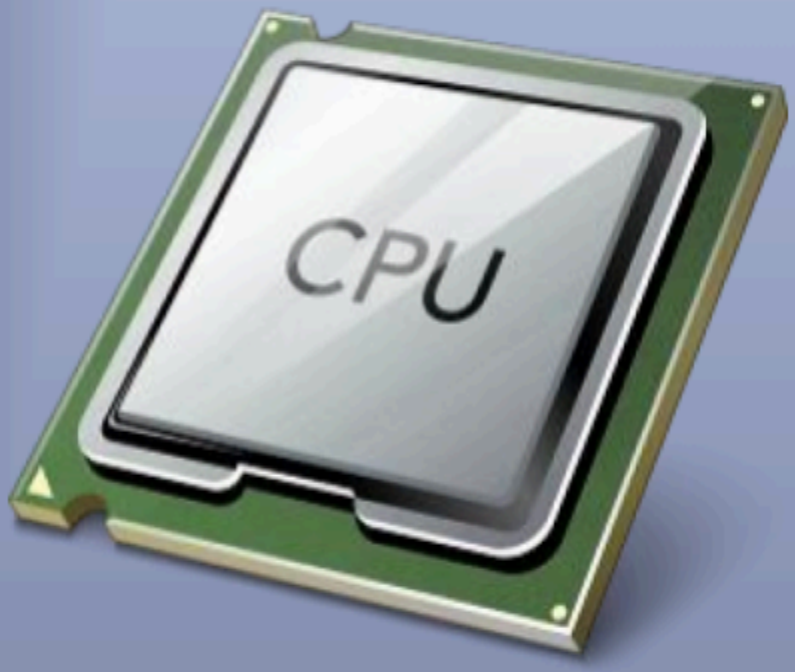


Реализация одноадресного процессора в Logisim

Том 1: Реализация ЦП в серии Logisim

В монографии реализован простой одноадресный процессор с использованием Logisim. Создан и объяснен работающий программируемый одноадресный процессор, включая язык ассемблера, используемый для процессора, ассемблер для перевода языка ассемблера в машинный код и то, как процессор использует машинный код для реализации программы.

Перевод by Bread

Реализация одноадресного процессора в Logisim

Описание
У большинства пользователей компьютеров есть неверная, но полезная когнитивная метафора: пользователь говорит (или набирает, или нажимает) что-то, и происходит нечто мистическое — почти интеллектуальное или магическое. Пользователи считают, что компьютеры подчиняются законам магии: вводится некое "заклинание", и компьютер отвечает ожидаемым, но магическим поведением.

На самом деле такого волшебного компьютера не существует. В реальности компьютер — это машина, и каждое его действие сводится к набору механических операций. Первое полное определение работающего компьютера было дано для механической машины, разработанной Чарльзом Бэббиджем в 1834 году, которая должна была работать на паровой тяге.

Вероятно, самым большим успехом компьютерной науки (КН) в XX веке стала разработка абстракций, скрывающих механическую природу компьютеров. Тот факт, что обычные люди используют компьютеры, никогда не задумываясь об их механистичности, — триумф специалистов по компьютерным наукам.

Цель данной монографии — разрушить абстрактное понимание компьютера и объяснить его поведение исключительно в механистических терминах. В частности, речь пойдет о центральном процессоре (ЦП), поскольку именно в нем происходит "волшебство". Все остальные компоненты компьютера можно рассматривать лишь как инструменты для предоставления информации процессору.

В этой монографии рассматривается конкретный тип процессора — одноадресный процессор. Его работа объясняется с использованием только стандартных логических элементов: **AND**, **OR**, **NOT**, **NAND** и **XOR**, а также четырех базовых интегральных схем (ИС) — **дешифратора**, **мультиплексора**, **сумматора** и **триггера (flip-flop)**. Все эти элементы можно описать как механические преобразователи входных сигналов в выходные, и тогда процессор в целом предстает как механическое устройство.

Ключевые слова
Цифровые микросхемы, архитектура системы, организация ЭВМ, интегральные микросхемы, логика ЭВМ, центральный процессор (ЦП), архитектура процессора, мультиплексор, дешифратор, арифметико-логическое устройство, регистровый файл, триггер (flip-flop), память, защелка памяти, сумматор, полный сумматор, полусумматор, автомат, конечный автомат, счетчик по модулю 4, серия 7400, лабораторное руководство по цифровым схемам, электронные схемы, электронные проекты, цифровые проекты, компьютерные науки онлайн, онлайн-лабораторное руководство.

Комментарии
В прилагаемом ZIP-файле должны содержаться все материалы к тексту, включая ассемблер, схемы Logisim, программы и рисунки. По мере разработки новых материалов они могут быть доступны в виде бета-версий на сайте http://chuckkann.com.

Вперед

Цель монографий этой серии — помочь студентам и другим людям, интересующимся компьютерными науками (CS), понять механическую природу самой важной части компьютера — центрального процессора. Эта серия предназначена для студентов и практиков CS, которые хотят глубже понять устройство процессора, но она также ориентирована на любителей, интересующихся тем, как компьютеры работают на самом деле.

Это первая монография в запланированной серии документов, описывающих и реализующих различные архитектуры центральных процессоров. Здесь представлен одноадресный центральный процессор (ЦП), реализованный автором в Logisim. За исключением некоторых простых интегральных схем (ИС) — дешифратора, мультиплексора, сумматора и триггера (flip-flop) — этот ЦП разработан с использованием только базовых логических элементов (AND, OR, NOT, XOR и NAND) и булевой логики. Таким образом, материал должен быть доступен даже любителям, желающим разобраться в принципах работы ЦП.

Этот процессор также можно использовать в курсах по организации или архитектуре компьютеров. В планируемой рабочей тетради для данного процессора будет представлен ряд проектов, которые студенты или энтузиасты смогут реализовать в Logisim для модификации и расширения его возможностей. Это поможет лучше понять принципы работы процессора. Проекты включают:

- Усовершенствование языка ассемблера.
- Модификацию ассемблера для обработки изменений.
- Аппаратные доработки для поддержки новых функций.

Книга может использоваться вместе с другими работами автора для создания полноценного курса по организации компьютера. Например:

- **«Проекты цифровых схем»** — реализация основных микросхем, используемых в данном учебнике, на макетных платах.
- **«Введение в программирование на языке ассемблера MIPS»** — знакомство с реальным ассемблером для архитектуры MIPS, включая структурированное программирование, вызов подпрограмм, работу с памятью (стек, куча, статические данные).

Автор использует эти три текста в своем курсе «Архитектура компьютера». Все они доступны для бесплатного скачивания (кроме рабочей тетради, которая распространяется за номинальную плату). Дополнительные ресурсы можно найти на сайте автора: http://chuckkann.com.

Данная монография — первая в серии, которая будет дополняться студенческими исследовательскими проектами. В планах — рассмотрение различных архитектур процессоров, включая:

- Различия между архитектурами Фон Неймана и Гарварда.
- 0-адресные, одноадресные и 2/3-адресные архитектуры.

Это поможет читателям понять принципы проектирования процессоров. Все проекты будут основаны на RISC-архитектуре, но возможно включение и CISC-архитектуры с 3 адресами, чтобы продемонстрировать её сложность и объяснить причины её упадка.

Содержание

Рисунки

1 Введение

У большинства пользователей компьютеров есть неверная, но полезная когнитивная метафора, в которой пользователь что-то говорит (или набирает, или нажимает), и происходит мистическое, почти интеллектуальное или магическое поведение. Пользователи компьютеров считают, что компьютеры подчиняются законам магии: вводится некое магическое заклинание, и компьютер отвечает ожидаемым, но магическим поведением.

На самом деле такого волшебного компьютера не существует. В реальности компьютер — это машина, и каждое действие, выполняемое компьютером, сводится к набору механических операций. Первое полное определение работающего компьютера — это механическая машина, разработанная Чарльзом Бэббиджем в 1834 году, которая должна была работать на паровой тяге.

Вероятно, самым большим успехом компьютерной науки (CS) в XX веке стала разработка абстракций, скрывающих механическую природу компьютеров. Тот факт, что обычные люди используют компьютеры, никогда не задумываясь об их механической сути, является триумфом разработчиков.

Цель данной монографии — разрушить абстрактное понимание компьютера и объяснить его поведение полностью в механистических терминах. Речь пойдет именно о центральном процессоре (ЦП) компьютера, поскольку именно в нем происходит «волшебство». Все остальные части компьютера можно рассматривать лишь как поставщиков информации для работы центрального процессора.

В этой монографии мы рассмотрим конкретный тип ЦП — одноадресный процессор — и объясним его работу, используя только стандартные логические элементы: AND, OR, NOT, NAND и XOR, а также 4 базовые интегральные схемы (ИС) — дешифратор, мультиплексор, сумматор и триггер. Все эти элементы и компоненты можно описать как механические преобразователи входных данных в выходные, и тогда процессор в целом можно рассматривать как механическое устройство.

Хотя для чтения этого текста не обязательно знать детали реализации этих микросхем (достаточно понимать, как они используются), их создание не представляет сложности. Бесплатную книгу по реализации этих интегральных схем можно получить у автора по адресу http://cupola.gettysburg.edu/oer/1/. В оставшейся части этой главы будет дан базовый обзор логических элементов и интегральных схем, используемых в данном тексте (раздел 1.1), а затем — обзор различных способов организации и архитектуры центрального процессора.

1.1 Основные компоненты процессора

В этом разделе рассматриваются основные компоненты процессора: логические элементы, а также четыре распространенные микросхемы, используемые в процессоре — сумматор, декодер, мультиплексор и регистр.

1.1.1 Булевы операции

Гейты - это аппаратная реализация булевых операций. Булевы операции - это операции, которые принимают одно или несколько двоичных значений и вычисляют результат. Например, операция AND принимает 2 двоичных значения (с 0 = ложь и 1 = истина) и вычисляет двоичный выход. Для операции AND входы 0 AND 0, 0 AND 1 и 1 AND 0 дают 0 (ложь), а вход 1 AND 1 дает 1 (истина). Обычно это реализуется с помощью таблицы истинности, как показано ниже:

Ввод		Вывод
A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1

Таблица 1-1: Таблица истинности для AND-шлюза

В этом тексте используются пять булевых операторов: **AND**, **OR**, **NOT**, **XOR** и **NAND**. Оператор **NOT** — унарный (принимает только один вход), его таблица истинности приведена в Таблице 1-2.

Ввод	Вывод
A	NOT
0	1
1	0

Таблица 1-2: Таблица истинности для затвора NOT

Операторы **AND**, **OR**, **XOR** и **NAND** — бинарные (принимают два входа), их таблицы истинности приведены в Таблице 1-3.

Ввод		Вывод			
A	B	AND	OR	XOR	NAND
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

Таблица 1-3: Таблица истинности для ворот AND, OR, XOR и NAND

1.1.2 Интегральные микросхемы

Интегральная микросхема (ИМС) - это набор затворов, которые используются для создания компонентов, реализующих определенное поведение. Компоненты ИС - это простые затворы, и все ИС в этой главе могут быть легко сведены к затворам AND, OR, NOT, XOR и NAND. Так же как ворота механически преобразуют вход в выход, микросхемы также механически преобразуют входы в выходы.

ИС, которые будут описаны в этой главе, - это сумматор, декодер, мультиплексор и триггер (flip-flop).

1.1.3 ALU (сумматор)

Арифметико-логическое устройство (АЛУ) - центральный компонент центрального процессора. Он выполняет все арифметические и логические операции над данными. Все остальное в процессоре предназначено для предоставления данных для работы ALU.

Обычно ALU представляет собой "черный ящик", который обеспечивает выполнение операций процессора над двумя операндами. Этот "черный ящик" отвечает за все операции, которые выполняет процессор, включая не только целочисленные и логические операции, но и вычисления с плавающей точкой. Такие операции, как вычисления с плавающей точкой, очень сложны и часто реализуются в сопроцессорах. Чтобы не усложнять ситуацию, в этом тексте для центрального процессора будут использоваться только целые типы данных, а также только целочисленные и логические операции.

Обзор АЛУ можно увидеть на примере типичного АЛУ, показанного ниже. АЛУ принимает два аргумента и реализует операции, такие как сложение, вычитание, умножение и деление, над этими двумя операндами. ALU также позволяет выполнять такие операции, как булевы операции (AND, OR, XOR и т. д.), сдвиг битов и сравнение.

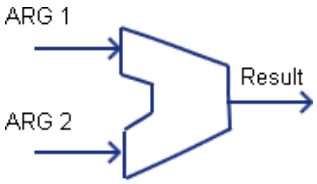


Рисунок 1-1: АЛУ

Поскольку единственной операцией АЛУ, рассматриваемой в рекомендуемом учебнике по ИС, является сумматор, АЛУ, используемое в реализации CPU в Logisim, будет содержать только схему сумматора. С помощью сумматора реализуется как сложение, так и вычитание. Более надежную конфигурацию АЛУ можно найти в дополнительных примечаниях, доступных к этому тексту.

1.1.4 Декодер

Декодер - это микросхема, разделяющая n-битное число на 2n отдельных выходных линий. Например, рассмотрим 2-битное число, которое может иметь 4 значения, 0x0 ... 0x3. Декодер принимает на вход 2 входные линии, представляющие 2-битное число, и включает одну (и только одну) из четырех выходных линий. Включенная линия соответствует значению 2-битного входа. Так, на следующей схеме, если на 2-битном входе обе линии имеют высокий уровень (представляющий "11"), включается выходная линия 3.

