СОДЕРЖАНИЕ

ВВЕДЕНИЕ 2

**1** РАЗРАБОТКА ОБЩЕЙ СТРУКТУРЫ МИКРОЭВМ 4

**1.1** Функциональный состав микро-ЭВМ 4

**1.2** Разработка системы команд 6

**1.3** Описание взаимодействия блоков микро-ЭВМ при выполнении

команд программы 8

**2** РАЗРАБОТКА ОСНОВНЫХ УСТРОЙСТВ МИКРО-ЭВМ 10

**2.1** Запоминающие устройства 10

**2.2** Устройство управления 12

**2.3** Арифметико-логическое устройство 19

**2.4** Блок регистров общего назначения 20

**2.5** Стек 21

**2.6** Организация кэш-памяти процессора 22

**2.7** Описание системы предсказания переходов 25

**2.8** Описание арбитра шины 27

**3** ФУНКЦИОНАЛЬНОЕ МОДЕЛИРОВАНИЕ 29

**3.1** Функциональное моделирование АЛУ 29

**3.2** Функциональное моделирование стека 29

**3.3** Функциональное моделирование блока РОН 30

**3.4** Функциональное моделирование блока памяти 31

**3.5** Функциональное моделирование кэш 32

**3.6** Функциональное моделирование арбитража 33

**3.7** Функциональное моделирование предсказателя переходов 34

**3.8** Функциональное моделирование системы в целом 34

**4** АНАЛИЗ И ОПТИМИЗАЦИЯ РАЗРАБОТАННОЙ МИКРО-ЭВМ 37

ЗАКЛЮЧЕНИЕ 38

СПИСОК ЛИТЕРАТУРЫ 39

ВВЕДЕНИЕ

Цель данной курсовой работы – разработка микро-ЭВМ согласно заданному варианту.

В общем случае, ЭВМ – это устройство или система, которая способна выполнять какие-либо заданные операции: манипулирование данными, операции ввода вывода.

При проектировании архитектуры наиболее важной частью будет являться тип этой самой архитектуры. В моём случае этим типом является гарвардская архитектура. В этом типе архитектура команды и данные хранятся раздельно. Использование такого типа памяти, как правило, позволяет выиграть в скорости по сравнению с Принстонской архитектурой. Запись в область команд невозможна в принципе. Сделано это в целях безопасности: невозможно случайно записать что-то в область команд и испортить код.

По заданию мне необходимо использовать два типа памяти ОЗУ и ПЗУ. Мной было принято решение сделать ПЗУ для команд и ОЗУ для данных, что как раз удовлетворяет всем требованиям к памяти команд, которая не может быть изменена.

Адресации в данной схеме наблюдается такие как прямая и базовая со смещением, что позволяет указать в команде адрес в памяти или номер регистра, откуда возьмем адрес, плюс смещение, и получим нужный нам адрес, где лежат данные.

Сам ход выполнения выглядит следующим образом: выборка и декодирование команд, вычисление адресов, выборка операндов, выполнение и/или запись.

Децентрализованный арбитраж шин не предполагает наличие главного арбитра, связанного со всеми ведущими устройствами, а предполагает наличие нескольких линий, ответственных за установление текущего приоритетного устройства, которое будет само следить за доступностью шины. Эта схема не идеальна, так как приоритет будет всегда фиксированным.

Предсказатель позволяет угадывать, будет ли совершен условный переход или нет. В моем случае используется схема А4, шаблоном будет служить программный счетчик. На лекциях я узнал, что это не самый лучший вариант шаблона, так лучшие показатели выдает вариант глобальная таблица шаблонов плюс программный счетчик, однако и мой вариант достаточно приемлем.

В качестве кэш используется схема 2-way associative, которая сочетает в себе достоинства кэш с полностью ассоциативным отображением и кэш с прямым отображением. Данные с 1 в старшем бите могут попасть в один сет, с 0 – в другой сет и никак иначе.

В АЛУ есть 4 команды: арифметическая команда CMP, логические команды NXOR и NOTZ, сдвиговая операция ROL.

Разработка проекта велась в Altera Quartus II 9.1. В ней возможно проектирование схем любой сложности, есть базовые примитивы, можно промоделировать работу результата с помощью временных диаграмм сигналов. Есть мощные заготовки megafunctions, благодаря которым можно легко реализовывать очень сложные схемы. Кроме того, я получил достаточно большой опыт работы в этой программе на лабораторном практикуме.

**1** РАЗРАБОТКА ОБЩЕЙ СТРУКТУРЫ МИКРОЭВМ

В этом разделе будет описан функциональный состав разрабатываемой ЭВМ, разработка системы команд и описание того, как блоки взаимодействуют между собой.

* 1. Функциональный состав микро-ЭВМ

Любая микро-ЭВМ – это не монолитное устройство, а набор некоторых блоков, выполняющих определенные функции. Данная микро-ЭВМ состоит из:

– арифметико-логическое устройство (АЛУ);

– блок памяти;

– устройство управления;

– кэш;

– предсказатель переходов;

– стек;

– блок регистров общего назначения.

Так же в схеме может присутствовать функция арбитража, но в моем случае арбитраж децентрализованный, поэтому в отдельном функциональном блоке необходимость отпадает: каждый блок, присоединенный к шине, следит за всем сам. Работает это следующим образом: есть два типа линий: линии занятости шины, линии запроса шины. Каждое устройство, которое работает с шиной, подключается также и к этим линиям. Логика в этом случае такая, что самым приоритетным будет самое левое устройство на схеме, за ним правое от него и так далее. Схема, конечно, не лишена недостатков, однако в общем случае она справляется с задачами арбитража шин.

Арифметико-логическое устройство получает операнды (один или два) и производит некоторые действия над ними. Все операции, выполняемые данным блоком, разделяют на три типа: арифметические, логические, сдвиговые. В данном проекте мне необходимо было реализовать операцию сравнения CMP (арифметическая операция), операцию исключающее ИЛИ-НЕ NXOR (логическая операция), операцию инверсия по флагу Z NOTZ (логическая операция) и циклический сдвиг влево ROL (сдвиговая операция). Некоторые результаты необходимо сохранять. Тут может быть несколько вариантов реализации этого: сохранять в память, сохранять в указанный регистр общего назначения, сохранять всегда в один регистр общего назначения, можно даже сохранять в стек. Мной был выбран, пожалуй, наиболее простой вариант: сохранение в определенный регистр. Такой метод широко себя зарекомендовал, например, в архитектуре х86, или в микропроцессоре MC68HC11, с которым я довольно плотно ознакомился на лабораторном и теоретическом практикуме дисциплины АВМиС.

Блок памяти представлен двумя блоками ОЗУ и ПЗУ. Он необходим для хранения данных (в ОЗУ) и команд (в ПЗУ). Разделение это удовлетворяет характеристике Гарвардской архитектуры, где данные и команды разделены. Кроме того, содержимое ПЗУ, как и команды, не могут быть изменены.

Устройство управления является одним из самых главных блоков микро-ЭВМ. Оно связывает внутренние блоки микро-ЭВМ и реализует управление ими. Именно этот блок реализует выборку команд, разбор, посылает микрокоманды всем остальным блокам. В устройстве управления находится программный счетчик (Program Counter – PC), который указывает на адрес текущей выполняемой команды. Именно благодаря наличию этого счетчика существует возможность реализации команд перехода (jmp). Необходимо лишь загрузить в PC новое значение. В моем случае в устройстве управления находится также и регистр флагов (Flag Register). Его можно было разместить и в другом месте, однако такой вариант показался мне наиболее удобным.

Кэш необходим для минимизации обращения к памяти и служит буфером между памятью и процессором. В общем случае обращение к памяти – крайне затратная по времени операция, а кэш, если не произошло промаха, позволяет выдавать данные процессору в кратчайшие сроки. Схема 2-way associative подразумевает наличие двух наборов. Данные с 1 в старшем бите адреса попадут во второй набор, с 0 в старшем бите – в первый набор. Таким образом, set в данном случае будет старший бит; tag – все остальные биты адреса. Данная схема сочетает в себе достоинства кэш с прямым отображением и кэш с полностью ассоциативным отображением. Также важный параметр кэш – это схема замещения строк. В моем случае анализа не проводится, что пагубно сказывается на производительности, но также и благоприятно на простоте схемы. Последний параметр кэш – это метод синхронизации с памятью. Подход, реализованный мной – это простая отложенная запись. Суть его в том, что данные, хранящиеся в кэш, будут записаны обратно в память в том случае, если они находятся в строке, которую необходимо выгрузить. Это немного уступает по производительности флаговой отложенной записи, но все равно достаточно быстро.

