Подмножество ядра процессора RISC-V

Оглавление

Введение

В данной главе рассматривается создание реализации ядра процессора RISC-V, позволяющей выполнять тестовую программу сложения чисел от 1 до 9. Впоследствии функциональность ядра будет дополнена.

В этой главе вы узнаете:

- роль основных компонентов базовой микроархитектуры ЦП;
- способы описания цифровой логики с помощью TL-Verilog;
- особенности процесса отладки проектов в Makerchip, включая:
    1. интерпретацию сообщений в журнале событий;
    2. использование визуальной отладки для понимания общего поведения цифровой схемы;
    3. использование средств просмотра временных диаграмм для понимания детального поведения цифровой схемы;
    4. отслеживание некорректного поведения схемы от предупреждения об ошибке до обнаружения ее причины.
- создание экземпляров существующих компонентов Verilog и TL-Verilog.

Настройка рабочего окружения для выполнения практической работы по разработке процессорного ядра

Видеоматериал: Визуализация моделирования процессора RISC-V

Видеоматериал расположен по ссылке [1].

Демонстрация: Начальный код

Видеоматериал расположен по ссылке [2].

Демонстрация: Эталонные решения

Видеоматериал расположен по ссылке [3].

Демонстрация

Теперь, когда вы готовы к созданию чего-то более существенного, чем простой учебный пример, разберем несколько вариантов того, как можно разрабатывать, сохранять и демонстрировать свой проект.

Проекты можно сохранять на GitHub (или на другой платформе хостинга Git). Если вы ищете или собираетесь искать работу, ваш профиль на GitHub часто может сказать потенциальному работодателю больше, чем ваше резюме. GitHub также является отличным местом для хранения кода по мере его разработки. Вы можете создать новый репозиторий для своей работы или сделать форк репозитория курса [4]. Можно редактировать файлы непосредственно через веб-интерфейс GitHub и вставлять свой код из Makerchip в браузере, или можно клонировать свой репозиторий на локальный диск и вставить код в текстовый редактор.

Есть удобная опция для работы с локальными файлами на рабочем столе, независимо от того, находятся ли они в git-репозитории или нет. Можно запускать Makerchip с рабочего стола для работы с локальным исходным файлом TL-Verilog. Makerchip запускается в браузере, но автоматически сохраняется на рабочем столе.

Если вы хотите попробовать данный способ, сначала скачайте на локальный диск исходный код, клонировав репозиторий курса с GitHub [4]:

`git clone https://github.com/stevehoover/LF-Building-a-RISC-V-CPU-Core.git` (и введите ваши учетные данные).

После этого установите приложение Makerchip [5]:

```
pip3 install makerchip-app
```

Использование кода точки старта проекта

Откройте шаблон начального кода проекта:

1. Если вы используете приложение Makerchip для редактирования кода на локальном компьютере, сначала скопируйте код из файла `LF-Building-a-RISC-V-CPU-Core/risc-v_shell.tlv` туда, где вы хотите его отредактировать, затем выполните: `makerchip <путь>/risc-v_shell.tlv`.
2. В противном случае просто нажмите на ссылку [6], чтобы открыть исходный код в браузере, и, как только код загрузится, выберите пункт меню «Сохранить как новый проект».
3. Симулятор должен запуститься. Во вкладке LOG должно появиться сообщение «Simulation FAILED!!!» (и так будет до тех пор, пока вы успешно не выполните все шаги, описанные в данной главе). На вкладке VIZ должен показать тестовую программу и сигналы, которые еще не были реализованы. (Проведите мышкой вниз или используйте кнопку «-», чтобы показать их полностью).

Независимо от того, используете вы приложение Makerchip или нет, после лабораторной работы убедитесь, что ваши файлы были правильно автоматически сохранена в облаке или локальной системе, проверив строку состояния в Makerchip.

Обычно, если вы допустили ошибку при разработке и вам нужно вернуться к стабильной версии вашего кода, достаточно нажать Ctrl-Z, но иногда имеет смысл сохранять промежуточную версию проекта (или git-коммит).

Микроархитектура процессорного ядра

Процессоры бывают разных видов: от небольших микроконтроллеров, оптимизированных для того, чтобы они занимали малую площадь на кристалле и имели низкое энергопотребление, до настольных и серверных процессоров, оптимизированных под высокую производительность. В течение нескольких часов вы разработайте ядро процессора, которое может быть использовано в качестве микроконтроллера. Создание процессорного чипа для настольного компьютера или сервера потребовало бы нескольких лет и команды из сотен опытных инженеров.

Разрабатываемый процессор будет полностью выполнять одну инструкцию за один такт. Выполнение команды в течение одного такта возможно только в том случае, если тактирование происходит относительно медленно(что и предполагается в этой реализации ядра).

Начнем с реализации процессора, достаточного для выполнения простой тестовой программы. По мере добавления каждого нового функционального блока, вы будете видеть на панели VIZ визуализацию работы команд. Все больше и больше частей тестовой программы будет выполняться правильно, пока она не начнет успешно суммировать числа от одного до девяти. Затем мы вернемся к реализации поддержки основной части набора команд RV32I.

Рассмотрим компоненты разрабатываемого процессора, проследив за прохождением команды через компоненты процессора (рисунке 1). Примерно в таком же порядке мы будем реализовывать логику работы процессора.

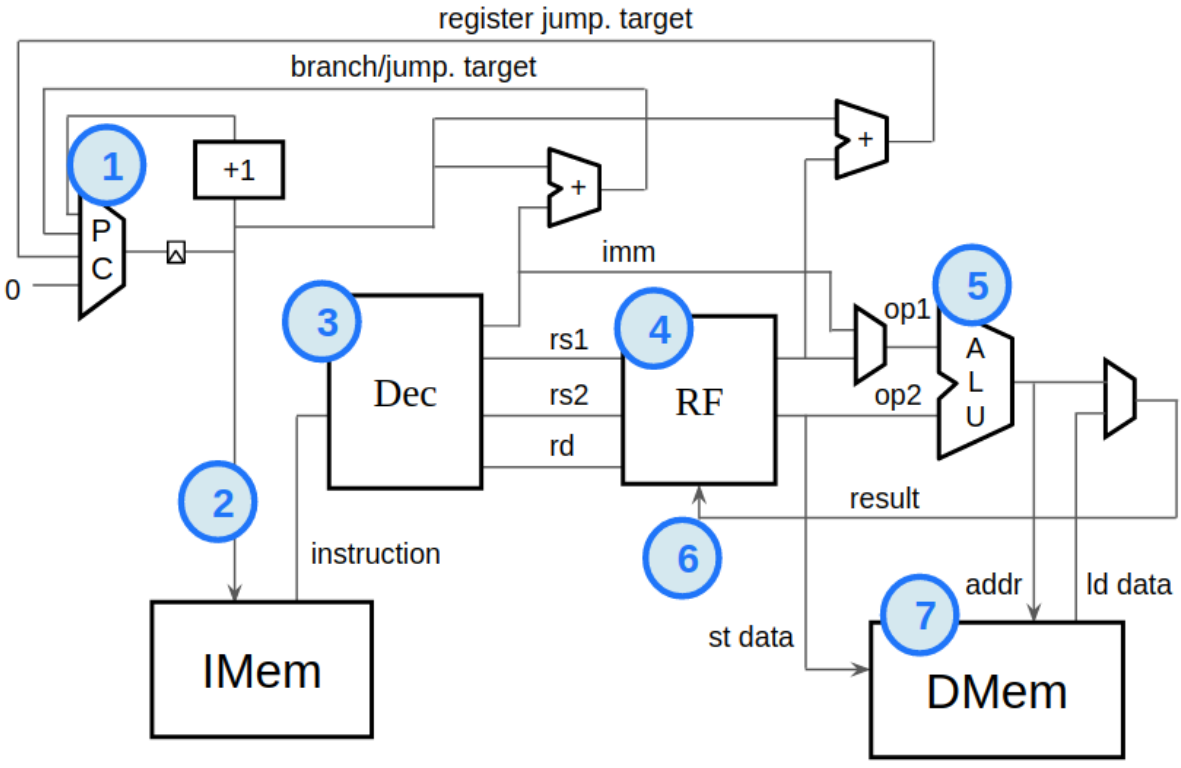


Рисунок 1 – Компоненты блока процессорного ядра RISC-V

1. **Счетчик команд (PC)**

Этот блок отвечает за PC (**Program Counter, счетчик команд**). Он хранит адрес команды, которую процессорное ядро выполнит следующей. Большинство команд выполняются последовательно, поэтому основное поведение PC заключается в инкрементировании (увеличении) адреса выполняемой команды на каждом такте. Но команды ветвления (branch) и перехода (jump), не являются последовательными. Они определяют целевую инструкцию, которая должна будет выполниться следующей, поэтому значение PC обновляется соответствующим образом.

