**Введение (Слайд 2)**

В этой главе рассматривается создание одной из реализаций ядра процессора RISC-V. Оно будет способно 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 журналах;
  2. использование визуальной отладки для понимания общего поведения цифровой схемы;
  3. использование средств просмотра вейвформ для понимания детального поведения цифровой схемы;
  4. отслеживание некорректного поведения схемы от предупреждения об ошибке до обнаружения ее причины.
* инстанцирование уже существующих компонентов Verilog и TL-Verilog.

**Настройка рабочего окружения для выполнения практической работы по разработке процессорного ядра (Слайд 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 дизайн.

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. Вы можете редактировать файлы непосредственно через веб‑интерфейс GitHub и вставлять свой код из Makerchip в браузере, или вы можете клонировать свой репозиторий на локальный диск и вставить свой 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:

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

После этого установите приложение Makerchip]:

pip3 install makerchip-app

**Использование кода точки старта проекта**

Откройте шаблон кода точки старта 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 ином случае просто нажмите на [ссылку](https://makerchip.com/sandbox?code_url=https:%2F%2Fraw.githubusercontent.com%2Fstevehoover%2FLF-Building-a-RISC-V-CPU-Core%2Fmaster%2Frisc-v_shell.tlv) [**Ошибка! Источник ссылки не найден.**], чтобы открыть исходный код в браузере, и, как только код загрузится, выберите пункт меню «Сохранить как новый проект».
3. Симуляция должна запуститься. В LOG должно появиться сообщение «Simulation FAILED!!!» (и так будет до тех пор, пока вы успешно не выполните все действия из этой главы). 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 ваша работа была правильно автоматически сохранена в облаке или локальной системе, проверив строку состояния в Makerchip.

Обычно, если вы допустили ошибки при разработке и вам нужно вернуться к стабильной версии вашего кода, достаточно нажать Ctrl-Z, но также рекомендуется сохранять промежуточные версии проектов (или git-коммиты).

**Микроархитектура процессорного ядра (Слайд 4)**

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 создадите 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 тактовый цикл. Выполнение команды в течение одного такта возможно только в том случае, если тактирование происходит относительно медленно, что и предполагается.

Начнем с реализации процессора, достаточного для выполнения простой тестовой программы. По мере добавления каждого нового элемента функциональности, вы будете видеть в панели VIZ реализованное поведение. 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 инструкций RV32I.

Рассмотрим компоненты разрабатываемого процессора, проследив за прохождением инструкции через составные компоненты схемы. Примерно в таком же порядке мы будем реализовывать логику работы процессора.

1. **Счетчик команд (PC)**

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

1. **IMem (Instruction Memory)**

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

1. **Декодер (Dec)**

Теперь, когда процессор имеет инструкцию для выполнения, нужно интерпретировать или декодировать ее. Необходимо разделить ее на области в зависимости от их типа. Эти области будут определять, какой регистр требуется прочесть, какую операцию выполнить, и т.д.

1. **Чтение регистрового файла (RF)**

Регистровый файл представляет собой небольшую область памяти с локальными данными и активно используется в зависимости от программы. Инструкция декодируется для определения, с каким регистром нужно работать. Соответствующие поля определяют какие регистры нужно прочесть из регистрового файла.

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

После получения значений из регистров происходит их обработка. АЛУ будет складывать, вычитать, умножать, сдвигать, и выполнять другие действия с данными из регистров в зависимости от типа операции, указанной в инструкции.

1. **Запись в регистровый файл**

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

1. **DMem (Data Memory)**

Используемая в данном курсе тестовая программа выполняется исключительно с помощью регистрового файла и не требует наличия памяти (Data Memory). Но ни одно процессорное ядро не будет завершенным без памяти. Данные в DMem записывается с помощью инструкций записи и считываются инструкциями загрузки.

В этом курсе акцент сделан только на ядре процессора. Вся логика, которая была бы необходима для взаимодействия с периферией, такая как контроллеры ввода-вывода (I/O), логика прерываний, системные таймеры и др., не рассматривается.