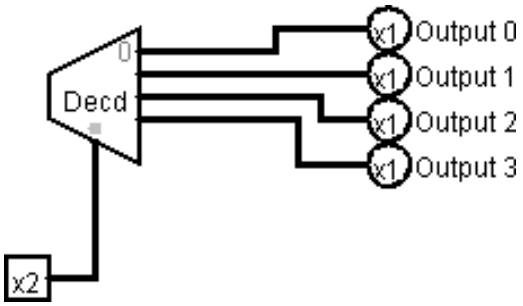


Рисунок 1-2: Декодер

1.1.5 Мультиплексор

Мультиплексор - это микросхема, которая выбирает между различными входами. На следующей схеме 8 бит, используемых выходом, могут поступать как из регистра 1, так и из регистра 2. MUX выбирает, какое 8-битное значение использовать. Если Select Input равен 0, выбирается регистр 1, а если Select Input равен 1, выбирается регистр 2.

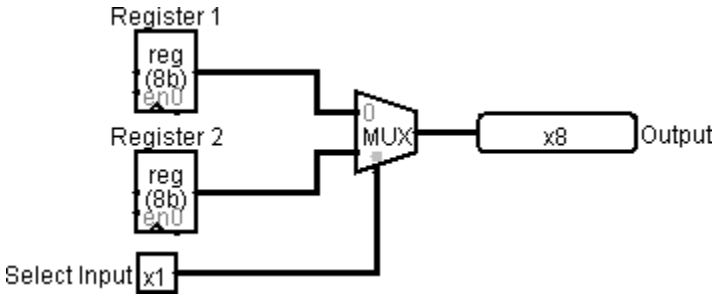


Рисунок 1-3: Мультиплексор

1.1.6 Регистры (D-триггеры) и память

Память отличается от других ИС тем, что она синхронная, где синхронность означает, что ячейка памяти имеет значение, которое находится в дискретных временных интервалах. Примером такого поведения является $ac в следующем фрагменте программы:

```
clac          ; time = t0, $ac = 5
addi 5        ; time = t1, $ac = 5
addi 7        ; time = t2, $ac = 5
subi 2        ; time = t3, $ac = 5
```

Эта программа показывает, что значение памяти, $ac, дискретно изменяется с течением времени. Это дискретное поведение обеспечивается системными часами. Системные часы - это электронный осциллятор, который вырабатывает квадратную волну с точной частотой. Ниже приведена иллюстрация квадратной волны.

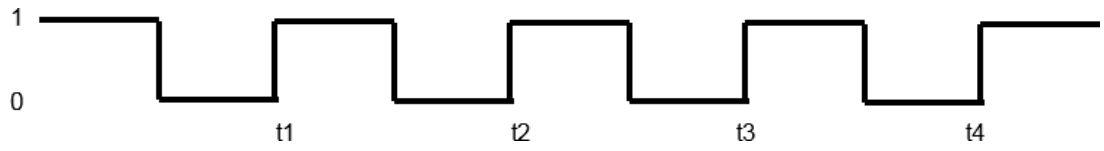


Рисунок 1-4: Квадратная волна

В квадратной волне значение всегда равно 0 или 1, и память использует переход от 0 к 1 (положительный фронт) для изменения значения всех компонентов памяти. Таким образом, ячейки памяти имеют дискретные значения, которые изменяются при каждом тактовом импульсе.

Любая ячейка памяти в процессоре обычно называется регистром. Регистровая память обычно состоит из статической памяти (SRAM) и реализуется с помощью флип-флопов. Основная память компьютера часто представляет собой динамическую память (DRAM), однако некоторые виды памяти, в частности кэш-память, могут быть реализованы с помощью SRAM. Специфика памяти выходит за рамки данного текста, и читателю достаточно знать, что регистровая память, как правило, состоит из SRAM и находится внутри процессора.

1.2 Сравнение компьютерных архитектур

Эта монография - первая в серии монографий, посвященных различным типам центральных процессоров, в которых два больших различия между типами процессоров - это формат адресов инструкций и то, как разделена память инструкций и данных процессора. В следующем разделе речь пойдет о формате адресов инструкций. В следующем разделе будут рассмотрены конструкции, в которых память инструкций и данных объединена (архитектура Фон Неймана) и разделена на память инструкций и данных (Гарвардская архитектура).

1.2.1 Нулевая, одноадресная и двух/трехадресная архитектура

Основное различие между архитектурами компьютеров с 0-, 1- и 2/3-адресами заключается в том, откуда берутся операнды для АЛУ. В этом разделе мы рассмотрим каждую из этих архитектур.

Обратите внимание, что во всех этих архитектурах операнды могут поступать из регистров/памяти, или же операнды могут быть частью самой инструкции. Например, значение, используемое в инструкции add A в одноадресном компьютере, поступает из ячейки памяти по адресу, соответствующему метке A, и инструкция добавляет значение в ячейке памяти A к значению $ac. В инструкции addi 5 значение операнда включено в инструкцию и называется немедленным значением. В этой серии монографий операторы, использующие немедленное значение, будут сопровождаться символом "i". Например, как показано выше, инструкция add использует значение памяти, а addi - немедленное значение.

1.2.1.1 0-Адресная архитектура

При обсуждении адресной архитектуры компьютера главный вопрос заключается в том, как извлекаются аргументы для АЛУ и где хранятся результаты работы АЛУ? 0-адресная архитектура извлекает (pops) два аргумента из верхней части стека операндов, выполняет операцию, а затем сохраняет (pushes) результат обратно в стеке операндов. Два операнда для ALU подразумеваются как два операнда на вершине стека, а операция, в данном случае add, не определяет никаких операндов. Поскольку оператор не принимает никаких явных операндов, 0-адреса включаются в операцию, и это называется 0-адресной архитектурой. Обратите внимание, что 0-адресную архитектуру часто называют стековой, поскольку она использует стек для передачи операндов в/из АЛУ.

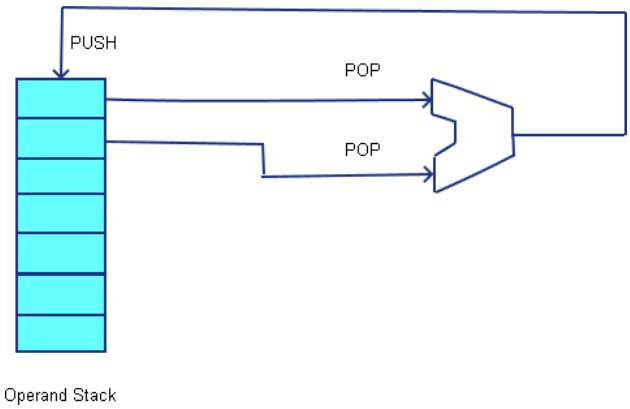


Рисунок 1-5: Архитектура 0-адреса

При написании ассемблерного кода для этой архитектуры операнды сначала заталкиваются в стек (из памяти или непосредственных значений) с помощью двух операций push. Затем выполняется операция, состоящая в том, чтобы вывести оба операнда из стека, выполнить АЛУ и вывести результат обратно в стек. Затем ответ сохраняется в памяти с помощью операции pop. Следующая программа, которая складывает значение переменной A и значение 5, а затем сохраняет результат обратно в переменную B, иллюстрирует простую программу с 0-адресом.

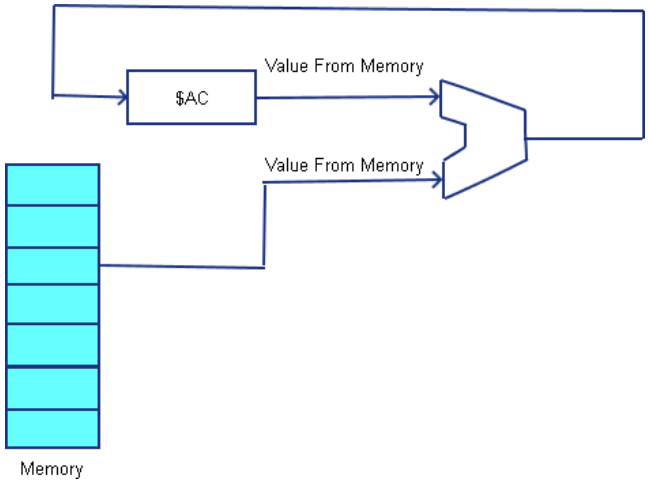
```
PUSH A      ; Загружаем значение переменной A в стек
PUSHI 5     ; Загружаем непосредственное значение 5 в стек
ADD         ; Складываем два верхних значения стека
POP B       ; Сохраняем результат в переменную B
```

Программа 1-1: 0-адресная программа для сложения двух чисел

Исторически существовали компьютеры с архитектурой 0-адреса, например, Burroughs 6500 и 7500, но в современных аппаратных архитектурах она используется редко, если вообще используется. Однако большинство современных языков, работающих на виртуальных машинах (ВМ), таких как Java Virtual Machine (JVM) или .Net Common Language Runtime (CLR), реализуют 0-адресную, или стековую, архитектуру.

1.2.2 Одноадресатная архитектура

В одноадресной архитектуре в процессоре ведется специальный регистр, называемый аккумулятором (Accumulator) или $ac. $ac всегда является подразумеваемым входным операндом для АЛУ, а также подразумеваемым местом назначения результата операции АЛУ. Вторым входным операндом является значение переменной памяти или мгновенное значение. Это показано на следующей диаграмме.



Программа 1-2: одноадресная архитектура

Ниже приведена простая одноадресная компьютерная программа, которая складывает значение 5 и значение переменной A и сохраняет результат в переменной B.

```
CLR     ; Установите AC в 0
ADDI 5  ; Добавьте 5 к $AC. Так как ранее он был равен 0, то загружается 5
ADD A ; Добавьте A+5 и сохраните результат в $AC
STOR B  ; Сохраните значение в $AC в переменной памяти B
```

Программа 1-3: одноадресная программа для сложения двух чисел

Поскольку в инструкции оператора ALU указывается только одно значение, такой тип архитектуры называется одноадресным. Поскольку в одноадресной архитектуре всегда есть аккумулятор, ее также называют аккумуляторной архитектурой.

Исторически многие ранние процессоры с одноадресной архитектурой, включая Intel 8080 и PDP-8, использовали аккумуляторную архитектуру. Из-за их простоты и возможности быть быстрее других архитектур некоторые микрокомпьютеры до сих пор используют аккумуляторную архитектуру, хотя в большинстве компьютеров применяются регистровые конструкции общего назначения.

1.2.3 Два/три - Архитектура адреса

Двухадресная и трехадресная архитектуры называются архитектурами регистров общего назначения. Двухадресная и трехадресная архитектуры работают одинаково. В обеих архитектурах некоторое количество регистров общего назначения используется для выбора двух входов в АЛУ, а результат работы АЛУ записывается обратно в регистры общего назначения.

Разница заключается в том, как задается результат операции ALU (регистр назначения). В трехадресной архитектуре 3 регистра - это регистр назначения (куда записываются результаты работы АЛУ), Rd, и два регистра-источника, предоставляющие значения для АЛУ, Rs и Rt. Это показано на следующем рисунке.

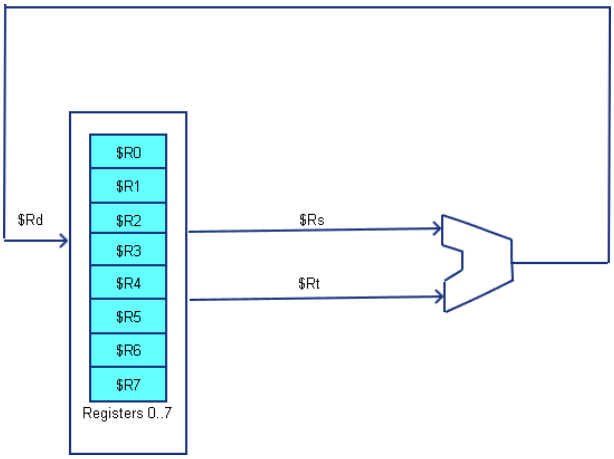


Рисунок 1-6: 3-адресная архитектура

2-адресная архитектура похожа на 3-адресную, с той лишь разницей, что в инструкции указываются только 2 регистра, первый из которых используется как для назначения операции, так и для первого источника в АЛУ.

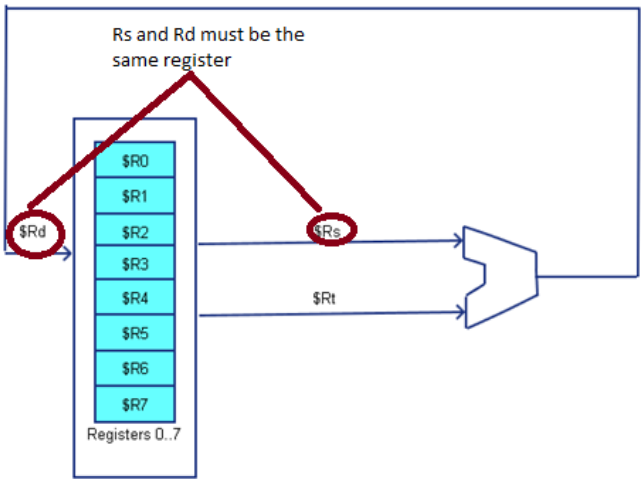


Рисунок 1-7: 2-адресная архитектура

Как показано на рисунках, центральный процессор выбирает два регистра общего назначения для отправки значений в ALU, и еще один выбор делается для записи значения из ALU обратно в регистр. В этой конструкции все значения, передаваемые в АЛУ, должны поступать из регистра общего назначения, а результаты работы АЛУ должны храниться в регистре общего назначения. Это требует доступа к памяти через операции загрузки и хранения, и правильное название для двух/трехадресного компьютера - "двух/трехадресный компьютер загрузки/хранения".