2. **IMem (Instruction Memory)**

Память команд хранит выполняемые процессором инструкции. Процесс чтения IMem или «выборки» (fetch) заключается в том, что процессор получает инструкцию, на которую указывает программный счетчик PC.

3. **Декодер (Dec)**

Для выполнения процессором команды ее нужно интерпретировать или декодировать. Необходимо разбить ее по полям, которые определят, какой регистр требуется прочесть, какую операцию выполнить, и т.д.

4. **Чтение регистрового файла (RF)**

Регистровый файл представляет собой небольшую область памяти с локальными данными и активно используется в зависимости от программы. В инструкции есть специальные поля, которые определяют какие регистры нужно прочесть из регистрового файла.

5. **Арифметико-логическое устройство (АЛУ, ALU)**

После получения значений из регистров происходит их обработка. АЛУ выполняет различные действия с данными из регистров в зависимости от типа операции, указанной в инструкции (например, складывает, умножает, вычитает, выполняет сдвиг и т.д.).

6. **Запись в регистровый файл**

Полученные из АЛУ данные с результатами операций можно записать обратно в определяемый инструкцией регистр вывода.

7. **DMem (Data Memory)**

Используемая в данном курсе тестовая программа выполняется исключительно с помощью регистрового файла и не требует наличия памяти (Data Memory). Но сложно считать ядро завершенным без блока памяти. Данные в DMem записывается с помощью команд записи и считываются при помощи команд загрузки.

В этом курсе акцент сделан только на ядре процессора. Дополнительные схемы, которые были бы необходимы для взаимодействия с периферией, такой как контроллеры ввода-вывода (I/O), логика прерываний, системные таймеры и др., не рассматривается.

В данной реализации используется упрощенная память. Обычно процессор общего назначения имеет больший объем памяти (содержащий как инструкции, так и данные,) для доступа к которому требуется много тактовых циклов. Чтобы снизить задержку доступа к памяти, используется кэш-память, в которой хранятся недавно полученные и часто используемые данные. Все эти сложные моменты реализации процессоров в целях обучения игнорируются. Для памяти команд и памяти данных используются отдельные небольшого объема запоминающие устройства. В промышленных процессорных ядрах обычно реализуется отдельная кэш-память для команд и данных со временем доступа, равным одному циклу, которые мало чем отличаются от IMem и DMem, используемых в этой главе.

Программный счетчик PC

Сначала реализуем только последовательную выборку, поэтому программный счетчик PC представляет собой простой счетчик (рисунок **.2**).

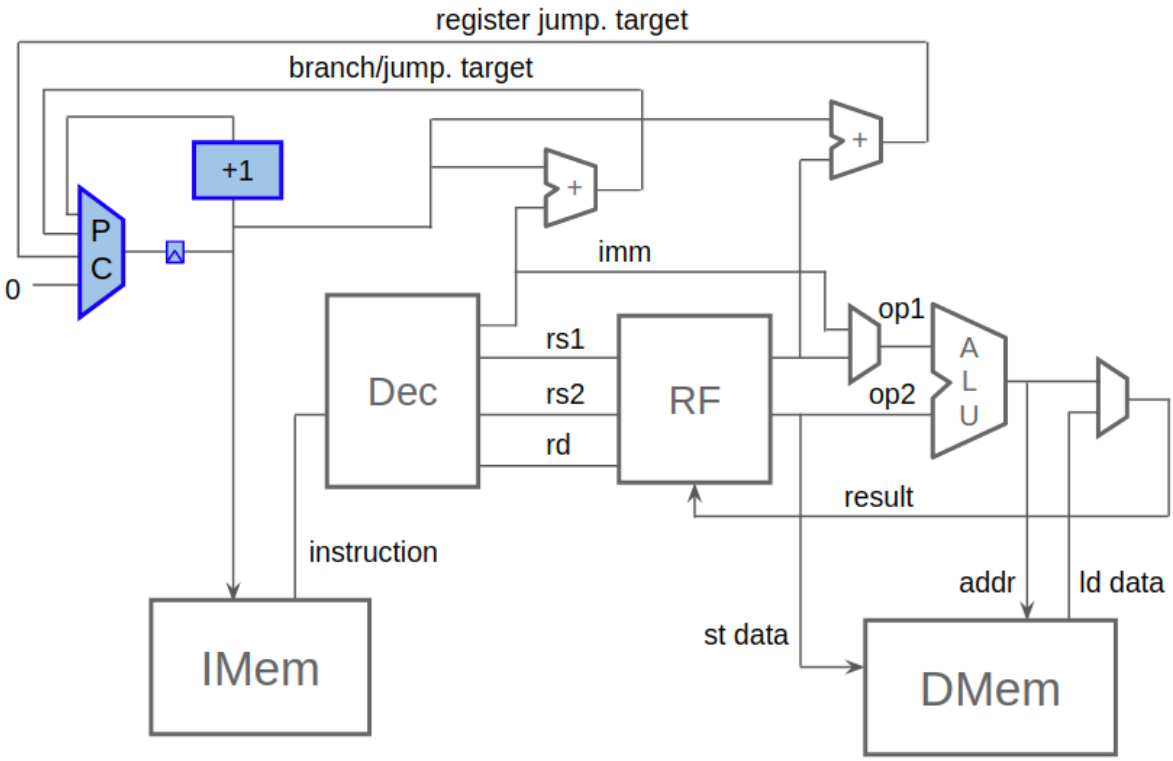


Рисунок 2 – PC на схеме упрощенного процессора RISC-V

Обратите внимание на то, что:

- PC хранит адрес в байтах, то есть он ссылается на первый байт инструкции в IMem. Инструкции имеют длину 4 байта, поэтому, хотя инкремент PC изображается как «+1» (инструкция), фактическое приращение адреса равно четырем.. В нормальном режиме работы два младших бита PC всегда должны быть нулевыми;
- выборка команд должна начинаться с нулевого адреса, поэтому первое значение `$pc` при приходе сигнала `$reset` должно быть нулевым, что реализовано в логической схеме на рисунке 3;
- в отличие от схемы счетчика, которая была разработана ранее, для удобства чтения используются уникальные имена для `$pc` и `$next_pc`, присваивая `$pc` предыдущему `$next_pc`.

