинге 6.5.

Листинг 6.5. Описание модуля «button controller»

```
module button_controller
(CLK, reset, LCLK, button,   // LCLK – сниженная частота тактирования
out_level, out_pulse);       // внутренние сигналы о нажатии
```

```
parameter delay=20;          // длительность задержки (время установки)
parameter long_press=100;          // длительность «долгого нажатия»
parameter speed=5;          // минимальная длительность между импульсами
input wire CLK, reset, LCLK, button;
output reg out_level, out_pulse;
reg [9:0] counter;
reg [9:0] max;
reg run, long;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                run<=0; counter<=0; max<=0; run<=0; long<=0;
                out_level<=0; out_pulse<=0;
        end
        else begin
                if (button) begin
                        if (!run) begin
                                run<=1'd1;
                                max<=delay;
                        end
                end
                else begin
                        run<=0; counter<=0;  max<=0; run<=0; long<=0;
                        out_level<=0; out_pulse<=0;
                end
                if (run) begin
                        case (counter)
                        max:    begin
                                        out_level<=1'd1;
                                        out_pulse<=1'd1;
                                        counter<=counter+1'd1;
                                end
                        max+1:          begin
                                        out_pulse<=0;
                                        if (long) begin
                                                if (max==speed) begin
                                                        max<=max;
                                                end
                                                else begin
                                                        max<=max-1'd1;
                                                end
                                                counter<=0;
                                        end
                                        else begin
```

```
                                        counter<=counter+1'd1;
                                    end
                                end
                    long_press:     begin
                                        long<=1'd1;
                                        counter<=0;
                                end
                    default: counter<=counter+1'd1;
                    endcase
                end // run
            end     // CLK
        end // always
        endmodule // button_controller
```

На рисунке 6.9 представлены диаграммы работы модуля «button_controller». Последние три сигнала являются внутренними и выведены для наглядности принципа работы.

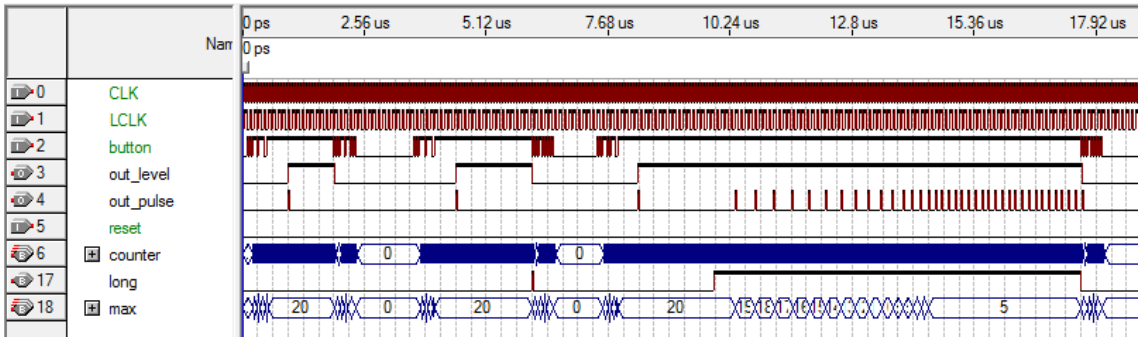


Рисунок 6.9 – Временные диаграммы, полученные при моделировании файла модуля «button_controller»

Модуль, описанный в листинге 6.5, имеет несколько параметров, позволяющих настроить его под различные органы управления с разной длительностью нестабильного состояния. Стоит заметить, что модуль требует дополнительного тактового сигнала, обычно его частота находится в диапазоне: 10–100Гц. Это сделано для того, чтобы снизить «затраты» логических элементов, занятых в проекте в тех случаях, когда таких модулей предполагается использовать несколько, т.е. потребуется один общий счетчик-делитель, и нет необходимости реализовывать его в каждом из таких

«button_controller» модулей. Кроме того, в модуль встроен детектор длительного нажатия, который удобно использовать для реализации режима быстрого изменения регулируемого параметра. При удержании кнопки происходит непрерывная генерация выходных импульсов, как было бы, если кнопка «нажималась» в подряд с высокой скоростью. Причем с увеличением времени удержания кнопки частота следования выходных импульсов увеличивается до максимальной, что хорошо видно из рисунка 6.9.

Часто вместо привычных тактовых кнопок или переменных поворотных резисторов в современной электронике применяют инкрементные энкодеры (рисунок 6.10).

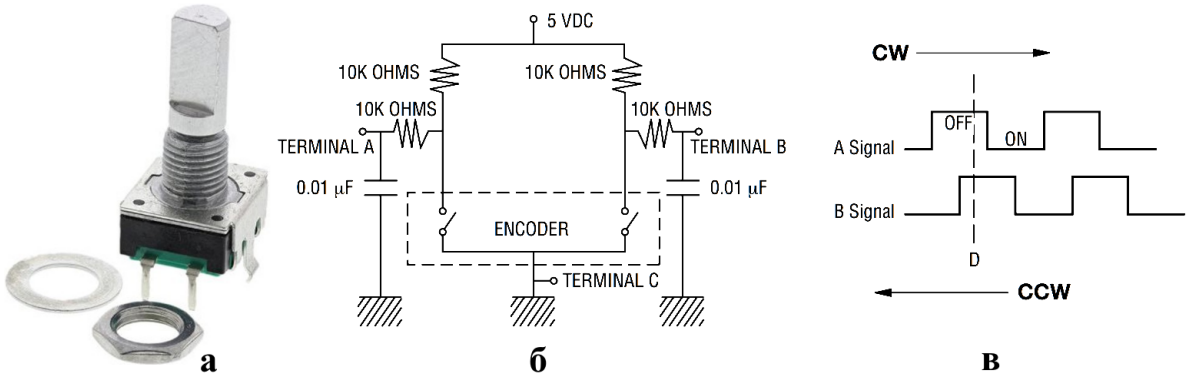


Рисунок 6.10 – Инкрементный энкодер Bourns PEC11R-4020F

Описание модуля, опрашивающего энкодер, приведено на листинге 6.6.

Листинг 6.6. Описание модуля «encoder controller»

```
module encoder_controller
(CLK,reset,
A, B,          // from encoder
pulse_A,       // left rotation
pulse_B);      // right rotation
input wire CLK, reset, A, B;
output reg pulse_A, pulse_B;
reg b_A, b_B, bb_A, bb_B; // bufers
reg [1:0] count_A; reg [1:0] count_B;
```

```
reg block_A,block_B;
always @(posedge B) begin
        if (A==1'd1) b_A<=1'd1; else b_A<=0;
end // always B
always @(posedge A) begin
        if (B==1'd1) b_B<=1'd1; else b_B<=0;
end // always A
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                pulse_A<=0; pulse_B<=0; bb_A<=0;bb_B<=0;
                count_A<=0; block_A<=0; count_B<=0; block_B<=0;
        end // reset
        else begin
                if (bb_B==0) begin
                        if (b_B==0) begin
                                if (b_A==1'd1)          bb_A<=1'd1;
                                if (b_A==0)    bb_A<=0;
                        end
                end
                if (bb_A==0) begin
                        if (b_A==0) begin
                                if (b_B==1'd1)          bb_B<=1'd1;
                                if (b_B==0)    bb_B<=0;
                        end
                end
                        if (bb_A==1'd1) begin
                                if (block_A==0) begin
                                        count_A<=count_A+1'd1;
                                        case (count_A)
                                        2'd0: pulse_A<=1'd1;
                                        2'd2:   begin
                                                        block_A<=1'd1;  pulse_A<=0;
                                                end
                                        default:;       endcase
                                end
                        end // bb_A==1'd1
                        else    block_A<=0;
                        if (bb_B==1'd1) begin
                                if (block_B==0) begin
                                        count_B<=count_B+1'd1;
                                        case (count_B)
```

```
                                    2'd0: pulse_B<=1'd1;
                                    2'd2:   begin
                                                block_B<=1'd1;  pulse_B<=0;
                                            end
                                    default:;       endcase
                                end
                            end // bb_B==1'd1
                            else    block_B<=0;
                    end // CLK
            end // always
            endmodule // encoder_controller
```

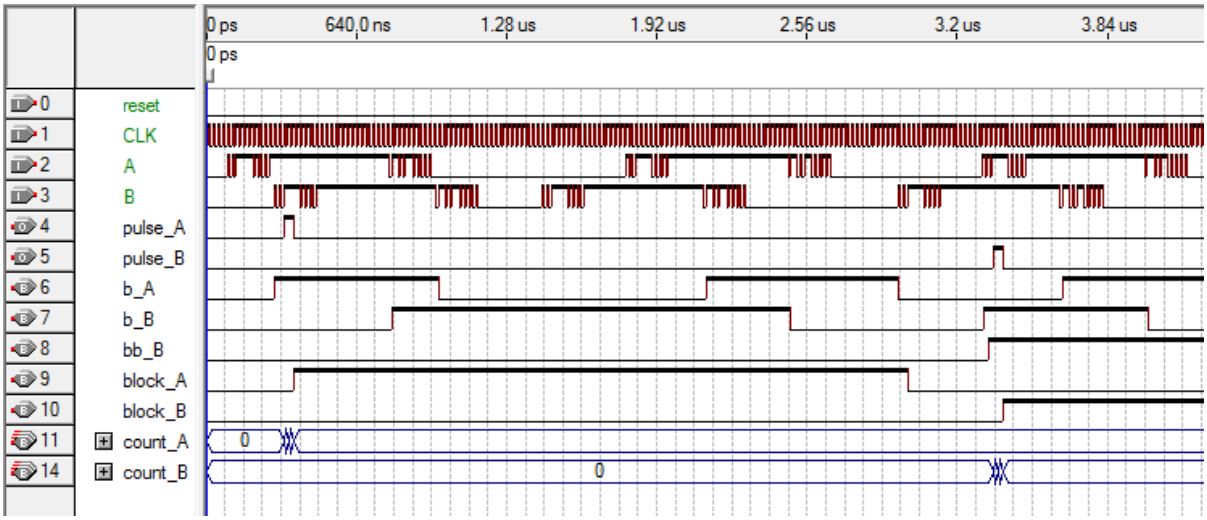


Рисунок 6.11 – Временные диаграммы, полученные при моделировании модуля «encoder controller»

Энкодер при вращении шкива генерирует последовательность парных импульсов со сдвигом фаз, зависящим от направления вращения. Энкодер обладает большей надёжностью и долговечностью, чем переменные резисторы, поэтому, где это возможно, в качестве органа управления желательно использовать именно их. Однако, оставаясь устройством с механическим контактом, энкодерам так же, как и тактовым кнопкам, свойственно явление «дребезга». Модуль, описанный в листинге 6.7, имеет средства борьбы с ним. При изменении направления вращения первый импульс пропускается для предотвращения случайного

воздействия при точной установке контролируемого параметра. Принципы работы модуля «encoder controller» и самого энкодера хорошо видны из рисунков 6.10, где представлены временные диаграммы модуля и рисунка 6.11 из технического описания на энкодер фирмы Bourns, где показан внешний вид и типовая схема подключения.

Почти любое электронное устройство нуждается в возможности преобразования аналогового сигнала в цифровой «оцифровке» с различной скоростью и точностью. Для этого используются микросхемы АЦП. Они различаются как по принципу преобразования, так и по интерфейсу связи с управляющим устройством. На листинге 6.8 представлен модуль, позволяющий подключать к ПЛИС АЦП AD7851 фирмы Analog Devices.

Листинг 6.7. Описание модуля «ADC AD7851»

```
//5023HB015 аналог  AD7851ARSZ. Режим ведомого SPI – SM1&SM2 - VCC
module ADC5023NV015
(CLK, reset,
dataout, ready, start,   // флаги и шина данных
CLKIN, NCONVST,
NSYNC, SCLK,
BUSY, DOUT);
parameter CLKIN_FIN=3; // CLKIN  = CLK / (2* CLKIN_FIN) [MHz]

input wire CLK, reset, start;
output reg [15:0] dataout;
output reg ready;
// signal from ADC:
output reg CLKIN;
output wire NCONVST; assign NCONVST=!CONVST;
reg CONVST;
input wire BUSY;
output wire  NSYNC; assign  NSYNC=!SYNC;
reg SYNC;
// SPI:
input wire DOUT;
```

```
output reg SCLK;
reg [7:0] counter_CLKIN;
reg [3:0] bitcounter;
reg [3:0] FSM;
always@(posedge CLK or posedge reset) begin
        if (reset) begin
                counter_CLKIN<=0; bitcounter<=0;
                FSM<=0; SCLK<=0; SYNC<=0;
                CLKIN<=0; CONVST<=0;
                dataout<=0; ready<=0;
        end // reset
        else begin
                case (FSM)
                4'd0:   begin
                                if (start&!BUSY) begin
                                        FSM<=FSM+1'd1; CONVST<=1'd1;
                                        SCLK<=1'd1;
                                end
                        end
                4'd1:   FSM<=FSM+1'd1;
                4'd2:   FSM<=FSM+1'd1;
                4'd3:   begin
                                if (BUSY) begin
                                        FSM<=FSM+1'd1;
                                        CONVST<=0;
                                end
                        end
                4'd4:   begin
                                if (!BUSY) begin
                                        SYNC<=1'd1; FSM<=FSM+1'd1;
                                        bitcounter[3:0]<=4'd15;        dataout<=0;
                                end
                        end
                4'd5:   begin
                                SCLK<=0;     FSM<=FSM+1'd1;
                        end
                4'd6:   FSM<=FSM+1'd1;
                4'd7:   FSM<=FSM+1'd1;
                4'd8:   begin
                                SCLK<=1'd1; dataout[bitcounter]<=DOUT;
                                FSM<=FSM+1'd1;
```

```
                end
        4'd9:           FSM<=FSM+1'd1;
        4'd10:          begin
                            if (bitcounter==0) begin
                                FSM<=FSM+1'd1;
                                ready<=1'd1;
                            end
                            else begin
                                bitcounter<=bitcounter-1'd1;
                                FSM<=4'd5;
                            end
                        end
        4'd11:          begin
                            FSM<=FSM+1'd1;
                            SYNC<=0;    ready<=0;
                        end
        4'd12:          FSM<=0;
        default:;
        endcase
    if (counter_CLKIN==CLKIN_FIN) begin
        CLKIN<=CLKIN+1'd1;
        counter_CLKIN<=0;
    end
    else    counter_CLKIN<=counter_CLKIN+1'd1;
    end // CLK
end // always
endmodule
```

На модуль с листинга 6.7 необходимо подать сигнал «start», после чего будут сгенерированы необходимые сигналы на выводах АЦП. По завершении процесса на выходе модуля «dataout» образуются данные, пропорциональные напряжению на аналоговом входе АЦП. По переходу флага «ready» в высокое состояние данные с выхода модуля могут быть захвачены, а процесс оцифровки повторен.

На листинге 6.9 представлено описание модуля для работы с другой ИМС АЦП AD7894.

Листинг 6.8. Описание модуля «ADC AD7894»

```
module ADC_7894
(CLK, reset,            // Глобальные сигналы
start,                  // Сигнал запуска модуля
CONVST_N, BUSY, SCLK, DOUT,         // Сигналы к АЦП
ADC_dat, ready_ADC);      // Внутренние сигналы
parameter t1=10;
parameter waiting = 10;
input wire CLK, reset, start;
// from IMS ADC:
input wire BUSY, DOUT;
output wire CONVST_N; assign CONVST_N=!CONVST;
output reg SCLK;
reg CONVST;
// internal signals:
output reg [15:0] ADC_dat;
output reg ready_ADC;
reg[7:0] FSM;
reg[3:0] bitcounter;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                CONVST<=0; SCLK<=0;
                ADC_dat<=0; ready_ADC<=0;
                FSM<=0; bitcounter<=0;
        end // reset
        else begin
                case (FSM)
                8'd0:           begin
                                        if (start) begin
                                                ready_ADC<=0;       ADC_dat<=0;
                                                FSM<=8'd1;   bitcounter<=4'd15;
                                                CONVST<=0;          SCLK<=0;
                                        end
                                end
                8'd1:           begin
                                        FSM<=FSM+1'd1;    CONVST<=1'd1;
                                end
                t1:             begin
                                        if (BUSY)      FSM<=FSM+1'd1;
                                        CONVST<=0;
                                end
```

```
                t1+4:           begin
                                        if (!BUSY)      FSM<=FSM+1'd1;
                                end
                t1+5:   FSM<=FSM+1'd1;
                t1+6:           begin
                                        SCLK<=1'd1; FSM<=FSM+1'd1;
                                end
                t1+7:   FSM<=FSM+1'd1;
                t1+8:   begin
                                        FSM<=FSM+1'd1;
                                        ADC_dat[bitcounter]<=DOUT;
                                        SCLK<=0;
                                end
                t1+9:           begin
                                        if (bitcounter==0)      FSM<=FSM+1'd1;
                                        else begin
                                                bitcounter<=bitcounter-1'd1;
                                                FSM<=t1+5;
                                        end
                                end
                t1+10+waiting:          begin
                                                ADC_dat[15:14]<=0;
                                                ready_ADC<=1'd1;
                                                FSM<=0;
                                        end
                default:FSM<=FSM+1'd1;
                endcase
        end // CLK
end // always
endmodule // ADC_7894
```

Некоторые блоки, поддерживающие подключение к софт ЦП по шине периферии, представлены в приложении А.

6.3 Пример разработки учебного процессора для синтезируемой SoC

Разрабатываемая SoC может выступать soft-заменой несложного микроконтроллера, например семейства AVR classic. Система оптимизирована к применению в небольших FPGA, например

5578ТС034 [5] и внешних ИМС постоянной памяти для программ, например 1645РТ3У [20]. SoC построена на базе процессорного модуля «NIGHTMARE», далее – «NMR» (расширенной версией «SIP»), который является процессором с гарвардской архитектурой, т.е. содержит отдельную шину команд, а также шину данных и несколько специальных шин периферии. Процессор является 32-битным, так разрядность адреса памяти программ составляет 32 бита, а разрядность шины инструкций уменьшена до 16 бит, хотя сами инструкции могут иметь переменную длину: 16 бит или 32 бита. Это сделано специально для возможности подключения широко распространённых ИМС внешней памяти с 16-битной шиной данных. Структура командного слова процессора представлена на рисунке 6.12.

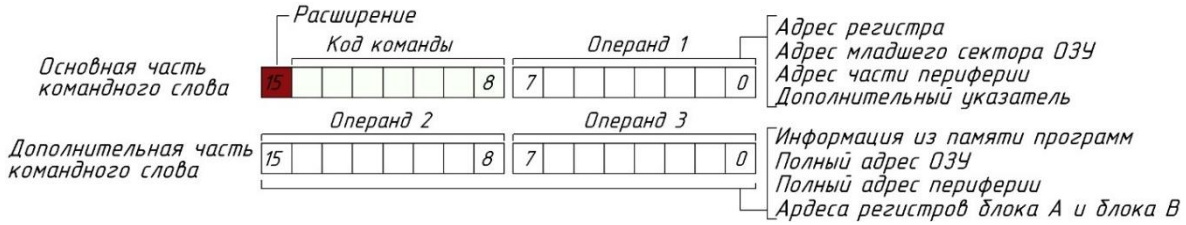


Рисунок 6.12 – Структура командного слова ЦП «NIGHTMARE»

Разделение некоторых команд на две части аппаратно предусмотрено архитектурой ядра и не отражается на программе. Система управления процессора по типу команды автоматически определяет смещение адреса, структурно она представляет из себя конечный автомат, в разных состояниях которого происходит обработка, выполнение инструкций и их частей. Важно, что параллельно с выполнением инструкций идет процесс считывания из памяти дополнительной части командного слова, что обусловлено оптимизацией под внешнюю 16-битную память программ. Такой подход позволяет избавиться от «простоя» ЦП, пока выполняется считывание всей инструкции из памяти, которая в случае с внешней ИМС – может быть медленнее максимальной скорости ЦП

(100нс для 1645РТ3У). Длительность процессорного такта в среднем короче или соответствует длительности считывания 32-битного командного слова из памяти программ, что исключает реализацию принципа конвейеризации инструкции. Использование внешней памяти в сочетании с 32-битной шиной адреса и программ позволяет гибко увеличивать необходимый объем в зависимости от программы. Такой адресуемый объем для памяти программ недоступен для любых ИМС ПЛИС, если говорить о внутренних блоках RAM. Структура ЦП «NMR» представлена на рисунке 6.13, а описание модуля ядра на Verilog HDL в приложении Б к данному пособию.