Стек представляет собой набор из 5 регистров (в моем случае), каждый размером 2 байта, счетчика Stack Pointer (SP) и логики управления. Над ним можно производить две операции: POP – для доставания данных с верхушки, PUSH – для укладывания данных в стек. Стоит сказать, для чего он применяется: стек применяется в случаях, когда необходимо организовать прерывания вызовов и возвратов, либо в случаях, когда нужно организовать временное хранилище данных (переменные, параметры функции).

Блок регистров общего назначения – это просто набор из 14 регистров, объединенных вместе для удобства. Так как все пересылки в блок регистров общего назначения идут через устройство управления (в моем случае), то никаких конфликтов быть не должно и нет необходимости во временном разделении. Эти регистры, как правило, без ограничений используются в арифметических операциях, операциях пересылки.

* 1. Разработка системы команд

Под командой понимают совокупность сведений, представленных в виде двоичных кодов, необходимых процессору для выполнения очередного шага. В ходе команды для сведений о типе операции, адресной информации о нахождении обрабатываемых данных, а также для информации о месте хранения результатов выделяются определенные разряды (поля). Главное поле – это код операции, уникальный идентификатор команды. Остальные два поля – это операнды (их адреса или номера регистров)

Разрабатывая систему команд, я учитывал свой предыдущий не самый приятный опыт лабораторного практикума, где коды операций были выбраны просто перебором, т.е. не поддавались никакой закономерности, из-за чего процесс декодирования усложнился в разы, так как требовалось большое количество дополнительных действий для установления типа адресации, типа команды и так далее.

Всего необходимо было реализовать 13 команд (учитывая разные типы адресации для команд пересылки данных). Было решено использовать 8 бит для кода операции, что может показаться довольно избыточным (8 бит могло бы хватить для 256 команд), однако это оказалось необходимым для разделения команд по блокам.

Все команды можно условно разделить на 4 группы:

– команды пересылки;

– команды стека;

– команды АЛУ;

– команды хода выполнения.

Для 4-х групп выделяется 4 бита в качестве идентификатора блока: установленный в единицу соответствующий идентификатору группы бит говорит о том, что команда принадлежит к определенной группе и будет использовать тот или иной блок.

Далее, в моем варианте задания необходимо реализовать два типа адресации: прямую адресацию в память и базовая адресация со смещением. Мной было решено выделить в коде операции 1 бит, который бы говорил о том, какой тип адресации используется в данной команде.

Оставшиеся три бита кода операции – это непосредственно идентификатор конкретной команды. Трех бит хватает для восьми команд. Количество команд в группе не велико, однако этого оказалось вполне достаточно.

Таким образом, вышла иерархическая структура, которую достаточно просто декодировать. Отпадает необходимость в больших массивах элементов ИЛИ для установления схожих команд.

Поле кода операции идут операнды. Для операндов, как уже было сказано ранее, используется прямая адресация и базовая адресация со смещением. Структура вычисления адреса второго типа представлена на рисунке 1.1.

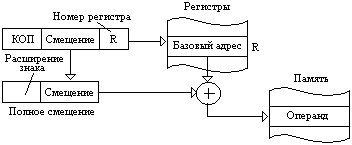


Рисунок 1.1 – Вычисление адреса операнда

Таким образом, необходимо передавать смещение и номер регистра. Я решил использовать 8 бит для смещения и 4 бита для номера регистра. Если используется прямая адресация, то поле регистра просто не учитывается, если одни из операндов просто номер регистра (например, при пересылке), то его номер будет записан в поле смещения.

Структура прямого адреса представлена на рисунке 1.2.

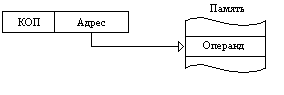


Рисунок 1.2 – Прямая адресация.

В сумме команда равна 32 битам, что довольно удачно, учитывая то, что шина данных в моем случае равна 16 бит, а это значит, что всю команду полностью можно прочитать за два такта.

Итоговый формат команды представлен на рисунке 1.3.

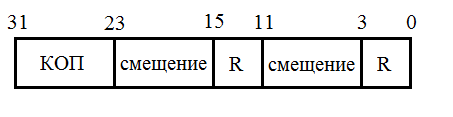


Рисунок 1.3 – Формат команды

Для более ясного понимания структура поля кода операции показана в таблице 1.1.

Таблица 1.1 – Формат поля КОП

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа | | Адр. | Команда | Мнемоническая запись |
| mov | 1000 | 0 | 001 | mov addr, reg |
| 1000 | 0 | 010 | mov reg, addr |
| 1000 | 1 | 001 | mov addr, reg |
| 1000 | 1 | 010 | mov reg, addr |
| stack | 0100 | 0 | 001 | push reg |
| 0100 | 0 | 010 | pop reg |
| ALU | 0010 | 0 | 001 | cmp reg, reg |
| 0010 | 0 | 010 | notz reg, reg |
| 0010 | 0 | 100 | rol reg, reg |
| 0010 | 0 | 000 | nxor reg, reg |
| program flow | 0001 | 0 | 001 | jmp addr |
| 0001 | 1 | 001 | jmp addr |
| 0001 | 0 | 010 | hlt |
| NOP | 0000 | 0 | 000 | nop |

В этой таблице addr – адрес, причем это поле подразумевает под собой либо просто адрес при прямом типе адресации, либо смещение и номер регистра при базовой адресации со смещением; reg – номер регистра.

**1.3** Описание взаимодействия блоков микро-ЭВМ при выполнении

команд программы

Всю работу инициирует устройство управления. Начинается все с подачи на него тактирующего сигнала. Блок управления посылает адреса на блок памяти, чтобы вычитать оттуда команду за два такта.

После того, как команда получена, она декодируется, и вычисляются настоящие адреса операндов (если необходимо), после чего идет опять запрос на блок памяти либо блок регистров общего назначения с целью получения самих операндов.

После первых двух стадий начинается выполнение. Тут ход может пойти несколькими путями в зависимости от команды.

Если это операция пересылки данных в регистры общего назначения, то в соответствующий регистр (номер регистра посылается на вход блока регистров общего назначения) будут записаны данные и наоборот, если это операция пересылки данных в память из регистров общего назначения, то по вычисленному адресу произойдет запись.

Если это логическая или арифметическая операция, то будет задействовано АЛУ. Так как прежде операнды уже были гарантированно подготовлены, то на вход АЛУ подаются эти операнды и требуемая команда. Команда сразу же выполняется, и результат отправляется обратно на устройство управления. Устройство управления, если необходимо, запишет результат в определенный регистр блока регистров общего назначения. Если результатом был флаг, то он запишется в регистр флагов, который все также находится в устройстве управления.

Если это команда работы со стеком, например, push, то, опять-таки, все данные уже готовы, и их устройству управления лишь необходимо отправить их на вход стека и указать соответствующую операцию. Если это команда pop, то данные сначала отправятся в устройство управления, а затем в регистры общего назначения.

Как видно из описания выше, абсолютно все пересылки идут через устройство управления, что не очень хорошо сказывается на быстродействии, однако устраняет большое количество проблем синхронизации и значительно упрощает реализацию схемы.

**2** РАЗРАБОТКА ОСНОВНЫХ УСТРОЙСТВ МИКРО-ЭВМ

В данном разделе описываются состав и принципы работы основных блоков разработанной микро-ЭВМ.

**2.1** Запоминающие устройства

В качестве запоминающего устройства используется комбинация модулей LPM\_RAM\_IO и LPM\_ROM (для памяти данных и команд соответственно). Quartus II 9.1 предоставляет мощный инструмент настройки этих модулей.

Важной особенность работы этих блоков является то, что они работают в два такта: на первом такте выставляется адрес, но втором происходит считывание или запись.

Объединив LPM\_RAM\_IO и LPM\_ROM и добавив логики управления к ним, я получил модуль памяти, представленный на рисунке 2.1.

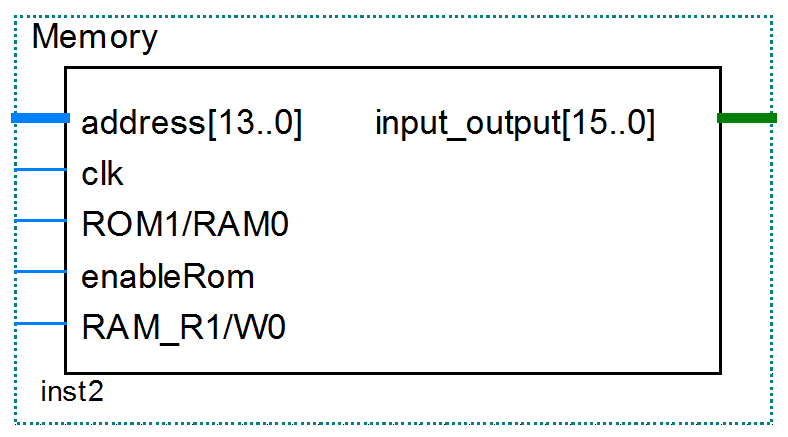


Рисунок 2.1 – Блок памяти

Рассмотрим предназначение входов и выходов.