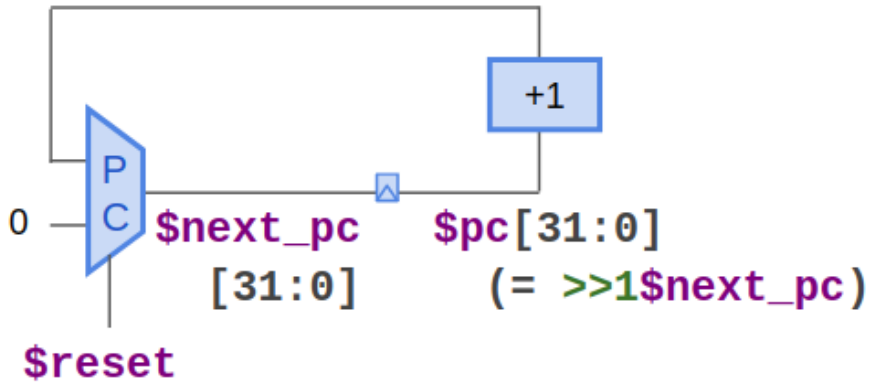


Рисунок 3 – Схема работы PC

Выполните следующее:

Реализуйте схему выше (заменив комментарий «YOUR CODE HERE»). Во вкладках VIZ и WAVEFORM убедитесь, что значение в PC начинает счет с нуля и увеличивается корректно.

Если возникли проблемы, эталонные решения [7] для всех лабораторных работ RISC-V находятся в репозитарии курса на GitHub [4].

Память команд IMem

Память команд IMem (Instruction Memory, рисунок 4) реализуется путем реализации макроса Verilog (рис. 5). Этот макрос принимает на вход адрес команды, а на выходе выдает 32-битные данные для чтения. Создать экземпляр макроса можно следующим образом:

```
`READONLY_MEM($addr, $$read_data[31:0])
```

Перед написанием макроса в Verilog ставится гравис (backquote, не путать с одинарной кавычкой). В выражениях, которые синтаксически не отличают назначенные сигналы от используемых, необходимо идентифицировать назначенные сигналы с помощью префикса «`$$`». Назначенным сигналам объявляется битовый диапазон. Таким образом, выше используется `$$read_data[31:0]`.

Этот макрос значительно упрощен по сравнению с макросом для массива:

1. Не существует способа записи в массив. Программа, указанная в шаблоне, заносится в этот массив перед моделированием.
2. Как правило, массив имеет вход разрешения на чтение. Это разрешение на чтение указывает в каждом цикле, следует ли выполнять чтение. В разработанном массиве чтение будет происходить всегда. Разрешение на чтение нужно для экономии энергопотребления, но в данном примере это проигнорировано для упрощения.
3. Обычно модуль памяти, подобный IMem, реализуется с помощью физического примитива, называемой статической памятью с произвольным доступом, или SRAM (Static Random Access Memory). Адрес предоставляется за один такт, а данные считываются в следующем такте. Но разрабатываемый процессор однотактный и команда выполняется в один такт. Массив возвращает выходные данные в том же тактовом цикле, на котором поступил входной адрес. Таким образом, реализация с использованием триггеров будет гораздо менее оптимальной, чем с помощью SRAM.

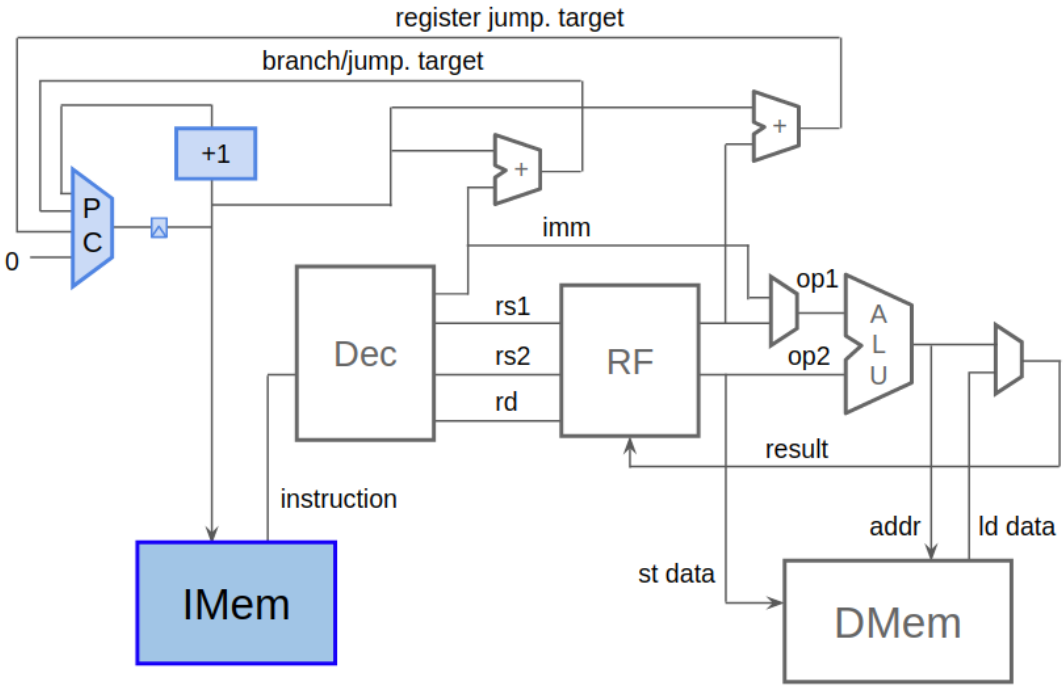


Рисунок 4 – IMem на схеме упрощенного процессора RISC-V

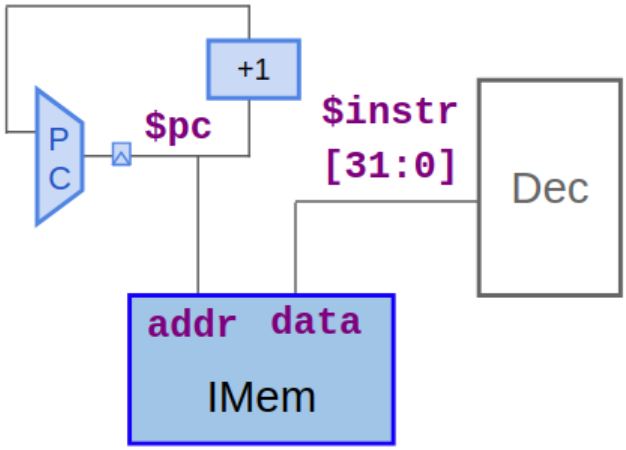


Рисунок 5 – Подключение памяти команд IMem

Реализуйте выборку команд из IMem:

1. Установите макрос `READONLY_MEM после логики PC, указав `$pc` в качестве адреса и `$$instr[31:0]` в качестве вывода. Обязательно выровняйте его с другими участками кода, используя отступ в три пробела.
2. Убедитесь, что: LOG указывает на вывод `$instr`, во вкладке DIAGRAM отображается нужная диаграмма, а во вкладку VIZ выводятся инструкции, считываемые из IMem. При некорректной работе модуля выполните отладку кода с помощью окна WAVEFORM и убедитесь, что сигналу `$instr` больше не соответствует значение «To Be Implemented» («Требуется реализовать») во вкладке VIZ.

Декодер команд

Типы команд

В этом разделе рассмотрен модуль декодера (decode) и все, что нужно знать про инструкции. Сам модуль в схеме процессора показан на рисунке 6.