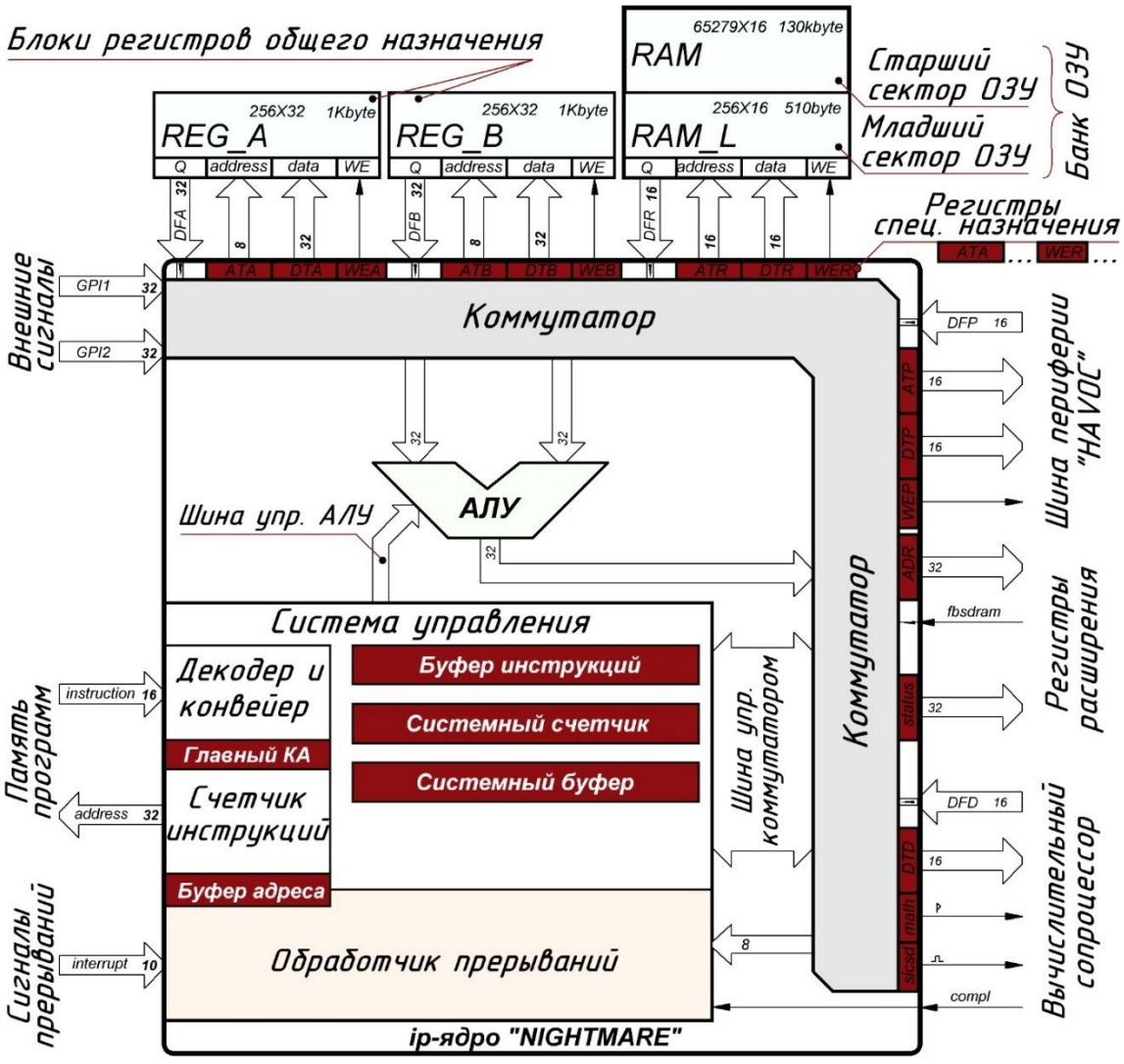


Рисунок 6.13 – Архитектура ЦП «NIGHTMARE»

В архитектуру входит массив регистров общего назначения (РОН), разделенный на два независимых блока REG_A и REG_B. Каждый банк содержит по 256 32-битных регистров. С «регистровой» памятью возможны все доступные операции, кроме специальных инструкций из системы команд процессора. Вся система команд ЦП «NMR» приведена в таблицах из приложения В. Блок работы с оперативной памятью состоит из нескольких частей. Первая часть для работы с быстрым внутренним банком ОЗУ, который имеет строение, схожее с регистрами общего назначения и использует внутреннюю ОЗУ ПЛИС, которая имеет разрядность шин данных и адреса по 16 бит, что позволяет адресовать до 131 кбайт памяти. Первые 256 ячеек ОЗУ (младший сектор) позволяют выполнять большее количество доступных операций, чем с остальным объемом ОЗУ (доступна инструкция «mark»), что снижает нагрузку на блоки РОН. В ЦП «NMR» предусмотрена возможность подключения внешней ОЗУ через ip-контроллер, для чего предназначены регистры расширения и входные линии внешних сигналов. В зависимости от типа внешней ОЗУ (SRAM, SDRAM, DDR) ip-контроллер может отличаться. В системе команд процессора предусмотрены инструкции для работы с динамической памятью, такие как пакетная запись и чтение блоками объемом до 131 кбайт. Остальные функции по взаимодействию непосредственно с ИМС ОЗУ выполняет ip-контроллер внешней памяти. Процессор содержит коммутатор, внешние порты которого буферизированы регистрами специального назначения (РСН). РСН необходимы для работы с блоками регистровой памяти, банком ОЗУ, а также с внешними шинами процессора. В системе команд (таблица 6.2) содержатся инструкции прямого взаимодействия с РСН, что позволяет осуществлять гибкое управление (различные варианты косвенной адресации) и ускорять операции чтения – записи (доступно копирования секторов, потоковое чтение периферии регистровой памяти и ОЗУ).

При использовании стандартных операций, связанных с перемещением данных, система управления процессором автоматически выполняет все операции с системными регистрами: адресацию, тактирование блоков, выставление и чтение данных с банков памяти. Все периферийные модули подключаются через универсальную адресуемую шину «HAVOC» (Hexadecimal Addressable Vast Outer Connection). Для чтения данных из шины «HAVOC» используется внешний модуль мультиплексора шины, управляемый адресным пространством данной шины. Ядро «NMR» универсально и было испытано на разных ИМС ПЛИС. Полученные характеристики ядра сведены в таблицу 6.2.

Таблица 6.2 – Характеристики ядра «NIGHTMARE» в разных ИМС ПЛИС

Параметр	ПЛИС								
	5578 TC034	**EPF10K 100EBC 356-3**	EPF10K 100EBC 356-1	**P2C8F 56C6**	**P3C16 484C6**	**P3C40 484C6**	**P355F 80C8**	P3C12 0780C7	EP3SE5 0F780C2
Необход. объем	4807 (96 %)	4807 (96 %)	4807 (96 %)	519 (43%)	540 (23%)	535 (9 %)	560 (6%)	625 (3 %)	2413+401 (6%)
Занимаемая память [бит]	32 768 (67%)	32 768 (67%)	32 768 (67%)	47 456 (85%)	1920 (82%)	47 456 (13%)	78 528 (12%)	64 960 (27%)	1 064 960 (20%)
Кол-во РОН	512	512	512	512	512	512	512	512	512
Объем ОЗУ	2КБ	2КБ	2КБ	16КБ	8КБ	16КБ	32КБ	131КБ	131КБ
Операций в секунду	6,4 MIPS	7,5 MIPS	13,2 MIPS	5,4 MIPS	47, MIPS	45,3 MIPS	37,2 MIPS	43,7 MIPS	72,8 MIPS
Макс. частота	20Мгц	23,5Мгц	40,5Мгц	79Мгц	149Мгц	141Мгц	116Мгц	136Мгц	225Мгц
Система команд	108 инстр.	108 инстр.	108 инстр.	полная	полная	полная	полная	полная	полная
Доп. проц.	внешний	внешний	внешний	Akeron	Akeron	Akeron + Sip CPU	Akeron + Sip CPU	Akeron II + Sip CPU	Akeron II + Sip CPU
Время отклика на прерывание, не более	300нс	255нс	150нс	76нс	40нс	42нс	51нс	44нс	27нс

В систему команд (приложение В) включены команды «mark», позволяющие делать аппаратные отметки по ходу исполнения программы. Для данной операции доступны как блоки РОН, так и младший сектор ОЗУ. Команда «mark» записывает в указанную область следующий от текущего указателя на адрес памяти программ.

Условные переходы при выполнении условия позволяют увеличить значения счетчика инструкции от 1 до 255. В пропущенной области может находиться ветвь программы, обрабатываемая при невыполнении условия, либо переход на другой сектор памяти, если условной ветви программы 255 ячеек памяти программ недостаточно.

Логические и арифметические операции выполняются на 32-битном АЛУ. Доступные на АЛУ операции представлены в системе команд. Ядро Ц.П. «NMR» разработано как универсальное решение для любых ИМС FPGA, поэтому из команд исключены операции аппаратного деления и умножения. Так как не во всех ИМС есть соответствующие аппаратные блоки, и не всегда данные функции необходимы, так для программ управления периферией обработка математики может быть избыточной функцией, а в случае необходимости целесообразно возложить эти задачи на вычислительный сопроцессор. Ядро сопроцессора для ЦП «NMR» будет описано далее, главное достоинство наличия вычислительного сопроцессора в том, что он может выполнять свои операции параллельно с основной программой, запущенной на ЦП. Для подключения сопроцессора предусмотрена выделенная высокоскоростная шина. Сопроцессор может быть размещен как на основной FPGA, так и вынесен в отдельную ИМС.

Графический файл процессора с ядром «NIGHTMARE» представлен на рисунке 6.14.

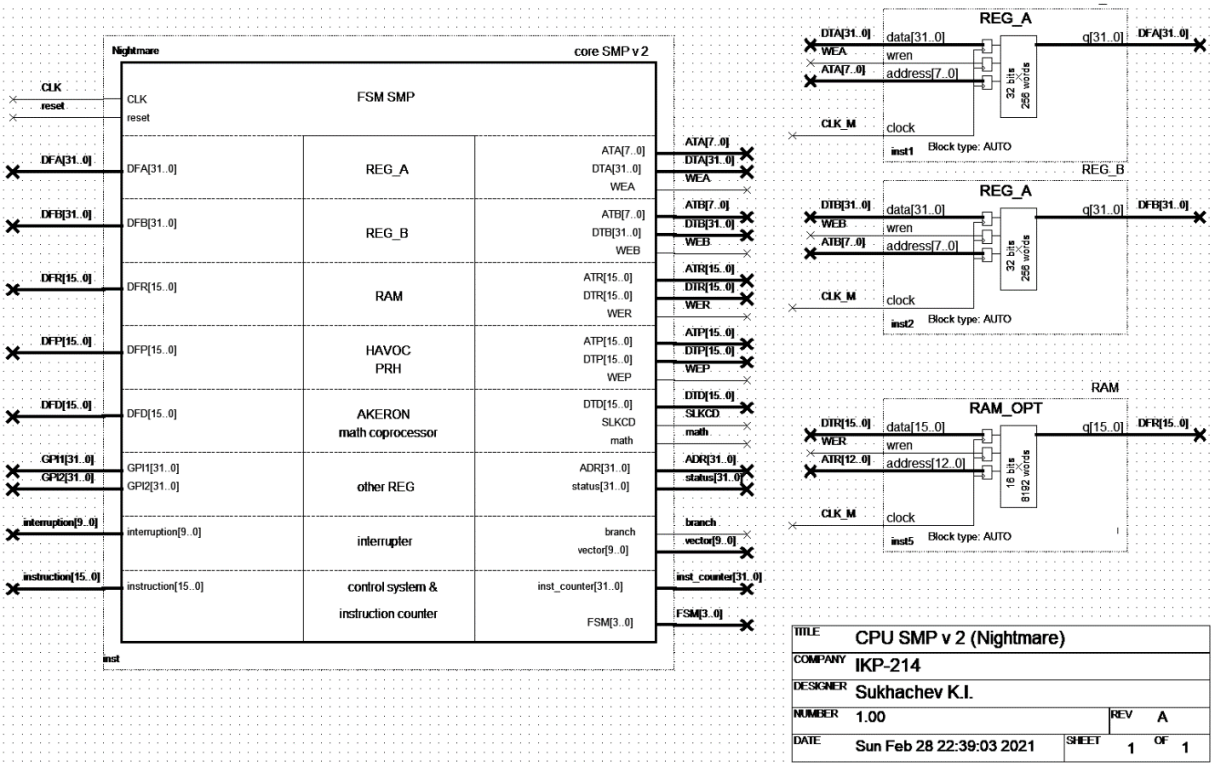


Рисунок 6.14 – Графическое представление ЦП «NIGHTMARE»

Для удобства написания программы применяется транслятор команд. Пример программы для ЦП и этапы её преобразования показаны в таблице 6.3. Программа состоит из объявления параметров памяти, обнуления необходимых РОН, инициализации блоков периферии, в том числе контроллера прерывания. Далее запускается бесконечный цикл с периодической отправкой на внешний порт инкрементируемой переменной. Процесс прерывается по сигналу о готовности от модуля передатчика, который захватывает значение переменной и начинает отправку, по завершении которой вновь вызывает прерывание. Сама программа прерывания описана по адресу начиная с 8000-й ячейки.

На рисунках 6.15 и 6.16 показаны осциллограммы выполнения программы из таблицы 6.3. На тестовой плате с EP3C25 на частоте ядра и периферии 50МГц.

Таблица 6.3 – Этапы преобразования программы

Исходная программа	Команды	Данные в MIF

#memory size: 8192
#first address: 1024
#interrupt file: C:\NMR\inter1.txt
// обнуление регистра A0 (не обязательно):
A0<=**16'd**0
A0>
A0<=**16'd**0
// запись в старший сектор регистра:
B0<=**16'd**0
B0>
B0<=**16'd**100
// обнуление регистра A1 (не обязательно):
A1<=**16'd**0
A1>
A1<=**16'd**200
// инициализация модуля по HAVOC:
PRH41<=**16'd**25
PRH42<=**16'd**25
PRH40<=**A**0
// активация «1го» вектора прерывания:
PRH1<=**16'd**1
// присваивание имен (не обязательно):
name A0 **as** counter;
name B0 **as** limit;
let PRH40 **as** SINT;
let PRH65535 **as** outport1;
// программа:
pointA: // указатель перехода
while (counter<limit){
outport1=counter;
counter++;
}
A0<=**16'd**0
goto pointA;
// сектор программы, вызываемый прерыванием:
address>>8000 *// указатель адреса сектора*
delayA1
SINT=counter; *// отправка сообщения*
intoff *// выход из прерывания*
	A0<=**16'd**0
	A0>
	A0<=**16'd**0
	B0<=**16'd**0
	B0>
	B0<=**16'd**100
	A1<=**16'd**0
	A1>
	A1<=**16'd**200
	PRH41<=**16'd**25
	PRH42<=**16'd**25
	PRH40<=**A**0
	PRH1<=**16'd**1
	marck RAM1
	marck RAM2
	if(A0<**B**0**)up**4
	PRH65535<=**A**0
	A0++
	jamp RAM2
	A0<=**16'd**0
	jamp RAM1
	address>>8000
	delayA1
	PRH40<=**A**0
	intoff
		0 : 1024;
		1 : 8000;
		2 : 8100;
		3 : 1024;
		…
		1022 : 1024;
		1023 : 7424;
		1024 : 0;
		1025 : 7936;
		1026 : 7424;
		1027 : 0;
		1028 : 7680;
		1029 : 0;
		1030 : 8192;
		1031 : 7680;
		1032 : 100;
		1033 : 7425;
		1034 : 0;
		1035 : 7937;
		1036 : 7425;
		1037 : 200;
		1038 : 7209;
		1039 : 25;
		1040 : 7210;
		1041 : 25;
		1042 : 4352;
		1043 : 40;
		1044 : 7169;
		1045 : 1;
		1046 : 10753;
		1047 : 10754;
		1048 : 12032;
		1049 : 4;
		1050 : 4352;
		1051 : 65535;
		1052 : 16384;
		1053 : 11522;
		1054 : 7424;
		1055 : 0;
		1056 : 11521;
		1057 : 0;
		…
		8000 : 32001
		8001 : 4352
		8002 : 40
		8003 : 32515

На рисунке 6.15 представлен полный цикл, в котором выполняется основная программа (инкремент и вывод переменной из РОН в ШП). Сигнал №4 на рисунке 6.15 – младший выводимый разряд переменной «counter». Проверка условия и условный переход, инкремент одного из РОН, вывод значения в ШП и прыжок на проверку условия (основная программа) занимает примерно 260нс. На рисунке 6.16 показана реакция на прерывание и разброс времени реакции.

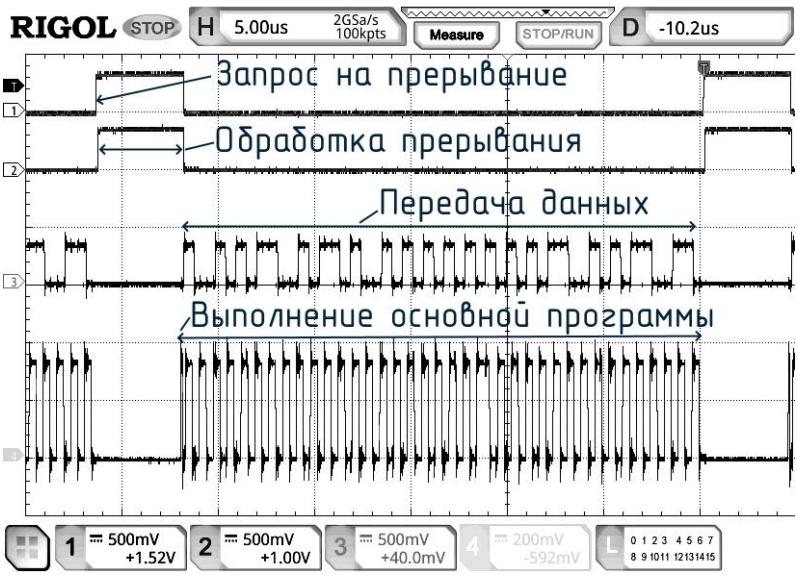


Рисунок 6.15 – Осциллограммы исполнения тестовой программы из табл. 7

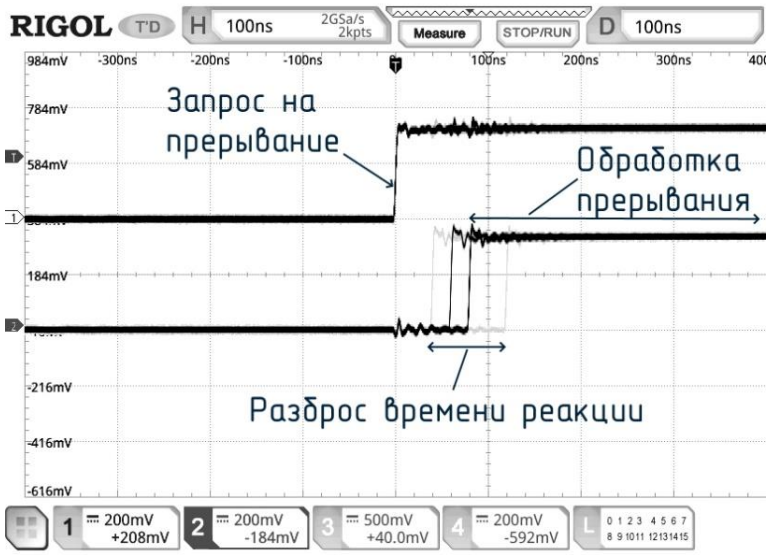


Рисунок 6.16 – Реакция на прерывание

6.4 Пример разработки учебного вычислительного сопроцессора для синтезируемой SoC

Выделение вычислительных функций на внешний относительно ЦП модуль позволяет, во-первых, увеличить производительность последнего, а во-вторых, сделать систему более гибкой. Это возможно благодаря тому, что функционал сопроцессора может быть скорректирован под текущую задачу, при этом не требуется вносить изменения в основное ядро, менять его систему команд и оптимизировать компилятор. Если не планируется производить вычислений в конкретном применении, то сопроцессор можно вообще исключит из проекта, а освободившуюся шину использовать в качестве, например, просто внешних портов. Целесообразно сопроцессор снабдить собственной оперативной памятью, которая ему потребуется для сложных операций, а также для хранения начальных данных и результатов вычислений.