Входы:

– address[13..0] – шина адреса. Адрес подается одновременно на оба устройства LPM\_RAM\_IO и LPM\_ROM;

– clk – тактирующий сигнал, глобальный для всей схемы;

– ROM1/RAM0 – сигнал, определяющий, будет ли вестись работа с LPM\_ROM или с LPM\_RAM\_IO;

– enableRom – отдельный сигнал для перекрытия выхода с LPM\_ROM. Сделан этот вход с той целью, чтобы полностью перекрыть подачу данных с LPM\_ROM на шину данных, например, в момент записи данных в LPM\_RAM\_IO или просто когда блок памяти не используется;

– RAM\_R1/W0 – сигнал, определяющий будем ли мы записывать данные в LPM\_RAM\_IO или считывать из него данные. Если в данный момент на сигнальной линии ROM1/RAM0 логическая единица (то есть в данный момент выбран LPM\_ROM), то эта линия никак не будет влиять на работу блока памяти.

Выход:

– input\_output[15..0] – двунаправленная шина данных, по которой ведется как чтение, так и запись.

На рисунке 2.2 представлена временная диаграмма работы ОЗУ (LPM\_RAM\_IO).

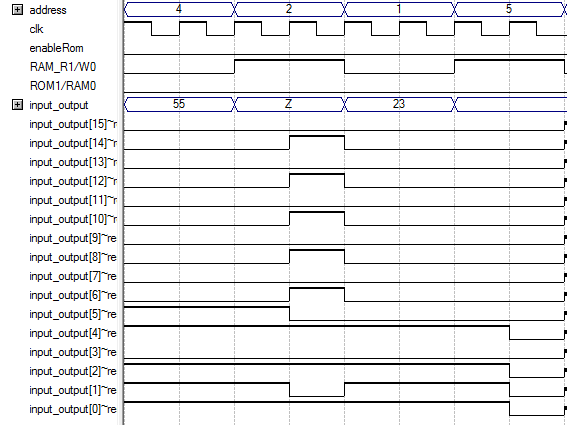


Рисунок 2.2 – Временная диаграмма работы ОЗУ.

Дампы памяти ОЗУ до и после симуляции представлены на рисунках 2.3 и 2.4.

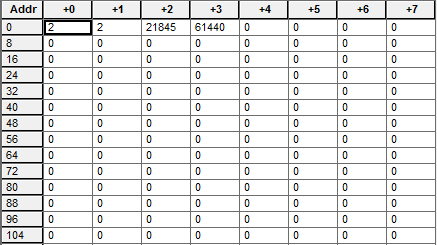


Рисунок 2.3 – Дамп памяти до симуляции

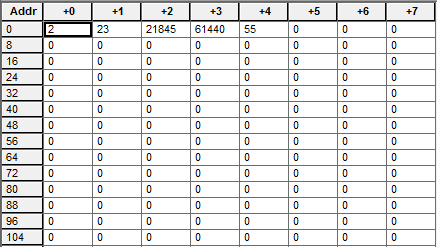


Рисунок 2.4 – Дамп памяти после симуляции

**2.2** Устройство управления

Устройство управления, пожалуй, оказалось самым комплексным в данной работе.

Внешний вид устройства управления представлен на рисунке 2.5.

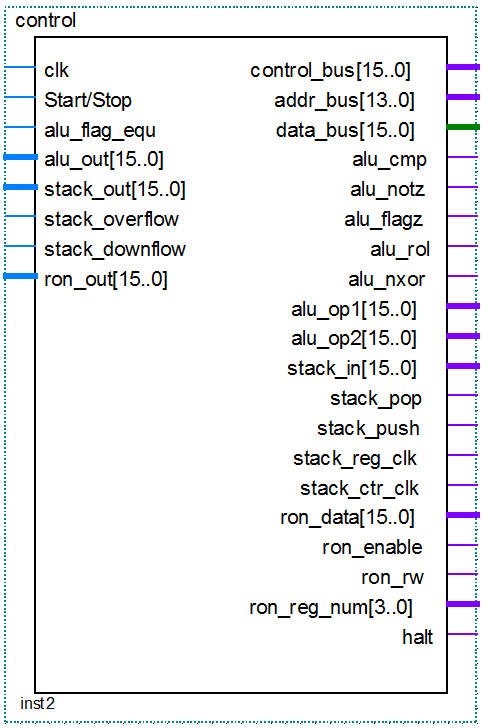


Рисунок 2.5 – Устройство управления

Как уже отмечалось выше, именно этот блок ответственен за управление абсолютно всеми остальными блоками. В нем происходит выборка команд, декодирование, отправка управляющих сигналов на АЛУ, стек, регистры общего назначения, память и так далее.

Рассмотрим подробнее, за что отвечают входы и выходы.

Входы:

– clk – тактирующий сигнал, глобальный для всей схемы. Он будет отправлен на блок делителя частоты для обеспечения возможно тактирования разных устройств в разное время;

– Start/Stop – сигнал для делителя частоты. По этому сигналу делитель запустится и начнет формировать синхросигналы;

– alu\_flag\_equ – специальный флаг, генерируемый в АЛУ. Он сигнализирует о том, что два операнда равны (если была запрошена операция CMP). Значение логического уровня на этой линии будет записано в регистр флагов;

– alu\_out[15..0] – на эту шину будет подаваться результат работы АЛУ, если были запрошены команды NXOR, ROL или NOTZ. Отдельный вход сделан с той целью, чтобы организовать независимые пересылки через устройство управления;

– stack\_out[15..0] – так же как и в случае с alu\_out[15..0], на эту шину подается выход стека с целью дальнейшей пересылки через устройство управления в блок регистров общего назначения;

– stack\_overflow – линия, сигнализирующая об одном из особых состояний стека – переполнение, указатель стека указывает за границу стека сверху. Значение логического уровня на этой линии будет записано в регистр флагов;

– stack\_overflow – линия, сигнализирующая об одном из особых состояний стека – указатель стека указывает за границу стека снизу. Значение логического уровня на этой линии будет записано в регистр флагов;

– ron\_out[15..0] – так же как и в случае с шинами alu\_out[15..0] и stack\_out[15..0], на эту шину подается выход регистров общего назначения с целью дальнейшей пересылки через устройство управления в стек или в память.

Выходы:

– control\_bus[15..0] – специальная шина управления внешними устройствами. В данном проекте внешним устройством будет являться только блок памяти. Задействованы лишь 3 линии данной шины, остальные линии не удалялись с целью будущего наращивания устройств, которого не произошло, однако было решено оставить шину в таком виде. Предназначение линий данной шины показано в таблице 2.1.

Таблица 2.1 – Предназначение линий шины управления

|  |  |
| --- | --- |
| № линии | Предназначение |
| 0 | выбор ROM либо RAM блока памяти |
| 1 | Разрешение ROM на выдачу данных |
| 2 | Запись либо чтение из RAM |

– addr\_bus[13..0] – шина адреса, которая будет заведена на блок памяти. По одной этой шине будут передаваться адреса данных и адреса команд;

– data\_bus[15..0] – шина данных, которая будет заведена на блок памяти. По одной этой шине будут передаваться как данные, так и команды;

– alu\_cmp – управляющий сигнал для АЛУ, сигнализирующий ему о том, что необходимо выполнить операцию сравнения двух операндов, поданных на его вход. Именно эта операция будет генерировать сигнал, который после отправить на вход alu\_flag\_equ;

– alu\_notz – управляющий сигнал для АЛУ, сигнализирующий ему о том, что необходимо выполнить операцию НЕ по флагу Z;

– alu\_flagz – специальная линия, по которой передается флаг Z, используемый командой NOTZ;

– alu\_rol – управляющий сигнал для АЛУ, сигнализирующий ему о том, что необходимо выполнить операцию циклического сдвига влево;

– alu\_nxor – управляющий сигнал для АЛУ, сигнализирующий ему о том, что необходимо выполнить операцию исключающее ИЛИ-НЕ;

– alu\_op1[15..0] – шина, по которой передается первый операнд операции для АЛУ;

– alu\_op1[15..0] – шина, по которой передается второй операнд операции для АЛУ. Если операции нужен лишь один операнд, то эта не используется;

