**Введение (Слайд 2)**

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 (Слайд 3)**

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 (Слайд 4-5)**

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

* Проделайте следующие действия: за исключением команд загрузки и хранения (LB, LH, LW, LBU, LHU, SB, SH, SW), реализуйте логику декодирования для оставшихся не обведенных инструкций выше ($is\_<instr> = ...). Помните, что можно использовать «x» для незначащих битов;
* В данной реализации все загрузки и все сохранения будут рассматриваться одинаково, поэтому назначение $is\_load должно быть основано только на коде операции. $is\_s\_instr уже идентифицирует сохранения, поэтому там не нужна дополнительная логика декодирования для сохранений;
* Скомпилируйте проект. Обратите внимание, что декодирование инструкций VIZ теперь показывает мнемоники инструкций. Обратите также внимание на то, что LOG будет содержать много предупреждений для всех неиспользуемых сигналов.

По мере увеличения размеров проекта возможно, что DIAGRAM может не сгенерироваться должным образом. Это может доставить неудобства, но это не обязательно связано с реализацией схемы.

**Арифметико-логический блок (Слайд 6)**

Добавьте в ALU поддержку оставшихся команд. Для этого необходимо расширить оператор присваивания для $result. Поскольку почти для каждой инструкции будет свое выражение, надо будет разработать много кода. Ниже приведены требуемые выражения, но их следует набрать самостоятельно, чтобы у вас была возможность подумать над каждой инструкцией. Если вы хотите получить больше информации об этих функциях, полезным справочником является зеленая карта RISC-V, или можно обратиться к спецификации RISC-V Unprivileged ISA Specification.

Выражения для ADD и ADDI довольно просты. Большинство других операций также имеют простые выражения, но некоторые из них сложнее. Некоторые из них имеют общие подвыражения, поэтому следует сначала создать назначения для этих подвыражений.

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 отладьте их.

**(Слайд 7-8)** Теперь нужно реализовать полный ALU. На слайде приведены выражения, некоторые из которых используют подвыражения, которые были реализованы выше.

* Увеличьте выражение для $result, чтобы завершить работу ALU для поддержки оставшихся команд;
* Скомпилируйте и проведите 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 (Слайд 9)**

ISA, в дополнение к условным ветвлениям, также поддерживает функции безусловного перехода (которые некоторые другие ISA называют "безусловными ветвлениями"). RISC-V имеет две формы инструкций перехода:

* JAL (Jump and link) – переход и связывание. Происходит переход на адрес PC + IMM (как и ветви, поэтому целью является $br\_tgt\_pc, уже назначенный);
* 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 АЛУ. (Регистр связывания особенно полезен для переходов, используемых для реализации вызовов функций, которые должны возвращаться по адресу перехода после выполнения функции).

**(Слайд 10)** 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 команд JAL и JALR должны переходить к следующей по порядку команде (как если бы безусловный переход вообще не происходил), за исключением JAL, который должен переходить сам на себя. Если предположить, что регистр x30 также правильно установлен в 1, этот последний 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 (Слайд 11)**

**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).

Однако, избежим этого нюанса и реализуем все инструкции 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 immediate:

**addr = rs1 + imm**

**Загрузка (Слайд 12)**

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

LOAD rd, imm(rs1).

Она использует формат команды I-типа.

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

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

**Хранение (Слайд 13)**

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

STORE rs2, imm(rs1).

Она имеет свой собственный формат команды S-типа.

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

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

**Адресная логика (Слайд 14)**

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

Выполните следующие действия:

Для загрузки / хранения ($is\_load/$is\_s\_instr), вычислите $result как адрес (rs1 + imm), как в инструкции ADDI. (Это изменение пока не будет заметно в VIZ).

**Память данных (Слайд 15)**

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

В отличие от регистра, который способен считывать два значения за каждый цикл и в том же цикле записывать значение, 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 поддерживается только операции загрузки / хранения с естественно выровненными адресами (поэтому младшие два бита адреса будут всегда нулевые).

Основываясь на обсуждении 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).

**(Слайд 16)** Выполните следующие действия:

* Аналогично тому, как и для файла регистров, 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 RF;
* 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 инструкций загрузки необходим новый мультиплексор, 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 LOG должен быть пустым (без ошибок и предупреждений);

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

* Изучите инструкции 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.

**(Слайд 17)** 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 репозитарий 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.