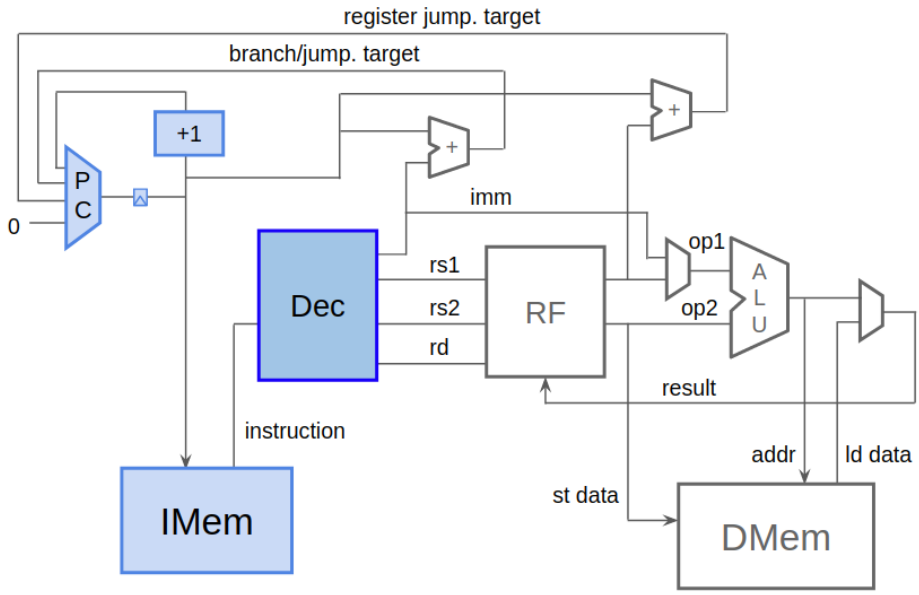


Рисунок 6 – Декодер в схеме упрощенного процессора RISC-V

В архитектуре RISC-V поддерживаются различные типы команд, которые определяют расположение полей команд, согласно таблице, отображенной на рисунке 7, из спецификаций RISC-V:

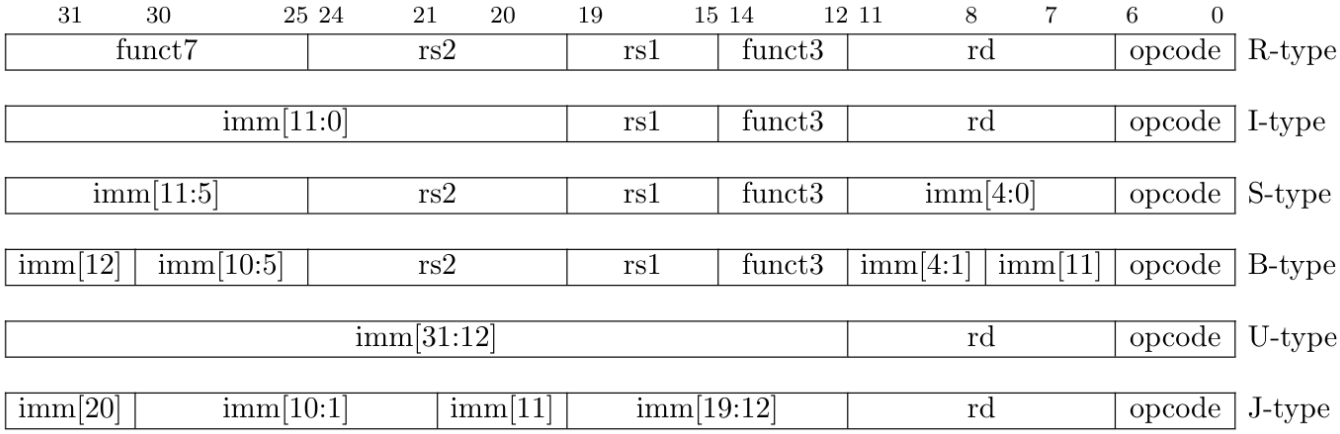


Рисунок 7 – Базовые форматы команд RISC-V и их структура

Прежде чем интерпретировать команду, нужно знать ее тип. Он определяется по ее коду операции opcode в `$instr[6:0]`. На самом деле, поле `$instr[1:0]` должно соответствовать `2'b11` для корректных (валидных) команд RV32I. Будем считать, что все инструкции корректны,

поэтому можно просто игнорировать эти два бита. ISA (Instruction Standard Architecture) определяет тип инструкции следующим образом:

instr[4:2] instr[6:5]	000	001	010	011	100	101	110	111
00	I	I	-	-	I	U	I	-
01	S	S	-	R	R	U	R	-
10	R4	R4	R4	R4	R	-	-	-
11	B	I	-	J	I (unused)	-	-	-

Рисунок 8 – Типы команд по opcode[6:2] (instr[6:2])

Для каждого типа команды нужно назначить логический сигнал, который указывает, относится ли инструкция к этому типу. Например, можно декодировать U-тип как:

```
$is_u_instr = $instr[6:2] == 5'b00101 ||
              $instr[6:2] == 5'b01101;
```

Проверьте следующее:

Изучите, как двоичные значения в этом выражении соответствуют двум U-образным ячейкам в таблице. В SystemVerilog есть оператор, который делает это сравнение немного проще:

```
$is_u_instr = $instr[6:2] ==? 5'b0x101;
```

Оператор `==?` позволяет исключить некоторые биты из сравнения, указав их как «`x`» (читается как «не важно»).

Выполните следующие шаги:

1. Добавьте оператор присваивания в свой код и напишите остальные 5 операторов для типов команд I, R, S, B и J. (Серые ячейки можно игнорировать, так как они не используются в RV32I).
2. Скомпилируйте и проведите моделирование кода. Изучите вывод LOG. В окне VIZ «Instr.Decode» должен указываться тип текущей инструкции. Отладьте при необходимости код, используя окно временных диаграмм WAVEFORM.

Поля команд

Основываясь на типе команды, можно правильно декодировать поля инструкции (рисунок 7). Большинство полей всегда состоят из одних и тех же битов, независимо от типа инструкции, но имеют значение только для некоторых из них. Исключением является поле `imm (immediate)` – значение (константа), встроенное в саму инструкцию. Оно состоит из разных битов в зависимости от типа инструкции.

Начнем с более простых полей, не требующих встроенного в инструкцию значения: `$funct3`, `$rs1`, `$rs2`, `$rd`, `$opcode`. В рассматриваемых примерах `$funct7` использоваться не будет, поэтому это поле можно пропустить:

1. Извлеките поля, например, таким образом: $rs2[4:0] = $instr[24:20].
2. Скомпилируйте и выполните моделирование кода, проверьте LOG (с предупреждениями о новых неподключенных сигналах) и выполните отладку. По мере добавления новых сигналов они должны удаляться из списка «To Be Implemented» («Требуют реализации») в VIZ.
3. Определите, когда эти поля валидны (за исключением `$opcode`, который действителен всегда). Например:

   ```
   $rs2_valid = $is_r_instr || $is_s_instr || $is_b_instr;
   ```

   Определите также сигнал `$imm_valid`, назначая для всех типов, кроме R, даже если вы еще не определили `$imm`.
4. Скомпилируйте и выполните моделирование кода, проверьте LOG (теперь довольно длинный) и отладьте. Список сигналов «To Be Implemented» должен стать меньше.
5. Чтобы множество предупреждений в LOG не мешали, их можно скрыть, добавив следующий код:

   ```
   `BOGUS_USE($rd $rd_valid $rs1 $rs1_valid ...)
   ```

   В проекте не создается никакой логики, но для компилятора выглядит как работа с сигналами, поэтому предупреждения можно проигнорировать. Обратите внимание, что между сигналами нет запятых. Если растянуть это выражение на две строки, то вторая строка должна быть отделена пробелами относительно первой. Эта строка может быть удалена после использования сигналов, хотя некоторые из этих сигналов будут использоваться только для VIZ и останутся незадействованными в проекте.
6. Скомпилируйте и выполните моделирование кода и убедитесь, что LOG не содержит ошибок. Теперь вы должны увидеть индексы регистров в VIZ (в синей части декодирования команд). Убедитесь, что они выглядят корректно, и отладьте код, если необходимо.

Работа с непосредственным значением immediate немного сложнее. Оно состоит из битов из разных полей в зависимости от типа инструкции (рисунок 9).