– stack\_in[15..0] – шина, по которой передаются данные для последующей их записи в стек;

– stack\_pop – управляющий сигнал для стека, сигнализирующий ему о том, что необходимо выполнить операцию pop;

– stack\_push – управляющий сигнал для стека, сигнализирующий ему о том, что необходимо выполнить операцию push;

– stack\_reg\_clk – служебный управляющий сигнал для стека, сигнализирующий ему о том, в какой именно момент нужно записать данные в стек;

– stack\_ctr\_clk – служебный управляющий сигнал для стека, сигнализирующий ему о том, в какой именно момент нужно изменить значение указателя стека;

– ron\_data[15..0] – шина, по которой передаются данные для записи в блок регистров общего назначения;

– ron\_enable – управляющий сигнал для блока регистров общего назначения, сигнализирующий ему о том, что необходимо выдать данные на его выходе;

– ron\_rw – управляющий сигнал для блока регистров общего назначения, сигнализирующий ему о том, необходимо прочить либо записать данные;

– ron\_reg\_num[3..0] – шина, по которой передается номер регистра общего назначения, с которым должна вестись работа (запись либо чтение);

– halt – специальный сигнал, который приостанавливает работу процессора;

Теперь необходимо описать некоторые подробности работы данного блока.

Вся схема тактируется генератором опорных сигналов, который представлен на рисунке 2.6.

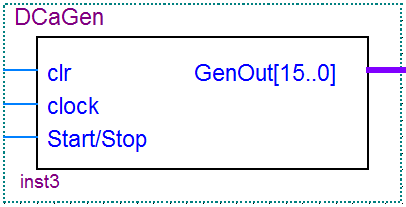


Рисунок 2.6 – Генератор опорных сигналов

Описание его входов:

– clr – сброс внутреннего счетчика;

– clock – глобальный тактирующий сигнал;

– Start/Stop – сигнал, отвечающий за запуск генератора.

Описание его выходов:

– GenOut[15..0] – выходная шина генератора. На каждой линии этой шины будут по очереди возникать сигналы, длящиеся один такт. После того, как сигнал появится на последней пятнадцатой линии, все начнется заново.

Благодаря данному генератору появляется возможность активировать разные участки схемы строго определенное время.

Первоначальный процесс выборки начинается с того, что с выхода регистра Instruction Pointer (IP) подается адрес необходимой команды на блок памяти. Так как ширина шины адреса по заданию равна четырнадцати битам, то и IP тоже такого же размера. В качестве IP используется управляющая обвязка вокруг LPM\_DFF, заключенная в отдельный блок, который представлен на рисунке 2.7.

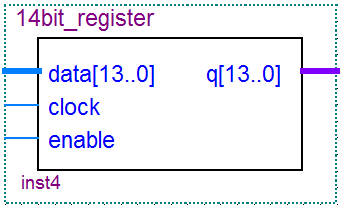


Рисунок 2.7 – Instruction Pointer

Предназначение входов и выходов здесь:

– data[13..0] – входная шина, по которой подаются данные;

– clock – тактирующий сигнал, по которому происходит запись в регистр;

– enable – разрешение тактирующего сигнала;

– q[13..0] – выход.

Работает IP в связке со счетчиком. В то время, как счетчик хранит следующий адрес, IP хранит текущий.

Так как команда 32-битная, а шина данных 16-битная, то вся команда считывается за два такта в два буферных регистра: buffer register 1 и buffer register 2. Так же как и IP, они являются управляющей обвязкой вокруг LPM\_DFF, заключенной в отдельный блок. По сути, эти регистры работают таким же образом, как и регистр 14bit\_register для IP, описанный выше, поэтому нет смысла повторяться.

После этих двух тактов содержимое подается на специальный блок composer, позволяющий разделить поля между собой. Он представлен на рисунке 2.8.

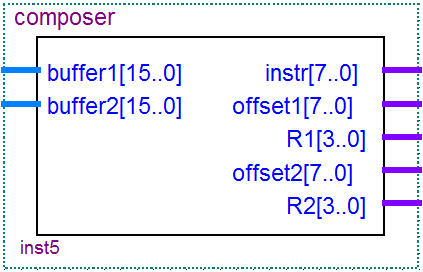


Рисунок 2.8 – Composer

Его входы buffer1 и buffer2 – это просто выход буферных регистров. Его выходы предназначены для следующего:

– instr[7..0] – на этом выходе будет находиться КОП текущей команды;

– offset1[7..0] – на этом выходе будет находиться смещение для вычисления адреса первого операнда текущей команды;

– R1[3..0] – на этом выходе будет находиться номер регистра для вычисления адреса первого операнда текущей команды;

– offset2[7..0] – на этом выходе будет находиться смещение для вычисления адреса второго операнда текущей команды;

– R2[3..0] – на этом выходе будет находиться номер регистра для вычисления адреса второго операнда текущей команды;

Теперь очень легко понять, где какое поле.

Выход instr[7..0] блока composer запишется в Instruction Register (который опять-таки функционирует по схожей с 14bit\_register схеме), после чего КОП подастся на специальный блок, который установит все информацию о данной команде. Этот называется command\_decoder и представлен на рисунке 2.9.

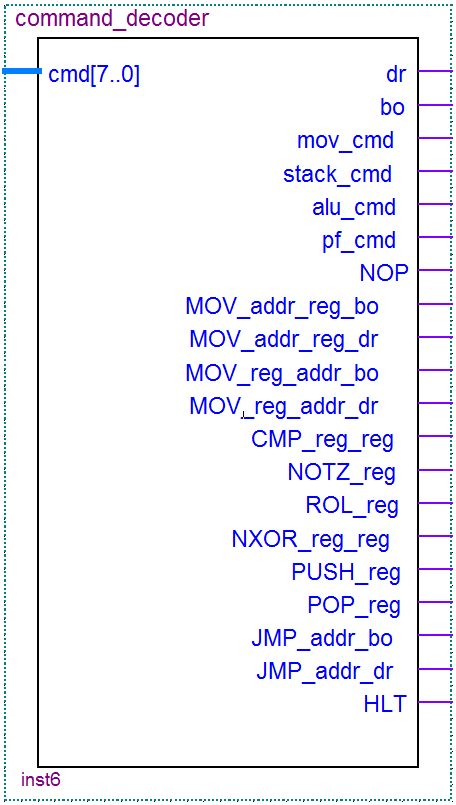


Рисунок 2.9 – command\_decoder

Вход cmd[7..0] – это просто текущая команда. Следует описать, о чем же говорят активные уровни на его выходах:

– dr – в данной команде используется прямая адресация;

– bo – в данной команде используется базовая адресация со смещением;

– mov\_cmd – команда пересылки данных;

– stack\_cmd – команда работы со стеком;

– alu\_cmd – команда работы с alu;

– pf\_cmd – команда изменения хода течения программы;

– NOP – простой;

– MOV\_addr\_reg\_bo – команда пересылки из регистра в память, использующая базовую адресацию со смещением;

– MOV\_addr\_reg\_dr – команда пересылки из регистра в память, использующая прямую адресацию;

– MOV\_reg\_addr\_bo – команда пересылки из памяти в регистр, использующая базовую адресацию со смещением;

– MOV\_reg\_addr\_dr – команда пересылки из памяти в регистр, использующая прямую адресацию;

– CMP\_reg\_reg – команда сравнения содержимого двух регистров;

– NOTZ\_reg – команда НЕ по флагу Z;

– ROL\_reg – команда циклического сдвига влево;

– NXOR\_reg\_reg – команда исключающее ИЛИ-НЕ;

– PUSH\_reg – команда записи содержимого регистра в стек;

– POP\_reg – команда записи содержимого вершины стека в регистр;

– JMP\_addr\_bo – команда безусловного перехода, использующая базовую адресацию со смещением;

– JMP\_addr\_dr – команда безусловного перехода, использующая прямую адресацию;

– HLT – команда приостановки процессора.

После того, как команда декодирована, сначала вычисляются нужные адреса и записываются в регистры Address Register 1 и Address Register 2 для первого и второго операндов соответственно. Если один из операндов находится в одном из регистров общего назначения, то в Address Register запишется номер регистра.

После того, как адреса известны, считываются сами данные из РОН либо из памяти в Data Register 1 и Data Register 2. После этого выполняется команда.

Управление другими блоками осуществляется с помощью выходов генератора опорных импульсов и декодера команд.

Последнее, о чем бы хотелось рассказать в рамках управляющего устройства – это регистр флагов (Flag Register – FR). В этом регистре фиксируются некоторые состояния АЛУ и стека. Регистр флагов представлен на рисунке 2.10.

