Завершение сборки процессора RISC-V

Оглавление

Введение

Теперь, когда тестовая программа выполняется правильно, доработаем проект, чтобы была реализована поддержка для оставшихся команд.

Цели обучения

Целью данной главы является:

- Закрепление знаний полученных в предыдущих главах;
- Формирование понимания базовой RISC-V ISA.

Тестовая программа

В этом разделе используется новая тестовая программа, которая проверяет каждую команду из набора команд RV32-I.

Вместо тестовой программы, использованной в предыдущей главе (все разграничено комментариями //----------), создайте макрос (с правильным отступом): `m4_test_prog()` и выполните компиляцию и моделирование кода.

Поскольку новая программа создана на основе встроенного макроса, вы больше не сможете увидеть или отредактировать ее исходный код. Она также не будет видна во вкладке NAV-TLV, но должна быть видна в VIZ.

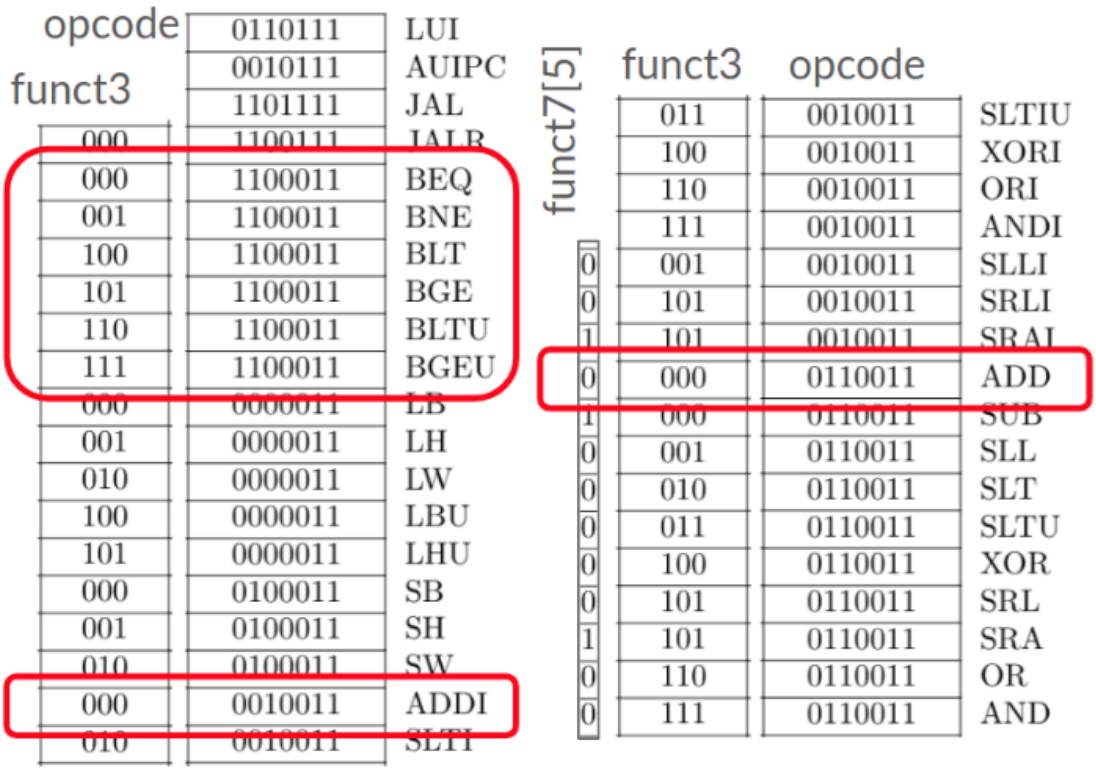
Для оставшихся упражнений будет легче производить отладку с представлением переменных в в шестнадцатеричном формате. Макрос `m4_test_prog()` настраивает вкладку VIZ для отображения значений регистров в шестнадцатеричном формате. Следует помнить, что каждая шестнадцатеричная цифра представляет собой четыре двоичных разряда.

Тестовая программа выполняет все команды один раз, каждая из которых записывает результат в уникальный регистр, начиная с x5 и далее по возрастанию. Для каждой команды выполняется XOR с таким значением, которое в случае правильного результата выполнения выдаст 1. Если все команды исполняются корректно , регистры x5-x27 будут содержать значение 1 при завершении теста (в регистры x28-x30 также будут содержать значение 1). Вы можете использовать VIZ, чтобы определить, какие команды выдали неправильные значения, и устранить ошибки. Из-за того, что большинство команд еще не реализовано, в большинство регистров будет храниться 0.