Далее будет рассмотрена архитектура сопроцессора «AKERON», специально разработанного для подключения к ЦП «NMR», рассмотренному ранее. На рисунке 6.17 представлена его структура в виде графического отображения в QuartusII.

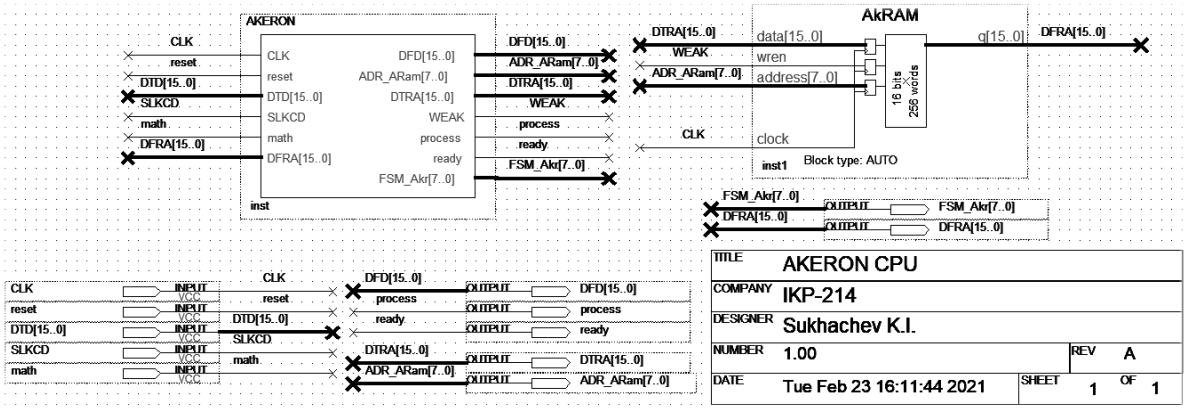


Рисунок 6.17 – Графическое представление вычислительного сопроцессора

Сначала рассмотрим логику работы сопроцессора, в основе которой лежит взаимодействие по выделенной шине с ЦП и запись

и чтение из ОЗУ сопроцессора. В нулевую ячейку ОЗУ необходимо записать данные, содержащие информацию о действии, которое необходимо выполнить, и количество слов в ОЗУ, которые будут сначала записаны, а потом задействованы в предстоящей операции. Формат нулевой ячейки ОЗУ представлен на рисунке 6.18. На листинге 6.9 приведено описание логики работы сопроцессора с шиной и памятью.

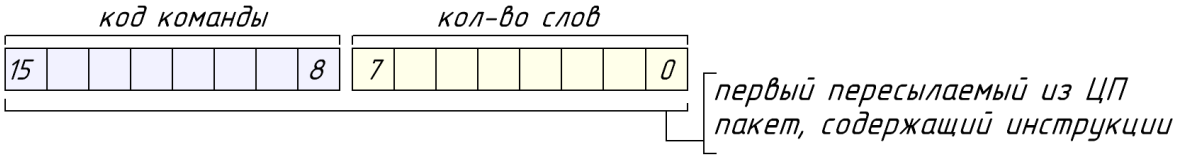


Рисунок 6.18 – Формат нулевой ячейки ОЗУ (пакет инструкций)

Листинг 6.9. Описание логики модуля «AKERON»

```
module AKERON
(CLK, reset, DTD, DFD, SLKCD, math,
ADR_ARam, DTRA, DFRA, WEAK,        // шина ОЗУ сопроцессора
process, ready,
FSM_Akr);
input wire CLK, reset;
input wire SLKCD, math;
input wire [15:0] DTD;
output wire [15:0] DFD;  assign DFD=ready?DFRA:16'd0;
input wire [15:0] DFRA;              //   шина данных от ОЗУ
output reg [7:0] ADR_ARam;                  //   шина адреса ОЗУ
output reg [15:0] DTRA;              //   шина данных к ОЗУ
output reg WEAK;                     //   разрешение записи
output reg ready, process;
reg [7:0] long; reg [7:0] instr;
reg [15:0] A; reg [15:0] B;
reg [63:0] Result;
output reg [7:0] FSM_Akr;
reg [15:0] FSM_exd;
reg flag, block, math_flg, math_blk, delay_fast_opr;
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                long<=0; instr<=0;
                process<=0; flag<=0; block<=0; ready<=0; math_flg<=0;
```

```
        FSM_Akr<=0; FSM_exd<=0; math_blk<=0;
        A<=0; B<=0; Result<=0; delay_fast_opr<=0;
        ADR_ARam<=0; DTRA<=0; WEAK<=0;
end // reset
else begin
// writing data to Akeron's memory for processing:
if (!process) begin
        if (SLKCD) begin
                DTRA[15:0]<=DTD[15:0];
                block<=0;
                WEAK<=1'd1;
                if (ADR_ARam==0) begin
                        long[7:0]<=DTD[7:0];
                        instr[7:0]<=DTD[15:8];
                        ADR_ARam<=0;
                        flag<=1'd1;
                end // ADR_ARam = 0
        end // SLKCD
        else begin
                if (flag&!block) begin
                        WEAK<=0;
                        block<=1'd1;
                        case (ADR_ARam)
                        long:   begin
                                        process<=1'd1; WEAK<=0;
                                        FSM_Akr<=0; FSM_exd<=0;
                                end
                        default:        ADR_ARam<=ADR_ARam+1'd1;
                        endcase
                end     // flag & ! block
                else begin
                        WEAK<=0;
                end     // flag ? block
        end     // SLKCD = 0
end //  (FINISH "writing data to Akeron's memory for processing")
// reading processed data from Akeron's memory:
if (math) begin
        math_flg<=1'd1;
        if (SLKCD)    math_blk<=1'd1;
        else begin
                if (math_blk) begin
                        ADR_ARam<=ADR_ARam+1'd1;
```

```
                        math_blk<=0;
                    end
                end
            end // math
            else begin
                if (math_flg)begin
                    long<=0; instr<=0;
                    process<=0; flag<=0; block<=0; ready<=0;
                math_flg<=0; math_blk<=0;
                    FSM_Akr<=0; FSM_exd<=0;
                    A<=0; B<=0; Result<=0;
                    ADR_ARam<=0; DTRA<=0; WEAK<=0; delay_fast_opr<=0;
                end
            end // math = 0
            // operation process:
            if (process) begin
                case (instr)
                8'd##:  begin
                                case (FSM_Akr)
                                // procedure for obtaining data for processing:
                                8'd##:  begin
                                            ***
                                            end
                                ***
                                default:;
                                endcase
                    end
                8'd##:  begin
                            ***
                    end
                ***
                default:;
                endcase
            end // process
            end // CLK
        end // always
        endmodule // AKERON
```

После завершения операции сопроцессор записывает результат в свою оперативную память и устанавливает флаг «ready», после чего ЦПУ может считать её. Причем в нулевой ячейке остается

код выполненной операции. Описание всех возможных вычислительных задач в листинге 6.9 ядра сопроцессора не представлен, в нем показана только основная логика взаимодействия с ЦПУ.

* Задание:

1. Добавить дополнительную инструкцию в систему команд SIP, внести соответствующие изменения в описание с листинга 6.1, проверить работоспособность внесенных изменений.
2. Предложить вариант по добавлению флагов состояния АЛУ для обоих вариантов ЦП.
3. Доработать описание модулей UART, внести функцию мажоритирования и проработать возможность буферизации при информационном обмене.

7 ПРАКТИЧЕСКИЕ ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОГО ВЫПОЛНЕНИЯ

В качестве отладочной платы для практических занятий №1 и №2 используется плата с ИМС EPM240, показанная на рисунке 7.1 Техническая информация на данную ИМС приведена в приложении Г.

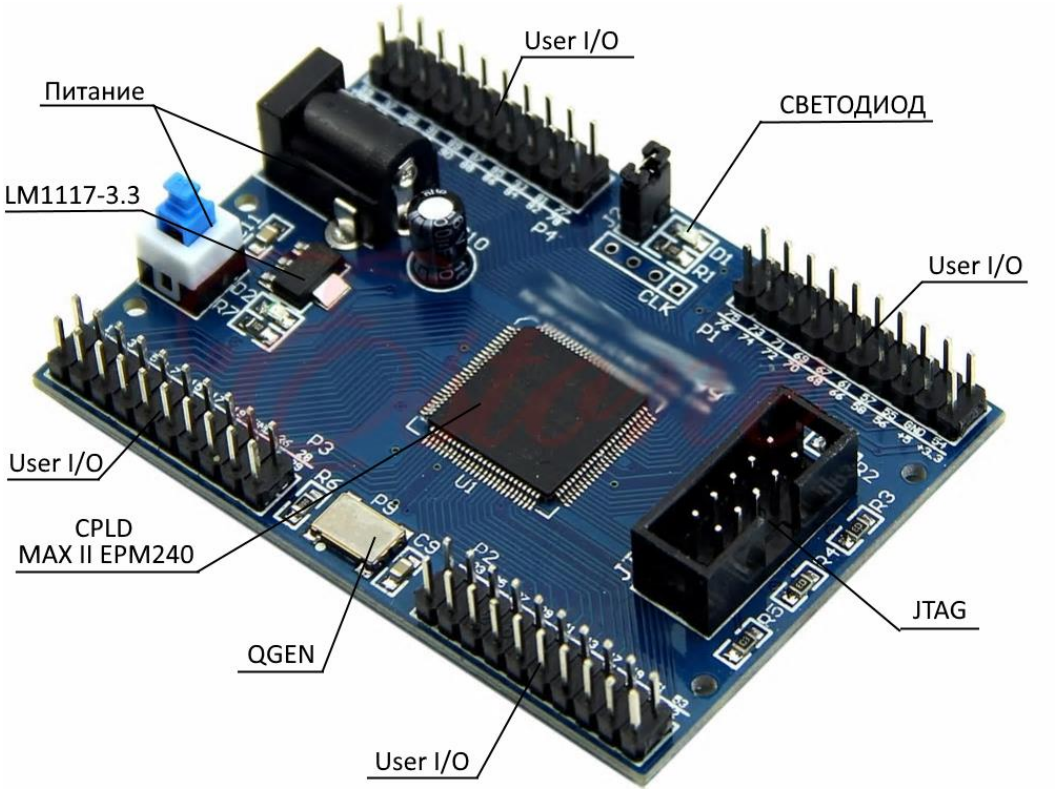


Рисунок 7.1 – Отладочная плата MAXII EPM240

Номера выводов микросхемы обозначены на самой ПП, прочие элементы показаны на рисунке 7.1.

7.1 Практическое занятие №1 Работа со счетчиками и разработка ШИМ контроллера

В ходе данной практической работы предлагается разработать демонстрационный проект, состоящий из двух блоков. Первый блок – управляемый ШИМ генератор позволяет непрерывно

генерировать импульсы с постоянной частотой и длительностью, зависящей от значения на входной шине блока. Второй блок – генератор значений, генерирующий на выходной шине, меняющийся по заданному закону в двоичный код. Выход второго блока соединен с управляющим входом ШИМ генератора. Выход ШИМ предполагается подать на ножку, соединенную со светодиодом LED на ПП. В результате правильной работы проекта светодиод должен загораться и гаснуть по заданному закону. Параметры блоков по вариантам представлены в таблице 7.1. Образец файла верхнего уровня в графическом формате показан на рисунке 7.2.

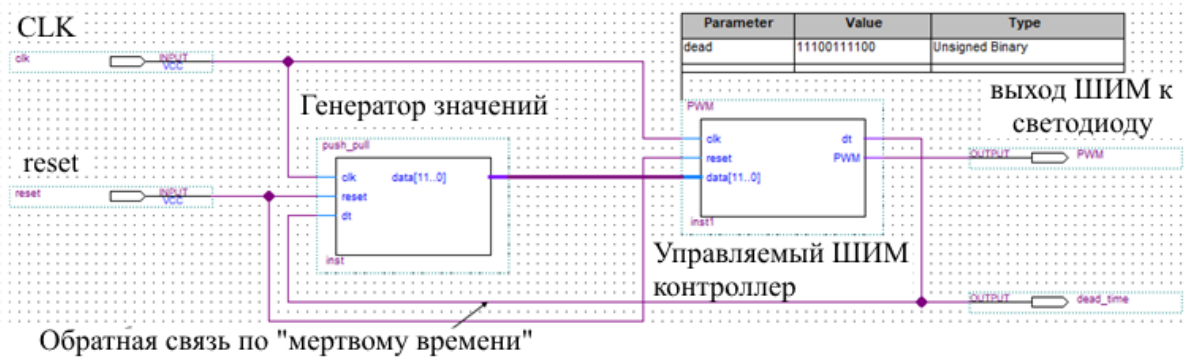
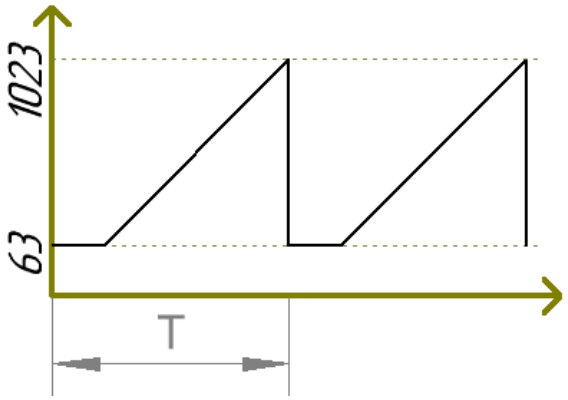
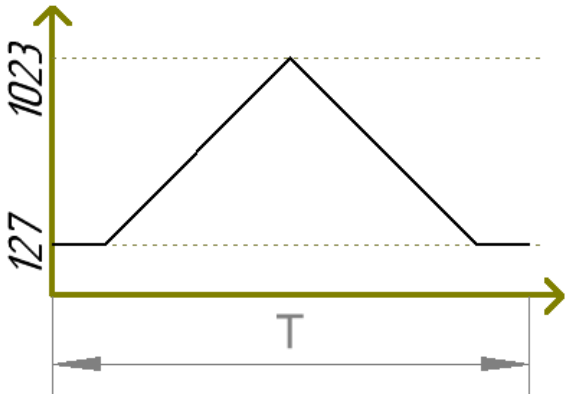
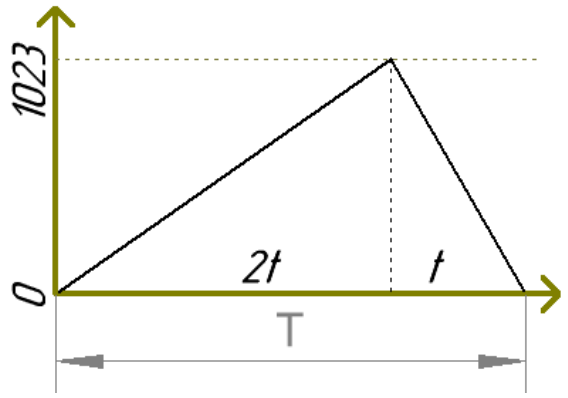
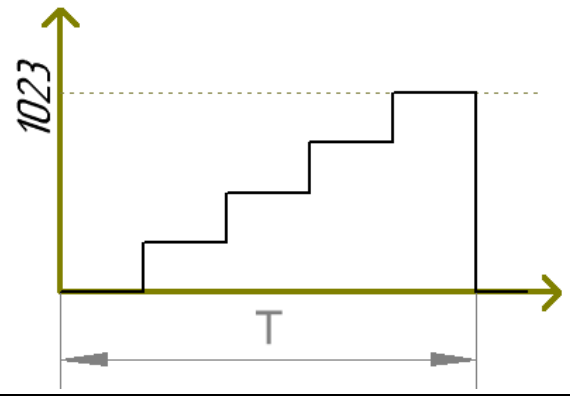


Рисунок 7.2 – Конфигурация верхнего уровня файла проекта

Генератор значений меняет число на выходной шине data[11..0] от минимального до максимального значения каждый раз по фронту сигнала обратной связи. Сигналом обратной связи является начало времени, при котором на выходе блока PWM всегда низкий уровень, независимо от значения на шине data[11..0]. Запись во внутренний регистр блока ШИМ генератора происходит на несколько тактов позже фронта на выводе dt. Общая длительность «мертвого времени» dt должна задается параметром блока PWM.

Таблица 7.1 – Варианты заданий для практического занятия №1

Вар.	Параметры генератора значений	Параметры ШИМ контроллера	Примечание
1		10-битный ШИМ	f = 1Гц
2		12-битный ШИМ	f = 1Гц
3		12-битный ШИМ	f = 0,333Гц
4		10-битный ШИМ	f = 0,775Гц

Дополнительное задание: управлять светодиодом по синусоидальному закону. Для этого предлагается использовать блок UFM для хранения таблицы значений.

Этапы выполнения:

1. Освоить теоретическую часть, получить допуск у преподавателя. Основным критерием допуска является повторение описанного в главе 3 проекта «blink» и демонстрация работы на отладочной плате.

2. Разработать структурную схему ШИМ контроллера и генератора значений согласно заданному варианту (таблица 7.1).

3. Разработать описание ШИМ контроллера в текстовом формате в Verilog HDL File. Произвести симуляцию и отладку модуля.

4. Разработать описание модуля генератора значений на Verilog HDL согласно заданному варианту (таблица 7.1). Произвести симуляцию и отладку модуля.

5. Объединить оба модуля в общий проект в графическом формате, согласно рисунку 7.2, произвести симуляцию и отладку проекта целиком.

6. Произвести прошивку отладочной платы, провести анализ работы устройства, снять осциллограммы с вывода PWM и dt, сравнить результаты измерений и симуляции.

7. Продемонстрировать результаты работы преподавателю и приступить к оформлению отчета.

Отчет должен быть представлен в печатной форме и должен содержать: название дисциплины; название лабораторной работы; этапы выполнения работы; описание разрабатываемых модулей, результаты их симуляции; реальные осциллограммы, снятые с отладочной платы; выводы по проделанной работе.

Контрольные вопросы

1. Какие преимущества дает увеличение степени интеграции элементной базы РЭС?

2. Какие существуют основные типы ИМС с изменяемым функционированием?

3. Как применение несинхронной логики влияет на стабильность и надежность цифровых РЭС? В каких случаях целесообразно применять синхронную и несинхронную логику?

4. В чем разница между CPLD и FPGA микросхемами?

5. Как устроена микросхема программируемой логики? Что такое логический программируемый блок?

6. В чем разница между блокирующим и неблокирующим присваиванием? Объяснить на примере участка кода:

```
input wire CLK;
input wire reset;
input wire [9:0] adres;
input wire record;
output reg [9:0] SASR;
output wire [9:0] adr_slv;
…
assign adr_slv = SASR;
always @(posedge CLK or posedge reset) begin
        if (reset) begin
…
        end // reset
        else begin
                if (record) begin
                        SASR<=adres;
                        data_slv<=data;
                        rec<=1'd1;
                end
                else begin
                        …
                end
        end // CLK
end // always
endmodule
```

7. Какими способами можно записать логическую конструкцию: *если **А** то **Б** иначе **С*** в Verilog?

7.2 Практическое занятие №2
Разработка многофазного контроллера блока питания

Практическое занятие позволяет закрепить ранее изученный материал и получить опыт в разработке контроллеров на базе ПЛИС, позволяющий управлять мощными регуляторами или источниками питания.

Сигнал, формируемый ШИМ контроллером, используется для управления электронными ключами, которые периодически, с частотой ШИМ сигнала, подключают и отключают нагрузку к линии питания. Амплитуда ШИМ сигнала должна быть такой, чтобы с его помощью можно было управлять электронным ключом. Таким образом, на выходе электронных ключей наблюдается последовательность прямоугольных импульсов с амплитудой линии питания и частотой следования, равной частоте ШИМ импульсов. Известно, что любой периодический сигнал может быть представлен в виде гармонического ряда Фурье. В частности, периодическая последовательность прямоугольных импульсов одинаковой длительности при представлении в виде ряда будет иметь постоянную составляющую, обратно пропорциональную скважности импульсов, то есть прямо пропорциональную их длительности. Пропустив полученные импульсы через фильтр нижних частот (ФНЧ) с частотой среза, значительно меньшей, чем частота следования импульсов, эту постоянную составляющую можно легко выделить, получив стабильное постоянное напряжение. Поэтому импульсные преобразователи напряжения содержат также низкочастотный фильтр, сглаживающий последовательность прямоугольных импульсов напряжения. Структурная схема такого импульсного понижающего преобразователя напряжения показана на рисунке 7.2.