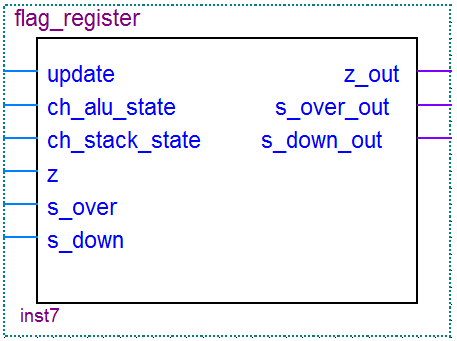


Рисунок 2.10 – Регистр флагов

Описание его входов:

– update – с приходом активного уровня на этом входе обновляется состояние регистра флагов;

– ch\_alu\_state – линия, активный уровень на которой сигнализирует о том, что нужно обновить состояние только тех бит регистра флагов, которые имею отношение к АЛУ;

– ch\_stack\_state – линия, активный уровень на которой сигнализирует о том, что нужно обновить состояние только тех бит регистра флагов, которые имею отношение к стеку;

– z – новее значение флага Z;

– s\_over – новее значение флага stack\_overflow;

– s\_down – новее значение флага stack\_downflow.

Описание его выходов:

– z\_out – текущее значение флага Z;

– s\_over\_out – текущее значение флага stack\_overflow;

– s\_down\_out – текущее значение флага stack\_downflow.

Управление блоками происходит довольно тривиально с помощью примитивов и, на мой взгляд, подробного описания не требует.

**2.3** Арифметико-логическое устройство

АЛУ, как уже писалось выше, выполняет арифметические, логические и сдвиговые операции. АЛУ представлено на рисунке 2.11.

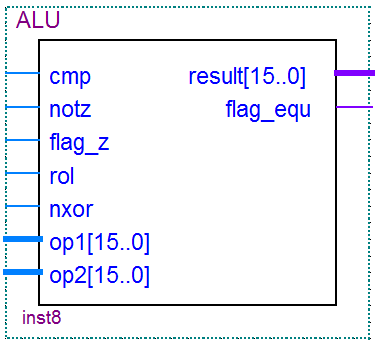


Рисунок 2.11 – АЛУ

Описание входов:

– cmp – вход, сигнализирующий АЛУ о том, что необходимо выполнить операцию сравнения операндов op1 и op2;

– notz – вход, сигнализирующий АЛУ о том, что необходимо выполнить операцию НЕ по флагу z операнда op1;

– flag\_z – флаг, используемый командой НЕ по флагу z. Инверсия будет осуществлена только в том случае, если флаг z имеет высокий логический уровень;

– nxor – вход, сигнализирующий АЛУ о том, что необходимо выполнить операцию исключающее ИЛИ-НЕ операндов op1 и op2;

– op1[15..0] – вход, на который подается первый операнд;

– op2[15..0] – вход, на который подается второй операнд. Если в операции используется один операнд, то этот данные на этом входе просто не учитываются.

Описание выходов:

– result[15..0] – на этой шине формируется результат запрошенной операции;

– flag\_equ – специальный выход, состояние его генерируется командой cmp. Именно этот флаг используется командой НЕ по флагу z.

Реализация АЛУ заняла мало времени, так как я использовал мощные заготовки Quartus II 9.1 – megafunctions. Благодаря этому блок вышел небольшим и достаточно тривиальным, описывать его подробно не имеет смысла.

**2.4** Блок регистров общего назначения

Регистры общего назначения, как отмечалось ранее, предназначены для хранения информации и, в общем случае, требуют наименьшего количества тактом для записи и чтения данных.

Блок регистров необходим для пересылки данных в стек, хранения операндов и результатов АЛУ.

Блок регистров общего назначения представлен на рисунке 2.12.

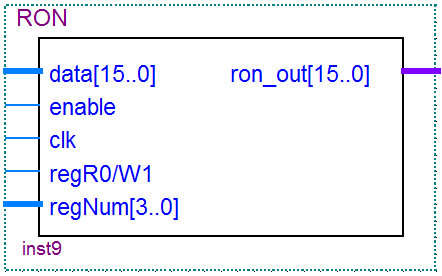


Рисунок 2.12 – РОН

Предназначение входов:

– data[15..0] – на этот вход подаются данные, которые необходимо записать в один и регистров общего назначения;

– enable – разрешение блоку регистров общего назначения выдавать данные на своем выходе;

– clk – глобальный тактирующий сигнал;

– regR0/W1 – логический уровень на этом входе говорит блоку о том, будет ли производиться запись или чтение данных;

– RegNum[3..0] – на этот вход подается номер регистра, с которым будут вестись операции чтения или записи.

Внутри блока находится 14 регистров, каждый представлен в виде блока 16bit\_register, который уже описывался выше.

Выбор необходимого регистра осуществляет декодер, который был создан на основе встроенного LPM\_DECODE. Таким образом, унитарный код переводится в унитарный, логическая единица на какой-либо линии декодера активирует соответствующий регистр.

**2.5** Стек

Стек – одна из самых необходимых структур хранения данных в компьютере. Благодаря стеку можно как осуществлять просто хранение, так и, например, передавать параметры в функции, хранить точки возврата и так далее.

Блок стека представлен на рисунке 2.13.

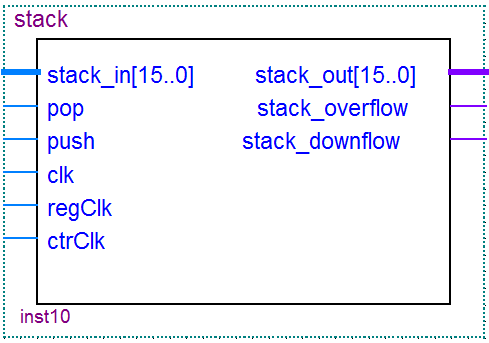


Рисунок 2.13 – Стек

Описание входов:

– stack\_in[15..0] – вход, на который подаются данные для записи в стек при команде push;

– pop – вход, активный уровень на котором сигнализирует блоку о том, что необходимо произвести операцию pop, то есть чтение данных из вершины стека;

– push – вход, активный уровень на котором сигнализирует блоку о том, что необходимо произвести операцию push, то есть запись данных в вершину стека;

– clk – глобальный тактирующий сигнал;

– regClk – тактирующий сигнал, положительному перепаду на этом входе произойдет запись данных в стек;

– ctrClk – тактирующий сигнал, положительному перепаду на этом входе произойдет изменение текущего значения указателя стека (stack pointer – SP);

Описание выходов:

– stack\_out[15..0] – выход, на который выдаются с вершины стека при команде pop;

– stack\_overflow – выход, сигнализирующий о особом состоянии стека – overflow.

– stack\_downflow – выход, сигнализирующий о особом состоянии стека – down.

Суть работы стека проста и схожа работой РОН, с той лишь разницей, что запись/считывание ведется не в указанный регистр, а согласно указателю стека. SP указывает на свободную ячейку стека.

При команде записи (push), данные записываются в тот регистр, на который указывает SP, который после записи увеличивается. Если была произведена запись в последнюю ячейку стека, то SP будет указывать за реальные пределы стека, что сформирует активный уровень на выходе stack\_overflow, что будет отмечено в регистре флагов. В этом случае запись производить больше нельзя.

При команде чтения (pop), сначала уменьшается SP, затем вычитываются данные из регистра, но который он указывает. Если стек был пуст, но команда pop все равно была запрошена, то это приведет к другой особой ситуации – downflow. В этом случае указатель стека будет уже с другой стороны указывать за пределы стека, что вызовет формирование активного уровня на выходе stack\_downflow, что также отметится в регистре флагов. Как только произойдет запись этого события в регистр флагов, указатель стека вернется на нулевое положение, чтобы не нарушать работу стека.

Сами ячейки стека представлены все теми же регистрами 16bit\_register, которые были описаны выше.

**2.6** Организация кэш-памяти процессора

Как уже писалось выше, кэш-память служит некоторым буфером между процессором и памятью, сохраняя в себе наиболее часто используемые данные.

В моем варианте задания необходимо было реализовать 2-way associative кэш, замещение без анализа, простую отложенную запись в память.

2-way associative говорит о том, что будет существовать два набора. Согласно полю set, которым у меня служит старший бит адреса, данные могут попасть только в один набор, и никогда в другой. Внутри набора они могут занимать любые строки кэш. Такой метод сочетает в себе достоинства fully-associative кэш и direct mapped кэш.

Внешне блок кэш выглядит, как показано на рисунке 2.14.