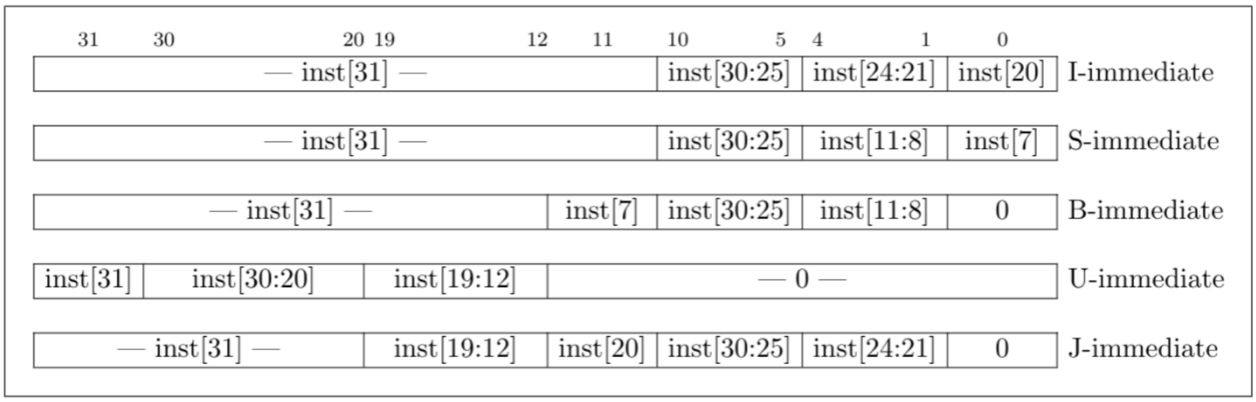


Рисунок 9 – Значения поля immediate в соответствии с типами команд

Непосредственное значение для команд типа I, например, формируется из 21 копии 31-го бита инструкции, за которым следует `inst[30:20]` (который выше разбит на три поля для согласованности с другими форматами).

Поле immediate может быть сформировано на основе этой таблицы с помощью логического выражения, подобного следующему. В нем используется комбинация извлечения битов (`$instr[30:20]`), репликации битов (`{21{...}}`) и конкатенации битов (`{..., ...}`):

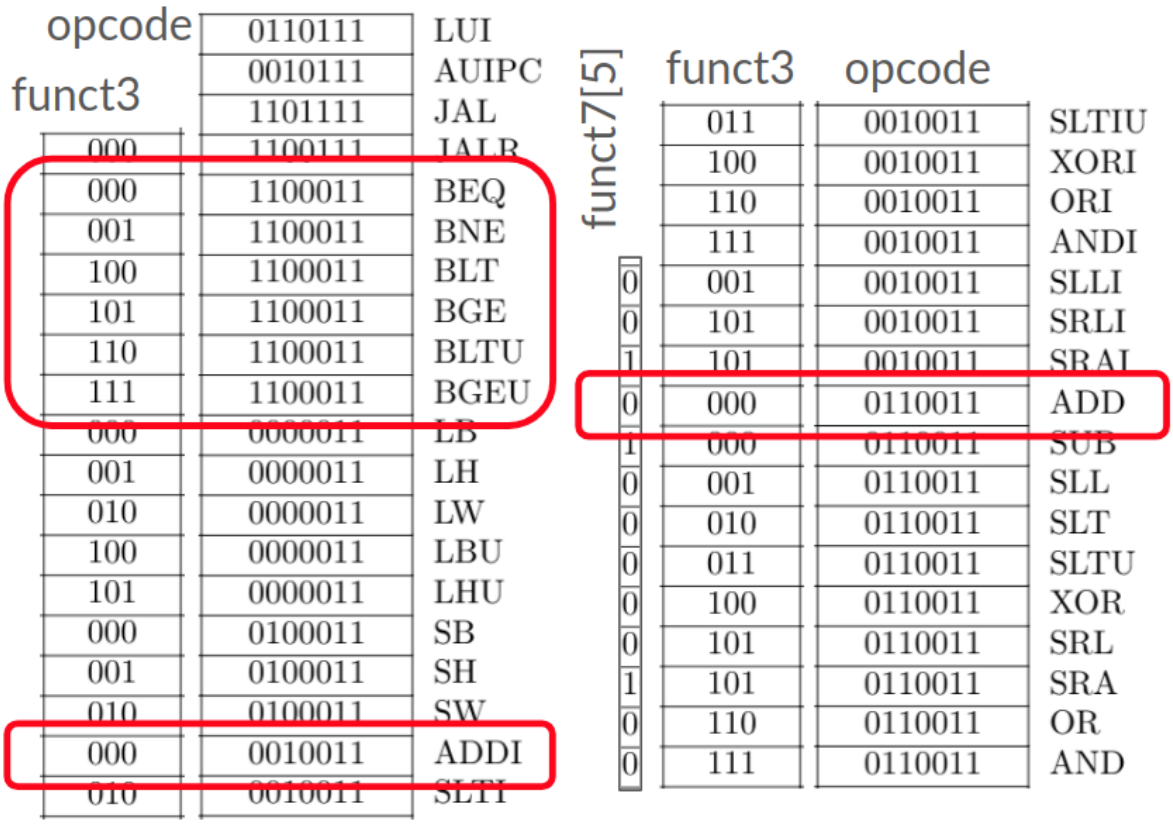
```
$imm[31:0] = $is_i_instr ? { { {21{$instr[31]}}, $instr[30:20] } :
                $is_s_instr ? {...} :
                ...
                32'b0; // По умолчанию
```

Выполните следующие шаги:

1. Заполните приведенное выше логическое выражение для `$imm`.
2. Проверьте в WAVEFORM и VIZ, что значение `$imm` соответствует инструкциям в тестовой программе. От вас потребуется понимание двоичной, десятичной и шестнадцатеричной систем счисления. Например, инструкция ADDI, x12, x10, 1010 содержит непосредственное значение в двоичном формате, VIZ представляет значение в десятичном формате как i[10], а WAVEFORM покажет шестнадцатеричное «a».

Команда

Теперь нужно определить конкретную команду. Она определяется из кода операции (opcode), полей `instr[30]` и `funct3` следующим образом. Обратите внимание, что `instr[30]` – это `$funct7[5]` для R-типа или `$imm[10]` для I-типа. В таблице ниже он обозначен как «`funct7[5]`»:



opcode	funct3		
0110111			LUI
0010111			AUIPC
1101111			JAL
1100111	000		JALR
1100011	000		BEQ
1100011	001		BNE
1100011	100		BLT
1100011	101		BGE
1100011	110		BLTU
1100011	111		BGEU
0000011	000		LB
0000011	001		LH
0000011	010		LW
0000011	100		LBU
0000011	101		LHU
0100011	000		SB
0100011	001		SH
0100011	010		SW
0010011	000		ADDI
0010011	010		SLTI
0010011	011		SLTIU
0010011	100		XORI
0010011	110		ORI
0010011	111		ANDI
0010011	001	0	SLLI
0010011	101	0	SRLI
0010011	101	1	SRAI
0110011	000	0	ADD
0110011	000	1	SUB
0110011	001	0	SLL
0110011	010	0	SLT
0110011	011	0	SLTU
0110011	100	0	XOR
0110011	101	0	SRL
0110011	101	1	SRA
0110011	110	0	OR
0110011	111	0	AND

Рисунок 10 – Таблица декодирования команд

Выполните следующие шаги:

1. Для удобства выполните конкатенцию соответствующих полей в один битовый векторный сигнал. Например: `$dec_bits[10:0] = {$instr[30],$funct3,$opcode};`
2. Для каждой из команд, обведенных красным (к остальным мы вернемся позже), определите, идентифицирует ли `$dec_bits` эту команду. Например:
   `$is_beq = $dec_bits ==? 11'bx_000_1100011;`
   Обратите внимание, что подчеркивание здесь необязательно, это нужно чтобы визуально разграничить поля. Также обратите внимание, что используется «x» для обозначения бита `instr[30]`, который не используется инструкцией BEQ (или любой другой инструкцией в левой колонке).