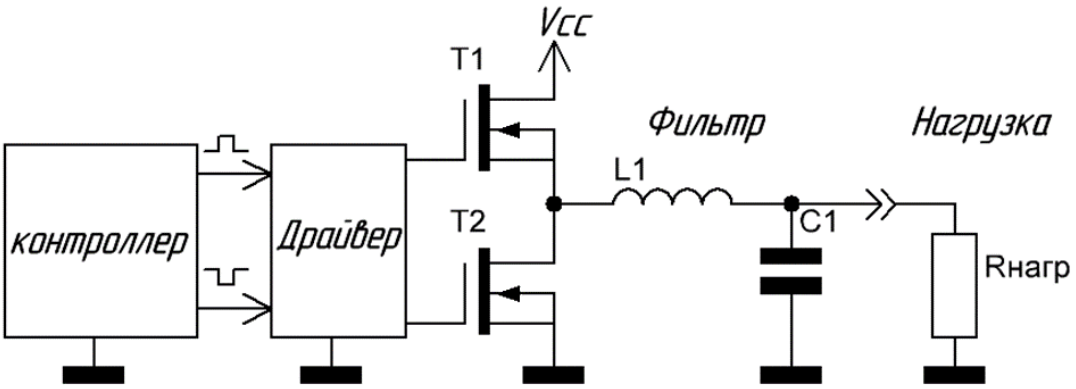


Рисунок 7.2 – Структурная схема преобразователя

В многофазных импульсных регуляторах напряжения каждая фаза образована драйвером управления переключениями MOSFET-транзисторов, парой самих MOSFET-транзисторов и сглаживающим LC-фильтром. При этом используется один многоканальный ШИМ контроллер, к которому параллельно подключается несколько фаз питания. Применение N-фазного регулятора напряжения позволяет распределить ток по всем фазам, а следовательно, ток, протекающий по каждой фазе, будет в N раз меньше тока нагрузки. Если использовать 4-фазный регулятор напряжения с ограничением по току в каждой фазе 30 А, то максимальный ток через нагрузку может составлять 120 А. Если нагрузке нужна большая мощность, то можно применить не 4-фазный, а 6-фазный импульсный регулятор напряжения питания, иначе необходимо использовать в каждой фазе питания дроссели, конденсаторы и MOSFET-транзисторы, рассчитанные на больший ток. Для уменьшения пульсации выходного напряжения в многофазных регуляторах напряжения все фазы работают синхронно с временным сдвигом друг относительно друга. Если T – это период переключения MOSFET-транзисторов (период PWM-сигнала) и используется N фаз, то временной сдвиг по каждой фазе составит T/N. За синхронизацию PWM-сигналов по каждой фазе с временным сдвигом отвечает PWM-контроллер.

Таблица 7.2 – Варианты заданий для практического занятия №2

Вар.	Кол-во фаз	Разрядность	Тип модуляции
1	4	12 бит	ШИМ
2	6	10 бит	
3	1	16 бит	Дельта-сигма
4	1	12 бит	ЧИМ
5	5	10 бит	ШИМ

Этапы выполнения:

1. Освоить теоретическую часть, получить допуск у преподавателя. Основным критерием допуска является повторение описанного в главе 6 модуля «PWM controller» и демонстрация его работы на отладочной плате или в симуляторе.

2. Разработать структурную схему многофазного ШИМ согласно заданному варианту (таблица 7.2).

3. Разработать описание многофазного ШИМ контроллера в текстовом формате в Verilog HDL File. Произвести симуляцию и отладку модуля.

4. Произвести программирование отладочной платы, провести анализ работы устройства, снять осциллограммы с выводов PWM, сравнить результаты измерений и симуляции.

5. Продемонстрировать результаты работы преподавателю и приступить к оформлению отчета.

Отчет должен быть представлен в печатной форме и должен содержать: название дисциплины; название лабораторной работы; этапы выполнения работы; описание разрабатываемых модулей, результаты их симуляции; реальные осциллограммы, снятые с отладочной платы; выводы по проделанной работе.

Контрольные вопросы

1. Преимущества и недостатки цифрового управления силовой и импульсной электроникой.

2. Какие существуют основные типы управления источниками питания?

3. Какие преимущества в многофазном управлении источниками питания, области применения? Привести примеры.

4. Сравнение использования ПЛИС для управления импульсными преобразователями, микроконтроллеров и специализированных ИМС.

7.3 Практическое занятие №3
Работа с конечными автоматами

В данном практическом занятии предлагается разработать простейший конечный автомат Мили с входным сигналом X и выходным сигналом Y. За основу взять схему, представленную на рисунке 7.3.

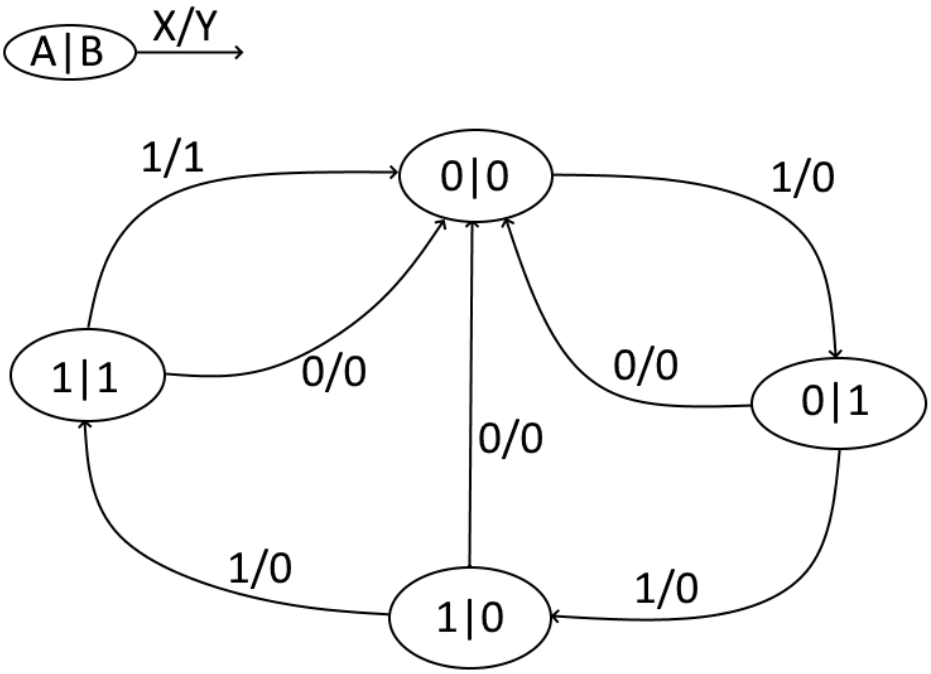


Рисунок 7.3 – Граф конечного автомата

Этапы выполнения:

1. Составить таблицу состояний по графу из рисунка 7.3, продемонстрировать таблицу преподавателю, получить указания по следующему пункту.
2. Разработать цифровую схему, реализующую конечный автомат с рисунка 7.3.
3. Разработать в Quartus графическим способом и на HDL, провести моделирование обоих вариантов, сравнить результаты и продемонстрировать их преподавателю.
4. Оформить отчет о проделанной работе.

Контрольные вопросы

1. Что такое конечный автомат?
2. Какие виды конечных автоматов вы знаете?
3. Варианты описания конечного автомата на Verilog.
4. Привести пример описания с одним и двумя always блоками для одной и той же задачи.

7.4 Практическое занятие №4
Разработка простого учебного процессорного ядра

В ходе данного практического занятия будет усвоен принцип работы процессорной техники и разработано учебное процессорное ядро с несложной системой команд. В качестве основы для проведения этого занятия рекомендуется использовать Verilog HDL описание, приведенное в разделе 6.5 данного пособия, или описание из приложения Б. При выполнении практической части задания будет применяться отладочная плата на базе FPGA, представленная на рисунке 7.4, описание которой приведено в приложении Д.

Данное занятие, кроме получения практических навыков и закрепления теоретической информации по курсу, также предполагает развитие навыков командной работы. Для этого разработка

процессора будет вестись командами до трех человек и с четким разделением обязанностей в команде: разработчик описания на HDL, разработчик архитектуры и системы команд, ответственный руководитель, он же организатор процесса разработки.

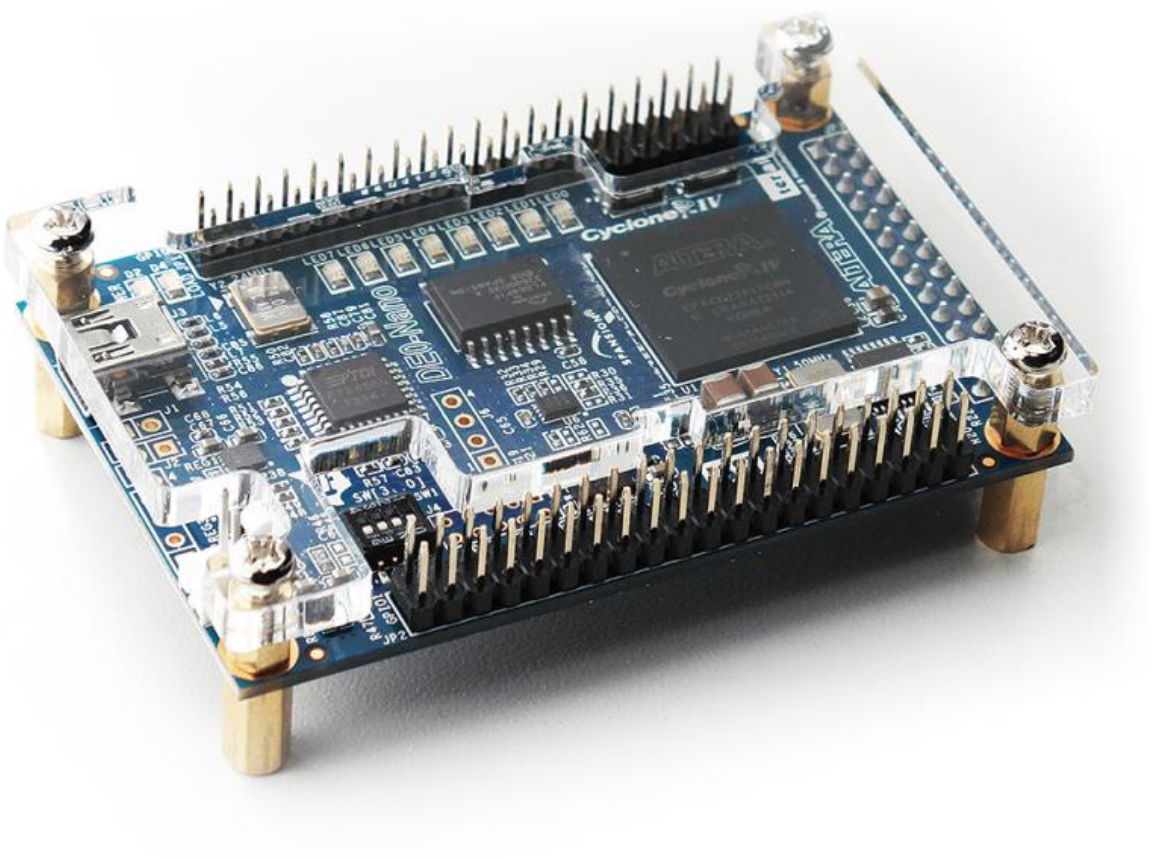


Рисунок 7.4 – Отладочная плата DE0-Nano ALTERA – Cyclone IV

По результатам выполнения данной работы будет проведен конкурс среди команд. Процессорное ядро будет оцениваться по таким критериям, как:

1) количество занимаемых модулей в ПЛИС;

2) максимальная рабочая частота ядра;

3) система команд (кол-во команд процессора, необходимое для выполнения тестовой команды);

4) наличие аппаратных возможностей ядра процессора.

Этапы выполнения:

1. Разделиться на команды по три человека и определиться с обязанностями в команде.

2. Усвоить теоретический материал по синтезируемым процессорам, разобраться со структурой и принципом работы учебного процессора из раздела 6.3 данного пособия. Получить допуск у преподавателя, ответив на теоретические вопросы.

3. Разработать ТЗ на разработку ядра ЦП и продемонстрировать понимание этапов выполнения и задач, стоящих перед командой.

4. Осуществить переработку системы команд (добавить недостающие или удалить лишние из описания процессора в разделе 6.3). Утвердить систему команд внутри группы разработчиков и у преподавателя.

5. Добавить в HDL описание, приведенное в разделе 6.3, обработчик прерываний и недостающие команды для работы с прерываниями.

6. Реализовать возможность перехода процессора в спящий режим.

7. Доработать и отладить описание на HDL_Verilog согласно ранее принятым решениям.

8. Получить у преподавателя тестовую программу для проверки работоспособности разработанного ядра.

9. Сгенерировать файл инициализации ROM памяти для хранения программы.

10. Продемонстрировать работоспособность ядра ЦП в симуляторе.

11. Получить у преподавателя отладочную плату с FPGA, назначить выводы и осуществить конфигурирование ПЛИС в энергозависимую память с программой для ЦП, позволяющую «мигать» светодиодом на отладочной плате.

12. Приступить к оформлению отчета.

Отчет должен быть представлен в печатной форме и должен содержать: название дисциплины; название лабораторной работы; этапы выполнения работы; описание разрабатываемых модулей, результаты их симуляции; реальные осциллограммы, снятые с отладочной платы; выводы по проделанной работе.

Контрольные вопросы

1. Принцип работы процессора, отличия процессора от классического конечного автомата состояний.
2. Чем софт-процессор отличается от аппаратного процессорного ядра?
3. Преимущества и недостатки синтезируемых внутри ПЛИС процессорных ядер. Какие есть особенности и ограничения?
4. Примеры синтезируемых процессоров от производителей ПЛИС, их характеристики.
5. Что такое система команд процессора, классификация процессоров по системам команд?
6. Отличие RISC, CISC, VLIW и архитектуры с неоднородными вычислениями. Какие архитектуры больше подходят для ПЛИС и почему?
7. Что такое система на кристалле? Привести примеры СнК. Какие методы реализации СнК существуют?
8. На примере учебного процессора объяснить назначение всех шин и сигналов.
9. Классификация процессоров по принципу разделения доступа к памяти и шинам.
10. Что такое прерывание? Как процессор может обрабатывать прерывания?
11. Варианты реализации многозадачности. Что такое вытесняющая многозадачность.
12. Варианты реализации параллельных вычислений.

7.5 Практическое занятие №5
Разработка ip-модулей интерфейсов

На данном занятии предлагается разработать модули приемника и передатчика для организации последовательного интерфейса, который может быть подключен как устройство периферии к разработанному ранее процессору. Данный интерфейс будет использован для связи между двумя отладочными платами с максимальной протяженностью линии связи до 10м. При разработке физического уровня интерфейса не допускается применение дополнительных активных элементов и ИМС. Должна быть предусмотрена возможность гальванической развязки.

В качестве метода кодирования рекомендуется использовать манчестерский код по IEEE 802.3 или Thomas, структуру пакета данных, представленную на рисунке 7.5.

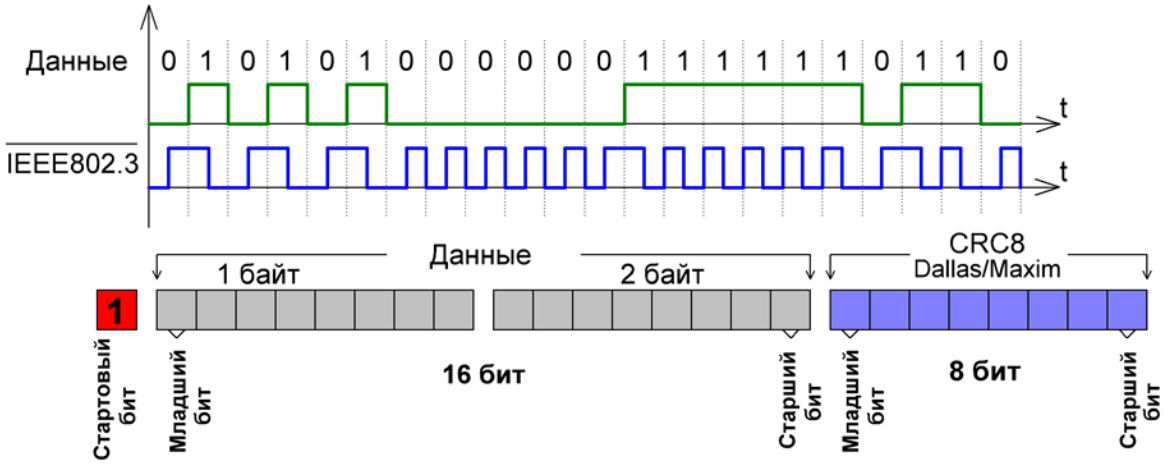


Рисунок 7.5 – Структура пакета данных и метод шифрования

Применение такого кодирования позволяет реализовывать самотактирующийся протокол с посылками неограниченной длительности без необходимости точной синхронизации часов приемника и передатчика. Кроме того, сигнал, закодированный по IEEE 802.3, не обладает постоянной составляющей, что позволяет в качестве гальванической развязки применять сигнальные трансформаторы без дополнительных аппаратных средств.

Задание выполняется по командам, команда включает до трех студентов: предполагается, что один участник команды разрабатывает модуль приемника, второй – модуль передатчика, а третий осуществляет общую координацию проекта и является ответственным исполнителем. Варианты заданий для нескольких команд представлены в таблице 7.3.

Таблица 7.3 – Варианты заданий для практического занятия №5

Вар.	Параметры передачи	Верификация	Примечание
1	1 Мб/с, IEEE802.3	CRC8: $x^8+ x^5+ x^4+1$	При обнаружении ошибки выставлять только флаг ошибки
2	5 Мб/с, Thomas	CRC8: $x^8+x^7+x^6+ x^2+1$	
3	0.5МБ/с, Дифференциальное манчестерское кодирование	5-битная контрольная сумма	
4	2МБ/с Дифференциальное манчестерское кодирование	CRC5: $x^5+ x^2+1$	Выставление флага готовности и дополнительно флага ошибки при её наличии
5	0,5 Мб/с, IEEE802.3	Трехкратное временное резервирование	
6	1 Мб/с, Thomas	CRC8: $x^8+x^7+x^6+ x^2+1$	

Этапы выполнения:

1. Изучить теоретическую часть.
2. Утвердить вариант задания и разбиться на команды.
3. Получить допуск у преподавателя. Для этого необходимо повторить UART приемник и передатчик из раздела 6.4 данного пособия. Продемонстрировать его работоспособность. Разработать ТЗ на модули согласно утвержденному варианту и продемонстрировать четкое понимание задачи на разработку модулей. Утвердить флаги состояния разрабатываемых модулей, обосновать их достаточность и необходимость.

4. Разработать описание на Verilog_HDL модуля приемника и передатчика, провести функциональное, а затем временное моделирование, для целевого семейства ИМС ПЛИС MAXII.

5. Провести совместное моделирование модулей, определить границы ресинхронизации опорных тактовых сигналов для модулей.

6. Продемонстрировать результаты преподавателю и получить исходные данные для второго этапа работы. Формат исходных данных представлен в таблице 7.3.

7. Продемонстрировать результаты моделирования и получить от преподавателя отладочные платы и тестовую линию связи. Собрать схему, показанную на рисунке 7.4.

8. В присутствии преподавателя осуществить передачу информации.

9. Контроль флагов состояния модуля передатчика осуществлять запоминающим осциллографом.

10. Приступить к оформлению отчета.

Таблица 7.4 – Исходные данные
для второго этапа практического занятия №5

Вар.	Данные для отправки	Верификация	Тип соединения модулей
1	X (число 16 бит)	Выводить на LED флаг ошибки. Остальные флаги выводить на свободные выводы платы	Прямое соединение
2	Данные с входной параллельной шины	Выводить параллельную шину с передатчика и сигнал ошибки	
3	Z (число 16 бит)	Выводить все флаги состояния	Через линию связи с развязкой

Отчет должен быть представлен в печатной форме и должен содержать: название дисциплины; название лабораторной работы; этапы выполнения работы; описание разрабатываемых модулей, результаты их симуляции; реальные осциллограммы, снятые с отладочной платы; выводы по проделанной работе.