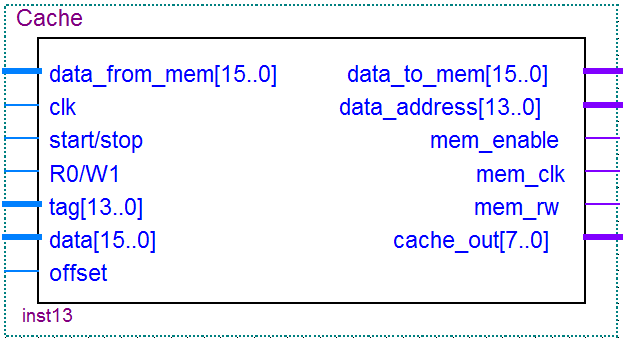


Рисунок 2.14 – Кэш

Описание входов:

– data\_from\_mem[15..0] – вход, на который подаются данные из памяти, после чего они сохраняются в кэш;

– clk – глобальный тактирующий сигнал;

– start/stop – вход для сигнала запуска внутреннего генератора опорных сигналов;

– R/0W1 – вход, определяющий тип текущей команды: чтение либо запись;

– tag[13..0] – тэг, согласно которому будут размещены данные в кэш. Тегом служит адрес данных в памяти;

– data [15..0] – вход, на который подаются данные от процессора, если необходимо записать в память какие-либо данные.

– offset – смещение внутри строки: есть возможность выбрать, старший либо младший биты будут поданы на выход стека.

Описание выходов:

– data\_to\_mem[15..0] – выход, который должен быть соединен с памятью. Он необходим для выгрузки данных из кэш.

– data\_address[13..0] – выход, который должен быть соединен с памятью. На нем формируются реальные адреса запрашиваемых данных.

– mem\_enable – управляющий блоком памяти выход. Разрешает выдачу данных на выход;

– mem\_clk – управляющий блоком памяти выход. Организует особое тактирование для блока памяти;

– mem\_rw – управляющий блоком памяти выход. Определяет тип текущей операции: чтение либо запись;

– cache\_out[7..0] – выход кэш, на который выдаются данные из кэш, служит для отправки данных процессору.

С использованием кэш уже не управляющее устройство будет управлять памятью, а блок кэш, для чего и появились специальные выходы.

Внутреннее устройство блока довольно сложное, поэтому стоит описать внутренние блоки, которые в нем находятся.

Главные внутренние элементы кэш – это наборы (sets). Внутри сета находятся непосредственно строки кэш. Сет представлен на рисунке 2.15.

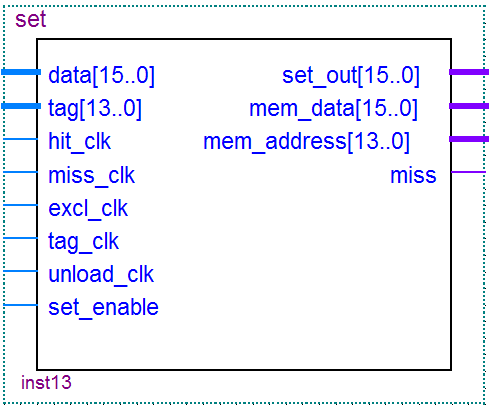


Рисунок 2.15 – Set

Описание входов:

– data[15..0] – вход, на который подаются;

– tag[13..0] – тэг, согласно которому будут размещены данные в наборе. Тегом служит адрес данных в памяти;

– hit\_clk – отдельное тактирование для случая, когда промаха не произошло;

– miss\_clk – отдельное тактирование для случая, когда промах произошел;

– hit\_clk – отдельное тактирование для случая, когда промаха не произошло;

– excl\_clk – отдельное тактирование для схемы определения замещаемой строки;

– tag\_clk – отдельное тактирование для сохранения тэгов;

– unload\_clk – отдельное тактирование для схемы замещения строки;

– set\_enable – вход, активный уровень на котором сигнализирует о том, что набор активен и с ним ведется работа.

Описание выходов:

– set\_out[15..0] – выход набора, на нем будут формироваться данные при попадании;

– mem\_data[15..0] – отдельная шина для выгрузки данных в память;

– mem\_address[15..0] – отдельная шина для указания адреса выгружаемых данных;

– miss – активный уровень на данной линии сигнализирует о том, что в наборе отсутствует строка с запрашиваемым тегом.

Таким образом, тэг подается сразу на оба набора. В зависимости от старшего бита адреса активизируется только один из них и с ним и будет вестись работа. Внутри набора строки кэш представлены тем же 16bit\_register, а тэги хранятся в 14bit\_register. Эти блоки уже описывались выше.

Выход каждого регистра, в котором хранится тэг, соединен с компаратором, который реализован на основе встроенного блока LPM\_COMPARE. В качестве второго операнда на компаратор подается искомый тэг. Если тэги равны, то это означает попадание. Если ни один тэг не равен искомому, то это означает промах. При промахе нужно дополнительно загрузить данные из памяти, предварительно вытесним содержимое вытесняемой строки.

**2.7** Описание системы предсказания переходов

Система предсказания переходов – критично важная часть конвейера, которая очень сильно влияет на производительность системы.

Суть системы предсказания переходов в следующем. Когда в конвейере появляется команда условного (и только условного) перехода, то существует вероятность того, что последующие команды, выполненные в конвейере, окажутся ненужными и их придется выгружать, так как произойдет переход в другое место программы. Также существует вероятность того, что условие в команде перехода будет ложным и ничего выгружать не нужно будет. Система предсказания позволяет угадать, действительно ли произойдет переход или нет, из-за чего можно будет не выгружать конвейер и выиграть в скорости.

В данном курсовом проекте в центральном процессоре не был реализован конвейер, поэтому эмуляция системы предсказания реализована в стороне.

Стоит подробнее описать о том, как работает предсказатель. Начнем с шаблона. Существует два основных типа шаблона: Global History Register (GHR) и Program Counter (PC). Также существует большое количество их комбинаций между собой. В данной работе согласно заданию используется Program Counter, то есть просто программный счетчик. Смысл в использовании PC в качестве шаблона в том, что предсказатель будет примерно запоминать, какие были результаты предсказания на разных значениях программного счетчика, то есть в разных участках программы. Если код не меняется, как, например, в микроконтроллерах, где постоянно работает одна и та же прошивка, то этот метод является наиболее предпочтительным.

Все шаблоны объединяются в структуру под названием Pattern History Table (PHT), где находятся все варианты шаблонов и биты предсказания для каждого шаблона. В моем случае в PHT будут находиться абсолютно все возможные варианты значения программного счетчика.

Биты предсказания в PHT меняются для каждого варианта шаблона согласно схеме переходов какого-либо автомата. В моем случае таким автоматом является A4.

Блок PHT представлен на рисунке 2.16.

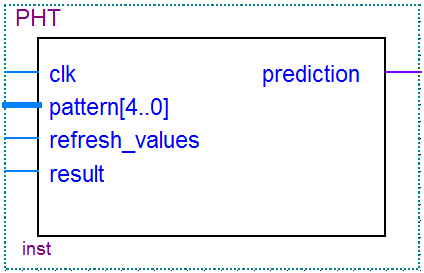


Рисунок 2.16 – Pattern History Table

Описание входов:

– clk – глобальный тактирующий сигнал;

– pattern[4..0] – шаблон, подаваемый в PHT для поиска соответствующих ему битов предсказания. В моем случае, как уже говорилось, это PC, но данная схема универсальна, и сюда может подаваться любой 5-битный шаблон;

– refresh\_values – тактирующий сигнал, по которому изменится значение битов предсказания согласно таблице переходов автомата;

– result – настоящий результат условного перехода, необходимый для изменения состояния автомата;

Внутри таблица реализована на основе LPM\_RAM\_DQ. Адресом в нем служит шаблон, а содержимое RAM – это и есть биты предсказания (состояние автомата).

Автомат А4 представлен на рисунке 2.17

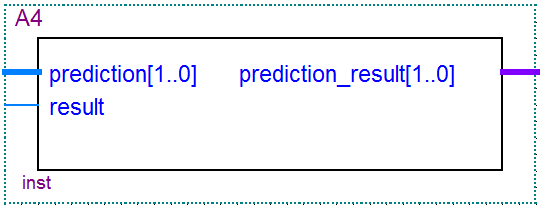


Рисунок 2.17 – Автомат А4

С автоматом все просто: он просто показывает, в какое состояние перейдет автомат.

Описание входов:

– prediction[4..0] – текущее состояние автомата;

– result – настоящий результат условного перехода – это сигнал, определяющий, как изменится состояние автомата.

Описание выхода:

– prediction\_result[1..0] – новое состояние автомата.