В этом разделе используется тот же тестбенч, что и раньше, – `m4+tb()`. Он выведет сообщение «Пройдено», когда программа завершится. Но следует помнить, что тестбенч не проверяет, что значения регистров равны 1. Это нужно проверить самостоятельно в VIZ.

Логика декодирования

В одном из предыдущих заданий вы реализовали логику декодирования для команд, обведенных красным цветом.



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

funct7[5]	funct3	opcode	
	011	0010011	SLTIU
	100	0010011	XORI
	110	0010011	ORI
	111	0010011	ANDI
0	001	0010011	SLLI
0	101	0010011	SRLI
1	101	0010011	SRAI
0	000	0110011	ADD
1	000	0110011	SUB
0	001	0110011	SLL
0	010	0110011	SLT
0	011	0110011	SLTU
0	100	0110011	XOR
0	101	0110011	SRL
1	101	0110011	SRA
0	110	0110011	OR
0	111	0110011	AND

Рисунок 1 –Таблица декодирования команд.

- Выполните следующее задание: за исключением команд загрузки и хранения (`LB`, `LH`, `LW`, `LBU`, `LHU`, `SB`, `SH`, `SW`), реализуйте логику декодирования для оставшихся не обведенных команд выше (`$is_<instr> = ...`). Помните, что можно использовать символ «x» для незначащих битов;
- В данной реализации все команды загрузки и сохранения будут рассматриваться одинаково, поэтому присвоение `$is_load` должно основываться только на коде операции. `$is_s_instr` уже идентифицирует сохранения, поэтому там не нужна дополнительная логика декодирования для сохранений;
- Скомпилируйте проект. Обратите внимание, что при декодировании полей команд во вкладке VIZ теперь показываются мнемоники команд. Обратите также внимание на то, что во вкладке LOG будет много предупреждений для всех неиспользуемых логических сигналов.

По мере увеличения размеров проекта возможно, что DIAGRAM может не сгенерироваться должным образом.

Арифметико-логический блок

Добавьте в ALU поддержку оставшихся команд. Для этого необходимо расширить оператор присваивания для `$result`. Поскольку почти каждая команда будет выполнять свою операцию, код будет индивидуальным под каждую команду. Ниже приведены требуемые выражения, но их следует набрать самостоятельно, чтобы у вас была возможность подумать над каждой командой. Если вы хотите получить больше информации об этих функциях, полезным справочником является зеленая карта RISC-V [1], или можно обратиться к спецификации RISC-V Unprivileged ISA Specification [2].

Код для реализации `ADD` и `ADDI` довольно прост. Большинство других операций также имеют простой код, но некоторые операции описываются более сложным кодом. Некоторые из них имеют общие фрагменты кода, поэтому следует сначала создать назначения для них.

```
// SLTU and SLTI (set if less than, unsigned) results:
$sltu_rslt[31:0]  = {31'b0, $src1_value < $src2_value};
$sltiu_rslt[31:0] = {31'b0, $src1_value < $imm};

// SRA and SRAI (shift right, arithmetic) results:
//   sign-extended src1
$sext_src1[63:0] = { {32{$src1_value[31]}}, $src1_value };
//   64-bit sign-extended results, to be truncated
$sra_rslt[63:0] = $sext_src1 >> $src2_value[4:0];
$srai_rslt[63:0] = $sext_src1 >> $imm[4:0];
```

Рисунок 2 – Код, необходимый для ALU.

Выполните следующие действия:

- Введите приведенные выше выражения присваивания перед существующим присвоением `$result`. Проанализируйте введенные выражения;
- Скомпилируйте и выполните моделирование разработанного кода. При возникновении ошибок в LOG произведите отладку.

Теперь нужно реализовать полный ALU. Ниже приведено описание ALU, некоторые, из команд используют описания, которые были реализованы выше.