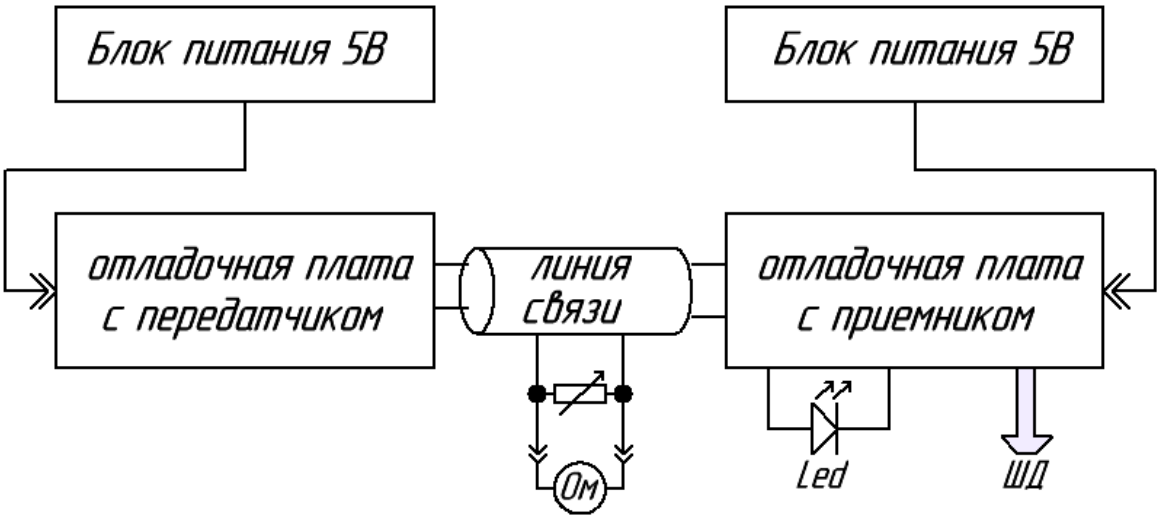


Рисунок 7.6 – Тестовая схема соединения модулей
для практического занятия №4

Контрольные вопросы

1. Модульный принцип описания структуры ПЛИС.
2. Принцип передачи информации по UART интерфейсу.
3. Предложить способ увеличения надежности приема по UART. Применить метод временной или структурной избыточности.
4. Принцип передачи по SPI интерфейсу.
5. Принцип передачи по I2C интерфейсу.
6. Принцип манчестерского кодирования, примеры использования.
7. Преимущества и недостатки перечисленных выше интерфейсов передачи.

8. Как вычисляется циклически избыточный код? В чем отличия от контрольной суммы?

9. Принцип верификации данных с помощью CRC.

10. Что такое временные домены в описании структуры ПЛИС, методы их синхронизации?

11*. Предложить вариант построения сети с использованием разработанных модулей.

7.6 Практическое занятие №6 Разработка аппаратных ускорителей математических вычислений

Практическая работа посвящена разработке цифровых схем ускорителей математических операций на RTL уровне регистровых передач с использованием Verilog HDL. В работе будет вестись разработка цифрового модуля, позволяющего выполнять заданную вариантом арифметическую операцию. Разработка должна осуществляться с учетом возможности подключения к ранее разработанному ЦП по шине периферии.

На ускоритель должны передаваться сначала исходные данные, необходимые для проведения вычислений. Функция, подлежащая вычислению, приведена в таблице 7.5. Алгоритм взаимодействия с СШ разрабатывается самостоятельно.

Таблица 7.5 – Варианты заданий для практического занятия №6

Вар.	Функция	Примечание
1	$y = a^2 + 2a - 3c^2$	Без использования аппаратных умножителей
2	$y = a^3 - a\sqrt{c}$	
3	$y = a^2 + \frac{\sqrt{a}}{2c}$	
4	$y = 1 + a + \frac{a^2}{2} + \frac{a^3}{6}$	Требуется максимальная скорость вычисления

Окончание табл. 7.5

Вар.	Функция	Примечание
5	$y = 38 + 14a - 3a^2$	
6	$y = 2\sqrt{a^2 + \sqrt{c}}$	
7	$y = \frac{8a}{c} + 3c$	Требуется минимальный объем занимаемой логики, скорость вычисления не важна.
8	$y = \sqrt{a} + \frac{b}{c}$	
9	$y = \frac{3a + \frac{b}{c} - \sqrt{c}}{2}$	
10	$y = \frac{a^c}{b}$	

В процессе реализации алгоритма работы разрабатываемого блока развиваются навыки проектирования цифровых схем конечных автоматов, которые используются для реализации управляющей логики блока и определения этапов вычислительного процесса.

Этапы выполнения:

1. Усвоить теоретическую часть.
2. Получить допуск у преподавателя и утвердить вариант задания.
3. Разработать описание модуля согласно полученному варианту.
4. Написать программу для разработанного ранее ЦП, выполняющую вычисления выражения согласно варианту.
5. Написать тестовую программу для разработанного ранее ЦП, позволяющую продемонстрировать работоспособность ускорителей вычислений, сравнить время выполнения вычислений с ускорителем и без него.
6. Оформить отчет о проделанной работе.

Контрольные вопросы

1. Как реализовать параллельные вычисления в процессорной системе на базе ПЛИС?

2. Предложить несколько вариантов машинного умножения.

3. Объяснить различие в алгоритмах машинного деления. С чем вызвана сложность данной операции?

4. Предложить алгоритм извлечения квадратного корня, отличающийся от алгоритма, приведенного в данном пособии.

5. Предложить алгоритм извлечения кубического корня.

6. Область применения аппаратных ускорителей. Привести примеры.

7.7 Практическое занятие №7
Разработка сопроцессора

Для выполнения более сложных вычислений и задач обработки данных аппаратному модулю ускорения вычислений может потребоваться наличие своей ОЗУ, своего набора команд и АЛУ. Таким образом, модуль превращается в сопроцессор, который может быть подключен к основному ЦП либо через ШП, либо через выделенную шину, если требуется большая скорость обмена. Данное занятие посвящено разработке такого вычислительного сопроцессора, реализующего ряд стандартных операций по обработке массива информации и разгружающего основной процессор, тем самым повышая быстродействие процессорной системы.

Предлагается модернизировать разработанный ранее модуль ускорения вычислений, реализовать одну дополнительную вычислительную операцию, разработать алгоритм взаимодействия ЦП и сопроцессора и разработать описания сопроцессорного модуля.

Взаимодействие с ЦП необходимо осуществить по шине подключения сопроцессора, предусмотренной архитектурой ЦП из раздела 6.5 данного пособия, используя специальные команды из системы ЦП для работы с сопроцессором.

7.8 Практическое занятие №8
Разработка СнК на базе ПЛИС

Занятие является завершающим и подразумевает объединение ранее разработанных модулей периферии, специальных блоков и ЦП в одной ИМС. В рамках практических и лабораторных занятий необходимо разработать СнК или аналог микроконтроллера с параметрами, приведенными в таблице 7.6, и загрузить его в отладочную плату с FPGA для демонстрации работы тестовой программы.

Таблица 7.6 – Параметры СнК для практического занятия №7

Модуль или блок	Кол-во	Примечание
ЦП	2	Раздельная память программ, связь между ЦП через ШП (или разработанный ранее)
Сопроцессор	2	По одному для каждого процессорного ядра
ускоритель вычислений	8	Общение через ШП
UART интерфейс	4	Приемник и передатчик
Спец. интерфейс	4	Приемник и передатчик
ШИМ контроллер	8	
Контроллер выходов	2	2 порта по 16 бит
Контроллер входа	2	2 порта по 16 бит
Таймер счетчик *	2	16-битный и 32-битный
Сторожевой таймер	1	
Специальный блок	1	Система из первого практического занятия

Контрольные вопросы

1. Что такое СнК? В чем отличие от системы на плате?
2. Что может входить в состав СнК?
3. Что такое сторожевой таймер в микроконтроллере, варианты реализации в синтезируемой СнК?
4. Зачем нужен таймер-счетчик? Назвать область применения с примерами. Объяснить, что такое прерывание по таймеру. Какие существуют возможности у аппаратных таймеров-счетчиков в современных МК?
5. На уровне блок-схем предложить вариант применения разработанной СнК в системах управления.

СПИСОК ЛИТЕРАТУРЫ

1. Смирнов, Д.С. Основы разработки встраиваемых систем на ПЛИС с использованием процессора NIOS II: учебное пособие / Д.С. Смирнов, И.Г. Дейнека, А.С. Алейник, И.А. Шарков. – Санкт-Петербург, 2019. – С. 97.

2. Строганов, А.В. Системное проектирование программируемых логических интегральных схем: учебное пособие / А.В. Строганов. – Воронеж: ФГБОУ ВПО «Воронежский государственный технический университет», 2012. – 322 с.

3. Воронов, К.Е. Разработка бортового модуля управления на базе вычислительного IP-ядра / К.Е. Воронов, К.И. Сухачев, Д.С. Воробьев // Ракетно-космическое приборостроение и информационные системы. – 2021. – №1. – Т. 8. С. 24–38.

4. Кастеншмидт, Ф. Плис и параллельные архитектуры для применения в аэрокосмической области. Программные ошибки и отказоустойчивое проектирование / Ф. Кастеншмидт, П. Реха. – Москва: Техносфера, 2018. – С. 312.

5. АО «Воронежский Завод Полупроводниковых Приборов – Сборка». Каталог изделий 2020 г. – URL: http://www.vzpp-s.ru/production/catalog.pdf (дата обращения 11.02.2021).

6. Cyclone V Device Handbook. – URL: https://www.intel.com-/content/dam/www/programmable/us/en/pdfs/literature/hb/cy-clonev/cv_5v2.pdf.

7. Зоев, И.В. Устройство на основе плис для распознавания рукописных цифр на изображениях / И.В. Зоев, А.П. Береснев, Н.Г. Марков, А.Н. Мальчуков // Компьютерная оптика – 2017. – Т. 41. – №6. – С. 938–949.

8. Николенко, С. Глубокое обучение, погружение в мир нейронных сетей / С. Николенко, А. Кадурин, Е. Архангельская. – Издательство «Питер», 2018.

9. Standard Verilog Hardware description language. URL: https://inst.eecs.-berkeley.edu/~cs150/fa06/Labs/verilog-ieee.pdf.

10. Стешенко, В.Б. Основы HDL Verilog как средства проектирования цифровых устройств: учебное пособие / В.Б. Стешенко,

Т.В. Попова, Д.Б. Малашевич // Московский государственный институт электронной техники. – Москва, 2006. – С. 95.

11. Поляков, А.К. Языки VHDL и Verilog в проектировании цифровой аппаратуры / А.К. Поляков. – Москва: Издательство «Солон-пресс», 2003. – С. 315.

12. Акчурин, А.Д. Программирование на языке Verilog: учебное пособие / А.Д. Акчурин., К.М. Юсупов // Казанский федеральный университет. Институт физики. – Казань, 2016.

13. Акчурин, А.Д. Основы работы в среде Quartus II: учебно-методическое пособие / А.Д. Акчурин., К.М. Юсупов, Колчев А.А. // Казанский федеральный университет. Институт физики. – Казань, 2017.

14. Ефремов, Н.В. Введение в систему автоматизированного проектирования QuartusII: учебное пособие / Н.В.Ефремов // Издательство Московского государственного университета леса, 2011. – С. 148.

15. ЯЗЫК AHDL. URL: https://studylib.ru/doc/148582/yazyk-ahdl.

16. Timing Analysis Overview, Quartus II Handbook Volume 3: Verification. URL: https://www.intel.com/content/dam/www/programmable/-us/en/pdfs/literature/hb/qts/qts_qii53030.pdf.

17. Антонов, А.П. Основы временного анализа при проектировании в Quartus Prime: учебно-методические материалы / А.П. Антонов, С.И. Кошелев, А.А. Федотов, А.С. Филиппов // Санкт-Петербургский политехнический университет Петра Великого, Институт компьютерных наук и технологий, Высшая школа интеллектуальных систем и суперкомпьютерных технологий. – Санкт-Петербург, 2021.

18. URL: https://habr.com/ru/post/352276/.

19. Лупал, А.М. Теория автоматов: учебное пособие / А.М. Лупал // СПбГУАП. – Санкт-Петербург, 2000 – 119 с.

20. Микросхемы памяти АО «ПКК Миландр». – URL: https://ic.-milandr.ru/products/mikroskhemy_pamyati/.

Приложение А
Описание периферийных модулей на Verilog

Модуль, реализующий I2C интерфейс:

```
// I2C module
developed by Sukhachev K.I.*/
module SMP_I2C_V
(CLK, reset, run,
str, stp,
DTS, DFS, ready, full,
SDA, SDA_in, SCL);

parameter URL=20;//500; // fCLK/100_000

input wire CLK, reset, run;
input wire str, stp;
output reg ready; output reg full;
input wire [7:0] DTS;
output reg [7:0] DFS;
output wire SDA, SCL; assign SDA=fSDA; assign SCL=fSCL;
input wire SDA_in;

reg [8:0] trans; reg [7:0] resiv;
reg action, transmission, future, start, stop, first;
reg fSDA, fSCL;
reg [15:0] counter; reg [3:0] bitcounter;

always @(posedge CLK or posedge reset) begin
        if (reset) begin
                fSDA<=1'd1; fSCL<=1'd1;
                counter<=0; bitcounter<=0;
                action<=0; transmission<=0; future<=0; start<=0; stop<=0; first<=0;
                trans<=0;  resiv<=0;
                ready<=1'd1; full<=0; DFS<=0;
        end
        else begin
        if (str) begin
                start<=1'd1;
                stop<=0;
                full<=0;
```

```
        ready<=0;
        first<=1'd1;
end
if (stp) begin
        start<=0;
        stop<=1'd1;
        full<=0;
        ready<=0;
        first<=0;
end
if (run&!start&!stop) begin
        full<=0;
        ready<=0;
        trans[8:1]<=DTS[7:0];
        action<=1'd1;
        resiv<=0;
        if (first) begin
                future<=!DTS[0];
        end
end // run
        if (start) begin
                case (counter)
                8*URL:          begin
                                fSDA<=0;
                                counter<=counter+1'd1;
                                end
                10*URL:         begin
                                fSCL<=0;
                                counter<=counter+1'd1;
                                end
                12*URL+1:       begin
                                transmission<=1'd1;
                                counter<=0;
                                start<=0;
                                ready<=1'd1;
                                end
                default:        counter<=counter+1'd1;
                endcase
        end
        if (stop) begin
                case (counter)
                0:              begin
```

```
                counter<=counter+1'd1;
                fSDA<=0;
            end
        2*URL:  begin
                fSCL<=1'd1;
                counter<=counter+1'd1;
            end
        4*URL:  begin
                fSDA<=1'd1;
                counter<=counter+1'd1;
            end
        20*URL: begin
                counter<=0;
                stop<=0;
                ready<=1'd1;
                first<=1'd1;;
                future<=0;
            end
        default: counter<=counter+1'd1;
        endcase
end
if (action) begin
        case (counter)
        0:  counter<=counter+1'd1;
        URL: begin
                if (transmission) begin
                    fSDA<=trans[4'd8-bitcounter];
                    if (bitcounter==4'd8) begin
                        fSDA<=1'd1;
                    end
                end
                else begin
                    if (bitcounter==4'd8) begin
                        fSDA<=0;
                    end
                    else begin
                        fSDA<=1'd1;

                    end
                end
                counter<=counter+1'd1;
            end
```

```
                2*URL:      begin
                            fSCL<=1'd1;
                            counter<=counter+1'd1;
                        end
                3*URL:      begin
                            if (!transmission) begin
                                if (bitcounter!=4'd8) begin
                                    resiv[4'd7-
bitcounter]<=SDA_in;
                                end
                            end
                            counter<=counter+1'd1;
                        end
                4*URL:      begin
                            fSCL<=0;
                            if (bitcounter!=4'd8) begin
                                bitcounter<=bitcounter+1'd1;
                                counter<=0;
                            end
                            else begin
                                counter<=counter+1'd1;
                                if (!transmission) begin
                                    DFS<=resiv;
                                    fSDA<=0;
                                end
                                else begin
                                    fSDA<=1'd1;
                                end
                                bitcounter<=0;
                                if (first) begin
                                    first<=0;
                                    transmission<=future;
                                end
                            end
                        end
                4*URL+1:    begin
                                if (first|transmission) begin
                                    ready<=1'd1;
                                end
                                else begin
                                    ready<=1'd1;
                                    full<=1'd1;
```

```
                                        end
                                        action<=0;
                                        full<=1'd1;
                                        counter<=0;
                                    end
                            default:    counter<=counter+1'd1;
                            endcase
                        end // action
                end // CLK
        end // always
        endmodule
```

Модули приемника и передатчика, реализующие помехоустойчивый интерфейс с манчестерским кодированием:

```
        /* SINT_TR: (the simplest interface)
        developed by Sukhachev K.I.*/
        module SINT_TR
        ( CLK, reset,
        data, set,
        ready,
        TX);
        parameter t = 25;
        input wire CLK, reset;
        // PRH SMK bus:
        input wire [15:0] data;
        input wire set;
        // Flags:
        reg busy;
        output wire ready; assign ready=!busy;
        output reg TX;
        // translator state machine element:
        reg [4:0] bitcounter;
        reg [10:0] count_clk;
        reg [24:0] databuf;
        // CRC elements of state machines:
        reg [4:0] CRCcount;
        reg [7:0] CRC;
        // sector of synchronous logic, synchronism along the front CLK:
        always @(posedge CLK or posedge reset) begin
                if (reset) begin
```

```
                TX<=0; busy<=0;
                count_clk<=0; bitcounter<=0; databuf<=0;
                CRCcount<=0; CRC<=0;
        end // reset
        else begin
                if (set) begin
                        databuf[16:1]<=data[15:0];
                        busy<=1'd1;;
                        databuf[24:17]<=0; CRCcount<=0; CRC<=0;
                        databuf[0]<=1'd1;
                end
                else begin
                        if (busy) begin
                                count_clk<=count_clk+1'd1;
                                case (count_clk)
                                0:              begin
                                                        if (databuf[bitcounter]==1'd1)
begin
                                                                TX<=1;
                                                end // bit = 1
                                                else begin
                                                        TX<=0;
                                                end
                                        end // count_clk=0
                                t:              begin
                                                        if (databuf[bitcounter]==1'd1)
begin
                                                                TX<=0;
                                                end // bit = 1
                                                else begin
                                                        TX<=1'd1;
                                                end
                                        end // count_clk=t/2
                                t+t:    begin
                                                        if (bitcounter<5'd24) begin

        bitcounter<=bitcounter+1'd1;
                                                                count_clk<=0;
                                                end
                                                else begin
                                                        count_clk<=0;
                                                        bitcounter<=0;
```

```
                                                busy<=0;
                                                databuf<=0;
                                                TX<=0;
                                        end
                                end // count_clk=t
                        default:;
                        endcase // count_clk
        ////////////////// CRC8 //// begin
                        case (CRCcount)
                        5'd0:   CRCcount<=CRCcount+1'd1;
                        5'd17:  begin
                                        CRCcount<=CRCcount+1'd1;
                                        databuf[24:17]<=CRC[7:0];
                                end
                        5'd18:  CRCcount<=CRCcount;
                        default:        begin
                                        CRCcount<=CRCcount+1'd1;
                                        CRC[0]<=CRC[7]^databuf[CRC-
count];
                                        CRC[1]<=CRC[0];
                                        CRC[2]<=CRC[1];
                                        CRC[3]<=CRC[7]^CRC[2];
                                        CRC[4]<=CRC[7]^CRC[3];
                                        CRC[5]<=CRC[4];
                                        CRC[6]<=CRC[5];
                                        CRC[7]<=CRC[6];
                                end
                        endcase  // CRCcount
        ////////////////// CRC8 //// end
                                end // busy
                        end // WEP
                end // CLK
        end // always
        endmodule
```