В остальном схема работает достаточно просто и подробного разъяснения не требует.

**2.8** Описание арбитра шины

Когда по шине передаются данные лишь между двумя блоками, то все функционирует довольно просто и, в общем случае, никакой особой логики управления не требуется. Но как только устройств становится три и более, то возникает проблема того, как разделить во времени доступ к шине. Эта задача называется арбитражем.

Существует несколько основных подходов к реализации арбитража: это централизованный и децентрализованный арбитраж. В свою очередь каждый из них может быть последовательным и параллельным.

В моем варианте задания используется параллельный децентрализованный арбитраж. Суть его заключается в том, что нет никакого центрального устройства, арбитра, который бы следил за запросами от остальных устройств и выдавал бы им шину. Здесь каждое устройство само следит за шиной. Минусом данной схемы является то, что приоритеты строго определены и не могут быть изменены динамически: наивысший приоритет всегда получает одно и то же устройство. Впрочем, иногда и это может быть полезно. Возможно также реализация смены приоритетов в реальном устройстве с помощью, например, перемычек.

В данном проекте я не стал организовывать арбитраж в связке с остальными блоками, а лишь реализовал симуляцию его работы.

В схеме представлено несколько устройств, каждое из которых имитирует какой-то вычислительный процесс и запрос к шине после него. Изображение такого устройства представлено на рисунке 2.18.

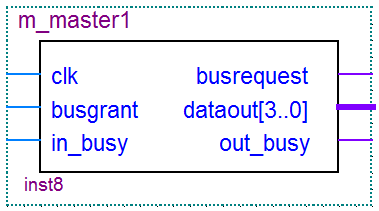


Рисунок 2.18 – Имитация устройства

Описание входов:

– clk – глобальный тактирующий сигнал;

– busgrant – вход, активный уровень на котором сигнализирует о том, что устройство получает доступ к шине;

– in\_busy – вход, активный уровень на котором сигнализирует о том, что шина занята и устройству следует подождать.

Описание выходов:

– busrequest – выход, активный уровень на котором сигнализирует о том, что устройство желает получить доступ к шине;

– dataout[3..0] – выход, на который устройство подает данные, которые отправятся на шину;

– out\_busy – выход, активный уровень на котором сигнализирует о том, что устройство сейчас заняло шину, и другие не могут сделать то же самое.

В моей схеме первое устройство всегда получит доступ к шине при конфликте первым, второе устройство получить доступ к шине в том случае, если первому устройству не требуется доступ к шине и так далее. Таким образом и реализуется фиксированный приоритет, описанный выше.

В остальном схема достаточно тривиальна и подробного описания не требует.

**3** ФУНКЦИОНАЛЬНОЕ МОДЕЛИРОВАНИЕ

В этом разделе я покажу моделирование разработанных блоков, а именно временные диаграммы, сгенерированные в программе Quartus 9.1.

**3.1** Функциональное моделирование АЛУ

Временная диаграмма теста арифметико-логического устройства представлена на рисунке 3.1.

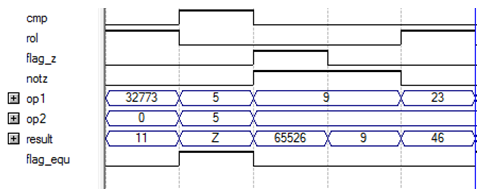


Рисунок 3.1 – Временная диаграмма АЛУ

С помощью линий cmp, rol и notz задается типа операции. Линия flag\_z отвечает за доставку флага для операции notz. Шины op1[15..0] и op2[15..0] нужны для передачи операндов. По выходной шине result[15..0] выдается результат команд, выходная линия flag\_equ – результат команды cmp.

**3.2** Функциональное моделирование стека

Временная диаграмма работы стека представлена на рисунке 3.2.

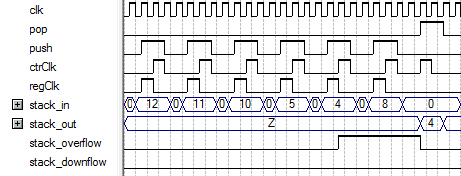


Рисунок 3.2 – Временная диаграмма работы стека. Команды push

Здесь видно, как постепенно заполняется весь стек и в итоге переполняется, о чем сигнализирует высокий логический уровень на лини stack\_overflow. После этого производится одна операция pop, которая освобождает одну ячейку и стек перестает быть переполненным.

На рисунке 3.3 показано продолжение моделирования.

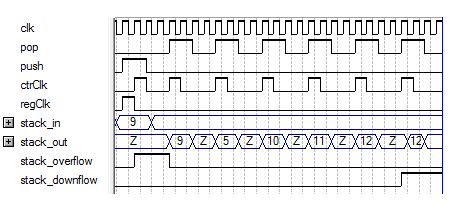


Рисунок 3.3 – Временная диаграмма работы стека. Команды pop

Здесь видно, как в освободившуюся ранее ячейку вновь записывается значение, затем идет череда команд pop. Постепенно выгружаются все записанные значения до тех пор, пока не произойдет выход за пределы стека снизу, о чем сигнализирует высокий логический уровень на линии stack\_downflow.

**3.3** Функциональное моделирование блока РОН

Временная диаграмма работы блока регистров общего назначения представлена на рисунке 3.4.

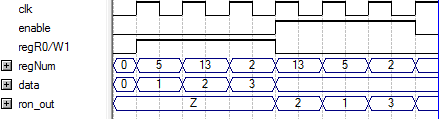


Рисунок 3.4 – Временная диаграмма блока РОН

Блок РОН представлен 14 регистрами (согласно заданию). На данной диаграмме видно, как по очереди происходит запись в 5-й, 13-й, 2-й регистры, затем из них производится считывание.

Сигналов regR0/W1 управляется режим работы (чтение либо запись), сигнал enable позволяет подавать блоку что-либо на выход.

В целом блок достаточно тривиален и дополнительно объяснения не требует.

**3.4** Функциональное моделирование блока памяти

Временная диаграмма работы блока памяти представлена на рисунке 3.5.

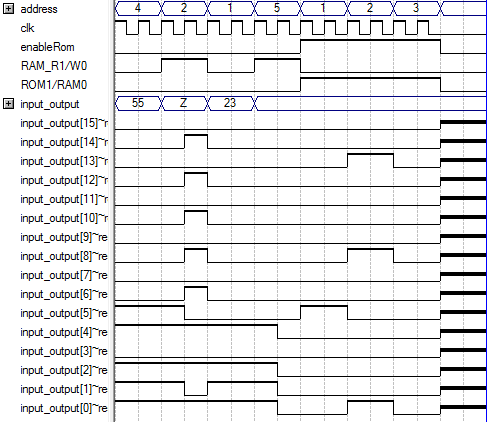


Рисунок 3.5 – Временная диаграмма работы блока памяти

Для более полного понимания происходящего на диаграмме стоит также показать содержимое ROM и RAM, которое представлено на рисунках 3.6 и 3.7.

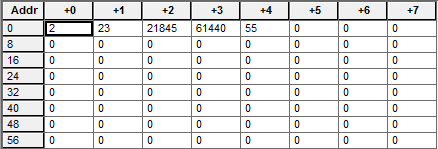


Рисунок 3.6 – Содержимое RAM

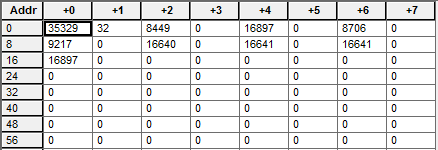


Рисунок 3.7 – Содержимое ROM

В начале диаграммы запись и чтение RAM. Результат записи можно увидеть на рисунке 3.6.

Во второй половине диаграммы происходит чтение из ROM. О правильности результата чтения можно удостовериться, взглянув на рисунок 3.7.

**3.5** Функциональное моделирование кэш

Временная диаграмма работы кэш представлена на рисунках 3.8 и 3.9, ввиду своей обильности.

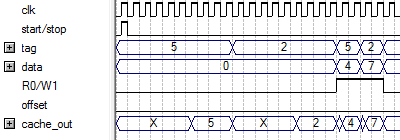


Рисунок 3.8 – Временная диаграмма работы кэш

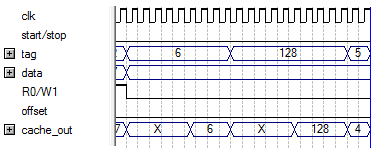


Рисунок 3.9 – Временная диаграмма работы кэш. Продолжение

Сначала на рисунке 3.8 видно, как запрашиваются данные с адресом 5, затем с адресом 2. Столь долгое время ожидания обусловлено тем, что первоначально таких строк в кэш не было. Затем видно, как происходит запись по адресам 5 и 2, которые уже находятся в кэш. Хорошо видно насколько быстрее происходит доступ, если данные уже находятся внутри.