[1] https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf

[2] https://riscv.org/technical/specifications/

```
ANDI:   $src1_value & $imm
ORI:    $src1_value | $imm
XORI:   $src1_value ^ $imm
ADDI:   $src1_value + $imm
SLLI:   $src1_value << $imm[5:0]
SRLI:   $src1_value >> $imm[5:0]
AND:    $src1_value & $src2_value
OR:     $src1_value | $src2_value
XOR:    $src1_value ^ $src2_value
ADD:    $src1_value + $src2_value
SUB:    $src1_value - $src2_value
SLL:    $src1_value << $src2_value[4:0]
SRL:    $src1_value >> $src2_value[4:0]
SLTU:   $sltu_rslt
SLTIU:  $sltiu_rslt
LUI:    {$imm[31:12], 12'b0}
AUIPC:  $pc + $imm
JAL:    $pc + 32'd4
JALR:   $pc + 32'd4
SLT:    ( ($src1_value[31] == $src2_value[31]) ?
              $sltu_rslt :
              {31'b0, $src1_value[31]} )
SLTI:   ( ($src1_value[31] == $imm[31]) ?
              $sltiu_rslt :
              {31'b0, $src1_value[31]} )
SRA:    $sra_rslt[31:0]
SRAI:   $srai_rslt[31:0]
```

Рисунок 3 – Описание ALU на TL-Verilog для каждой команды.

- Доработайте код для `$result`, чтобы завершить работу над ALU;
- Проведите синтез и моделирование разработанного кода, устраните любые возникшие ошибки в `LOG`. Сигналы `$is_<instr>` больше не должны быть неиспользуемыми;
- Если какая-либо из новых команд не приводит к значениям регистра 1 в VIZ, выполните ее отладку. В конце моделирования значения регистров должны быть равны 1, кроме x0-4, x27 и x31. Сохраните проект Makerchip.

Логика безусловного перехода

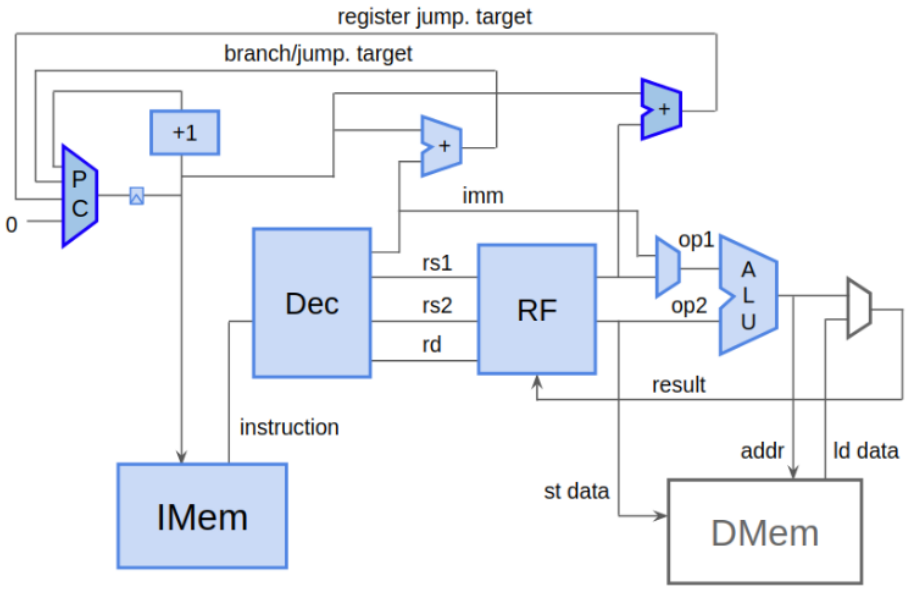


Рисунок 4 – Реализация логики переходов.

ISA, в дополнение к командам условных переходов, также поддерживает функции безусловного перехода (которые некоторые другие ISA называют "безусловными ветвлениями"). RISC-V имеет два вида команд перехода:

- JAL (Jump and link) – переход и связывание. Происходит переход на адрес PC + IMM;
- JALR (Jump and link register) – регистр перехода и связывания. Переход на адрес SRC1 + IMM;

Понятие «связывания» означает тот факт, что эти команды сохраняют свой исходный PC + 4 в регистр назначения, который задан ALU. (Регистр связывания особенно полезен для переходов, используемых для реализации вызовов функций, которые должны возвращаться по адресу перехода после выполнения функции).