Приемник:

```
        /* SINT_RS: (the simplest interface)
        developed by Sukhachev K.I.*/
        module SINT_RS
        (
        CLK, reset,
```

```
Dataout, full, Error,
//busy,RS_complit, NO_ERROR,
RX
);
parameter t=25;
parameter stops=3000;
localparam tt=t+(t/2);
input wire CLK, reset, RX;
// Flags:
reg busy;
reg RS_complit;
output wire full; assign full=RS_complit;
output reg [15:0] Dataout;
reg NO_ERROR;
output wire Error; assign Error=!NO_ERROR;
// translator state machine element:
reg [7:0] bitcounter;
reg [10:0] count_clk;
reg [23:0] databuf;
reg flag;
reg PR_bit;
reg process;
reg [11:0] stoper;
reg [1:0] FSM;
// CRC elements of state machines:
wire [7:0] CRCt;
reg [7:0] CRC;
assign CRCt[7:0] = databuf[23:16];
// sector of synchronous logic, synchronism along the front CLK:
always @(posedge CLK or posedge reset) begin
        if (reset) begin
                Dataout<=0;
                busy<=0; RS_complit<=0;
                count_clk<=0; bitcounter<=0; databuf<=0;
                busy<=0; flag<=0; PR_bit<=0; process<=0;
                CRC<=0;
        end // reset
        else begin
                        if (!RX) begin
                                if (stoper==stops) begin
                                        count_clk<=0;
                                        bitcounter<=0;
```

```
                busy<=0;
                PR_bit<=0;
                process<=0;
                flag<=0;
                FSM<=0;
                CRC<=0;
                NO_ERROR<=0;
                RS_complit<=0;
            end
            else begin
                stoper<=stoper+1'd1;
            end
        end
        else begin
            stoper<=0;
        end

if (busy==0) begin
        if (RX) begin
            busy<=1'd1;
            PR_bit<=1'd1;
            RS_complit<=0;
            NO_ERROR<=0;
        end
end // NO busy
else begin
        if (flag!=RX) begin
                case (FSM)
                2'd0:   begin
                            if (flag) begin
                                FSM<=2'd1;
                            end
                            else begin
                                FSM<=2'd2;
                            end
                        end
                2'd1:   begin
                            if (PR_bit) begin
                                process<=1'd1;
                            end
                            FSM<=2'd3;
                        end
```

```
                2'd2:   begin
                            if (PR_bit==0) begin
                                process<=1'd1;
                            end
                            FSM<=2'd3;
                        end
                2'd3:   begin
                            FSM<=0;
                            flag<=RX;
                        end
                default:;
                endcase
        end // flag!=RX
        if (process) begin
            count_clk<=count_clk+1'd1;
            case (count_clk)
            tt:     begin
                        PR_bit<=RX;
                        databuf[bitcounter]<=RX;
                    end
            tt+1:   begin
                        if (bitcounter<5'd16) begin
                            CRC[0]<=CRC[7]^PR_bit;
                            CRC[1]<=CRC[0];
                            CRC[2]<=CRC[1];
                            CRC[3]<=CRC[7]^CRC[2];
                            CRC[4]<=CRC[7]^CRC[3];
                            CRC[5]<=CRC[4];
                            CRC[6]<=CRC[5];
                            CRC[7]<=CRC[6];
                        end
                    end
            tt+2:   begin
                        if (bitcounter<5'd24) begin
                            count_clk<=0;
                            bitcounter<=bitcounter+1'd1;
                            process<=0;
                        end
                        else begin
                            count_clk<=count_clk+1'd1;
                        end
                    end
```

```
                        tt+3:   begin
                                    if (CRC==CRCt) begin
                                        NO_ERROR<=1'd1;
                                    end
                                    Dataout[15:0]<=databuf[15:0];
                                end
                        tt+4:   begin
                                    RS_complit<=1'd1;
                                    count_clk<=0;
                                    bitcounter<=0;
                                    busy<=0;
                                    PR_bit<=0;
                                    process<=0;
                                    flag<=0;
                                    FSM<=0;
                                    CRC<=0;
                                end
                        default:;
                        endcase
                end // process
        end // busy
end // CLK
end // always
endmodule // SINT_RS
```

Модуль SPI:

```
/*SPI tr/rs developed by Sukhachev K.I.*/
module SPI
(CLK, reset,
adr, data, set,
ready, Dataout,
MOSI, MISO, SCK);
parameter add_set = 16'd40;
parameter add_out = 16'd44;

input wire CLK, reset, set, MISO;
input wire [15:0] adr; input wire [15:0] data;
output reg  MOSI, SCK;
output wire ready; assign ready = !busy;
output reg [7:0] Dataout;
reg [9:0] URT;
```

```
reg busy;
reg [7:0] bufdata_out; reg [7:0] bufdata_inp;
reg [10:0] maincounter;
reg [3:0] bitcounter;
wire [8:0] t; assign t[8:0]=URT [9:1];
always @(posedge CLK or posedge reset) begin
if (reset) begin
        busy<=0;  MOSI<=0; SCK<=0;
        URT<=0;
        bufdata_out<=0; bufdata_inp<=0;
        maincounter <=0; bitcounter<=0;
end // reset
else begin
        if (set) begin
                bufdata_inp<=0;
                maincounter<=0;
                bitcounter<=0;
                case (adr)
                add_set:        begin
                                URT[9:0]<=data[9:0];
                                busy<=!data[15];
                        end
                add_out:        begin
                                bufdata_out[7:0]<=data[7:0];
                                busy<=1'd1;
                        end
                default:;
                endcase
        end // set
                if (busy) begin
                        case (maincounter)
                        10'd1:          begin
                                                MOSI<=bufdata_out[bitcounter];
                                                maincounter<=maincounter+1'd1;
                                        end
                        t:              begin
                                                SCK<=1'd1;
                                                        maincounter<=maincounter+1'd1;
                                                end
                        URT:                    begin
                                                        SCK<=0;
                                                        maincounter<=0;
```

```
                                        bufdata_inp[bitcounter]<=MISO;
                                        if (bitcounter==4'd7) begin
                                                busy<=0;
                                                bitcounter<=0;
                                                Dataout<=bufdata_inp;
                                        end
                                        else begin

bitcounter<=bitcounter+1'd1;
                                        end // bitcounter<8
                                end
                        default:        maincounter<=maincounter+1'd1;
                        endcase
                end
end // CLK
end // always
endmodule // SPI_16
```