На рисунке 3.9 видно, как продолжаются запросы и считываются новые данные. Размер одного набора кэш равен 32 битам. Когда мы запрашиваем данные с адресом 6, какая-то из строк должна была выгрузиться. Можно узнать, какая именно из них выгрузилась, взглянув на дампы памяти до начала работы кэш и после. Они представлены на рисунке 3.10 и 3.11.

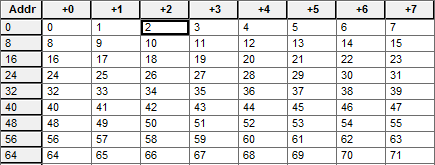


Рисунок 3.10 – Дамп памяти до начала работы

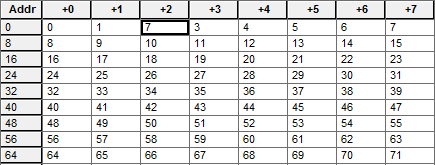


Рисунок 3.11 – Дамп памяти после работы кэш

На рисунках 3.8, 3.10 и 3.11 хорошо видно, как сначала мы запросили данные по адресу 2, затем записали туда 7, а после того, как запросили данные по адресу 6, строка кэш, содержащая 2, выгрузилась в память, и там уже мы наблюдаем 7 по адресу 2.

**3.6** Функциональное моделирование арбитража

Временная диаграмма работы арбитража представлена на рисунке 3.12. В этой схеме есть эмуляция трех устройств, которые имитируют некоторую вычислительную деятельность и занимают шину на различное время.

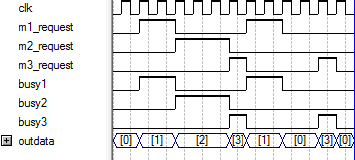


Рисунок 3.12 – Временная диаграмма работы арбитража

Судя по рисунку 3.12 показаться, что конфликтов не возникает, так как не встречается два запроса к шине в один момент времени. Однако на самом деле шина запрашивается несколькими устройствами одновременно, они просто не выдают сигнал до тех пор, пока шина занята. Такого поведения требует сама реализация децентрализованного арбитража.

**3.7** Функциональное моделирование предсказателя переходов

Временная диаграмма работы арбитража представлена на рисунке 3.13.

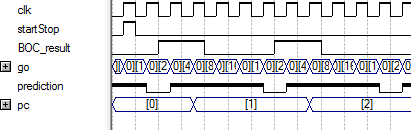


Рисунок 3.13 – Временная диаграмма работы предсказателя переходов

На рисунке 3.13 можно увидеть, как на линии prediction появляются предсказания. В данном случае предсказатель все 3 раза предположил, что перехода не будет. Вообще должно довольно много времени, прежде чем накопится достаточно истории для того, чтобы совершить точное предсказание. Это называется «холодный старт». Кроме того, В реальных системах предсказатель пытается найти закономерность в коде программы, в моем же случае закономерности нет: это просто симуляция.

**3.8** Функциональное моделирование системы в целом

Процессор представляет собой, как бы, черный ящик. Мы подаем что-то на вход и ожидаем что-то на выходе. Поэтому сама по себе временная диаграмма почти ничего не покажет. На рисунке 3.14 представлена временная диаграмма работы всей системы.

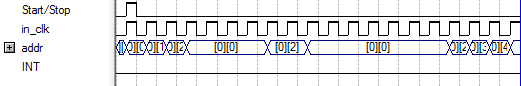


Рисунок 3.14 – Временная диаграмма работы всей системы

Процессор сам выполняет весь код, поданный на его вход, извне требуется минимальное вмешательство.

Программный код, выполняемый процессором, находится в ROM. Содержимое ROM показано на рисунке 3.15.

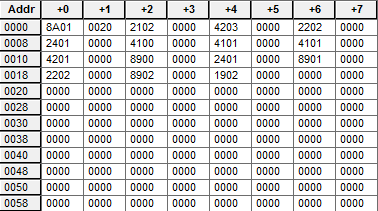


Рисунок 3.15 – Содержимое ROM

В таблице 3.1 дано описание того, что означает каждая ячейка

Таблица 3.1 – Описание содержимого ROM

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Числовая запись команды | | | | Мнемоническая запись |
| 1-я часть | 2-я часть | 31-15 биты | 15-0 биты |
| 8A01 | 0020 | 1000101000000001 | 0000000000100000 | mov R1, 2+R0 |
| 2102 | 0000 | 0010000100000010 | 0000000000000000 | cmp R2, R0 |
| 4203 | 0000 | 0100001000000011 | 0000000000000000 | pop R3 |
| 2202 | 0000 | 0010001000000010 | 0000000000000000 | notz R2 |
| 2401 | 0000 | 0010010000000001 | 0000000000000000 | rol R1 |
| 4100 | 0000 | 0100000100000000 | 0000000000000000 | push R0 |
| 4101 | 0000 | 0100000100000001 | 0000000000000000 | push R1 |
| 4101 | 0000 | 0100000100000001 | 0000000000000000 | push R1 |
| 4201 | 0000 | 0100001000000001 | 0000000000000000 | pop R1 |
| 8900 | 0000 | 1000100100000000 | 0000000000000000 | mov 0, R0 |
| 2401 | 0000 | 0010010000000001 | 0000000000000000 | rol R1 |
| 8901 | 0000 | 1000100100000001 | 0000000000000000 | mov 1, R0 |
| 2202 | 0000 | 0010001000000010 | 0000000000000000 | notz R2 |
| 8902 | 0000 | 1000100100000010 | 0000000000000000 | mov 2, R0 |
| 1902 | 0000 | 0001100100000010 | 0000000000000000 | jmp 2 |

Теперь покажем содержимое RAM до и после выполнения программы на рисунках 3.16 и 3.17.

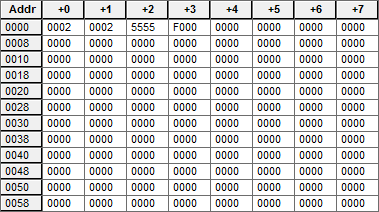


Рисунок 3.16 – Содержимое RAM до выполнения программы

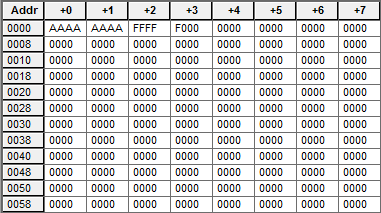


Рисунок 3.17 – Содержимое RAM после выполнения программы

# 4 АНАЛИЗ И ОПТИМИЗАЦИЯ РАЗРАБОТАННОЙ МИКРО-ЭВМ

Оптимизация, которую можно было бы сделать в данном проекте – это конвейеризация.

Конвейеризация любого процесса подразумевает его разделение на

примерно одинаковые по длительности этапы и позволяет выполнять

эти этапы одновременно и параллельно для нескольких объектов, над которыми производится конвейеризируемый процесс.

Как правило, цикл выполнения центральным процессором любой

команды состоит из следующих этапов:

– выборка команды;

– декодирование команды;

– вычисление адресов операндов;

– выборка операндов;

– исполнение команды;

– запись результата.

Несмотря на гипотетическую возможность выравнивания большинства из приведенных этапов по времени, на практике все этапы выполняются различное количество тактов.

Сама по себе реализация данного проекта очень плохо смогла бы лечь на схему конвейеризации, так как многие блоки не подразумевают параллельного чтения и записи.

ЗАКЛЮЧЕНИЕ

В данном курсовом проекте я реализовал микро-ЭВМ. Я получил достаточно хорошие знания в этой области. Сама ЭВМ вышла достаточно простой, однако свои функции она выполняет. Несмотря на то, что в данном проекте нет КПДП и некоторые блоки выполнены в стороне от основных блоков, все же данная ЭВМ способна функционировать правильно, вычислять, сохранять и загружать данные из памяти, работать со стеком, АЛУ, регистрами общего назначения и делать это достаточно быстро.

Кроме того, мной получены знания и о других аспекта работы ЭВМ благодаря реализации остальных блоков.

СПИСОК ЛИТЕРАТУРЫ

Столлингс, У. Структурная организация и архитектура компьютерных систем. 5-е изд. Пер. с англ. – М.: «Вильямс», 2001 – 892 стр.

Таненбаум, Э. Архитектура компьютерных систем. 4-е изд. Пер. с англ. – М.: «ПИТЕР», 2002 – 698 стр.