Выполните следующие действия:

- Вычислите `$jalr_tgt_pc[31:0]` (SRC1 + IMM).
- Обновите логику PC, чтобы выбрать правильный $next_pc для JAL ($br_tgt_pc) и JALR ($jalr_tgt_pc). В тестовой программе команды JAL и JALR должны переходить к следующей по порядку команде (как если бы безусловный переход вообще не происходил), за исключением JAL, которая переходит на себя. Если предположить, что регистр x30 также правильно установлен в 1, последняя итерация JAL приведет к тому, что тест выдаст сообщение «Пройден» в LOG и VIZ (хотя загрузка и сохранение еще не работают). Проведите проверку работы разработанного модуля в VIZ.

Загрузка, хранение и память данных

Адресация памяти

До сих пор все разработанные команды в качестве операндов использовали значениями регистров. Процессор не может считаться полноценным, если у него нет памяти. Нужно добавить память. Для этого нужно предусмотреть операции загрузки и сохранения, которые будут читать данные из памяти и записывать в нее.

Для выполнения команд загрузки и сохранения требуется адрес, с которого производится чтение или на который производится запись. Функции загрузки и сохранения могут читать или записывать отдельные байты (1 байт/8 бит), полуслова (2 байта/16 бит) или слова (4 байта / 32 бита).

Рисунок 5 –Слово, полуслово, байт.

Реализуем все команды загрузки/сохранения для работы со словами, предполагая, что два младших бита адреса равны нулю. Другими словами, предлагается работа загрузки/сохранения с естественно выровненными адресами.

Адрес для загрузки/сохранения вычисляется на основе значения из исходного регистра и значения смещения (часто нулевого), предоставляемого в качестве константы в поле команды immediate:

addr = rs1 + imm

Загрузка

Команда загрузки (`LW`, `LH`, `LB`, `LHU`, `LBU`) имеет вид:

`LOAD rd, imm(rs1).`

Загрузка использует формат команды I-типа:

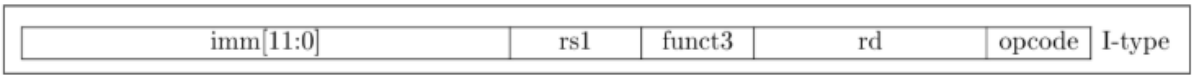


Рисунок 6 –Формат команды I-типа.

Команда загрузки записывает в свой регистр назначения значение, считанное из указанного адреса памяти, который можно представить как:

```
rd <= DMem[addr] (where, addr = rs1 + imm)
```

Хранение

Операция хранения (`SW`, `SH`, `SB`) имеет вид:

```
STORE rs2, imm(rs1).
```

Она имеет свой собственный формат команды S-типа:

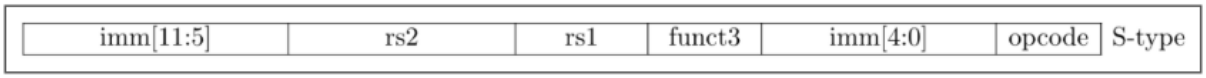


Рисунок 7 –Формат команды S-типа.

Она записывает в указанный адрес памяти значение из исходного регистра rs2:

```
DMem[addr] <= rs2 (where, addr = rs1 + imm)
```

Логика адресации

Вычисление адреса `rs1 + imm` – это то же самое вычисление, что и выполняемое командой `ADDI`. Поскольку функции загрузки и хранения не требуют ALU, можно использовать ALU для этого вычисления.

Выполните следующие действия:

Для загрузки / хранения (`$is_load/$is_s_instr`), вычислите `$result` как адрес (`rs1 + imm`), как в команде `ADDI`. (Это изменение пока не будет заметно в VIZ).