3. Скомпилируйте и проведите моделирование проекта. Будет сгенерировано много предупреждений для неиспользуемых сигналов. Можно снова использовать `BOGUS_USE, чтобы сохранить LOG чистым.
4. Убедитесь, что VIZ теперь отображает правильные мнемоники команд в синей секции «Instr. Decode». Выполните отладку кода при необходимости.

Чтение регистрового файла

Как и IMem, регистровый файл (рисунок 11) представляет собой типичный модуль массива, поэтому для него можно использовать библиотечный компонент. На этот раз, вместо того чтобы использовать модуль или макрос Verilog, как это было сделано для IMem, воспользуйтесь определением массива TL-Verilog, расширенным макропрепроцессором M4.

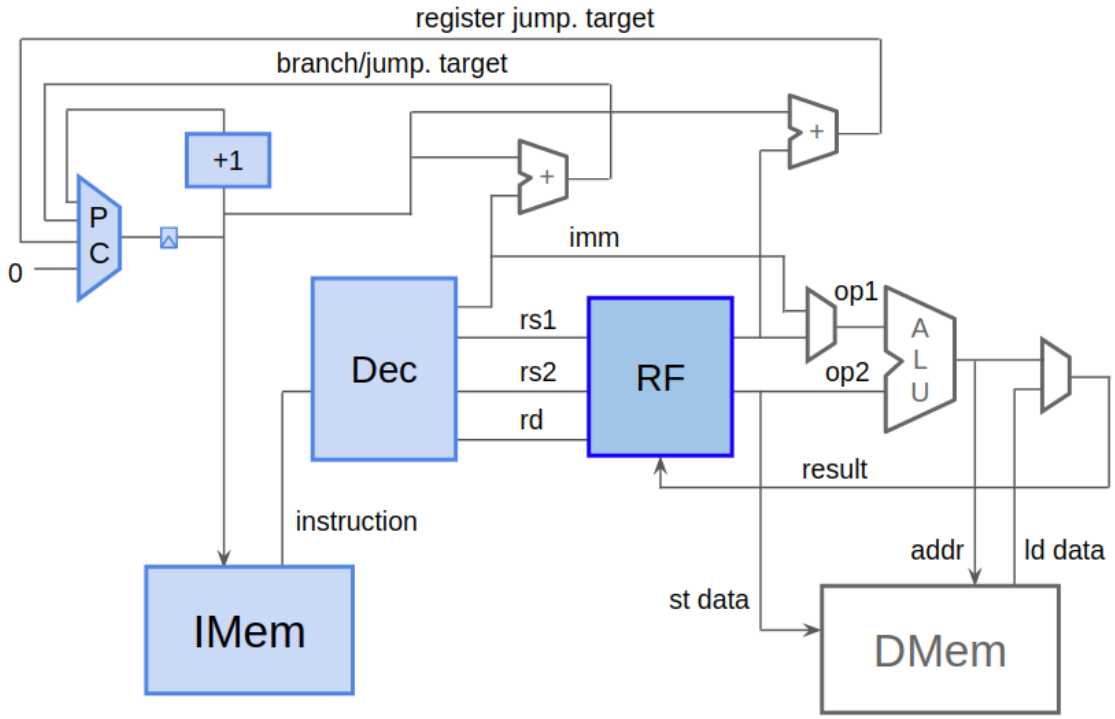


Рисунок 11 – Регистровый файл RF на схеме упрощенного процессора RISC-V

Внизу кода в закомментированном виде расположен следующий пример макроса регистрового файла RF:

```
//m4+rf(32, 32, $reset, $wr_en, $wr_index[4:0], $wr_data[31:0], $rd1_en,
$rd1_index[4:0], $rd1_data, $rd2_en, $rd2_index[4:0], $rd2_data)
```

Данный код создает регистровый файл, подключенный к заданным входным и выходным сигналам, как показано ниже. Каждый из двух портов чтения (rs1, rs2) получает на вход индекс регистра для чтения и разрешающий сигнал (rd), который должен подаваться, когда требуется чтение. Прочитанные данные подаются на выход в том же цикле:

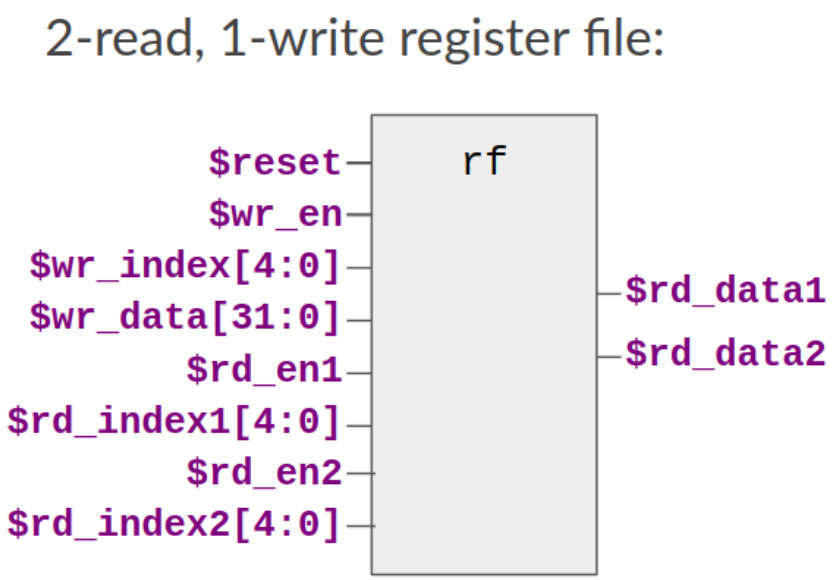


Рисунок 12 – Пример регистрового файла (до модификации)

Например, для чтения регистра 5 (x5) и регистра 8 (x8), сигналы на входах `$rd_en1` и `$rd_en2` должны быть установлены в 1, а на `$rd_index1` и `$rd_index2` поданы значения 5 и 8.

Несколько замечаний:

- для этого макроса аргументами выходного сигнала являются имена сигналов, а входными аргументами являются выражения;
- обратите внимание, что используется сокращение «rd» как сокращение от read (чтение). Такое сокращение легко спутать с обозначением поля команды, отвечающего за регистр назначения (которое в RISC-V также обозначается «rd»).

Измените аргументы макроса, связанные с чтением регистров, чтобы правильно считывать значения исходных регистров команд:

1. Раскомментируйте инстанцирования rf (регистрового файла). (Учитывая, что макросы M4 являются экспериментальной конструкцией, запись макроса нужно представлять в одной (очень длинной) строке. Кроме того, рекомендуется оставить его там, где он есть, в самом низу файла, чтобы улучшить читабельность кода во вкладке NAV-TLV.
2. Хотя макрос еще не подключен должным образом, скомпилируйте и проведите его моделирование. В NAV-TLV можно будет увидеть расширение этого макроса. В нем используется синтаксис, который ранее в примерах не использовался, но с его помощью можно увидеть предупреждения и ошибки, выделенные из-за неподключенных входных и выходных сигналов. По мере подключения сигналов на следующих шагах, эти предупреждения будут постепенно убираться. Также во вкладке LOG либо в журнале событий появятся соответствующие сообщения. Во вкладке DIAGRAM также отобразятся

изменения. Во вкладке VIZ появится регистровый файл, но его поведение будет некорректным, пока его входы не будут правильно подключены.

Схема декодирования инструкции вырабатывает сигналы, необходимые для чтения регистрового файла. На основе кода типа команды она определяет, нужны ли исходные регистры. Она извлекает поля rs1 и rs2, которые задают индексы для этих регистров, если они корректные.