Следующие две программы выполняют ту же самую программу, B=A+5, что и в предыдущих примерах. В первом примере используется 3-адресный формат, а во втором - 2-адресный.

```
LOAD  $R0, A          ; Загружаем значение переменной A в регистр R0
LOADI $R1, 5          ; Загружаем константу 5 в регистр R1
ADD   $R0, $R0, $R1   ; Складываем значения в R0 и R1, результат в R0
STORE B, $R0          ; Сохраняем результат из R0 в переменную B
```

Программа 1-4: 3-адресная программа для сложения двух чисел

```
LOAD   $R0, A        ; Загружаем значение переменной A в R0
LOADI  $R1, 5        ; Загружаем константу 5 в R1
ADD     $R0, $R1     ; Суммируем R0 и R1, результат в R0
STORE  B, $R0        ; Сохраняем результат в переменную B
```

Программа 1-5: 2-адресная программа для сложения двух чисел

1.3 Архитектуры фон Неймана и Гарварда

При обсуждении того, как осуществляется доступ к памяти на уровне процессора, следует рассмотреть две конструкции. Первая - это архитектура фон Неймана, а вторая - Гарвардская архитектура. Основное различие между этими двумя архитектурами заключается в том, что в архитектуре фон Неймана вся память способна хранить все элементы программы, данные и инструкции; в гарвардской архитектуре память разделена на две памяти, одну для данных и одну для инструкций.

Для данной монографии основным вопросом при выборе архитектуры является то, что некоторые операции должны обращаться к памяти как для получения инструкции для выполнения, так и для доступа к данным для выполнения операции. Поскольку доступ к памяти возможен только один раз за такт, в принципе, архитектуре фон Неймана требуется не менее двух тактов для выполнения инструкции, в то время как гарвардская архитектура может выполнить инструкцию за один такт.

Способность гарвардской архитектуры выполнять инструкцию за один такт приводит к тому, что дизайн процессора становится намного проще и чище, чем при использовании фон-неймановской архитектуры. Для этой первой монографии будет использована реализация гарвардской архитектуры. В последующих монографиях будет рассмотрена реализация центрального процессора с использованием архитектуры Фон Неймана.

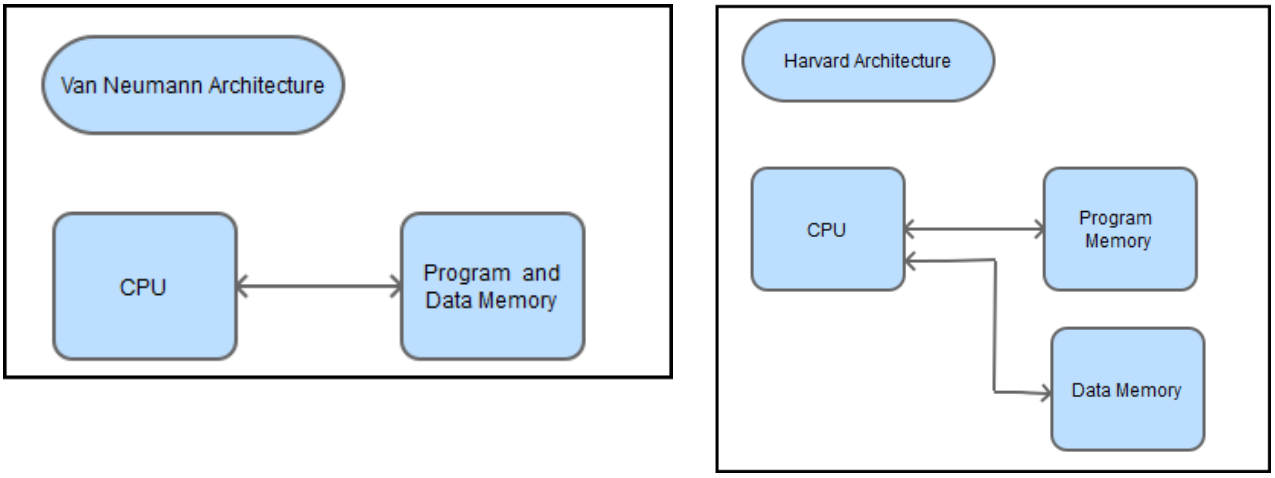


Рисунок 1-8: Разница между архитектурой фон Неймана и Гарвардской архитектурой

2 Язык ассемблера

2.1 Что такое язык ассемблера

Язык ассемблера - это очень низкоуровневый, читаемый человеком и программируемый язык, в котором каждая инструкция на языке ассемблера соответствует инструкции машинного кода компьютера. Программы на языке ассемблера напрямую транслируются в инструкции машинного кода, причем каждая инструкция ассемблера транслируется в одну инструкцию машинного кода.

После выбора базового формата адреса для архитектуры определяется формат языка ассемблера, называемый архитектурой набора инструкций (ISA). Следующий шаг - разработка всего языка ассемблера, который будет транслироваться и выполняться в центральном процессоре.

Ниже перечислены этапы разработки процессора.

1. Сначала был разработан язык ассемблера, который можно использовать для написания программ для этого процессора. Этот язык тестируется путем реализации простых программ на языке ассемблера.
2. Для языка ассемблера будет написано представление машинного кода. Процессор способен интерпретировать только двоичную информацию, поэтому язык ассемблера необходимо перевести в двоичные данные, которые будут понятны процессору.
3. Затем процессор разрабатывается для выполнения инструкций машинного кода.

Процесс разработки языка, создания машинного кода для него и реализации процессора обычно итеративен, но в данной монографии будет представлен только конечный продукт этих этапов.

Первый этап, создание языка ассемблера, является темой этой главы.

Для создания языка ассемблера необходимо определить три основных ограничения на язык.

1. Необходимо определить данные, которые будут обрабатываться в этом процессоре. В языках более высокого уровня это будут типы, такие как integer, float или string. В центральном процессоре типов не существует. Вместо этого процессор занимается такими вопросами, как размер слова в компьютере и то, как адреса памяти будут использоваться для получения данных.
2. Набор директив ассемблера для управления ассемблерной программой во время ее выполнения. Директивы определяют такие вопросы, как тип обращаемой памяти (текст или данные), метки для указания адресов в программе, способ выделения и хранения данных программы, а также способ определения комментариев.

[1] Обычно это соответствие один к одному между инструкциями языка ассемблера и инструкциями машинного кода устраняется. В языках ассемблера определяются псевдо инструкции, которые переводятся в несколько инструкций машинного кода. Эти псевдо инструкции предназначены для облегчения написания программ на языке ассемблера. Эта монография призвана показать читателю, как работает центральный процессор, поэтому каждой инструкции языка ассемблера будет соответствовать одна инструкция машинного языка.

3. Должен быть определен полный набор инструкций ассемблера.

В следующем разделе этой главы будут приведены некоторые предостережения для программистов, перешедших с языка более высокого уровня, о вопросах, которые они должны учитывать при программировании на языке ассемблера. В трех последующих разделах будут определены данные, используемые в ассемблере, директивы ассемблера и инструкции ассемблера.

2.2 Предостережения по языку ассемблера

Программисты, изучавшие языки более высокого уровня, такие как Java, C/C++, C# или Ada, часто имеют способы мышления о программе, которые неприемлемы для языков и систем низкого уровня, таких как язык ассемблера. В этом разделе мы дадим несколько советов программистам, впервые обращающимся к языку ассемблера.

Первое, что следует учесть, - это то, что все инструкции должны реализовывать примитивные операции. Языки более высокого уровня позволяют использовать короткую руку, которая подразумевает множество инструкций. Например, оператор B=A+5 подразумевает операцию загрузки, которая готовит переменные A и 5 для отправки в АЛУ. Затем необходимо выполнить операцию сложения в АЛУ. И наконец, необходимо выполнить операцию сохранения результата работы АЛУ обратно в переменную B. В ассемблере программист должен указать все необходимые примитивные операции. Не существует коротких путей.

Второе, что следует учесть: несмотря на то, что вы, возможно, слышали о том, что оператор goto - это плохо, не существует способа реализовать управление программой, например, операторы if или циклы, без использования инструкции branch, которая является эквивалентом оператора goto. Это не означает, что конструкции структурированного программирования не могут быть использованы эффективно. Если программист запутался в том, как реализовать конструкции структурированного программирования на ассемблере, в бесплатной книге по ассемблеру MIPS, написанной автором этой монографии, есть глава, в которой объясняется, как это можно сделать.

Третий важный момент, связанный с языком ассемблера, заключается в том, что данные не имеют контекста. В языке более высокого уровня обычно переменные A и B, а также число 5 задаются как целые числа. Язык более высокого уровня знает, что это целые числа, и затем предоставляет контекст для их интерпретации. Известно, что операция сложения - это целочисленное сложение, и компилятор генерирует инструкцию для выполнения целочисленного варианта, а не операции с плавающей точкой. Если бы объявление чисел было изменено на float, операция сложения в языке более высокого уровня была бы изменена на сложение с плавающей точкой. Язык более высокого уровня знает тип переменных и может предоставить правильный контекст для их интерпретации.

В языке ассемблера для любых данных не существует контекста. Данные могут быть целым числом, булевым значением, числом с плавающей точкой, символами ASCII или даже инструкциями программы. Ассемблер не имеет представления о типе и просто выполнит указанную операцию. В ассемблере можно выполнять бессмысленные операции, например, складывать вместе две программные инструкции. Язык ассемблера с радостью позволит вам делать бессмысленные и совершенно бессодержательные вещи и никоим образом не предупредит вас о том, что это бессмысленно. У ассемблера нет контекста для данных, и нет способа исправить эту проблему, потому что с точки зрения ассемблера проблемы не существует.

При программировании на языке ассемблера важно, чтобы программист знал текущий контекст программы. Именно программист знает, являются ли два элемента данных целыми числами, и, следовательно, уместна ли операция целочисленного сложения. Программист должен знать, являются ли значения, с которыми он работает, адресами или значениями, и выполнять соответствующие операции разыменования. Ничто, кроме знаний программиста, не гарантирует, что программа будет выполнять правильные операции над соответствующими типами данных.

2.3 Руководство по ассемблеру

Директивы ассемблера - это указания ассемблеру выполнить какое-либо действие или изменить параметр. Директивы ассемблера не представляют собой инструкции и не переводятся в машинный код.

В этом ассемблере все директивы начинаются с "." или "#" (комментарий - это #), и директива должна находиться на отдельной строке от любой другой директивы или инструкции ассемблера. Существует 4 директивы ассемблера и тег комментария.

- .text - Директива .text сообщает ассемблеру, что следующая информация - это текст программы (инструкции ассемблера), а транслированный машинный код должен быть записан в текстовый сегмент памяти.
- .data - Директива .data указывает ассемблеру, что следующая за ней информация является данными программы. Информация, следующая за инструкцией .data, будет являться значениями данных и будет храниться в сегменте данных.
- .label - Метка - это адрес в памяти, соответствующий либо инструкции, либо значению данных. Это просто удобство, чтобы программист мог ссылаться на адрес по имени. Она будет использоваться следующим образом:

  Имя .label

  Метка - это метка, на которую можно ссылаться вместо адреса в любой инструкции ассемблера, которая может принимать метку/адрес. Метки и адреса могут использоваться как взаимозаменяемые.

- .number - Директива number указывает ассемблеру выделить 2 байта памяти под значение данных и инициализировать память заданным значением. Она часто используется вместе с директивой .label для установки метки на 2-байтовое значение в памяти и инициализации этого значения, как показано в следующем фрагменте кода.

```
.label var1        ; Здесь начинается история переменной
.number 5          ; Её судьба – хранить число пять
```

  Этот оператор выделяет место для переменной var1 и присваивает этому месту в памяти значение 5. Данные для этого процессора будут работать только с 2-байтовыми (16-битными) целыми числами, поэтому можно использовать дискретные значения от -32768...32767 (включительно). Все значения данных приведены в десятичной системе; ассемблер не распознает шестнадцатеричные значения.

- # - символ # (хэштег) используется для указания комментария. Все, что в строке начинается с "#", является строкой комментария и игнорируется ассемблером.

2.4 Типы данных

Хотя в языке ассемблера нет явных типов данных, существуют правила доступа к данным и их хранения. В этом разделе определяются правила доступа к данным.

В этом процессоре слово - это 16 бит. Все ячейки памяти имеют ширину 16 бит, и адресовать можно слова, а не байты. Таким образом, значение по адресу 0 содержится в битах 0...15, значение по адресу 1 - в битах 16...31 и т. д.

Каждый адрес относится к 2-байтовому значению или слову. Если эта ячейка памяти находится в памяти данных, то значение является целым числом; если адрес находится в текстовой памяти, то это 2-байтовая инструкция.

Всего в памяти данных и текстовой (программной) памяти 256 ячеек (адресуемых слов). Адреса обеих памяти начинаются с 0 и заканчиваются 255, что соответствует 8-битному беззнаковому значению. Хотя адреса памяти пересекаются, контекст запроса определяет, какая память будет использоваться. Только $pc будет использоваться для доступа к текстовой памяти, а все остальные адреса будут относиться к памяти данных.

При обращении к значениям в инструкциях (непосредственные значения и адреса) используется 8-битное значение. Это 8-битное значение может быть задано либо как числовое значение, либо как допустимая метка для адреса в программе. При использовании в качестве адреса это 8-битное значение будет беззнаковым и относится к числу в диапазоне 0...255. Для инструкций add, sub и stor это адрес в сегменте данных. Для инструкций ветвления, beqz, 8-битный адрес относится к текстовому сегменту.

Для непосредственных инструкций, addi и subi, значение операнда является 8-битным целым числом и имеет значение от -128...127.

2.5 Проектирование языка ассемблера

При разработке языка ассемблера, языка для работы с процессором, существует три основных проблемы:

1. Передача данных из основной памяти во внутреннюю память процессора (регистры или стек операндов).
2. Набор операций, которые может выполнять АЛУ над данными, например сложение, вычитание, и, сдвиг и т. д.
3. Способ управления программой, например, для реализации структур типа ветвления (if) и зацикливания (for или while) в программе. Обычно структура управления обеспечивается операцией ветвления.