Используется упрощенное представление о памяти. Процессор общего назначения обычно имеет больший объем памяти, содержащий как инструкции, так и данные. Как правило, для доступа к памяти требуется много тактовых циклов. Чтобы снизить задержку доступа к памяти, используют кэши, в которых хранятся недавно полученных данные в памяти вблизи ядра процессора. 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 реализуются отдельные кэши команд и данных с временем доступа равным одному циклу, которые мало чем отличаются от IMem и DMem, используемых в этой главе.

**Программный счетчик PC (Слайд 5)**

Сначала реализуем только последовательную выборку, поэтому программный счетчик PC представляет собой простой счетчик.

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 должно быть на 4 байта. 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;
* 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.

Выполните следующее:

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

Если возникли проблемы, эталонные решениядля всех лабораторных работ RISC-V находятся в репозитарии курса на GitHub.

**Память инструкций IMem (Слайд 6)**

Память инструкций IMem (Instruction Memory) реализуется путем инстанцирования макроса Verilog. Этот макрос принимает на вход байтовый адрес, а на выходе выдает 32‑битные данные для чтения. Макрос может быть инстанцирован, например, следующим образом:

`READONLY\_MEM($addr, $$read\_data[31:0])

Перед инстанцированием макроса в Verilog ставится гравис (backquote, не путать с одинарной кавычкой). В выражениях, подобных этому, которые синтаксически не отличают назначенные сигналы от используемых, необходимо идентифицировать назначенные сигналы с помощью префикса «$$». Назначенный сигнал объявляет свой битовый диапазон. Таким образом, выше используется $$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. Это приведет к тому, что реализация с использованием триггеров будет гораздо менее оптимальной, чем с помощью SRAM.

Реализуйте выборку инструкций, инстанцировав 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, окно DIAGRAM выглядит правильно, а VIZ показывает инструкции, считываемые из IMem. Если что-то выглядит неправильно, выполните отладку кода с помощью окна WAVEFORM и убедитесь, что $instr больше не появляется в сигналах VIZ «To Be Implemented» («Требуется реализовать»).

**Декодер инструкций (Слайд 7)**

**Типы инструкций**

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

Прежде чем интерпретировать инструкцию, 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) инструкций RV32I. Будем считать, что все инструкции валидны, поэтому можно просто игнорировать эти два бита. ISA (Instruction Standard Architecture) определяет тип инструкции следующим образом:

Для каждого типа инструкции 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»).

**(Слайд 8)** Выполните следующие шаги:

1. Добавьте оператор присваивания в свой код и напишите остальные 5 операторов для типов инструкций 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 осциллограмм WAVEFORM.

**Поля инструкций (Слайд 9)**

Основываясь на типе инструкции, можно извлечь поля инструкции. 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, 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 отладьте. 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.

1. Скомпилируйте и выполните моделирование кода, проверьте LOG (теперь довольно длинный) и отладьте. Список сигналов «To Be Implemented» должен стать меньше.
2. 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. Также обратите внимание, что если растянуть это выражение на две строки, то эта 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.

1. Скомпилируйте и выполните моделирование кода и убедитесь, что LOG теперь чист. Теперь вы должны увидеть индексы регистров в VIZ (в синей части декодирования инструкций). 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 **Ошибка! Источник ссылки не найден.**).

Непосредственное значение для инструкций 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

**(Слайд 10)** 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».

**Инструкция (Слайд 11)**

Теперь нужно определить конкретную инструкцию. 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.

Выполните следующие шаги:

1. Для удобства объедините соответствующие поля в один битовый векторный сигнал. Например: $dec\_bits[10:0] = {$instr[30],$funct3,$opcode};
2. Для каждой из инструкций, обведенных красным (к остальным мы вернемся позже), определите, идентифицирует ли $dec\_bits эту инструкцию. 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).

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

**Чтение регистрового файла (Слайд 12)**

Как и IMem, регистровый файл 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.