Приложение Б
Описание процессорного ядра NMR на Verilog

```
        /* developed by Sukhachev K.I. for IKP-214 and education if Samara University/ 2020
*/
        module Nightmare // процессорное ядро
        (CLK,reset,                     //тактовый сигнал и сброс
        ATA, DTA, DFA, WEA,             //шина подключения блок регистров А
        ATB, DTB, DFB, WEB,             //шина подключения блок регистров Б
        ATR, DTR, DFR, WER,             //шина подключения внутреннего ОЗУ
        ATP, DTP, DFP, WEP,             //PRH, шина периферии HAVOC
        DTD, DFD, SLKCD, math,  //DSP, шина подключения сопроцессора
        ADR,                            //доп. регистр, может быть использован для адреса-
ции внешней памяти
        status,                         //регистр статуса
        instruction, inst_counter,              //шина памяти программ
        interruption, branch, vector,  //сигналы, идущие к контроллеру прерываний
        GPI1, GPI2,                     //порты ввода
        FSM );                          //главный автомат состояний
        input wire CLK, reset;                  //глобальные сигналы
        // ## сигналы блока РОН А:
        output reg [7:0] ATA;                   // адрес ячейки блока РОН А
        output reg [31:0] DTA;                  // информация В РОН А
        input wire [31:0] DFA;                  // информация ИЗ РОН А
        output reg WEA;                         // разрешение записи в РОН А
        // ## сигналы блока РОН Б:
        output reg [7:0] ATB;                   // адрес ячейки блока РОН Б
        output reg [31:0] DTB;                  // информация В РОН Б
        input wire [31:0] DFB;                  // информация ИЗ РОН Б
        output reg WEB;                         // разрешение записи в РОН Б
        // ## сигналы блока внутреннего ОЗУ:
        output reg [15:0] ATR;                  // адрес ячейки блока внутреннего ОЗУ (КЕШ
ОЗУ)
        output reg [15:0] DTR;                  // информация для записи В ОЗУ
        input wire [15:0] DFR;                  // информация ИЗ ОЗУ
        output reg WER;                         // разрешение записи в ОЗУ
        // ## шина периферии HAVOC:
        output reg [15:0] ATP;                  // адрес устройства на шине периферии
HAVOC
        output reg [15:0] DTP;                  // информация от ядра В периферию
        input wire [15:0] DFP;                  // информация ИЗ периферии
```

```
        output reg WEP;                 // разрешение записи в HAVOC
        // ## сигналы для подключения сопроцессора:
        output reg [15:0] DTD;          // информация В сопроцессор
        input wire [15:0] DFD;          // информация ИЗ сопроцессора
        output reg SLKCD;           // сигнал тактирования сопроцессора
        output reg math;                // флаг направления данных и доступа
        // ## внешние регистры:
        output reg [31:0] ADR;          // доп. регистр
        output reg [31:0] status;       // регистр статуса, задействован в программ-
ных прерываниях
        // ## шина памяти программ:
        input wire [15:0] instruction;      // входная 16-битная шина инструкций (ко-
манд) от памяти программ
        output reg [31:0] inst_counter;     // счетчик инструкций, он же адрес обращения
к памяти программ
        // ## сигналы и регистры обработки прерывания:
        input wire [9:0] interruption;      // входная внешняя шина прерываний
        wire [9:0] inter_interruption;      // внутренняя шина прерываний (связана с ре-
гистром статуса)
        assign inter_interruption[7:0]=interruption[7:0]|status[7:0];
        assign inter_interruption[9:8]=interruption[9:8];
        output reg branch;    //флаг, нахождения в прерывании
        output reg [9:0] vector;    //внешний сигнал, вектора прерывания
        reg [31:0] counter_buf;     // буфер для хранения состояния счетчика инструк-
ций при уходе в прерывные
        // ## входные порты:
        input wire [31:0] GPI1;     // первый внешний порт, доступный для считывания
в РОН
        input wire [31:0] GPI2;     // второй внешний порт, доступный для считывания в
РОН
        // ## основные регистры хранения инструкций и данных ПП, главный автомат
состояний:
        output reg [3:0] FSM;       // основной конечный автомат, необязательно назна-
чать выводом
        reg [15:0] inst_buf;    // буфер для хранения захваченных данных от памяти про-
грамм
        reg [15:0] data_buf;   // буфер для хранения захваченных данных от памяти про-
грамм
        reg [7:0] buf_logick;  // регистр для работы с инструкциями логических операций
        // ## дополнительные регистры:
        reg [9:0] FSM_2;// регистр для работы с сопроцессором, а также для потокового
чтения и записи106/107
```

```
reg [15:0] MFC;        // многофункциональный счетчик
reg [31:0] delay;      // счетчик для инструкции delay
// ## установка первоначальных состояний (not avalable for 5578):
initial begin
        ATA<=0;     DTA<=0;     WEA<=0;
        ATB<=0;     DTB<=0;     WEB<=0;
        ATR<=0;     DTR<=0;     WER<=0;
        ATP<=0;     DTP<=0;     WEP<=0;
        DTD<=0;     SLKCD<=0;
        ADR<=0;
        status<=0;
        inst_counter<=0;
        branch<=0;
        vector<=0;
        counter_buf<=0;
        FSM<=0;
        inst_buf<=0; data_buf<=0; buf_logick<=0;
        FSM_2<=0; delay<=0;
end
always @ (posedge CLK or posedge reset) begin
        if (reset) begin
                ATA<=0;     DTA<=0;     WEA<=0;
                ATB<=0;     DTB<=0;     WEB<=0;
                ATR<=0;     DTR<=0;     WER<=0;
                ATP<=0;     DTP<=0;     WEP<=0;
                ADR<=0;
                DTD<=0;     SLKCD<=0;
                inst_counter<=0;
                branch<=0; counter_buf<=0; inst_buf<=0; vector<=0; data_buf<=0;
                buf_logick<=0;
                FSM<=0;     FSM_2<=0;
        end //reset.
        else begin
case (FSM)
//################ главный автомат состояний FSM=0:
4'd0:   begin
// сброс регистров разрешения записи:
                        WEA<=0;     // разрешение записи по шине РОН A.
                        WEB<=0;     // разрешение записи по шине РОН B.
                        WEP<=0;     // разрешение записи по шине HAVOC.
                        WER<=0;     // разрешение записи по шине внутрен-
```
него ОЗУ.

```
                                    SLKCD<=0;  // тактирование шины сопроцессора.
        // работа с прерыванием, определение вектора перехода, обработка нулевого адреса ПП:
        case (inter_interruption)
        10'd0:  begin  // нет прерываний
                if (inst_counter[31:0]==0) begin      // нулевой адрес (запуск процессора)
                        FSM<=4'd10; end
                else begin                      // адрес не нулевой
                        FSM<=4'd2;   end
                end
        default:        begin  // обнаружен запрос на прерывание
                        if (branch==0) begin  // процессор не обрабатывает прерывание
                                FSM<=FSM+1'd1;
                                counter_buf[31:0]<=inst_counter[31:0];      // сохранение счетчика инструкции
                                branch<=1'd1;//выставление флага ухода в прерывание.
                        end      // branch==0:
                        else begin              // процессор уже находится в прерывании
                                FSM<=4'd2;
                        end     // branch==1:
                end // default
        endcase // inter_interruption (обработка внутреннего сигнала прерывания)
        end // главный автомат состояний FSM=0
        //################ главный автомат состояний FSM=1:
        4'd1:   begin
                        FSM<=4'd7;                              //пропуск такта
                        inst_counter[31:10]<=0;                         //обнуление старших разрядов счетчика инстр.
                        inst_counter[9:0]<=inter_interruption[9:0]; //переход на ячейку памяти программ по вектору
                        vector[9:0]<=inter_interruption[9:0];         //внешний вывод информационный (тестовый)
        end // главный автомат состояний FSM=1
        //first cycle processing main script (determinete of instruction): FSM=2>>
        //################ главный автомат состояний FSM=2:
        4'd2:   begin
                        inst_buf[15:0]<=instruction[15:0];   // сохранение первых 16 бит инструкций
                        inst_counter<=inst_counter+1'd1;    // указатель адреса инкрементируется
        //первый этап обработки полученных инструкций:
                        case (instruction[15:8])                // анализ команды
```

```
                8'd0:   begin
                            FSM<=0;
                        end
// запись в РСН адресов блоков РОН А и РОН Б:
                8'd1:   begin
                            FSM<=4'd15;
                            ATA[7:0]<=instruction[7:0];
                        end // ATA<=8'd255
                8'd2:   begin
                            FSM<=4'd15;
                            ATB[7:0]<=instruction[7:0];
                        end // ATB<=8'd255
// запись в блок РОН А из ОЗУ по заранее установленным адресам:
                8'd4:   begin
                            FSM<=4'd3;
                            case (instruction[7:0])
                            8'd1:   DTA[15:0]<=DFR[15:0];
                            8'd2:   begin
                                        DTA[15:0]<=DFR[15:0];
                                        ATR<=ATR+1'd1;
                                    end
                            8'd3:   begin
                                        DTA[15:0]<=DFR[15:0];
                                        ATR<=ATR+1'd1;
                                        ATA<=ATA+1'd1;
                                    end
                            8'd4:   DTA[15:0]<=DFP[15:0];
                            8'd5:   begin
                                        DTA[15:0]<=DFP[15:0];
                                        ATP<=ATP+1'd1;
                                    end
                            8'd6:   begin
                                        DTA[15:0]<=DFP[15:0];
                                        ATP<=ATP+1'd1;
                                        ATA<=ATA+1'd1;
                                    end
                            default:;
                            endcase
                        end // (4.1) A*<=RAM*/PRH*
// запись в блок РОН Б из ОЗУ по заранее установленным адресам:
                8'd5:   begin
                            FSM<=4'd3;
```

```
                    case (instruction[7:0])
                    8'd1:   DTB[15:0]<=DFR[15:0];
                    8'd2:   begin
                                DTB[15:0]<=DFR[15:0];
                                ATR<=ATR+1'd1;
                            end
                    8'd3:   begin
                                DTB[15:0]<=DFR[15:0];
                                ATR<=ATR+1'd1;
                                ATB<=ATB+1'd1;
                            end
                    8'd4:   DTB[15:0]<=DFP[15:0];
                    8'd5:   begin
                                DTB[15:0]<=DFP[15:0];
                                ATP<=ATP+1'd1;
                            end
                    8'd6:   begin
                                DTB[15:0]<=DFP[15:0];
                                ATP<=ATP+1'd1;
                                ATB<=ATB+1'd1;
                            end
                    default:;
                    endcase
                end // (5.1) B*<=RAM*/PRH*
// запись в ОЗУ по заранее предустановленным адресам:
                8'd6:   begin
                            FSM<=4'd3;
                            case (instruction[7:0])
                            8'd1:   DTR[15:0]<=DFA[15:0];
                            8'd2:   begin
                                        DTR[15:0]<=DFA[15:0];
                                        ATR<=ATR+1'd1;
                                    end
                            8'd3:   begin
                                        DTR[15:0]<=DFA[15:0];
                                        ATR<=ATR+1'd1;
                                        ATA<=ATA+1'd1;
                                    end
                            8'd4:   DTR[15:0]<=DFB[15:0];
                            8'd5:   begin
                                        DTR[15:0]<=DFB[15:0];
                                        ATR<=ATR+1'd1;
```

```
                    end
                8'd6:   begin
                            DTR[15:0]<=DFB[15:0];
                            ATR<=ATR+1'd1;
                            ATB<=ATB+1'd1;
                        end
                8'd7:   DTR[15:0]<=DFP[15:0];
                8'd8:   begin
                            DTR[15:0]<=DFP[15:0];
                            ATR<=ATR+1'd1;
                        end
                default:;
                endcase
            end // (6.1) RAM*<=A*/B*/PRH
// запись в Периферию по заранее предустановленным адресам:
            8'd7:   begin
                        FSM<=4'd3;
                        case (instruction[7:0])
                        8'd1:   DTP[15:0]<=DFA[15:0];
                        8'd2:   begin
                                    DTP[15:0]<=DFA[15:0];
                                    ATP<=ATP+1'd1;
                                end
                        8'd3:   begin
                                    DTP[15:0]<=DFA[15:0];
                                    ATP<=ATP+1'd1;
                                    ATA<=ATA+1'd1;
                                end
                        8'd4:   DTP[15:0]<=DFB[15:0];
                        8'd5:   begin
                                    DTP[15:0]<=DFB[15:0];
                                    ATP<=ATP+1'd1;
                                end
                        8'd6:   begin
                                    DTP[15:0]<=DFB[15:0];
                                    ATP<=ATP+1'd1;
                                    ATB<=ATB+1'd1;
                                end
                        8'd7:   DTP[15:0]<=DFR[15:0];
                        8'd8:   begin
                                    DTP[15:0]<=DFR[15:0];
                                    ATP<=ATP+1'd1;
```

```
                    end
                default:;
                endcase
            end // (7.1) PRH*<=A*/B*/RAM
        // набор инструкций, связанных с первоначальной установкой адреса РОН Б:
        8'd8, 8'd16, 8'd18, 8'd21, 8'd22, 8'd24, 8'd30, 8'd32, 8'd34, 8'd43, 8'd61, 8'd63, 8'd65,
8'd67, 8'd81, 8'd83, 8'd86, 8'd87, 8'd88, 8'd90, 8'd111:    begin
                    FSM<=4'd3;
                    ATB[7:0]<=instruction[7:0];
                end // ATB<=instruction
        // набор инструкций, связанных с первоначальной установкой адреса РОН А:
        8'd9, 8'd10, 8'd11, 8'd15, 8'd17, 8'd19, 8'd20, 8'd23, 8'd29, 8'd31, 8'd33, 8'd44, 8'd46,
8'd47, 8'd48, 8'd49, 8'd60, 8'd62, 8'd60, 8'd64, 8'd66, 8'd80, 8'd82, 8'd84, 8'd85, 8'd89, 8'd91,
8'd92, 8'd95, 8'd96, 8'd100, 8'd101, 8'd110, 8'd125:    begin
                    FSM<=4'd3;
                    ATA[7:0]<=instruction[7:0];
                end // ATA<=instruction
        // набор инструкций, связанных с первоначальной установкой адреса младшего
сектора ОЗУ:
        8'd27, 8'd45:    begin
                    FSM<=4'd3;
                    ATR[15:8]<=0;
                    ATR[7:0]<=instruction[7:0];
                end // ATR<=instruction
        // запись в младшую область  Периферии  с адресом (от 0 по 255) из ПП, 16-бит-
ного числа из ПП:
        8'd28:  begin
                    FSM<=4'd3;
                    ATP[15:8]<=0;
                    ATP[7:0]<=instruction[7:0];
                end // PRH255<=16'd65535
        // команда записи текущего значения счетчика инструкций в РОН А или РОН Б
или ОЗУ:
        8'd40:  begin
                    FSM<=4'd6;
                    ATA[7:0]<=instruction[7:0];
                    DTA[31:0]<=inst_counter[31:0]+1'd1;
                end // marck A255
        8'd41:  begin
                    FSM<=4'd6;
                    ATB[7:0]<=instruction[7:0];
                    DTB[31:0]<=inst_counter[31:0]+1'd1;
```

```
            end // marck B255
        8'd42:  begin
                    FSM<=4'd5;
                    ATR[15:8]<=0;
                    ATR[7:0]<=instruction[7:0];
                    DTR[15:0]<=inst_counter[15:0]+1'd1;
        end // marck RAM255

        // логические операции:
        8'd68, 8'd69, 8'd70, 8'd71, 8'd72, 8'd73, 8'd74, 8'd75, 8'd76, 8'd77, 8'd78, 8'd79:
                    begin
                    FSM<=4'd3;
                    buf_logick[7:0]<=instruction[7:0]; // target address storage buffer for
logick and math
        end // A255<=A254 ??? B253 // B255<=A254 ??? B253
        // увеличение значения во дополнительном регистре ADR на число из ПП:
        8'd99:  begin
                    FSM<=0;
                    ADR[31:0]<=ADR[31:0]+instruction[7:0];
            end // adr+8'd255
        // спящий режим до пробуждения по прерыванию:

        8'd126: begin
                    if (interruption!=10'd0) begin
                        FSM<=0;
                    end
            end // sleep
        //завершение обработки текущего прерывания:
        8'd127:     begin
                    FSM<=4'd15;
                    branch<=0;
                    vector[9:0]<=0;
                    inst_counter[31:0]<=counter_buf[31:0];
            end // intoff
        default: FSM<=4'd3; // если код команды не распознан, переход на следующее
состояние FSM
        endcase  // анализ команды
        end // главный автомат состояний FSM=2
        //################ главный автомат состояний FSM=3:
        4'd3:   begin
            case (inst_buf[15:8])
        // запись в блок регистров A по заранее установленным адресам:
```

```
        8'd4:   begin
                    FSM<=0;
                    WEA<=1'd1;
                end // (4.2) A*<=RAM*/PRH*
// запись в блок регистров Б по заранее установленным адресам:
        8'd5:   begin
                    FSM<=0;
                    WEB<=1'd1;
                end // (5.2) B*<=RAM*/PRH*
// запись в ОЗУ по заранее предустановленным адресам:
        8'd6:   begin
                    FSM<=0;
                    WER<=1'd1;
                end // (6.2) RAM*<=A*/B*/PRH
// запись в Периферию по заранее предустановленным адресам:
        8'd7:   begin
                    FSM<=0;
                    WEP<=1'd1;
                end // (7.2) RAM*<=A*/B*/PRH
        8'd8:   begin
                    FSM<=0;
                    ATA[7:0]<=DFB[7:0];
                end // (8.2) ATA<=B255
        8'd9:   begin
                    FSM<=0;
                    ATB[7:0]<=DFA[7:0];
                end // (9.2) ATB<=A255
        8'd10:  begin
                    FSM<=0;
                    ATR[15:0]<=DFA[15:0];
                end // (10.2) ATR<=A255
        8'd11:  begin
                    FSM<=0;
                    ATP[15:0]<=DFA[15:0];
                end // (11.2) ATR<=A255
// Фрагмент обработки инструкции перемещения:
        8'd15:  begin
                    FSM<=4'd4;
                    DTR[15:0]<=DFA[15:0];
                end // (15.2) RAM65535<=A255
        8'd16:  begin
                    FSM<=4'd4;
```

```
                DTR[15:0]<=DFB[15:0];
            end // (16.2) RAM65535<=B255
        8'd17: begin
                FSM<=4'd4;
                DTP[15:0]<=DFA[15:0];
            end // (17.2) PRH65535<=A255
        8'd18: begin
                FSM<=4'd4;
                DTP[15:0]<=DFB[15:0];
            end // (18.2) PRH65535<=B255
        8'd23: begin
                FSM<=4'd4;
                DTB[31:0]<=DFA[31:0];
            end // (23.2) B255<=A254
        8'd24: begin
                FSM<=4'd4;
                DTA[31:0]<=DFB[31:0];
            end // (24.2) A255<=B254
        8'd31: begin
                FSM<=4'd6;
                DTA[15:0]<=DFA[31:16];
                DTA[31:16]<=DFA[15:0];
            end // A255><
        8'd32: begin
                FSM<=4'd6;
                DTB[15:0]<=DFB[31:16];
                DTB[31:16]<=DFB[15:0];
            end // B255><
        8'd33: begin
                FSM<=4'd4;
                DTA[31:0]<=DFA[31:0];
            end // (33.2) A255<=A254
        8'd34: begin
                FSM<=4'd4;
                DTB[31:0]<=DFB[31:0];
            end // (34.2) B255<=B254
// Фрагмент обработки инструкций переходов:
        8'd43: begin
                FSM<=4'd15;
                inst_counter[31:0]<=DFA[31:0];
            end // (43.2) jamp A255
        8'd44: begin
```

```
                        FSM<=4'd15;
                        inst_counter[31:0]<=DFB[31:0];
                    end // (43.2) jamp A255
                8'd45:  begin
                        FSM<=4'd15;
                        inst_counter[31:16]<=0;
                        inst_counter[15:0]<=DFR[15:0];
                    end // (43.2) jamp A255
        // Фрагмент обработки инструкций логических операций:
                8'd64:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=DFA[31:0]+1'd1;
                    end // A255++
                8'd65:  begin
                        FSM<=4'd6;
                        DTB[31:0]<=DFB[31:0]+1'd1;
                    end // B255++
                8'd66:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=DFA[31:0]-1'd1;
                    end // A255--
                8'd67:  begin
                        FSM<=4'd6;
                        DTB[31:0]<=DFB[31:0]-1'd1;
                    end // B255--
                8'd80:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=!DFA[31:0];
                    end // notA255
                8'd81:  begin
                        FSM<=4'd6;
                        DTB[31:0]<=!DFB[31:0];
                    end // notB255
                8'd84:  begin
                        FSM<=4'd6;
        DTA[15]<=DFA[0]; DTA[14]<=DFA[1]; DTA[13]<=DFA[2]; DTA[12]<=DFA[3];
DTA[11]<=DFA[4];
        DTA[10]<=DFA[5]; DTA[9]<=DFA[6]; DTA[8]<=DFA[7];       DTA[7]<=DFA[8];
DTA[6]<=DFA[9];
        DTA[5]<=DFA[10]; DTA[4]<=DFA[11]; DTA[3]<=DFA[12]; DTA[2]<=DFA[13];
DTA[1]<=DFA[14];
        DTA[0]<=DFA[15];
```

```
                end // rotAL
        8'd85:  begin
                        FSM<=4'd6;
    DTA[31]<=DFA[16]; DTA[30]<=DFA[17]; DTA[29]<=DFA[18];
DTA[28]<=DFA[19]; DTA[27]<=DFA[20];
    DTA[26]<=DFA[21]; DTA[25]<=DFA[22]; DTA[24]<=DFA[23];
DTA[23]<=DFA[24]; DTA[22]<=DFA[25]; DTA[21]<=DFA[26]; DTA[20]<=DFA[27];
DTA[19]<=DFA[28]; DTA[18]<=DFA[29]; DTA[17]<=DFA[30];
    DTA[16]<=DFA[31];
                end // rotAH
        8'd88:  begin // инструкция исключена из набора команд для небольших
ПЛИС
                        FSM<=4'd6;
                        DTB[31]<=DFB[0]; DTB[30]<=DFB[1]; DTB[29]<=DFB[2];
DTB[28]<=DFB[3];
    DTB[27]<=DFB[4]; DTB[26]<=DFB[5]; DTB[25]<=DFB[6]; DTB[24]<=DFB[7];
DTB[23]<=DFB[8];
    DTB[22]<=DFB[9]; DTB[21]<=DFB[10]; DTB[20]<=DFB[11];
DTB[19]<=DFB[12]; DTB[18]<=DFB[13];
    DTB[17]<=DFA[14]; DTB[16]<=DFB[15]; DTA[15]<=DFA[16];
DTA[14]<=DFA[17]; DTA[13]<=DFA[18]; DTA[12]<=DFA[19]; DTA[11]<=DFA[20];
DTA[10]<=DFA[21]; DTA[9]<=DFA[22]; DTA[8]<=DFA[23];
    DTA[7]<=DFA[24]; DTA[6]<=DFA[25]; DTA[5]<=DFA[26]; DTA[4]<=DFA[27];
DTA[3]<=DFA[28]; DTA[2]<=DFA[29]; DTA[1]<=DFA[30]; DTA[0]<=DFA[31];
                end // rotBF
        8'd91:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=DFA[31:0]<<1;
                end // <<A255
        8'd92:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=DFA[31:0]>>1;
                end // >>A255
    // Фрагмент обработки дополнительных команд:
        8'd95:  begin
                        FSM<=0;
                        ADR[31:0]<=DFA[31:0];
                end // adr<=A255
        8'd96:  begin
                        FSM<=4'd6;
                        DTA[31:0]<=ADR[31:0];
                end // A255<=adr
```

```
8'd100:         begin
                        FSM<=0;
                        status[31:0]<=DFA[31:0];
                end // stat<=A255
        8'd101:         begin
                        FSM<=4'd6;
                        DTA[31:0]<=status[31:0];
                end // A255<=stat
        8'd110:         begin
                        FSM<=4'd6;
                        DTA[31:0]<=GPI1[31:0];
                end // A255<=GPI1
        8'd111:         begin
                        FSM<=4'd6;
                        DTB[31:0]<=GPI2[31:0];
                end // A255<=GPI2
        8'd125:         begin
                        case (delay[31:0])
                        DFA[31:0]:    begin
                                        FSM<=0;
                                        delay<=0;
                                end
                        default:        delay<=delay+1'd1;
                        endcase
                end // delayA
        default: FSM<=4'd4;
        endcase  // inst_buf[15:8]
end // FSM = 3         (weiting ROM and operation)
//################ главный автомат состояний FSM=4:
4'd4:   begin
                inst_counter<=inst_counter+1'd1;
                data_buf[15:0]<=instruction[15:0];
//Фрагменты обработки инструкций на данном этапе:
        case (inst_buf[15:8])
// базовые операция со специальными регистрами:
        8'd3:   begin
                        FSM<=4'd15;
                        case(inst_buf[7:0])
                        8'd1:   ATR[15:0]<=instruction[15:0];
                        8'd2:   ATP[15:0]<=instruction[15:0];
                        default:;
                        endcase
```

```
                end // ATR<=16'd65535, ATP<=16'd65535
// команды перемещения по типу *ATR<=instruction*:
8'd15, 8'd16, 8'd19:    begin
                                FSM<=4'd5;
                                ATR[15:0]<=instruction[15:0];
                            end // (15/16.3)  (19.2) (21.2)
// команды перемещения по типу *ATP<=instruction*:
8'd17, 8'd18, 8'd20:    begin
                                FSM<=4'd5;
                                ATP[15:0]<=instruction[15:0];
                            end // (17/18.3) (20.2 (22.2)
// команды перемещения по типу * ATB<=instruction*/8'd23*

8'd23:  begin
                    FSM<=4'd6;
                    ATB[7:0]<=instruction[7:0];
                end // (23.2) B255<=A254
// команды перемещения по типу * ATA<=instruction*:
8'd24:  begin
                    FSM<=4'd6;
                    ATA[7:0]<=instruction[7:0];
                end // (24.2) A255<=B254
// команды записи:
8'd27:  begin
                    FSM<=4'd5;
                    DTR[15:0]<=instruction[15:0];
                end // (27.2)RAM255<=16'd65535
8'd28:  begin
                    FSM<=4'd5;
                    DTP[15:0]<=instruction[15:0];
                end // (28.2) PRH255<=16'd65535
8'd29:  begin
                    FSM<=4'd6;
                    DTA[15:0]<=instruction[15:0];
                end //(29.2) A255<=16'd65535
8'd30:  begin
                    FSM<=4'd6;
                    DTB[15:0]<=instruction[15:0];
                end //(30.2) B255<=16'd65535
// команды переходов:
8'd46, 8'd47, 8'd48, 8'd49:     begin
                                FSM<=4'd5;
```

```
                                ATB[7:0]<=instruction[15:8];
                            end //  if(A255 <???> B254)up253
// логические операции:
8'd60:  begin
                    FSM<=4'd6;
                    DTA[31:0]<=DFA[31:0]+instruction[15:0];
                end // A255+65535
8'd61:  begin
                    FSM<=4'd6;
                    DTB[31:0]<=DFB[31:0]+instruction[15:0];
                end // B255+65535
8'd62:  begin
                    FSM<=4'd6;
                    DTA[31:0]<=DFA[31:0]-instruction[15:0];
                end // A255-65535
8'd63:  begin
                    FSM<=4'd6;
                    DTB[31:0]<=DFB[31:0]-instruction[15:0];
                end // B255-65535
8'd68, 8'd69, 8'd70, 8'd71, 8'd72, 8'd73, 8'd74, 8'd75, 8'd76, 8'd77, 8'd78, 8'd79:
                        begin
                    FSM<=4'd5;
                    ATA[7:0]<=instruction[15:8];
                    ATB[7:0]<=instruction[7:0];
                end  // A255<=A254 ??? B253 // B255<=A254 ??? B253
8'd82:  begin
                    FSM<=4'd6;
                    DTA[15:0]<=DFA[15:0]&instruction[15:0];
                end // mask(65535)AL255
8'd83:  begin
                    FSM<=4'd6;
                    DTA[31:16]<=DFA[31:16]&instruction[15:0];
                end // mask(65535)AH255
8'd86:  begin
                    FSM<=4'd6;
                    DTB[15:0]<=DFB[15:0]&instruction[15:0];
                end // mask(65535)BL255
8'd87:  begin
                    FSM<=4'd6;
                    DTB[31:16]<=DFB[31:16]&instruction[15:0];
                end // mask(65535)BH255
8'd89:  begin
```

```
                    FSM<=4'd6;
                    case (instruction[15:8])
                    8'd0:   DTA[31:0]<=DFA[31:0]<<instruction[7:0];
                    8'd1:   DTA[31:0]<=DFA[31:0]>>instruction[7:0];
                    default:;
                    endcase
                end // A255<<32; A255>>32
8'd90:  begin
                    FSM<=4'd6;
                    case (instruction[15:8])
                    8'd0:   DTB[31:0]<=DFB[31:0]<<instruction[7:0];
                    8'd1:   DTB[31:0]<=DFB[31:0]>>instruction[7:0];
                    default:;
                    endcase
                end // B255<<32; B255>>32
default:FSM<=4'd5;
endcase  // inst_buf[15:8]
end // FSM = 4 (read the second part of an instruction word)
//################ главный автомат состояний FSM=5:
4'd5:   begin
        case (inst_buf[15:8])
// Фрагмент обработки операций копирования:
        8'd12:  begin
                    case (FSM_2)
                    10'd0:  begin
                                FSM_2<=1'd1;
                                WER<=0;
                                WEP<=0;
                                ATR<=ATR+1'd1;
                                DTR[15:0]<=DFP[15:0];
                                MFC<=MFC+1'd1;
                            end
                    10'd1:  begin
                                if (MFC[15:0]==data_buf[15:0]) begin
                                    FSM<=0;
                                    FSM_2<=0;
                                    MFC<=0;
                                end
                                else begin
                                    FSM_2<=0;
                                end
                                WER<=1'd1;
```

```
                        WEP<=1'd1;
                    end
                default:;
                endcase
            end // P=>R(65535)/(copy operation)
        8'd13:  begin
                case (FSM_2)
                10'd0:  begin
                        FSM_2<=1'd1;
                        WEP<=0;
                        ATR<=ATR+1'd1;
                        DTP[15:0]<=DFR[15:0];
                        MFC<=MFC+1'd1;
                    end
                10'd1:  begin
                        if (MFC[15:0]==data_buf[15:0]) begin
                            FSM<=0;
                            FSM_2<=0;
                            MFC<=0;
                        end
                        else begin
                            FSM_2<=0;
                        end
                        WEP<=1'd1;
                    end
                default:;
                endcase
            end // R(65535)=>P  (copy operation)
// Фрагмент обработки команд перемещения *record RAM*:
8'd15, 8'd16, 8'd27, 8'd42:    begin
                    FSM<=0;
                    WER<=1'd1;
                end // record RAM
// Фрагмент обработки команд перемещения * record PRH *:
8'd17, 8'd18, 8'd28:    begin
                    FSM<=0;
                    WEP<=1'd1;
                end // record PRH
8'd19:  begin
            FSM<=4'd6;
            DTA[15:0]<=DFR[15:0];
        end // (19.3) A255<=RAM65535
```

```
8'd20:  begin
                FSM<=4'd6;
                DTA[15:0]<=DFP[15:0];
        end // (20.3) A255<=PRH65535
8'd21:  begin
                FSM<=4'd6;
                DTB[15:0]<=DFR[15:0];
        end // (21.3) B255<=RAM65535
8'd22:  begin
                FSM<=4'd6;
                DTB[15:0]<=DFP[15:0];
        end // (22.3) B255<=PRH65535
// Фрагмент обработки команд переходов:
8'd46:  begin
                if (DFA[31:0]>DFB[31:0]) begin
                        FSM<=0;
                end // DFA[31:0]>DFB[31:0]
                else begin
                        inst_counter[31:0]<=inst_counter[31:0]+data_buf[7:0];
                        FSM<=4'd15;
                end
        end // if(A255>B254)up253
8'd47:  begin
                if (DFA[31:0]<DFB[31:0]) begin
                        FSM<=0;
                end // DFA[31:0]<DFB[31:0]
                else begin
                        inst_counter[31:0]<=inst_counter[31:0]+data_buf[7:0];
                        FSM<=4'd15;
                end
        end // if(A255<B254)up253
8'd48:  begin
                if (DFA[31:0]==DFB[31:0]) begin
                        FSM<=0;
                end // DFA[31:0]=DFB[31:0]
                else begin
                        inst_counter[31:0]<=inst_counter[31:0]+data_buf[7:0];
                        FSM<=4'd15;
                end
        end // if(A255=B254)up253
8'd49:  begin
                if (DFA[31:0]!=DFB[31:0]) begin
```

```
                FSM<=0;
            end // DFA[31:0]=!DFB[31:0]
            else begin
                inst_counter[31:0]<=inst_counter[31:0]+data_buf[7:0];
                FSM<=4'd15;
            end
        end // if(A255=B254)up253
// Фрагмент обработки команд логических операций:
8'd68:  begin
            FSM<=4'd6;
            DTA[31:0]<=DFA[31:0]&DFB[31:0];
            ATA[7:0]<=buf_logick[7:0];
        end // A255<=A254andB253
8'd69:  begin
            FSM<=4'd6;
            DTB[31:0]<=DFA[31:0]&DFB[31:0];
            ATB[7:0]<=buf_logick[7:0];
        end // B255<=A254andB253
8'd70:  begin
            FSM<=4'd6;
            DTA[31:0]<=DFA[31:0]|DFB[31:0];
            ATA[7:0]<=buf_logick[7:0];
        end // A255<=A254orB253
8'd71:  begin
            FSM<=4'd6;
            DTB[31:0]<=DFA[31:0]|DFB[31:0];
            ATB[7:0]<=buf_logick[7:0];
        end // A255<=A254orB253
8'd72:  begin
            FSM<=4'd6;
            DTA[31:0]<=DFA[31:0]^DFB[31:0];
            ATA[7:0]<=buf_logick[7:0];
        end // A255<=A254xorB253
8'd73:  begin
            FSM<=4'd6;
            DTB[31:0]<=DFA[31:0]^DFB[31:0];
            ATB[7:0]<=buf_logick[7:0];
        end // B255<=A254xorB253
8'd74:  begin
            FSM<=4'd6;
            DTA[31:0]<=DFA[31:0]+DFB[31:0];
```

```
                ATA[7:0]<=buf_logick[7:0];
        end // A255<=A254+B253
8'd75:  begin
                FSM<=4'd6;
                DTB[31:0]<=DFA[31:0]+DFB[31:0];
                ATB[7:0]<=buf_logick[7:0];
        end // B255<=A254+B253
8'd76:  begin
                FSM<=4'd6;
                DTA[31:0]<=DFA[31:0]-DFB[31:0];
                ATA[7:0]<=buf_logick[7:0];
        end // A255<=A254-B253
8'd77:  begin
                FSM<=4'd6;
                DTA[31:0]<=DFB[31:0]+DFA[31:0];
                ATA[7:0]<=buf_logick[7:0];
        end // A255<=B253+A254
8'd78:  begin
                FSM<=4'd6;
                DTB[31:0]<=DFA[31:0]-DFB[31:0];
                ATB[7:0]<=buf_logick[7:0];
        end // B255<=A254-B253
8'd79:  begin
                FSM<=4'd6;
                DTB[31:0]<=DFB[31:0]-DFA[31:0];
                ATB[7:0]<=buf_logick[7:0];
        end // B255<=B253+A254
// Фрагмент обработки дополнительных команд:
8'd93:  begin
                case (FSM_2)
                8'd0:   begin
                                ATR[15:0]<=data_buf[15:0];
                                FSM_2<=FSM_2+1'd1;
                        end
                8'd1:   begin
                                DTD[15:0]<=DFR[15:0];
                                FSM_2<=FSM_2+1'd1;
                        end
                8'd2:   begin
                                SLKCD<=1'd1;
                                FSM_2<=FSM_2+1'd1;
```

```
                end
            8'd3:   begin
                        if (MFC[15:0]==inst_buf[7:0])begin
                            FSM<=4'd15;
                                FSM_2<=0;
                        end
                        else begin
                            ATR<=ATR+1'd1;
                            MFC<=MFC+1'd1;
                            FSM_2<=8'd1;
                        end
                        SLKCD<=0;
                end
            default:;
            endcase
        end // R8420+255=>DTD
8'd94:  begin
            case (FSM_2)
            8'd0:   begin
                        ATR[15:0]<=data_buf[15:0];
                        FSM_2<=FSM_2+1'd1;
                        math<=1'd1;
                end
            8'd1:   begin
                        DTR[15:0]<=DFD[15:0];
                        FSM_2<=FSM_2+1'd1;
                end
            8'd2:   begin
                        SLKCD<=1'd1;
                        WER<=1'd1;
                        FSM_2<=FSM_2+1'd1;
                end
            8'd3:   begin
                        if (MFC[15:0]==inst_buf[7:0])begin
                            FSM<=4'd15;
                                FSM_2<=0;
                            math<=0;
                        end
                        else begin
                            ATR<=ATR+1'd1;
                            MFC<=MFC+1'd1;
```

```
                                        FSM_2<=8'd1;
                                    end
                                    SLKCD<=0;
                                    WER<=0;
                                end
                        default:;
                        endcase
                end // R8420+255<=DTD
        default:FSM<=4'd6;
        endcase  // inst_buf[15:8]
        end // FSM =5 (write or finished operation)
        //################ главный автомат состояний FSM=6:
        4'd6:   begin
                        case (inst_buf[15:8])
        //record REG_A:
        8'd19, 8'd20, 8'd24, 8'd29, 8'd31, 8'd33, 8'd40, 8'd60, 8'd62, 8'd64, 8'd66, 8'd68, 8'd70,
8'd72, 8'd74, 8'd76, 8'd77, 8'd80, 8'd82, 8'd83, 8'd84, 8'd85, 8'd89, 8'd91, 8'd92, 8'd96, 8'd101,
8'd110:                                         begin
                        FSM<=0;
                        WEA<=1'd1;
                end // record REG_A
        //record REG_B:
        8'd21, 8'd22, 8'd23,8'd30, 8'd32, 8'd34, 8'd41, 8'd61, 8'd63, 8'd65, 8'd67, 8'd69, 8'd71,
8'd73, 8'd75, 8'd78, 8'd79, 8'd81, 8'd86, 8'd87, 8'd88, 8'd90, 8'd111:
                                        begin
                        FSM<=0;
                        WEB<=1'd1;
                end // record REG_B
        default:FSM<=0;
        endcase  // inst_buf[15:8]
        end // FSM =6 (additional and finished operation)
        //################ главный автомат состояний FSM=7,8:
        4'd7:   begin
                        FSM<=FSM+1'd1;
                end // FSM =6 (interruption script)
        4'd8:   begin
                        FSM<=4'd15;
                        inst_counter[31:16]<=0;
                        inst_counter[15:0]<=instruction[15:0];
                end // FSM =8 (interruption script)
        ///################ главный автомат состояний FSM=10,11,15:
```

```
4'd10:  begin
                FSM<=4'd11;
                inst_counter[31:0]<=31'd1024;
        end // FSM =10 (interruption script)
4'd11:  begin
                FSM<=4'd2;
        end // FSM =11 (interruption script)
4'd15:  begin
                FSM<=0;
        end // FSM =15 (delay)
default:;
endcase // FSM
end // CLK
end // always
endmodule // core
```