Эти три основные проблемы и то, как они решаются в языке ассемблера, будут рассмотрены в следующих разделах. В последнем разделе этой главы будут приведены несколько программ, которые проиллюстрируют, как пишется программа на этом языке ассемблера.

2.5.1 Перенос данных из основной памяти во внутреннюю память процессора

Объем памяти, непосредственно доступный программисту в процессоре (например, регистры), очень ограничен. В случае одноадресной архитектуры программист может напрямую использовать только один слот памяти - $ac. Поэтому программы должны полагаться на основную память для хранения программных инструкций и данных.

Для переноса элементов из памяти данных в $ac используются инструкции add, sub и stor.

Для инструкций add и sub вторым операндом инструкции является метка или адрес памяти значения, которое нужно извлечь из памяти и передать в АЛУ. Так, например, для загрузки значения в $ac из ячейки памяти с меткой A используется следующий код.

```
clac     ; Очистить аккумулятор
add A    ; Прибавить значение переменной A
```

Обратите внимание, что перед загрузкой значения в $ac необходимо всегда устанавливать $ac в 0 (с помощью clac), иначе значение, хранящееся в $ac, будет результатом сложения значения в ячейке памяти A с текущим значением в $ac.

Для инструкции stor вторым операндом является метка или адрес памяти, в которую нужно поместить значение из $ac. Например, чтобы сохранить значение в $ac в память по адресу в метке B, используется следующий код.

```
stor B   ; Сохранить результат в переменную B
```

2.5.2 Набор допустимых операций АЛУ

Следующее соображение - это набор операций, которые АЛУ может выполнять над входными данными. Этот список зависит от сложности АЛУ. АЛУ в этом компьютере очень простое, и поэтому будет поддерживать только операции сложения и вычитания.

2.5.3 Управление программой (разветвление)

Для выполнения любой полезной программы необходимо поддерживать конструкции if и цикла. В реализованном одноадресном процессоре это достигается с помощью операции Branch-if-equal-zero (beqz). Для этой операции, если $ac равен 0, программа переходит по адресу текстовой памяти, который содержится в оператор ветвления. Этот адрес может быть либо меткой, представляющей адрес, либо числовым значением адреса ветвления. Так, в следующей инструкции программа перейдет по адресу метки EndLoop, если значение в $ac равно 0.

```
beqz EndLoop    ; Переход в конец цикла если равно нулю
```

Часто используется оператор безусловного ветвления, но его можно смоделировать, предварительно установив значение $ac в 0 перед оператором ветвления. Следующая инструкция реализует безусловное ответвление.

```
clac             ; Очистить аккумулятор
beqz StartLoop   ; Если ноль - переход к началу цикла
```

2.5.4 Инструкции ассемблера

На основе критериев предыдущего раздела определяется минимальный набор инструкций ассемблера для создания полезных программ. Этих инструкций достаточно для создания полезных программ, и несколько примеров будут показаны в конце этой главы.

- `add [label/address]` – Добавьте значение из памяти данных (dm) к текущему значению $ac.

```
$ac <- $ac + dm[address]   ; $ac ← $ac + dm[address]
```

  В этой инструкции может быть использована либо метка, либо фактический адрес значения. Таким образом, если метка A относится к адресу dm, равному 5, то следующие две инструкции будут одинаковыми.

```
add A     ; Прибавить значение переменной A
add 5     ; Прибавить непосредственное значение 5
```

- `addi immediate` - Добавьте непосредственное значение в этой инструкции к $ac. Непосредственное значение - это 8-битное целое значение в диапазоне -128...127

```
$ac <- $ac + immediate
```

  Ниже приведен пример инструкции `addi`, которая добавляет 15 к значению в $ac.

```
addi 15      ; Добавить непосредственное значение 15
```

- `beqz [label/address]` – Инструкция beqz изменяет значение в счетчике программ ($pc) на адрес текстовой памяти в инструкции, если значение в $ac равно 0. В этом процессоре $pc всегда определяет следующую инструкцию для выполнения, так что это имеет эффект изменения следующей инструкции для выполнения на адрес в инструкции. Это называется программной ветвью, или просто ветвью.

```
$pc <- address IF $ac is 0
```

  Ниже приведен пример инструкции beqz, которая выполняет ветвление на адрес 16, если $ac равен 0:

```
beqz 16      ; Переход на адрес 16 если равно нулю
```

- `clac` – Инструкция clac устанавливает значение $ac в 0. Это можно сделать с помощью набора операций stor и sub, поэтому инструкция служит в основном для удобства.

```
$ac <- 0
```

- `sub [label/address]` - Вычитание значения из памяти данных в текущее значение $ac.

```
$ac <- $ac - dm[address]
```

В этой инструкции может использоваться либо метка, либо фактический адрес значения. Таким образом, если метка A относится к адресу dm, равному 5, то следующие две инструкции будут одинаковыми.

```
sub A      ; Вычесть значение переменной A
sub 5      ; И ещё пять капель в никуда
```

- `subi immediate` - Подставляйте непосредственное значение этой инструкции в $ac. Непосредственное значение - это 8-битное целое значение в диапазоне -128...128.

```
$ac <- $ac - immediate
```

Ниже приведен пример инструкции subi, которая добавляет 15 к значению в $ac.

```
subi 15     ; Вычесть непосредственное значение 15
```

- `stor [label/address]` – Сохраните текущее значение в $ac в памяти данных.

```
dm[address] <- $ac
```

В этой инструкции может быть использована либо метка, либо фактический адрес значения. Таким образом, если метка A относится к адресу dm, равному 5, то следующие две инструкции будут одинаковыми.

```
stor A      ; Сохранить значение в переменную A
stor 5      ; Сохранить непосредственное значение 5
```

- `noop` – Этот оператор ничего не делает. Выполнение этого оператора не изменяет значение какой-либо памяти или регистров (кроме $pc) в системе. Он включен для того, чтобы текстовая память могла быть установлена в 0 и выполнена без изменения внутреннего состояния компьютера.

2.6 Программы на ассемблере

Следующие программы на ассемблере иллюстрируют, как язык ассемблера, определенный в этой главе, может быть использован для реализации некоторых простых программ.

2.6.1 Загрузка значения в AC

Эта первая программа загружает в регистр $ac мгновенное значение 5. После выполнения программы значение в регистре $ac будет равно 5.

```
.text         ; Начало текстового сегмента
    clac      ; Очистить аккумулятор
    addi 5    ; Добавить непосредственное значение 5
```

Программа 2-1: Загрузка значения в $ac из непосредственного значения

Эта вторая программа загружает значение из адреса памяти, соответствующего метке var1 в переменную $ac. Поскольку значение по адресу var1 равно 5, программа загружает значение 5 в переменную $ac.

```
.text             ; Начало секции кода
    clac          ; Обнуление аккумулятора
    add var1      ; Прибавить значение переменной var1

.data             ; Начало секции данных
.label var1       ; Метка переменной
    .number 5     ; Инициализация значением 5
```

Программа 2-2: Загрузка значения в память $ac с помощью метки

Эта третья программа добавляет значение по адресу 0 в сегменте данных к значению $ac. Поскольку значение 5 было загружено в качестве первого значения в сегменте данных .data, значение 5 загружается в $ac.

```
.text            ; Секция кода
    CLAC         ; Очистка аккумулятора
    ADD 0        ; Прибавить значение из ячейки 0

.data            ; Секция данных
    .number 5    ; Число 5 по адресу 0
```

Программа 2-3: Загрузка значения в $ac из памяти с помощью ссылки

2.6.2 Сложение двух непосредственных значений

Эта программа иллюстрирует сложение двух мгновенных значений в $ac. Инициализация $ac равна 0, а затем в $ac загружается первое значение из непосредственного значения в инструкции. Непосредственное значение из второй инструкции затем добавляется к $ac, и $ac содержит окончательный результат.

```
.text
    clac            ; Обнуление аккумулятора ($ac = 0)
    addi 5          ; $ac = $ac + 5 → $ac = 5
    addi 2          ; $ac = $ac + 2 → $ac = 7
    # Теперь в $ac находится результат: 7
```

Программа 2-4: Сложение двух мгновенных значений

2.6.3 Сложение двух значений из памяти и сохранение результатов

Эта программа добавляет из памяти два значения по меткам var1 и var2, складывает их и сохраняет результат обратно в значение по метке ans. Эта программа также вводит новую конструкцию, которую мы назовем halt.

halt - это набор инструкций, который создает бесконечный цикл в конце программы, чтобы программа не просто продолжала выполнять инструкции noop, пока не закончится память. Эта конструкция устанавливает $ac в 0, а затем снова и снова переходит к одному и тому же оператору. Программа выполняется, но не продвигает $pc и не изменяет состояние компьютера.

```
.text
    clac            ; Обнуляем аккумулятор ($ac = 0)
    add var1        ; $ac += значение var1 (5)
    add var2        ; $ac += значение var2 (2)
    stor ans        ; Сохраняем результат в ans

.label halt        ; Метка точки останова
    clac            ; Очищаем аккумулятор
    beqz halt       ; Безусловный переход (цикл на себе)

.data

# Ответ будет находиться в памяти по адресу ans (0)
.label ans
    .number 0       ; Ячейка для результата (инициализирована 0)

.label var1
    .number 5       ; Первое слагаемое

.label var2
    .number 2       ; Второе слагаемое
```

Программа 2-5: Сложение двух значений и сохранение в памяти

2.6.4 Умножение на итеративное сложение

Эта программа выполняет умножение двух чисел методом итерации. Это означает, что n*m вычисляется путем прибавления n к самому себе m раз, например, n + n + n... и т.д.

```
.text
# ================================================
#      Программа умножения через сложение
#      Алгоритм: multiplicand * multiplier = product
#      Реализовано через циклическое сложение
# ================================================

.label startLoop
      # Инициализация цикла
      clac                    ; Очистка аккумулятора
      add multiplier          ; Загрузка множителя
      sub counter             ; Вычитание счетчика
      beqz endLoop            ; Если (multiplier-counter) == 0, выход

      # Вычисление произведения
      clac                    ; Очистка аккумулятора
      add product             ; Текущее значение product
      add multiplicand        ; Прибавляем multiplicand
      stor product            ; Сохраняем новое значение

      # Инкремент счетчика и повтор цикла
      clac                    ; Очистка аккумулятора
      add counter             ; Текущее значение счетчика
      addi 1                  ; Увеличиваем на 1
      stor counter            ; Сохраняем новое значение
      clac                    ; Очистка аккумулятора
      beqz startLoop          ; Безусловный переход в начало

.label endLoop
      # Точка завершения программы
      clac                    ; Очистка аккумулятора
      beqz endLoop            ; Бесконечный цикл (остановка)

.data

# ================================================
#             Область данных программы
# ================================================
.label multiplicand           ; Множимое (5)
      .number 5

.label multiplier             ; Множитель (4)
      .number 4

.label counter                ; Счетчик итераций
      .number 0               ; Инициализирован нулем

.label product                ; Результат умножения
      .number 0               ; Инициализирован нулем
```

Программа 2-6: Умножение с использованием итеративного сложения

3 Машинный код

Машинный код - это представление программы на языке ассемблера, которое может понять аппаратное обеспечение центрального процессора. Поскольку процессор понимает только двоичную форму, машинный код - это двоичный язык, который управляет процессором. Когда мы пишем машинный двоичный код, чтобы человеку было легче его читать, код собирается в группы по 4 бита, а результат записывается в шестнадцатеричном виде (основание 16).

3.1 Обзор формата команд машинного кода

Все инструкции машинного кода для нашего компьютера будут состоять из двух 4-битных сегментов и одного 8-битного сегмента, как показано ниже.

Рисунок 3-1: Формат 16-битных машинных команд

Первый 4-битный сегмент будет представлять тип операции. Возможны следующие типы операций:

- 0 - Это инструкция без операции (или noop). Она не изменяет текущее состояние компьютера и просто переводит процессор к следующей инструкции.
- 1 - Этот опкод представляет собой немедленную операцию, которая использует АЛУ для получения результата. Эта инструкция состоит из 4-битного опкода, 4-битной опции ALU (ALUopt), указывающей ALU, какую операцию выполнять, и 8-битного значения операнда. В реализованном виде ALU выполняет только 2 операции, 0x0 - сложение и 0x1 - вычитание, хотя упражнения в конце текста добавляют больше операций. В процессоре может быть реализовано максимум 16 операций.

  Далее следуют примеры перевода этих инструкций ассемблера в машинный код.

  Инструкция:

  ```
  addi 2
  ```

  переводится в следующий машинный код:

  ```
  0x1002
  ```

  Инструкция:

  ```
  subi 15
  ```

  переводится в следующий машинный код

  ```
  0x110f
  ```

- 2 - Этот опкод представляет операцию с адресом памяти, которая использует ALU для получения результата. Эта инструкция состоит из 4-битного опкода, 4-битного ALUopt, указывающего ALU, какую операцию выполнять, и 8-битного адреса памяти данных для операнда. В реализованном виде ALU выполняет только 2 операции, 0x0 - сложение и 0x1 - вычитание, хотя упражнения в конце текста добавляют больше операций. В процессоре может быть реализовано максимум 16 операций.

Далее следуют примеры перевода этих инструкций ассемблера в машинный код.

Инструкция:

```
add 2
```

переводится в следующий машинный код:

```
0x2002
```

Инструкция:

```
sub 15
```

переводится в следующий машинный код:

```
0x210f
```

Обратите внимание, что в процессе сборки метки в ассемблерном коде транслируются в адреса, поэтому метки никогда не появятся в машинном коде.