Память данных

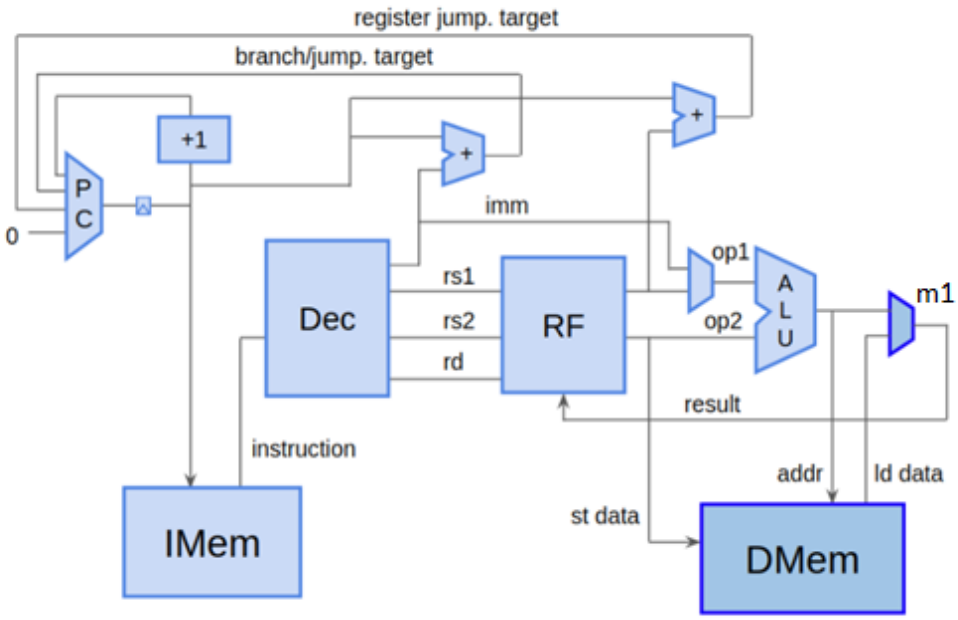


Рисунок 8 – Реализация памяти данных

Чтобы моделирование проходило быстрее, создайте память данных такого же размера, как и регистровый файл.

В отличие от регистра, который позволяет считывать два значения за каждый цикл и в том же цикле записывать значение, такая память должна считывать только одно значение или записывать одно значение каждый цикл, чтобы обработать команду загрузки или хранения. Как и регистровый файл, память DMem разделена по словам. При этом поддерживаются только операции загрузки / хранения с выровненными адресами (младшие два бита адреса будут всегда равны нулю).

Основываясь на суждениях выше:

- Запись разрешена для хранения (`$is_s_instr`);
- Чтение разрешено для загрузки (`$is_load`);
- Результат ALU (`$result`) используется для расчета адреса чтения / записи; это байтовый адрес, в то время как память состоит из 32-битных слов;
- регистр `rs2 ($src2_value)` содержит данные для записи;
- единственным выходом DMem являются данные для загрузки (которые называются `$ld_data`).

Выполните следующие действия:

- Аналогично тому, как и для регистрового файла, в коде подготовлен закомментированный макрос для `m4+dmem(32, 32, $reset, $addr[4:0], $wr_en, $wr_data[31:0], $rd_en, $rd_data)`. Раскомментируйте его;
- Задайте соответствующие аргументы макроса для подключения корректных входных и выходных сигналов. Обязательно извлеките соответствующие биты адреса байта для управления адресом слова DMem. Поскольку память имеет один порт чтения, для DMem требуется меньше аргументов, чем для регистрового файла;
- Скомпилируйте, проведите моделирование кода, а также отладку ошибок компиляции.

Данные загрузки (`$ld_data`), поступающие из DMem, должны быть записаны в регистровый файл. Для выбора `$ld_data` для команд загрузки необходим новый мультиплексор (m1), как показано на схеме (рисунок 8).

- Добавьте новый мультиплексор, чтобы записывать `$ld_data`, а не `$result`, в регистровый файл, когда `$is_load` установлен в 1;
- Отладьте ошибки компиляции при их наличии. На этом этапе журнал событий должен быть пустым (без ошибок и предупреждений);

Тестовая программа, ближе к концу, выполняет сохранение и загрузку шестнадцатеричного значения 32'h15.

- Изучите команды хранения (`SW`) и загрузки (`LW`) в VIZ. Убедитесь, что значение 'h15 сохраняется в ячейку памяти 2 и загружается в регистр x27;
- Убедитесь, что регистры x5-x30 равны 1 в конце цикла симуляции.

Работающее ядро RISC-V создано! Сохраните все свои наработки.

Следующие шаги

Если вы задаетесь вопросом, какими могут быть ваши следующие шаги на пути к освоению RISC-V, вот несколько из них:

- Продолжите изучать Makerchip, чтобы глубже погрузиться в TL-Verilog и его экосистему;
- Изучите репозиторий курса, который может быть обновлен с учетом последних изменений;
- Узнайте больше о RISC-V на сайте riscv.org.

Изучите другие предложения курсов от The Linux Foundation и следите за новыми курсами по RISC-V.