3. Измените соответствующие аргументы макроса RF, чтобы соединить выходные сигналы декодирования с входными сигналами чтения регистрового файла для чтения нужных регистров.
4. Подключите выходные данные чтения к новым сигналам с именами `$src1_value` и `$src2_value`, заменив соответствующие аргументы макроса на эти новые имена сигналов. (Битовые диапазоны не нужны, так как они явно указаны в определении макроса).
5. Скомпилируйте и выполните моделирование кода. Обратите внимание, что значение регистра в каждой записи регистрового файла равно индексу записи. Более типичным вариантом было бы инициализировать все значения нулем. Ненулевые начальные значения используются для того, чтобы упростить следующий шаг.
6. Убедитесь в правильности работы в VIZ, наблюдая за считыванием исходного регистра из RF. Сохраните проект.

Арифметико-логическое устройство

Теперь имеются исходные значения, с которыми можно оперировать, поэтому на этом шаге создайте арифметико-логическое устройство (АЛУ, рисунок 13). Схема АЛУ во многом похожа на схему калькулятора, который был разработан ранее. Оно вычисляет для каждой возможной инструкции результат и затем выбирает, основываясь на полях инструкции, какой из этих результатов является правильным и должен быть подан на выход.

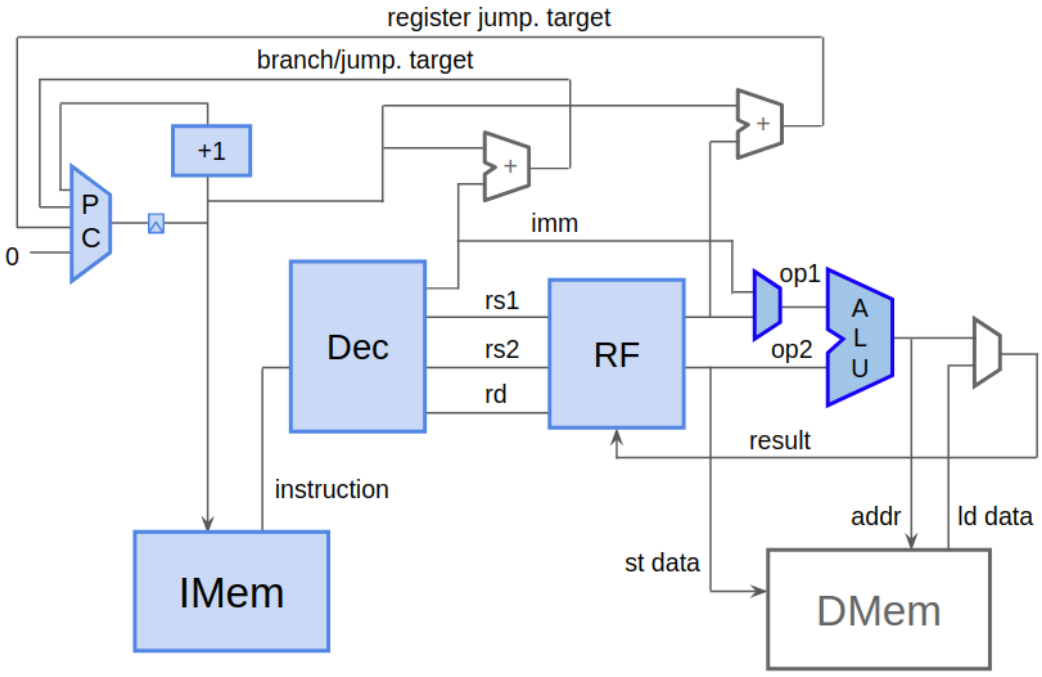


Рисунок 13 – АЛУ в структуре упрощенного процессора RISC-V

На данном этапе будет реализована поддержка команд, которые используются только в тестовой программе. Поскольку команды ветвления не возвращают значения результата, нужно обеспечить поддержку только команд ADDI (которая добавляет непосредственное значение к значению в регистре 1) и ADD (которая складывает два значения из регистров).

Обратите внимание на ошибку в схеме. Непосредственное значение используется вместо `op2 (src2)`, а не `op1 (src1)`.

1. Используйте следующую заготовку кода, чтобы присвоить выходу $result результат выполнения для команд ADDI и ADD:

```
$result[31:0] =    $is_addi ? $src1_value + $imm :
    ...
                   32'b0;
```

2. Скомпилируйте и выполните моделирование кода. Вычисленные результаты должны появиться в окне VIZ «Instr. Decode».

Запись в регистровый файл

Результат `$result` нужно записать обратно в регистр назначения (rd) в регистровом файле (если команда предполагает запись результата в регистр назначения). Выполните следующие шаги:

1. Подключите входы записи регистрового файла, чтобы выполнить запись для команд, имеющих корректный регистр назначения.
2. Скомпилируйте и выполните моделирование кода. Проверьте LOG. Убедитесь с помощью VIZ, что значение регистра назначения записывается в регистровый файл.

   В RISC-V, регистр `x0` называется «always-zero» («всегда нулевой»). Один из способов реализовать такое поведение – избегать записи `x0`.
3. В настоящее время тестовая программа не записывает в `x0`, поэтому нет возможности проверить это изменение. Добавьте инструкцию после ветвления, которая записывает ненулевое значение в `x0`, и посмотрите, как выглядит ее выполнение в VIZ.
4. Измените код так, чтобы убрать сигнал на вход разрешения записи в регистровый файл, если регистр назначения равен 0. Скомпилируйте и выполните моделирование, а также выполните отладку кода и убедитесь с помощью вкладки VIZ, что нулевой регистр не перезаписывается. Добавленную инструкцию проверки можно удалить, поскольку перезапись нулевого регистра невозможна на уровне компилятора.

Логика ветвления

Последняя часть архитектуры процессора для правильного выполнения тестовой программы – это реализация команд ветвления (рисунок Рисунок 14). Тестовая программа использует операцию BLT для повторения цикла, если следующее значение счетчика меньше десяти. Используется команда BGE для реализации бесконечного цикла в конце тестовой программы. Реализуем инструкции условного перехода, показанные на рисунке ниже.

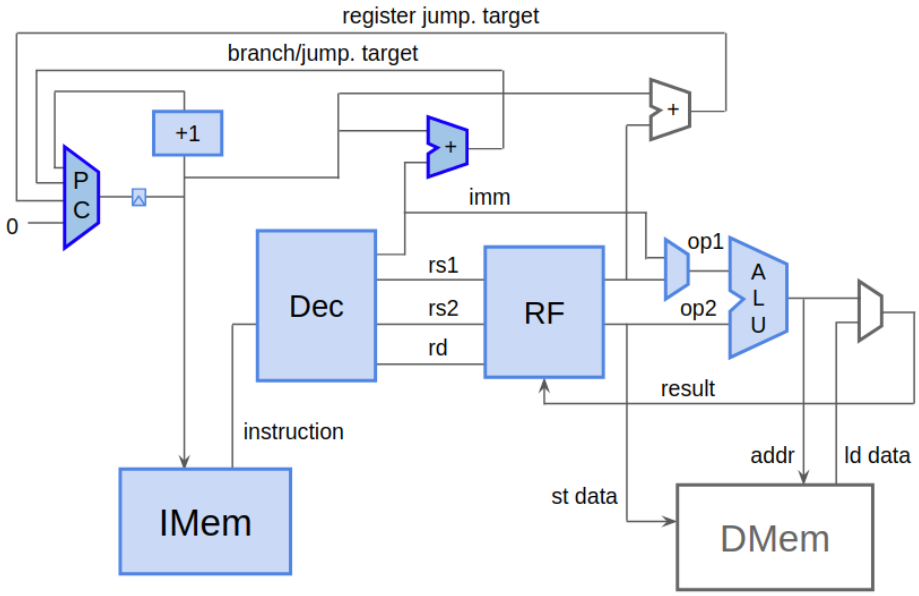


Рисунок 14 – Логика ветвления на схеме упрощенного процессора RISC-V

Инструкция условного перехода означает изменение адреса в регистре PC, если ее условие истинно. Условие – это результат сравнение двух значений регистров. Для реализации инструкции условного перехода требуется:

- определить, является ли инструкция ветвлением, которое должно выполниться (`$taken_br`);
- вычисление целевого адреса операции ветвления (`$br_tgt_pc`);
- обновление значения в PC (`$pc`).