- 3 - Этот опкод выполняет операцию clac (например, устанавливает $ac в 0). В этой инструкции все последующие биты после 0x3 игнорируются, поэтому они могут содержать любое значение. По соглашению, дополнительные биты всегда должны быть установлены в 0.

  Например, следующая инструкция по сборке:

  ```
  clac
  ```

  переводится как:

  ```
  0x3000
  ```

- 4 - Этот опкод выполняет операцию stor. В этой инструкции 4-битный опт ALU не используется и должен быть установлен в 0. Значение адреса - это адрес, по которому нужно сохранить значение в $ac. Например, следующая инструкция

  ```
  stor 15
  ```

  переводится как:

  ```
  0x400f
  ```

- 0x5 - Опкод выполняет операцию beqz. В этой инструкции 4-битная операция ALU не используется и должна быть установлена в пустое значение. В результате выполнения этой операции $pc устанавливается в значение адреса, если $ac равен нулю. Установка значения $pc приводит к ветвлению программы на этот адрес.

  Например, следующая инструкция:

  ```
  beqz 40
  ```

  Переводится как:

  ```
  0x502
  ```

4 Программа на ассемблере

Ассемблер - это программа, которая переводит программу на языке ассемблера в машинный код. Ассемблер считывает исходный текст программы, состоящий из инструкций ассемблера. Затем ассемблер записывает два файла, которые будут использоваться для выполнения программы в процессоре Logisim.

Первый выходной файл содержит сегмент машинного кода, содержащий программу в машинном коде, которая будет использоваться в процессоре. Это перевод инструкций ассемблера в машинные инструкции. Второй выходной файл содержит сегмент инициализированных данных, которые будут использоваться в процессоре. Это показано на следующем рисунке.

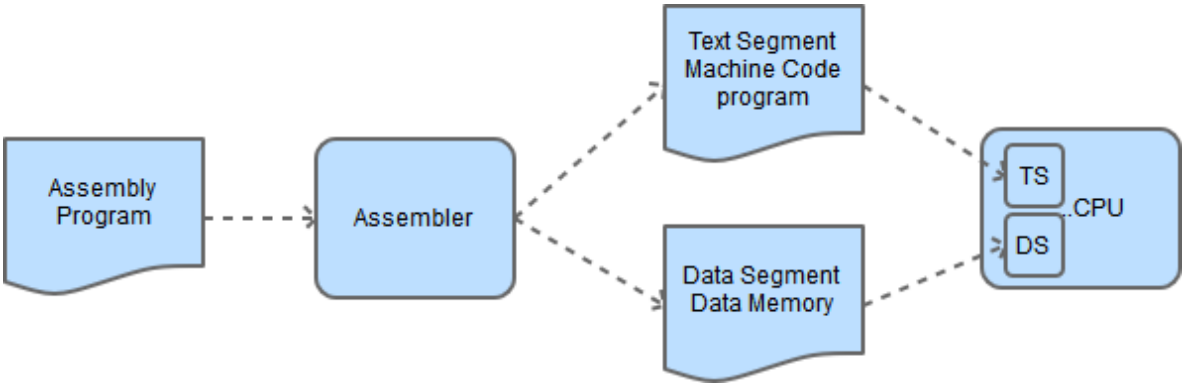


Рисунок 4-1: Процесс сборки

Ассемблер является двухпроходным ассемблером. Двухпроходной ассемблер считывает входной файл дважды, или за 2 прохода от начала до конца исходного ассемблерного файла. В первом проходе вычисляется адрес для каждой метки в программе, чтобы создать таблицу символов. Таблица символов - это список, содержащий метки в программе и их адреса в памяти. Второй проход переводит каждую инструкцию во входном файле в машинный код и записывает файл, который соответствует машинному коду для данных и текстовым сегментам для программы. Во время второго прохода по файлу используется таблица символов для разрешения ссылок на метки, которые используются в инструкциях ассемблера. Формат программы показан на следующем рисунке.

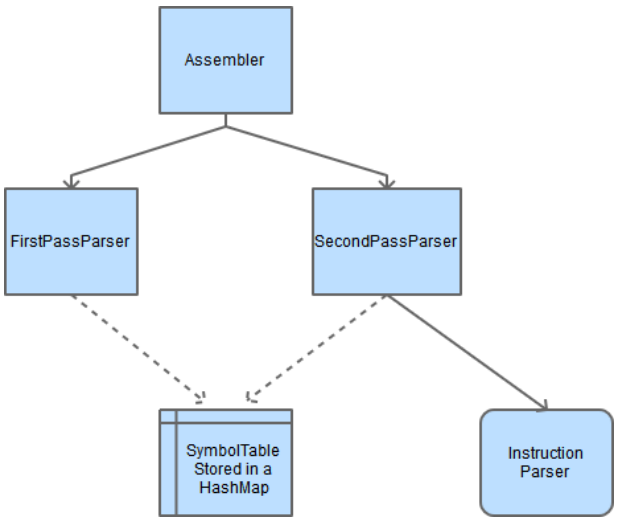


Рисунок 4-2: Обзор ассемблера

Ассемблер написан на языке Java. Нет никаких особых причин или преимуществ в написании этого ассемблера на Java. Основной проблемой ассемблера является написание обратных вызовов или виртуальных методов/функций для разбора отдельных инструкций. Вероятно, это было бы проще на C/C++, и уж точно быстрее при использовании указателей функций вместо полиморфизма. Но концепция обратного вызова сложна в любом языке для читателей, не знакомых с этой концепцией, а этот ассемблер более чем достаточно быстр.

Для реализации полиморфной структуры обратных вызовов для обработки команд ассемблер использует статические инициализаторы с фабричными шаблонами и синглтонные объекты. Это можно было бы сделать в большом операторе "if", но это полиморфное решение чище и легче расширяется. Кроме того, читателям, не знакомым с паттерном фабрики, паттерном синглтона и статическими инициализаторами, будет полезно с ними ознакомиться. Но ассемблер - это относительно короткая и простая программа, и читатели могут переписать ее, используя любую структуру или язык, который им больше нравится.

Ассемблер можно получить на сайте автора http://chuckkann.com. В zip-файле содержится ассемблер, файл определения схемы для одноадресного процессора, а также несколько программ и файлов для работы с процессором.

4.1 Выполнение программы на одноадресном процессоре

В следующих инструкциях подробно описано, как использовать одноадресный процессор.

1. Сначала скачайте файл One-AddressCPU.zip с сайта автора, http://chuckkann.com. Разархивируйте файл в подходящем месте.
2. В корневом каталоге этого zip-файла находится исполняемый JAR-файл с именем OneAddressAssembler.jar. Дважды щелкните на этом значке, и вы увидите следующее окно.

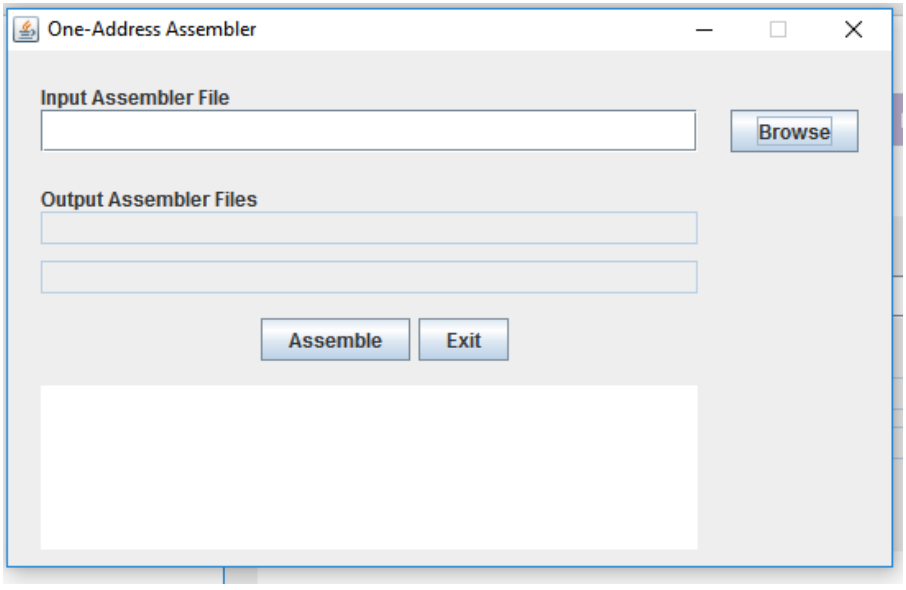


Рисунок 4-3: Запуск ассемблера - шаг 1

Нажмите кнопку Browse (Обзор), после чего появится диалоговое окно с файлами. Выберите каталог AssemblyPrograms.

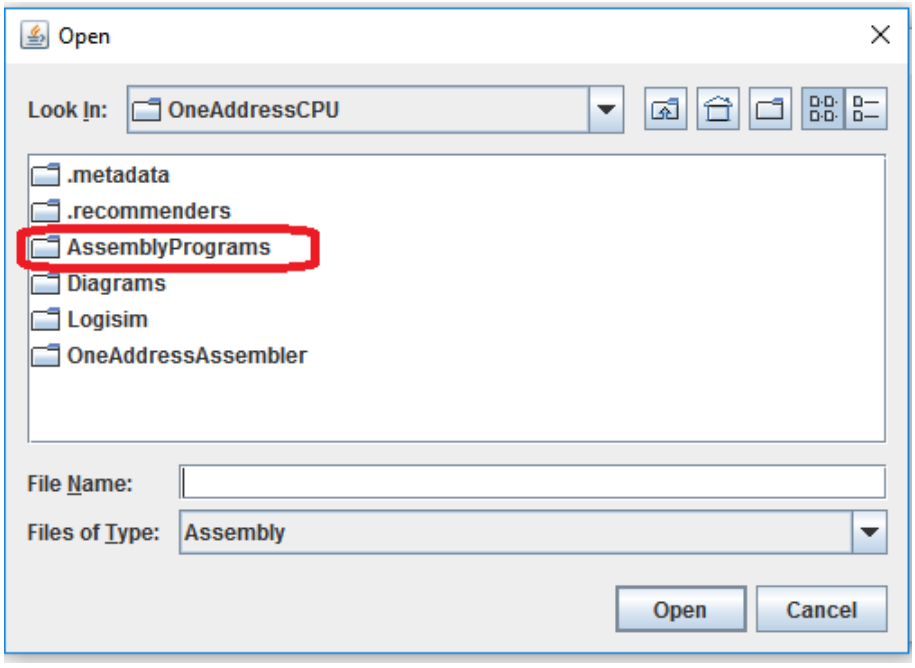


Рисунок 4-4: Запуск ассемблера - шаг 2

3. Выберите программу AdditionExample.asm. Это программа 2.5 из главы 2, которая складывает два значения из памяти и сохраняет результат в памяти.

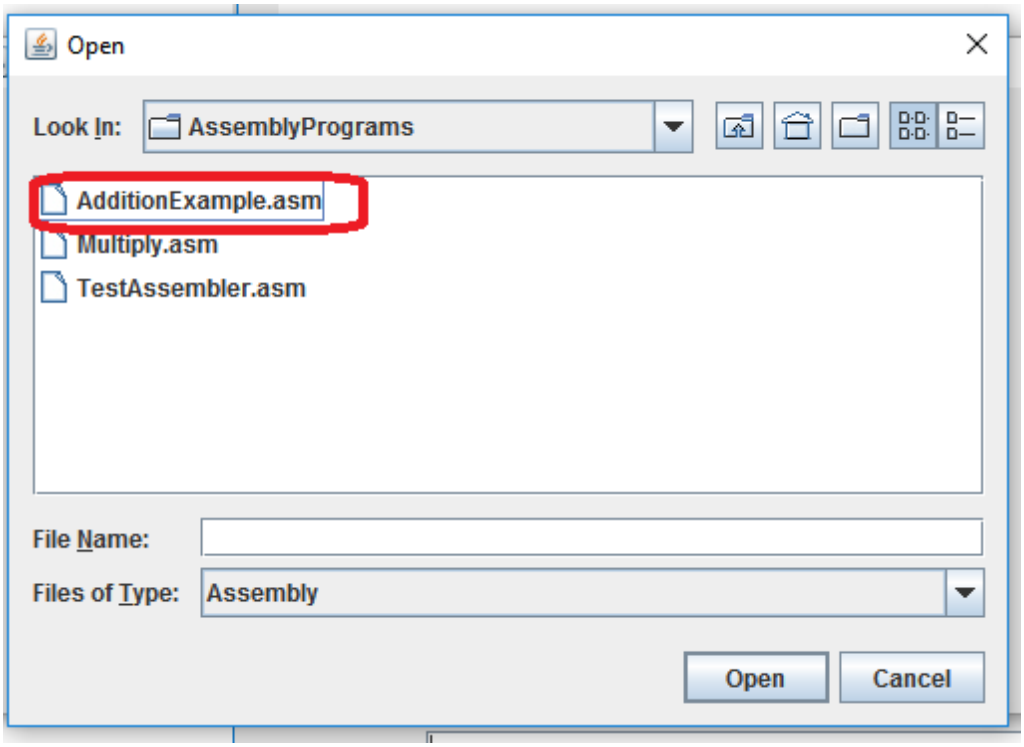


Рисунок 4-5: Запуск ассемблера - шаг 3

4. Программа выведет следующее окно, в котором говорится, что ассемблер использует в качестве входного файла AdditionExample.asm и при успешном завершении создаст два выходных файла - AdditionExample.mc (машинный код) и AdditionExample.dat (данные). Щелкните на кнопке Assemble (Сборка).