Рядом с нижней частью кода в закомментированном виде расположен следующий пример инстанцирования 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 32-входовый регистровый файл шириной 32 бита, подключенный к заданным входным и выходным сигналам, как показано ниже. Каждый из двух портов чтения получает на вход индекс регистра для чтения и разрешающий сигнал, который должен подаваться, когда требуется чтение. Прочитанные данные подаются на выход в том же цикле:

Например, для чтения регистра 5 (x5) и регистра 8 (x8), $rd\_en1 и $rd\_en2 оба 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 «rd» как сокращение для read (чтение), которое легко спутать с регистром назначения, записываемым инструкцией, которая в RISC-V также обозначается «rd».

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

1. Снимите комментарий с инстанцирования rf. (Учитывая, что макросы M4 являются экспериментальной конструкцией, запись макроса нужно выполнять в одной (очень длинной) строке. Кроме того, рекомендуется оставить его там, где он есть, в самом низу файла, чтобы его расширение не загромождало 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 мощью 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 в LOG появятся соответствующие сообщения, а в DIAGRAM появятся разъединения. Также в VIZ появится файл регистра, но его поведение будет случайным, пока его входы не будут правильно подключены.

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

1. Измените соответствующие аргументы макроса RF, чтобы соединить выходные сигналы декодирования с входными сигналами чтения регистрового файла для чтения нужных регистров.
2. 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).
3. 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.
4. Убедитесь в правильности работы в VIZ, наблюдая за считыванием исходного регистра из RF. Сохраните проект.

**Арифметико-логическое устройство (Слайд 13)**

Теперь имеются исходные значения, с которыми можно оперировать, поэтому на этом шаге создайте арифметико-логическое устройство (АЛУ). АЛУ во многом похож на схему калькулятора, который был разработан ранее. Он 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 АЛУ.

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

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

1. Используйте код, подобный следующему, чтобы присвоить АЛУ $result в одном выражении присвоения для инструкций ADDI и ADD:

$result[31:0] = $is\_addi ? $src1\_value + $imm :

...

32'b0;

1. Скомпилируйте и выполните моделирование кода. Вычисленные результаты должны появиться в окне VIZ «Instr. Decode» (хотя неправильные значения записываются обратно в регистровый файл).

**Запись в регистровый файл (Слайд 14)**

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

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

В RISC-V, регистр x0 (индексе регистра равен 0) называется «always-zero» («всегда нулевой»). Один из способов реализовать такое поведение – избегать записи x0.

1. 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.
2. 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 (Слайд 15)**

Последняя часть архитектуры процессора для правильного выполнения тестовой программы – это реализация инструкций ветвления. 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.

Инструкция условного перехода означает изменение адреса в PC, если ее условие истинно. Условие – это 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).

Реализуем проверку условия ветвления ($taken\_br). Каждая инструкция условного ветвления имеет свое выражение условия (на слайде), основанное на двух значениях регистра источника ($src1\_value и $src2\_value, обозначенных ниже как x1 и x2).

**(Слайд 16)** Аналогично тому, как это реализовано в АЛУ с помощью мультиплексора можно определять, нужно ли делать ветвление, выбирая соответствующий результат сравнения.

**(Слайд 17)** Выполните следующие действия:

1. Закодируйте схему в виде одного выражения, как в случае с АЛУ. В качестве стандартного случая тернарного оператора присвойте нулевое значение для инструкций без ветвления. Для каждой инструкции ветвления определите значение на основе условного выражения для этой инструкции, приведенного в таблице выше.

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

1. Запишите выражение для $br\_tgt\_pc[31:0] = ....
2. Если инструкция является ветвлением, то ее следующий PC должен быть целевым PC ветвления. Обновите существующее выражение $next\_pc, чтобы учесть это.
3. 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 упоминаются с предваряющим символом «\*».

**(Слайд 18)** Реализуйте проверку того, что PC программы повторяется, и что 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.