Ниже приведена схема логики работы операции ветвления:

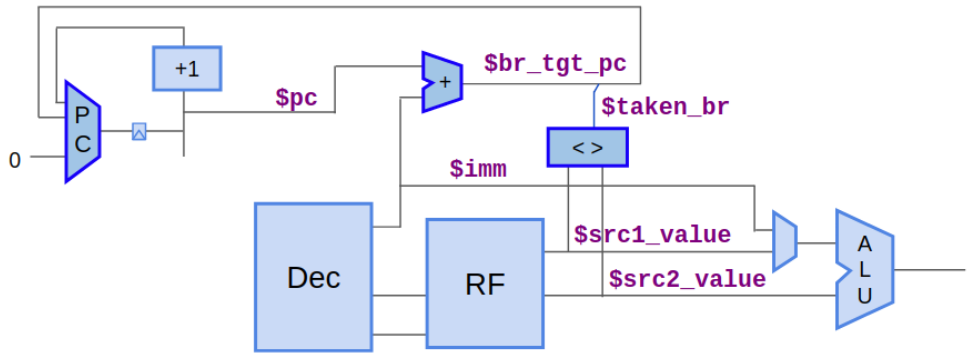


Рисунок 15 – Логика работы операции ветвления

реализуем проверку условия ветвления (`$taken_br`). Каждая инструкция условного ветвления имеет свое выражение условия (таблица Таблица 1), основанное на двух значениях регистра источника (`$src1_value` и `$src2_value`, обозначенных ниже как x1 и x2).

Таблица 1

Выражения условий для каждой инструкции условного перехода

Инструкция	**Описание**	**Выражение условия**
BEQ	Branch if equal (ветвление, если равно)	`x1 == x2`
BNE	Branch if not equal (ветвление, если не равно)	`x1 != x2`
BLT	Branch if less than (ветвление, если меньше)	`(x1 < x2) ^ (x1[31] != x2[31])`
BGE	Branch if greater than or equal (ветвление, если больше или равно)	`(x1 >= x2) ^ (x1[31] != x2[31])`
BLTU	Branch if less than, unsigned (ветвление, если меньше, беззнаковое сравнение)	`x1 < x2`

BGEU	Branch if greater than or equal, unsigned (ветвление, если больше или равно, беззнаковое сравнение)	**x1 >= x2**

Аналогично тому, как это реализовано в АЛУ с помощью мультиплексора можно определять, нужно ли делать ветвление, выбирая соответствующий результат сравнения:

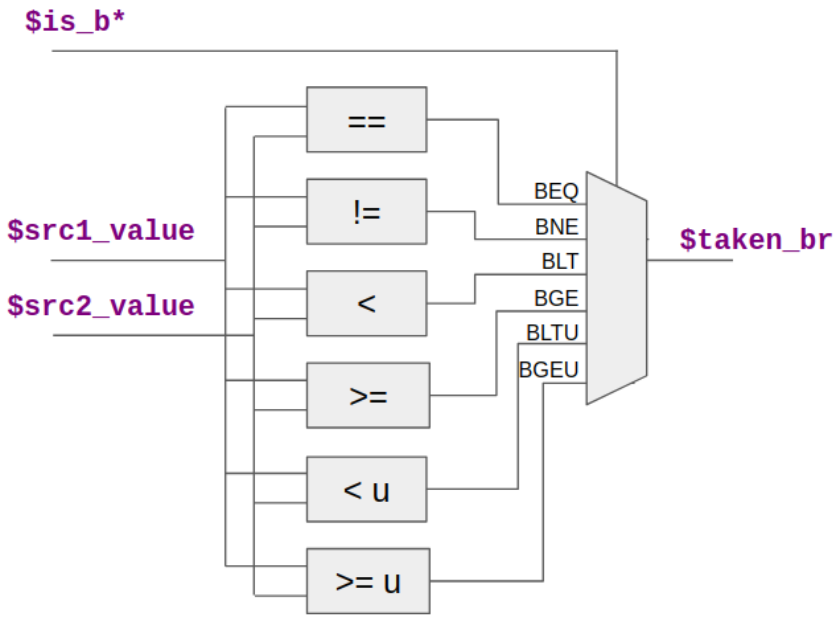


Рисунок 16 – Логическая схема установки флага ветвления

Выполните следующие действия:

1. Закодируйте схему (по примеру с АЛУ). В качестве значения по умолчанию присвойте нулевое значение для команд без ветвления. Для каждой инструкции ветвления определите значение на основе условного выражения для этой инструкции, приведенного в таблице выше.

Также нужно знать целевой адрес команды ветвления. Целевой адрес указывается в поле immediate как относительное смещение байта от текущего адреса (значения PC). Таким образом, целевой адрес команды ветвления – это текущее значение PC плюс значение поля immediate.

2. Запишите выражение для `$br_tgt_pc[31:0] = ....`
3. Если команда является командой ветвления, то следующий адрес считываемой инструкции (значение PC) должен быть целевым адресом команды ветвления. Обновите существующее выражение `$next_pc`, чтобы учесть это.
4. Скомпилируйте, проведите моделирование и отладку кода. Когда все будет работать корректно, программа должна зациклиться. Она должна прекратить цикл, как только будет

получена сумма значений от 1 до 9, равная 45. Заключительный ADDI вычитает из этой суммы 44 и, следовательно, должен поместить значение 1 в регистр x30. Затем конечный BGE должен зациклиться до бесконечности.

Теперь, когда тестовая программа работает корректно, добавьте автоматическую проверку кода. Можно сообщить платформе Makerchip, что тест был пройден успешно или не пройден, присвоив выходным сигналам Verilog соответствующие значения. В контексте `\TLV` сигналы Verilog маркируются с предваряющим символом «*».

Реализуйте проверку того, что значение PC повторяется, и что регистр x30 содержит значение 1.

1. Включите проверку, заменив строку `*passed = 1'b0` на `m4+tb()`.
2. Изучите макрорасширение, определяющее `*passed` в NAV-TLV. Проверьте LOG на наличие сообщения «Simulation PASSED!!!».
3. Сохраните проект Makerchip.

Список источников

1. Visualization of RISC-V CPU Simulation: [Электронный ресурс] // GtiLab. URL: https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/tree/master/RISC-V_CPU_core_building/Chapter_4.RISC-V-Subset_CPU/Visualization_of_RISC-V_CPU_Simulation
2. Starting-Point Code Demo: [Электронный ресурс] // GitLab. URL: https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/tree/master/RISC-V_CPU_core_building/Chapter_4.RISC-V-Subset_CPU/Starting_Point_Code
3. Reference Solutions: [Электронный ресурс] // GitLab. URL: https://git.miem.hse.ru/mtomarov/RISC-V_courses/-/tree/master/RISC-V_CPU_core_building/Chapter_4.RISC-V-Subset_CPU/Reference_Solutions
4. Building a RISC-V CPU Core: [Электронный ресурс] // GitHub. URL: https://github.com/stevehoover/LF-Building-a-RISC-V-CPU-Core
5. Makerchip app: [Электронный ресурс] // PyPI. URL: https://pypi.org/project/makerchip-app/
6. RISC-V shell.tlv: [Электронный ресурс] // Makerchip. URL: https://makerchip.com/sandbox?code_url=https:%2F%2Fraw.githubusercontent.com%2Fstevehoover%2FLF-Building-a-RISC-V-CPU-Core%2Fmaster%2Frisc-v_shell.tlv
7. Makerchip: [Электронный ресурс] // Makerchip. URL: https://makerchip.com/sandbox?code_url=https:%2F%2Fraw.githubusercontent.com%2Fstevehoover%2FLF-Building-a-RISC-V-CPU-Core%2Fmain%2Frisc-v_solutions.tlv