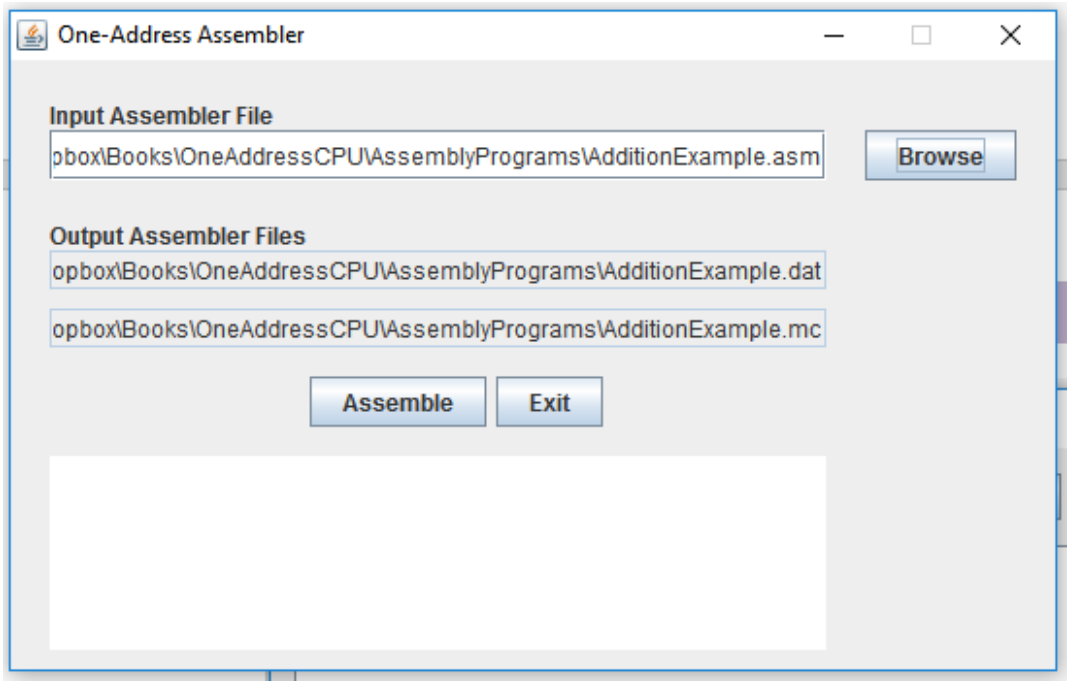


Рисунок 4-6: Запуск ассемблера - шаг 4

5. Вы должны получить следующее сообщение об успешном завершении сборки. Нажмите Exit, чтобы закрыть ассемблер.

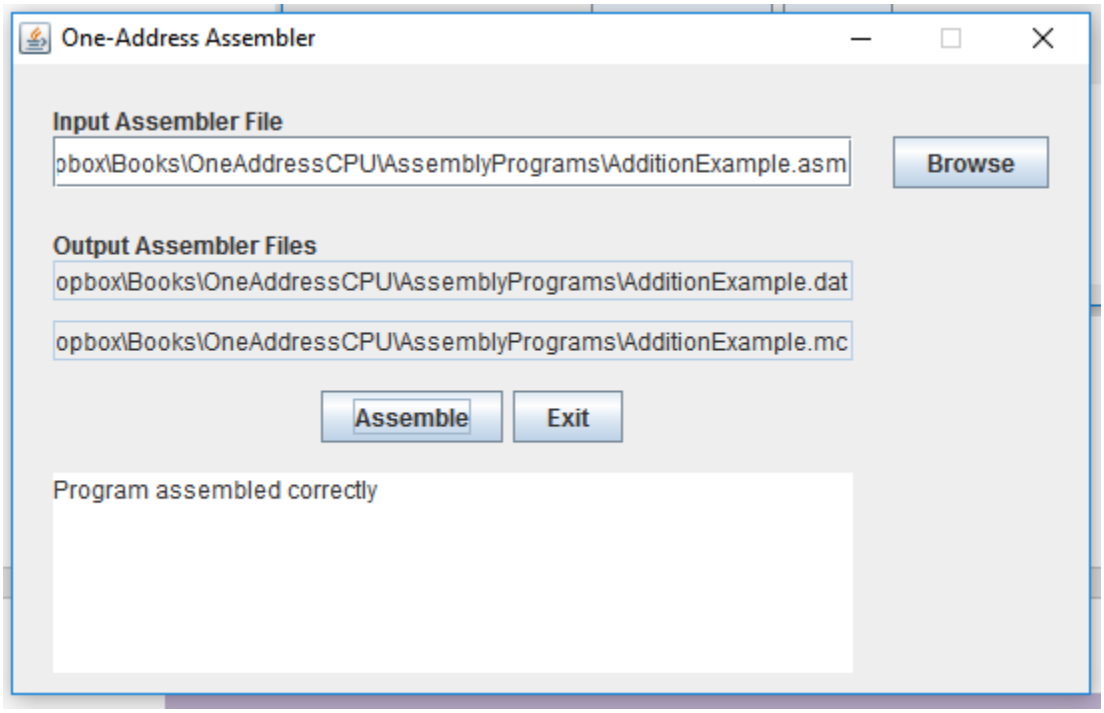


Рисунок 4-7: Запуск ассемблера - шаг 5

1. Перейдите в подкаталог под названием Logisim щелкните значок logisim-generic-2.7.1.jar. Откроется Logisim, и вы увидите следующий экран. Выберите *Файл->Открыть* выберите каталог Logisim и откройте файл OneAddressCPU.circ.

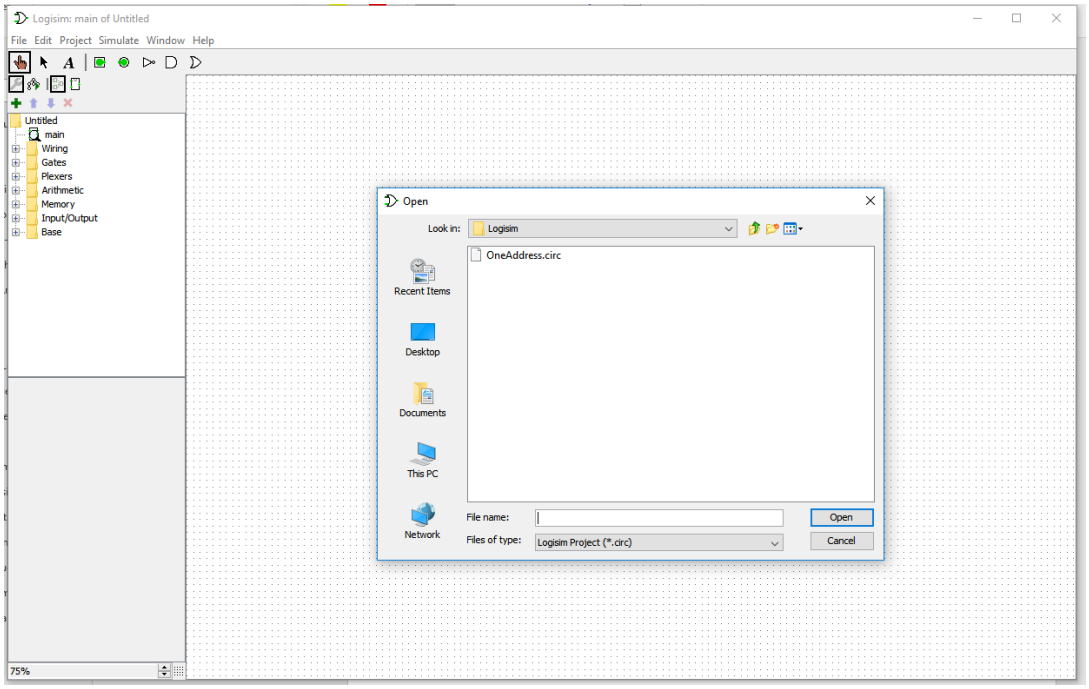


Рисунок 4-8: Запуск процессора - шаг 1

6. Теперь вы должны увидеть экран с одноадресными процессором. Установите размер Zoom (обведено на рисунке) на 75% (или 50%, если необходимо), чтобы увидеть всю диаграмму. Сейчас нас интересуют только две области процессора - текстовая память и память данных, которые обведены на диаграмме.

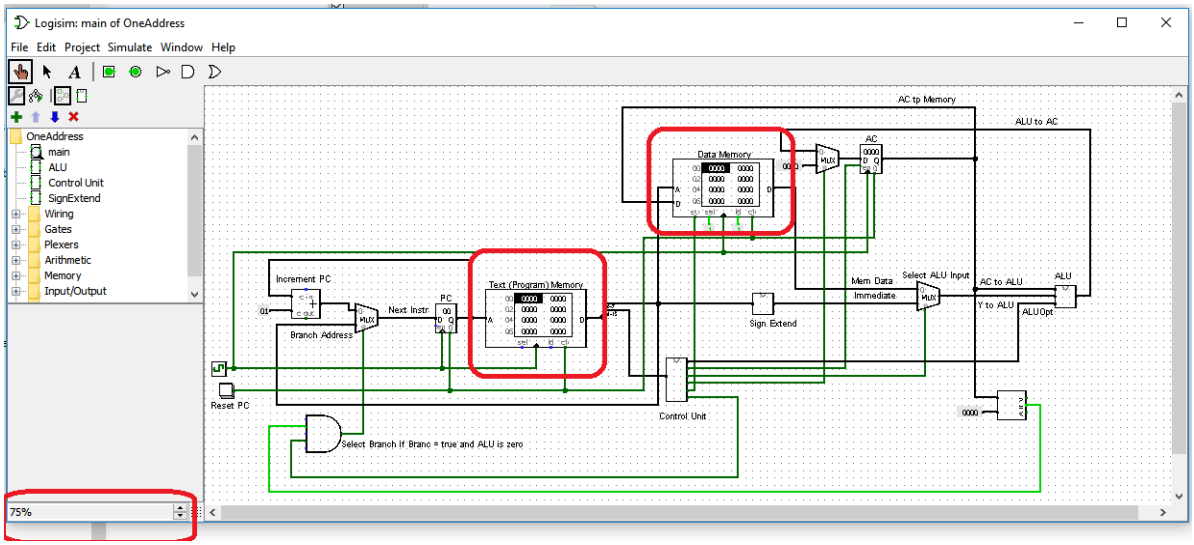


Рисунок 4-9 Запуск процессора - шаг 2

7. Щелкните правой кнопкой мыши на поле Text Memory и выберите опцию Edit Contents. На экране должен появиться следующий редактор для памяти. Обратите внимание, что входные значения для этой памяти - шестнадцатеричные, но мы будем заполнять ее из файлов, созданных ассемблером, а ассемблер записал эти файлы в шестнадцатеричном виде.

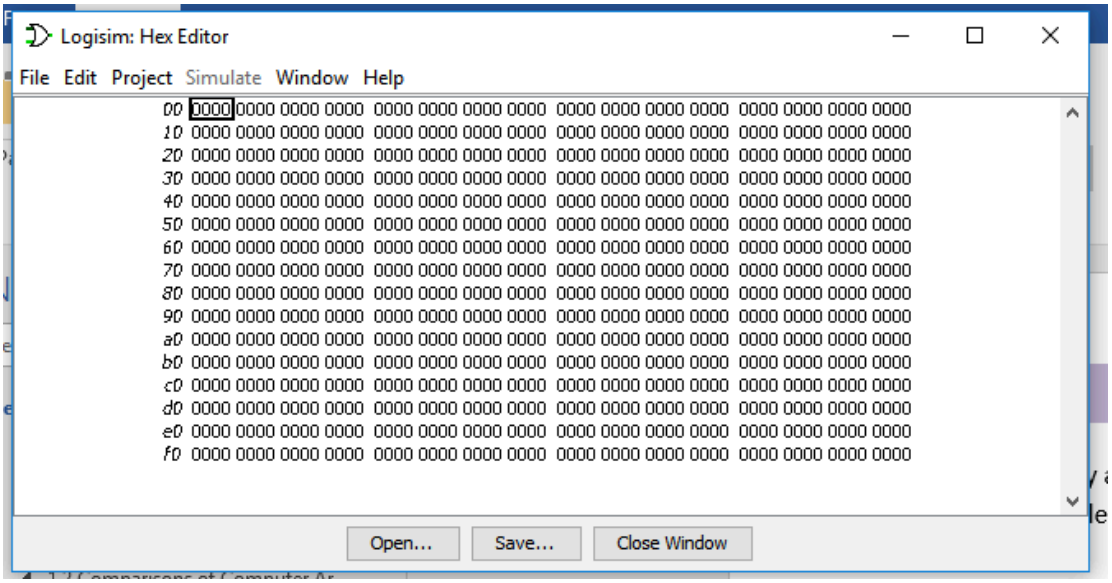


Рисунок 4-10: Запуск процессора - шаг 2

8. Выберите опцию Open, перейдите в каталог AssemblyPrograms и выберите файл AdditionExample.mc.