Приложение В
Система команд «NMR»

Команда	Основ. часть		Доп. часть		Описание команды
	Код	Оп. 1	Оп. 2	Оп. 3	
ATA<=X	1	X	-		Запись числа в системный регистр адреса ATA блока рег.А
ATB<=X	2	X	-		Запись числа в системный регистр адреса ATA блока рег.В
ATR<=X	3	1	X		Запись числа в системный регистр адреса ATR ОЗУ
ATP<=X	3	2	X		Запись числа в системный регистр адреса ATR ШП «HAVOC»
ATA<=B#	8	#	-		Запись в специальные регистры адресов (ATA, ATB, ATR, ATH) содержимого ячеек с адресом [#] из блоков рег.А или рег.В, в зависимости от команды
ATB<=A#	9	#	-		
ATR<=A#	10	#	-		
ATP<=A#	11	#	-		
A<=RAM	4	1	-		Запись в блок рег.А или рег.В по заранее установленным адресам в регистрах адреса ATA, ATB данных из ОЗУ или ШП «HAVOC» по также заранее установленным адресам в специальных регистрах адреса ATR, ATH соответственно
A<=PRH	4	4	-		
B<=RAM	5	1	-		
B<=PRH	5	4	-		
A<=RAM+	4	2	-		Запись в блок рег.А или рег.В по заранее установленным адресам в регистрах адреса ATA, ATB данных из ОЗУ или ШП «HAVOC» по также заранее установленным адресам в специальных регистрах адреса ATR, ATH соответственно, значения в которых после записи инкрементируются
A<=PRH+	4	5	-		
B<=RAM+	5	2	-		
B<=PRH+	5	5	-		
A+<=RAM+	4	3	-		Запись в блок рег.А или рег.В по заранее установленным адресам в регистрах адреса ATA, ATB данных из ОЗУ или ШП «HAVOC» по также заранее установленным адресам в специальных регистрах адреса ATR, ATH соответственно, значения всех регистров адреса инкрементируются
A+<=PRH+	4	6	-		
B+<=RAM+	5	3	-		
B+<=PRH+	5	6	-		
RAM<=A	6	1	-		Запись в ОЗУ или ШП «HAVOC» по заранее установленным адресам в регистрах ATR и ATH соответственно, данных из блоков рег.А или рег.В с также предустановленными адресами в регистрах ATA и ATB
RAM<=B	6	4	-		
PRH<=A	7	1	-		
PRH<=B	7	4	-		
RAM+<=A	6	2	-		Запись в ОЗУ или ШП «HAVOC» по заранее установленным адресам в регистрах ATR и ATH соответственно, данных из блоков рег.А или рег.В с также предустановленными адресами в регистрах ATA и ATB. Значения ATR или ATH после операции записи инкрементируются
RAM+<=B	6	5	-		
PRH+<=A	7	2	-		
PRH+<=B	7	5	-		
RAM+<=A+	6	3	-		Запись в ОЗУ или ШП «HAVOC» по заранее установленным адресам в регистрах ATR и ATH соответственно, данных из блоков рег.А или рег.В с также предустановленными адресами в регистрах ATA и ATB. Значения всех регистров адреса после записи инкрементируются
RAM+<=B+	6	6	-		
PRH+<=A+	7	3	-		
PRH+<=B+	7	6	-		
RAM<=PRH	6	7	-		Обмен данными между ОЗУ и ШП «HAVOC» с предустановленными адресами в РСН ATR и ATH с инкрементированием адреса в регистрах ATR и ATH или без в зависимости от команды
PRH<=RAM	7	7	-		
RAM+<=PRH	6	8	-		
PRH+<=RAM	7	8	-		

Команда	Основ. часть		Доп. часть		Описание команды
	Код	Оп. 1	Оп. 2	Оп. 3	
P=>R(X)	12		X		Запись из ШП в ОЗУ (код 12), или наоборот (код 13) блока данных объемом X слов, по предустановленным адресам в регистрах ATR, ATH с инкрементом адреса ОЗУ в ATR
R(X)=>P	13		X		
RAM●<=A#	15	#	●		Запись в ОЗУ или ШП «HAVOC» по адресу [●] содержимого ячейки [#] блоков рег.A или рег.B, в зависимости от команды
RAM●<=B#	16	#	●		
PRH●<=A#	17	#	●		
PRH●<=B#	18	#	●		
A#<=RAM●	19	#	●		Запись в ячейку [#] блоков рег.A или рег.B из адреса [●] ОЗУ или ШП, в зависимости от команды
A#<=PRH●	20	#	●		
B#<=RAM●	21	#	●		
B#<=PRH●	22	#	●		
B#<=A●	23	●	#		Запись в ячейку [#] блока рег.B из ячейки [●] блока рег.A
A#<=B●	24	●	#		Запись в ячейку [#] блока рег.A из ячейки [●] блока рег.B
RAM#<= X	27	#	X		Запись ОЗУ, ШП «HAVOC», или блоки рег.A или рег.B с адресом [#] числа ## из памяти программ. Для ОЗУ и ШП доступны только первые 256 адресов. Для 32-битных регистров, запись осуществляется в младшие 16 бит
PRH#<= X	28	#	X		
A#<= X	29	#	X		
B#<= X	30	#	X		
A#><	31	#	-		Обмен старших 16 бит на младшие в 32-битной ячейки с адресом [#] блоков рег.A или рег.B
B#><	32	#	-		
A#<=A●	33	●	#		Перемещение внутри блока одного блока РОН: запись в ячейку [#] ячейки [●]
B#<=B●	34	●	#		
mark A#	40	#	-		Сохранение в ячейке [#] блока регистров A или B текущего номера инструкции (значение счетчика инструкций)
mark B#	41	#	-		
mark RAM#	42	#	-		Сохранение в ячейке [#] младшего сектора ОЗУ текущего номера инструкции (значение счетчика инструкций)
jamp A#	43	#	-		Безусловный переход по адресу из ячейки [#] блоков регистров A или B, а также из ячейки [#] младшего сектора ОЗУ (присвоение счетчику инструкций значения из ячейки [#])
jamp B#	44	#	-		
jamp RAM#	45	#	-		
if(A#>B●)up X	46	#	●	X	Условный переход. Если условие выполняется происходит прыжок на X значений вперед, иначе выполнение программы идет по порядку. Для проверки условия берутся ячейки [#] и [●] из РАЗНЫХ блоков регистров A и B соответственно
if(A#<B●)up X	47	#	●	X	
if(A#=B●)up X	48	#	●	X	
if(A#!=B●)up X	49	#	●	X	
A#+X	60	#	X		Операция сложение содержимого ячейки [#] блоков регистров A или B и 16-битного числа X из памяти программ
B#+X	61	#	X		
A#-X	62	#	X		Операция вычитания содержимого ячейки [#] блоков регистров A или B и 16-битного числа X из памяти программ
B#-X	63	#	X		
A#++	64	#	-		Инкремент ячейки [#] блока регистров A
B#++	65	#	-		Инкремент ячейки [#] блока регистров B
A#--	66	#	-		Декремент ячейки [#] блока регистров A
B#--	67	#	-		Декремент ячейки [#] блока регистров B

Команда	Основ. часть		Доп. часть		Описание команды
	Код	Опр. 1	Опр. 2	Опр. 3	
A#<=A●andB◊	68	#	●	◊	Логические операции (and; or; xor) между ячейками РАЗНЫХ блоков регистров: Ячейка блока регистров A с адресом [●] и ячейка из блока регистров B с адресом [◊]; результат помещается в ячейку [#] блока регистров A, адрес помещения результата может совпадать с адресом [●] операнда блока регистров A, в таком случае исходное значение будет замещено результатом операции
B#<=A●andB◊	69	#	●	◊	
A#<=A●orB◊	70	#	●	◊	
B#<=A●orB◊	71	#	●	◊	
A#<=A●xorB◊	72	#	●	◊	
B#<=A●xorB◊	73	#	●	◊	
A#<=A●+B◊	74	#	●	◊	Математические операции сложения или вычитания между ячейками [◊], [●] РАЗНЫХ блоков рег.A и рег.B, результат операции помещается в ячейку [#] блоков рег.A и рег.B, в зависимости от команды
B#<=A●+B◊	75	#	●	◊	
A#<=A●-B◊	76	#	●	◊	
A#<=B●-A◊	77	#	●	◊	
B#<=A●-B◊	78	#	●	◊	
B#<=B●-A◊	79	#	●	◊	
notA#	80	#	-		Логическое побитовое отрицание содержимого ячейки [#] блоков рег.A и рег.B, в зависимости от команды
notB#	81	#	-		
mask(X)AL#	82	#	X		Побитовая маска младших (код 82) или старших (код 83) 16 бит 32-битной ячейки [#] блока регистров A с числом X
mask(X)AH#	83	#	X		
mask(X)BL#	86	#	X		Побитовая маска младших (код 86) или старших (код 87) 16 бит 32-битной ячейки [#] блока регистров B с числом X
mask(X)BH#	87	#	X		
rotAL#	84	#	-		Операция вращения числа в младшей (код 84) и старшей (код 86) области ячейки [#] блока рег.A. или всего 32-битного числа ячейки [#] рег.B. (код 88)
rotAH#	85	#	-		
rotBF#	88	#	-		
A#<< X	89	#	0	X	Сдвиг вправо или лево содержимого ячейки [#] блока регистров A или B на число X
A#>> X	89	#	1	X	
B#<< X	90	#	0	X	
B#>> X	90	#	1	X	
<<A#	91	#	-		Сдвиг вправо или лево содержимого ячейки [#] блока регистров A или B на 1 разряд
>>A#	92	#	-		
R#+X =>DTD	93	#	X		Отправка или получение в/от сопроцессора пакета данных из/в ОЗУ объемом до X слов начиная с адреса [#] ОЗУ
R#+X <=DTD	94	#	X		
adr<=A#	**95**	#	-		Запись в специальный регистр ADR ячейки [#] блока рег.A
A#<=adr	**96**	#	-		Запись в ячейку [#] блока рег.A содержимого регистра ADR
adr+8'd X	**97**	X	-		Увеличение содержимого специального регистра ADR на X
stat<=A#	**100**	#	-		Запись в специальный регистр Status ячейки [#] блока рег.A
A#<=stat	101	#	-		Запись в ячейку [#] блока рег.A содержимого регистра Status
A#<=GPI1	110	#	-		Запись данных со входа GPI1 в ячейку [#] блока рег.A
B#<=GPI2	111	#	-		Запись данных со входа GPI1 в ячейку [#] блока рег.A
intoff	127	-	-		Команда выхода из прерывания
sleep	126	-	-		Остановка процессора, запуск процессора возможен по сигналам прерывания
delayA255	125	#	-		Аппаратная задержка на кол-во тактов из ячейки [#] рег.A

Приложение Г
Описание отладочной платы и ИМС MAXII

ПЛИС семейства MAX II являются энергонезависимым (конфигурация хранится в блоке конфигурационной Flash-памяти) и готовы к работе сразу после включения питания. Микросхемы этого семейства поддерживают режим внутрисхемного программирования по JTAG-интерфейсу.

Архитектура семейства MAX II объединяет различные узлы: массив программируемой логики, пользовательскую Flash-память, встроенный RC-генератор, встроенный линейный регулятор напряжения.

Ресурсы СБИС MAX II EPM240T1100

Логические элементы, шт.	240
Макроячейки, шт.	192
Объем встроенной памяти, Кбит	8.192
Тип встроенной памяти	FLASH
t_{PD} (тип.), нс	4.7
I/O (макс.), шт.	80
F (макс.), МГц	304
V_{CC}, В	от 2.375 до 3.6
V_{CC} I/O, В	от 1.425 до 3.6
$I_{CC(STB)}$, мкА	12000
T_A, °C	от 0 до 70
Корпус ИМС	TQFP-100

URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/-literature/ug/max2_mii5v1_01.pdf

Приложение Д
Описание отладочной платы DE0-Nano

Макетная плата DE0-Nano является компактной платформой разработки ПЛИС, которая подходит для обучения, макетирования цифровых устройств, робототехники и других, в том числе портативных конструкций. Плата оснащена FPGA Cyclone IV с небольшим количеством логических элементов 22 320. Данная платформа, позволяет пользователю расширять функционал платы DE0-Nano при помощи двух внешних разъемов I/O (GPIO), обрабатывать большее количество данных и увеличить буфер при помощи встроенной памяти SDRAM и EEPROM. В состав платы входят светодиоды и кнопки, для отладки и взаимодействия с пользователем подключение к ПК не требует внешнего программатора. На плате реализованы три схемы питания: порт USB Mini-AB, разъем внешнего питания и два вывода 5В DC.

Состав платы:

– ПЛИС Cyclone IV EP4CE22F17C6N

– Встроенный USB Blaster, устройство EPCS16 с последовательной конфигурацией

– Устройства памяти: 32МБ SDRAM, 2КБ I2C EEPROM

– G-sensor ADI ADXL345, 3-осевой акселерометр с разрешением (13 бит)

– 8-канальный, 12-битный АЦП

– Встроенный кварцевый генератор 50МГц

Ресурсы СБИС Cyclone IV E

		Cyclone IV E (напряжения питания ядра 1.0 В и 1.2 В)								
		EP4CE6	**EP4CE10**	**EP4CE15**	**EP4CE22**	**EP4CE30**	**EP4CE40**	**EP4CE55**	**EP4CE75**	**EP4CE115**
Ресурсы	Кол-во логических элементов, тыс.	6	10	15	22	29	40	56	75	114
	Кол-во блоков встроенного ОЗУ M9K	30	46	56	66	66	126	260	305	432
	Объем строенного ОЗУ (Кбит)	270	414	504	594	594	1 134	2 340	2 745	3 888
	Кол-во умножителей 18 x 18	15	23	56	66	66	116	154	200	266
Архитектурные особенности	Кол-во глобальных цепей тактирования	0	0	0	0	0	0	0	0	0
	Кол-во блоков PLL	2	2	4	4	4	4	4	4	4
	Размер конфигурационного файла (Мбит)	2.8	2.8	3.9	5.5	9.1	9.1	14.2	19	27.2
Подсистема ввода-вывода	Поддерживаемые уровни напряжения ввода-вывода (В)	1.2, 1.5, 1.8, 2.5, 3.3								
	Поддерживаемые стандарты ввода-вывода	LVTTL, LVCMOS, PCI, PCI-X, LVDS, mini-LVDS, RSDS, LVPECL, Differential SSTL-15, Differential SSTL-18, Differential SSTL-2, Differential HSTL-12, Differential HSTL-15, Differential HSTL-18, SSTL-15 (I and II), SSTL-18 (I and II), SSTL-2 (I and II), 1.2-V HSTL (I and II), 1.5-V HSTL (I and II), 1.8-V HSTL (I and II)								
	Кол-во каналов LVDS	6	6	37	2	24	24	60	78	30
	Поддерживаемые интерфейсы внешней памяти	DDR2, DDR, QDR II, RLDRAM II, SDR								

URL: https://www.intel.cn/content/dam/alterawww/global/zh_CN/pdfs/literature/-hb/cyclone-iv/cyclone4-handbook.pdf

Учебное издание

Сухачев Кирилл Игоревич,
Родин Дмитрий Владимирович,
Дорофеев Александр Сергеевич

ОСОБЕННОСТИ ПРОЕКТИРОВАНИЯ ЦИФРОВОЙ ЭЛЕКТРОНИКИ НА БАЗЕ ПРОГРАММИРУЕМОЙ ЛОГИКИ

Учебное пособие

Техническое редактирование: *А.С. Никитина*
Компьютерная верстка: *А.С. Никитина*

Подписано в печать 04.05.2022. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 12,5.
Тираж 120 экз. (1-й з-д 1–25). Заказ № . Арт. – 3(Р1У)/2022.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.