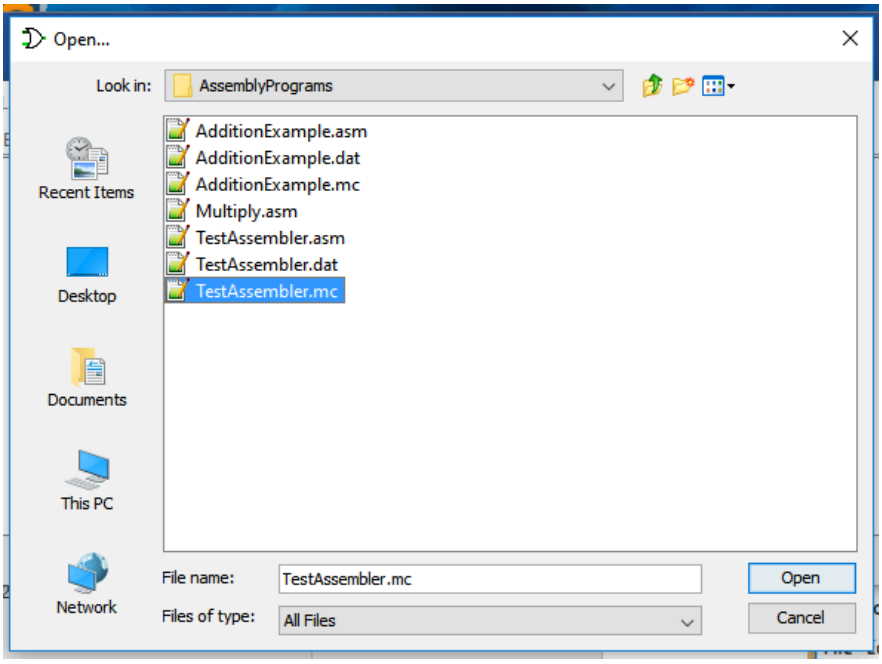


Рисунок 4-11: Запуск процессора - шаг 3

9. Теперь в этом блоке данных должен быть машинный код AdditionExample, как показано в следующем примере. Выберите опцию закрытия, и машинный код теперь доступен в процессоре.

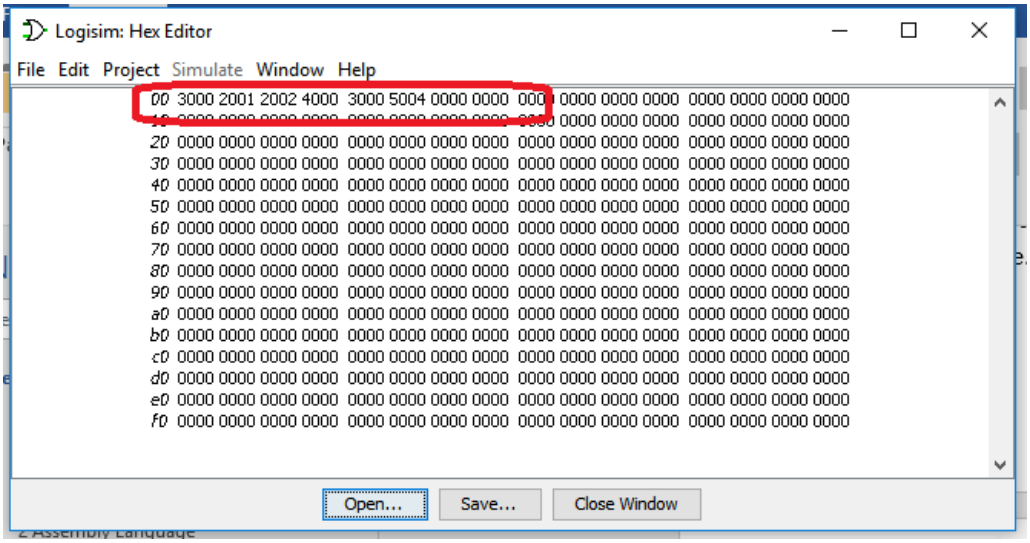


Рисунок 4-12: Запуск процессора - шаг 4

10. Повторите шаги 9-11 для памяти данных, используя файл AdditionExample.dat. Теперь память данных и память программы должны быть инициализированы.

11. Щелкните два раза на тактовой частоте (помните, что регистры и память изменяются только при срабатывании положительного фронта импульса, а не отрицательного), и будет выполнена инструкция clac. Это не приведет к изменению, потому что $ac уже равен нулю. Нажмите на часы еще два раза, и вы увидите, что программа находится на инструкции 2, а $ac изменился на 5.

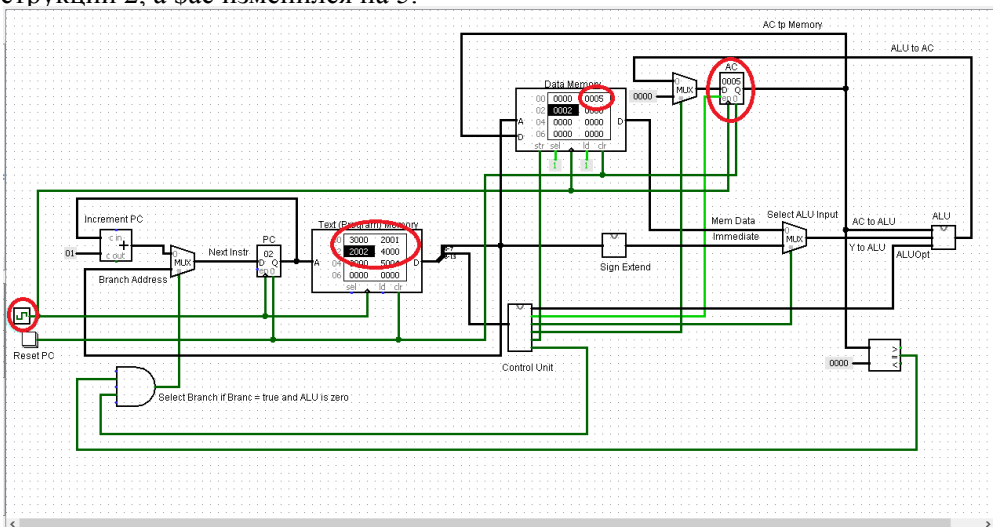


Рисунок 4-13: Запуск процессора - шаг 5

12. Щелчок на часах еще 6 раз показывает, что программа завершена, значение 7 (5+2) вычислено и сохранено по адресу памяти 0. Последние две инструкции, clac и beqz halt, просто переводят программу в бесконечный цикл, чтобы она не продолжала выполняться в оставшейся части памяти.

В Logisim есть несколько способов управления симуляцией с помощью опции Simulation. Особое значение имеет частота тиков (как часто происходит щелчок часов), включение тиков (что запускает часы с частотой тиков) и ведение журнала (что обеспечивает простой способ просмотра результатов).

5 Реализация CPU

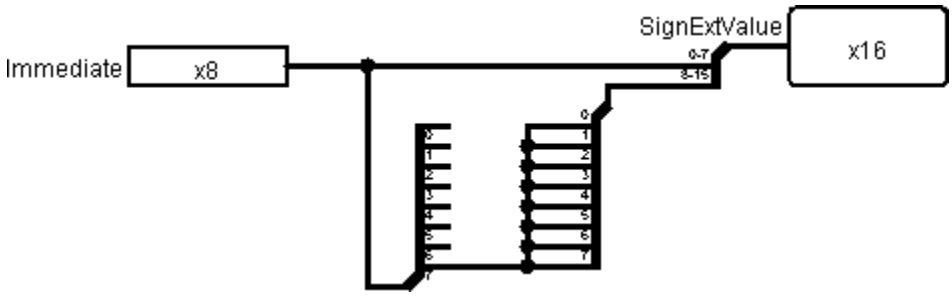
В этой главе будет рассмотрена реализация в Logisim одноадресного процессора. Эта реализация будет состоять из 3 подкомпонентов Logisim, необходимых для реализации CPU, и основного компонента, которым является CPU. Сначала будут описаны три подкомпонента: блок расширения знака, АЛУ и блок управления (CU), а затем будет подробно рассмотрен главный компонент.

5.1 Устройство для расширения знака

Непосредственные значения, которые могут быть частью инструкции, имеют размер 8 бит и могут быть использованы в качестве входа в ALU. Однако АЛУ принимает входные данные размером 16 бит. Поэтому непосредственные значения, передаваемые процессору, должны быть расширены, чтобы заполнить 16 бит. Вопрос в том, как заполнить старшие 8 бит при расширении мгновенных значений с 8 до 16 бит.

Помните, что все непосредственные значения, передаваемые процессору, являются целыми числами; старший (крайний левый) бит значения определяет знак. Если число, дополняющее 2, положительное, ведущие 0 не оказывают на него никакого влияния. Например, 01012 = 0000 01012 = 510. В отрицательном числе с дополнением 2 ведущие 1 не влияют на число. Таким образом, 10112 = 1111 10112 = -510. Таким образом, чтобы расширить целое число, крайний левый бит расширяется до новых двоичных цифр. Именно это и делает блок расширения знака, расширяя 7-й бит до позиций 8-15, чтобы преобразовать 8-битное целое число в 16-битное.

Рисунок 5-1: Блок расширения знака



5.2 АЛУ

ALU этого блока поддерживает сложение и вычитание, а также реализует флаг, позволяющий определить, не привела ли текущая операция к переполнению. Переполнение происходит, когда складываются два числа, которые слишком велики для хранения в 16-битных целых числах, реализованных в процессоре. Например, сложение 27000 + 25000 = 52000 - значение, превышающее максимальное обрабатываемое целое число, равное 32767. Аналогично -27000 + -25000 = -52000, число, которое слишком мало для минимального целого числа, которое можно обрабатывать, а именно -32768. Как АЛУ справляется с этими ситуациями, будет рассмотрено позже.

Простое АЛУ, реализующее только 16-битное сложение, легко реализовать, и оно показано на следующем рисунке. Два 16-битных значения ($ac и Y) отправляются в процессор, и сумматор использует их для сложения и получения результата.

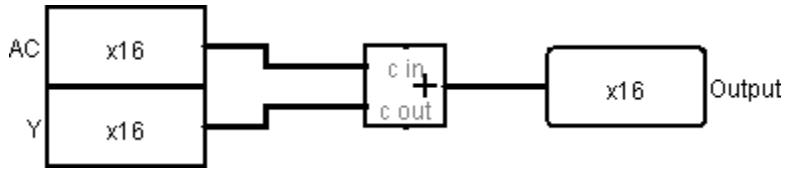


Рисунок 5-2: Простой сумматор

Чтобы реализовать вычитание, используется немного креативной математики.

1. Помните, что X ⊕ 0 = X; и X ⊕ 1 = X'.
2. Простая арифметика говорит, что X + Y = X + (-Y)
3. -Y = Y' + 1 (операция отрицания дополнения 2).
4. Вычитание можно реализовать, взяв каждый бит Y, XORировав его с 1 (получив дополнение), а затем добавив 1. Чтобы добавить 1, передайте этот бит в перенос сумматора.
5. Сложение можно реализовать, взяв каждый бит Y, XORировав его с 0 (чтобы он не изменился) и добавив 0. Чтобы добавить 0, передайте этот бит в перенос сумматора.
6. Таким образом, блок сложения/вычитания может быть реализован путем передачи флагового бита. Если бит равен 0, выполняется операция сложения, если бит равен 1, выполняется операция вычитания.

Эта процедура реализована в следующей схеме Logisim. Обратите внимание, что все, что она делает, - это XOR битов Y со значением флага 0/1, а затем добавляет флаг в сумматор через перенос в сумматор.

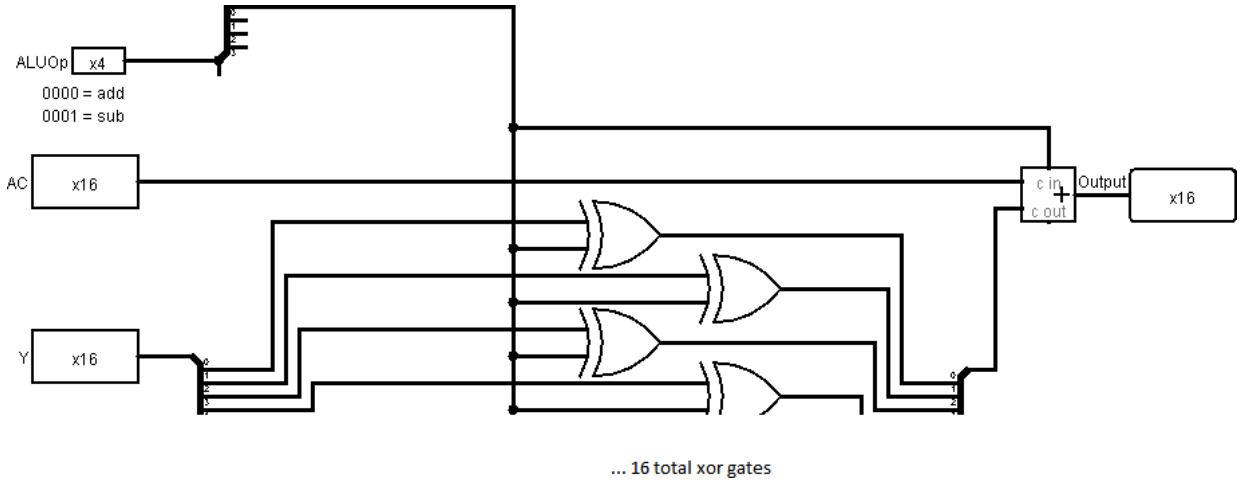


Рисунок 5-3: Сложение/вычитание

Осталось сделать последнее дополнение к АЛУ. Мы хотим проверить, есть ли переполнение или нет. Самый простой способ сделать это - проверить биты переноса и выноса последнего полного сумматора. Если они одинаковые, то переполнения нет, а если разные, то произошло переполнение. Эти два бита можно проверить с помощью XOR-гейта. Если они одинаковые (без переполнения), то XOR выдаст 0, а если разные (переполнение), то XOR выдаст 1.

Сумматор из Logisim не сигнализирует о переполнении, поэтому снова придется использовать схему творчески. Вместо 16-битного сумматора в ALU будет использоваться один 15-битный сумматор и 1-битный сумматор. Это позволяет проверить перенос и вынос последнего бита, но требует ряда переключателей, чтобы правильно определить количество линий к каждому компоненту. Однако это единственное изменение между последней версией ALU и окончательным вариантом, представленным на рисунке ниже.

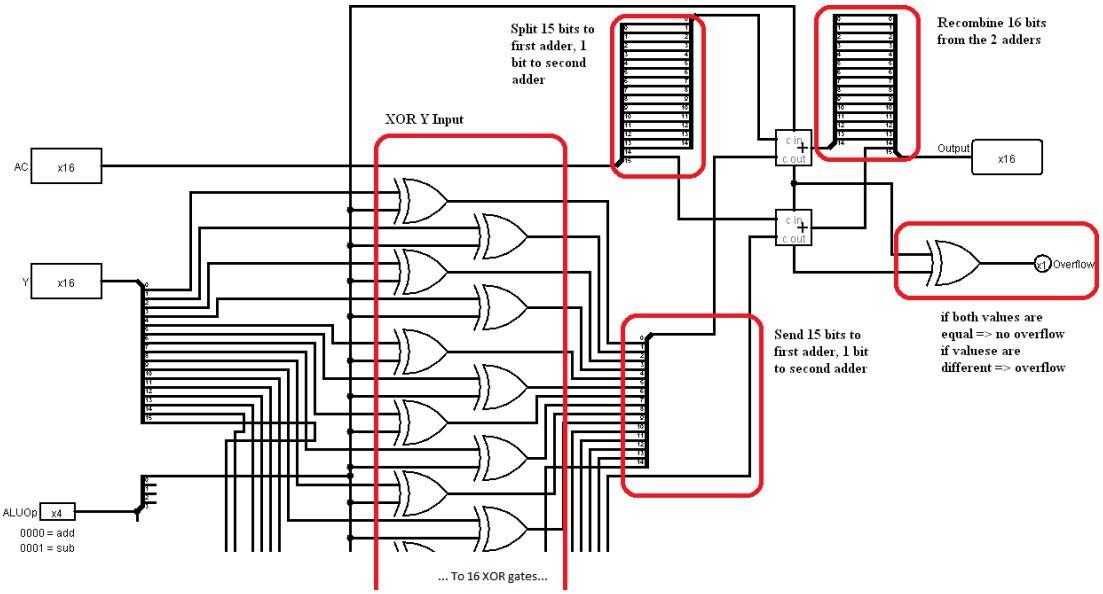


Рисунок 5-4: Сложение/вычитание с переполнением

5.3 Блок управления (CU)

CU - это "мозг" процессора. Он получает опкод из инструкции и устанавливает управляющие провода, которые контролируют, как процессор будет обрабатывать инструкцию. Поскольку установка управляющих проводов будет иметь смысл только после того, как будет понято их использование, объяснение CU будет дано после объяснения CPU.

5.4 Процессор

Центральный процессор объединяет все компоненты в единый пакет для выполнения программ. Процессор в этом тексте состоит из двух подразделов, и он был специально спроектирован достаточно просто, чтобы управляющие провода (кроме тактового генератора) для каждого подраздела процессора были полностью отделены от другого.

Первый подраздел выполняет все арифметические действия и управляет вводом/выводом данных в память/из памяти. В этом подразделе используются управляющие провода Clock, MemWr (запись в память), ClrAC (очистка $ac путем установки его в 0),

WriteAc (запись значения в $ac), ALUSrc (выбор источника для второго операнда ALU, либо памяти, либо мгновенного значения) и ALUOpt (4-битное значение, указывающее, какую операцию выполнять на ALU).

Второй подраздел управляет траекторией выполнения программы. В нем используются управляющие провода Clock и Beqz (ветвление при равенстве нулю).

Эти два подраздела будут рассмотрены отдельно.

Перед началом работы 0 в управляющем проводе означает "ничего не делать", поэтому инструкция noop - это двухбайтовое 0x0000. Если провод не используется, по умолчанию установите его в 0.

5.4.1 CPU- арифметический подраздел

Арифметический подраздел процессора охватывает регистр $ac, АЛУ и память данных. Здесь рассматриваются ассемблерные операции clac, add, addi, sub, subi и stor. Арифметический подраздел показан на следующей схеме.

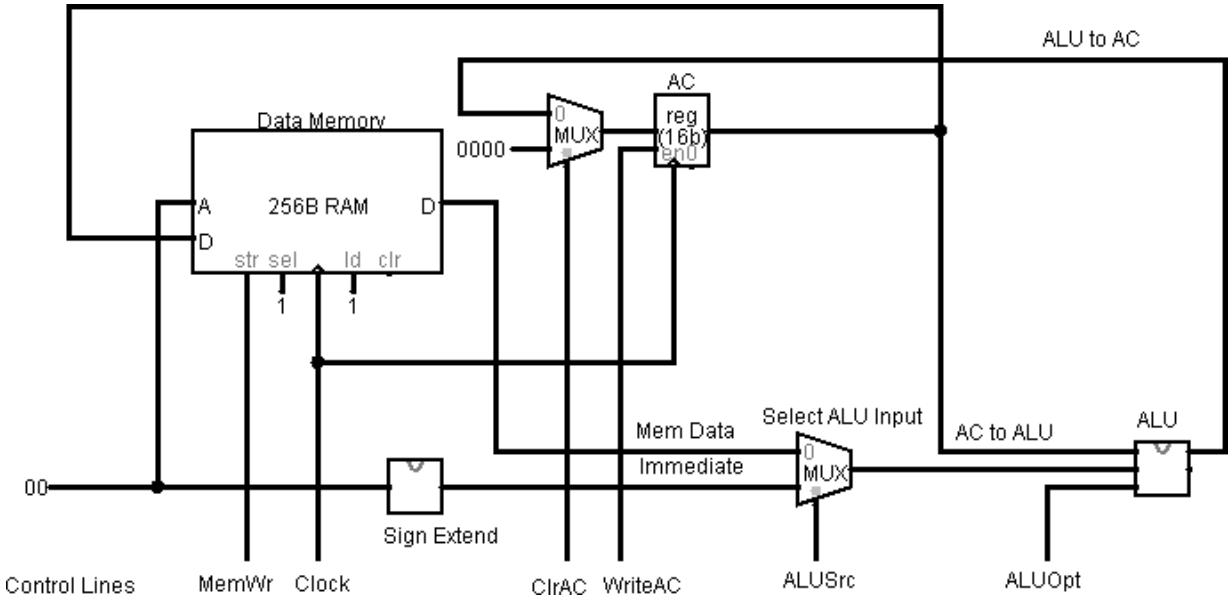


Рисунок 5-5: Центральный процессор - Подраздел арифметики

Операция clac выбирает константный вход 0x0000, используя mux перед $ac для установки значения $ac. Для этого линия ClrAC в mux равна 1 (выбор 0x0000), а линия WriteAC равна 1. Все остальные управляющие линии равны 0.

Для операций сложения и вычитания на вход ALU поступают данные из памяти, поэтому линия ALUSrc устанавливается в 0 для выбора Mem Data. Результат сохраняется обратно в $ac, поэтому линия ClrAC должна быть установлена в 0, чтобы выбрать выход из ALU, а линия WriteAC установлена в 1, чтобы записать результат ALU в AC. Для сложения ALUOpt имеет значение 0000, а для вычитания - 0001. Все остальные управляющие линии установлены в 0.

Для операций addi и subi на вход ALU подается немедленное значение, поэтому строка ALUSrc устанавливается в 1, чтобы выбрать немедленное значение данных. Все остальные строки устанавливаются так же, как для операций add и sub.

Для операции stor линия MemWr устанавливается в 1, что приводит к записи значения в порт D памяти данных в память по адресу, указанному в порту A (обратите внимание, что адрес берется из непосредственной части инструкции). Все остальные управляющие линии установлены в 0.

Некоторых читателей может обеспокоить тот факт, что в процессор передаются значения, которые не используются. Например, при выполнении операции stor значение все еще вычисляется в ALU и передается по проводу в $ac. Однако строка WriteAC равна 0, поэтому значение ALU не имеет никакого эффекта и игнорируется. То же самое касается значения Mem Data для немедленных операций типа addi. Mem Data генерируется, но игнорируется, поскольку ALUSrc выбирает немедленное значение. Многие линии устанавливаются в каждой инструкции, и большинство из них игнорируется, поэтому использование установки всех управляющих проводов в 0 в инструкции NOOP ничего не дает.

5.4.2 CPU - Путь выполнения

Второй подраздел процессора - путь выполнения, который показан на рисунке ниже.

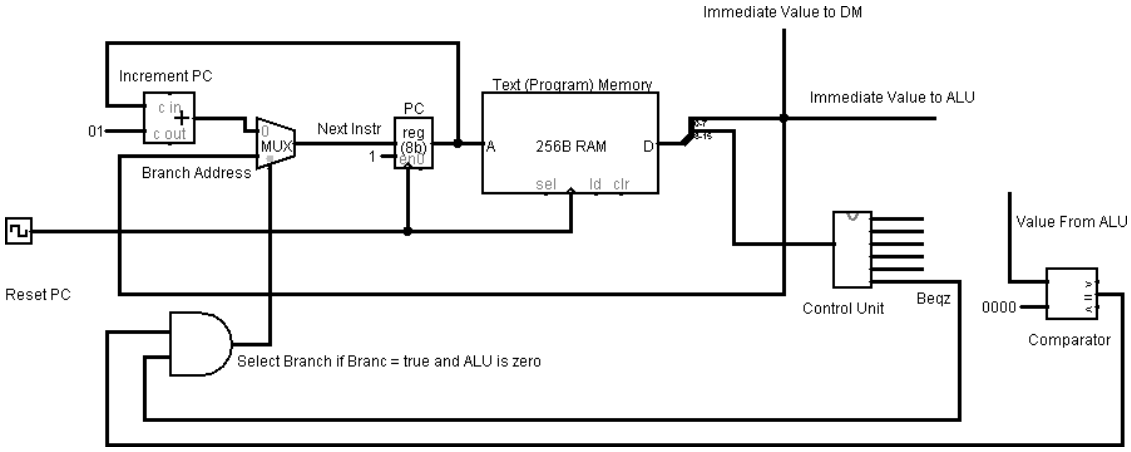


Рисунок 5-6: Процессор - Подраздел "Путь выполнения

Как показано на этом рисунке, значение регистра $pc используется для установки адреса чтения инструкции. Затем эта инструкция разделяется на управляющую половину (биты 8-15) и непосредственное значение (биты 0-7). Управляющие биты отправляются в CU для установки управляющих проводов, а непосредственное значение отправляется в память данных или в ALU для использования в арифметическом подразделе процессора.

При каждом выполнении инструкции регистр $pc изменяется, указывая на следующую выполняемую инструкцию. При последовательном выполнении программы следующая инструкция в памяти выбирается путем прибавления 1 к $pc в сумматоре с именем Increment PC, а мультиплексор устанавливается в 0 для выбора этой инструкции.

Единственный случай, когда следующая инструкция не выбирается, - это если провод Beqz имеет высокий уровень (что означает, что это инструкция beqz), И результаты компакта равны 1 (ALU имеет значение 0). Когда это происходит, mux выбирает адрес ветвления, который является непосредственным значением инструкции, и программа продолжает выполнение инструкции по новому адресу.

5.5 Реализация CU

Теперь можно указать, как установить управляющие провода от CU. Сначала ALU opt - это значение битов 8-11 ALU, поэтому они отделяются и передаются для управления ALU.

Верхние 4 бита, биты 12-15, используются для установки других управляющих проводов, и, как следует из предыдущего обсуждения, могут быть установлены в соответствии со следующей таблицей.

Операция	Code	WriteAc	ALUSrc	ClrAc	MemWr	Beqz
Немедленная операция[3]	0x1	1	1	0	0	0
Работа с памятью[4]	0x2	1	0	0	0	0
clac	0x3	1	x	1	0	0
stor	0x4	0	x	x	1	0
beqz	0x5	0	x	x	0	1

Таблица 5-1: Операционные и управляющие провода

Для реализации этой таблицы используется декодер, который разделяет отдельные операции. Затем эти операции объединяются для получения правильного поведения на выходе. CU показан на рисунке ниже.

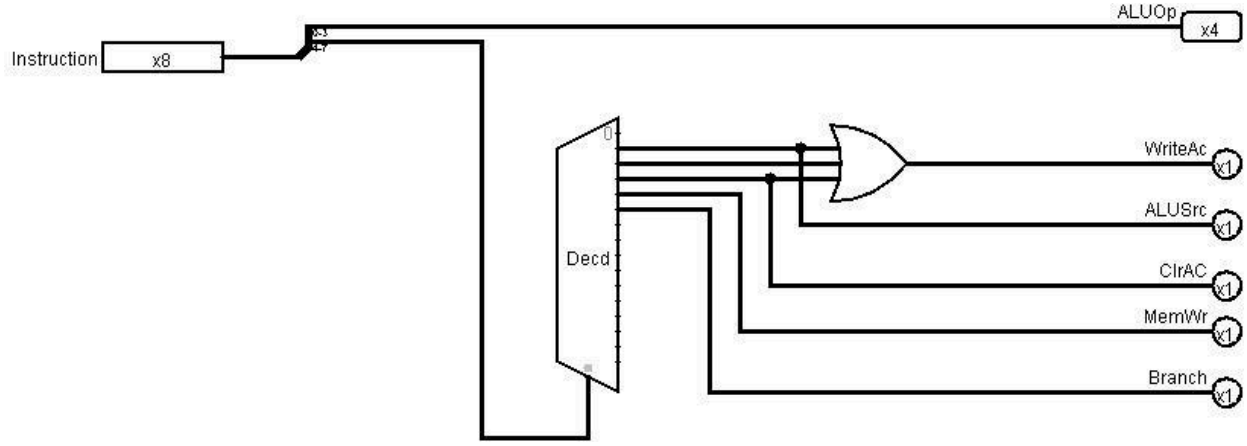


Рисунок 5-7: Блок управления

[2] Символ "x" в таблице означает условие "неважно", например, значение может быть либо 0, либо 1, так как оно не влияет на работу процессора. По традиции все значения x следует кодировать как 0.

[3] addi, subi и т. д.

[4] add, sub и т. д.