Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Компьютерных Систем и Сетей

Кафедра Информатика и технологии программирования

Дисциплина: Архитектура Вычислительных систем

Курсовой проект

на тему:

"Реализация однотактного RISC-V CPU с использованием Python фреймворка nMigen"

Студент гр.953506: Крапивницкий В.С.

Руководитель: Леченко А.В.

  Минск, 2021

# **Введение**

Войны между RISC и CISC, проходившие в конце 1990-х, уже давно отгремели, и сегодня считается, что разница между RISC и CISC совершенно не имеет значения. Многие заявляют, что наборы команд несущественны.  
  
 Однако на самом деле наборы команд важны. Они накладывают ограничения на типы оптимизаций, которые можно легко добавлять в микропроцессор.

Недавно я подробнее изучил информацию об архитектуре набора команд (instruction-set architecture, ISA) RISC-V и вот некоторые из аспектов, которые по-настоящему впечатлили меня в ISA RISC-V:

1. Это небольшой и простой в изучении набор команд RISC. Очень предпочтителен для тех, кому интересно получать знания о микропроцессорах.
2. Благодаря своей простоте, открытости и связи с университетскими профессорами он с большой вероятностью будет доминировать как архитектура, выбираемая для обучения процессорам в вузах.
3. Его продуманная структура позволяет разработчикам CPU создавать высокопроизводительные микропроцессоры на основе ISA RISC-V.
4. Благодаря отсутсвию лицензионных отчислений и нацеленности на простую аппаратную реализацию увлечённый любитель может, в принципе, создать за приемлемое время собственную конструкцию процессора RISC-V.

nMiegn

nMigen – обновлённый фреймворк языка Python для построения сложного цифрого оборудования. Хотя nMigen не закончен и находится в активной разработке, его уже можно использовать для реальных проектов. Язык nMigen не претерпит несовместимых изменений, хотя стандартная библиотека nMigen и система сборки будут модифицироваться до окончания проекта.

Несмотря на то, что это быстрее, чем ввод схемы, проектирование оборудования с помощью Verilog и VHDL остается утомительным и неэффективным по нескольким причинам. Событийно-ориентированное программирование создаёт проблемы и добавляет лишнего ручного кодирования, которые не нужны для синхронных схем, которые составляют львиную долю современных логических схем. Противоинтуитивные арифметические правила приводят к более крутым кривым обучения и создают благодатную почву для мелких ошибок в дизайне. Наконец, поддержка процедурной генерации логики (метапрограммированние) с помощью операторов «генерации» очень ограничена и ограничивает способы, которыми код может быть обобщен, повторно использован и организован.

Чтобы решить эти проблемы, разработчики из m-labs создали nMigen FHDL - библиотеку, которая заменяет событийно- ориентированную парадигму понятиями комбинаторных и синхронных выражений и операторов, имеет арифметические правила, которые заставляют целые числа всегда вести себя как математические целые числа, и, что наиболее важно, позволяет строить лигику и дизайн как программу на языке Python. Это позволяет разработчикам оборудования использовать все богатство языка Python - объектно-ориентированное программирование, параметризованные функции, генераторы, перегрузку операторов, библиотеки и т. д. - для создания хорошо организованных, многоразовых и элегантных проектов.

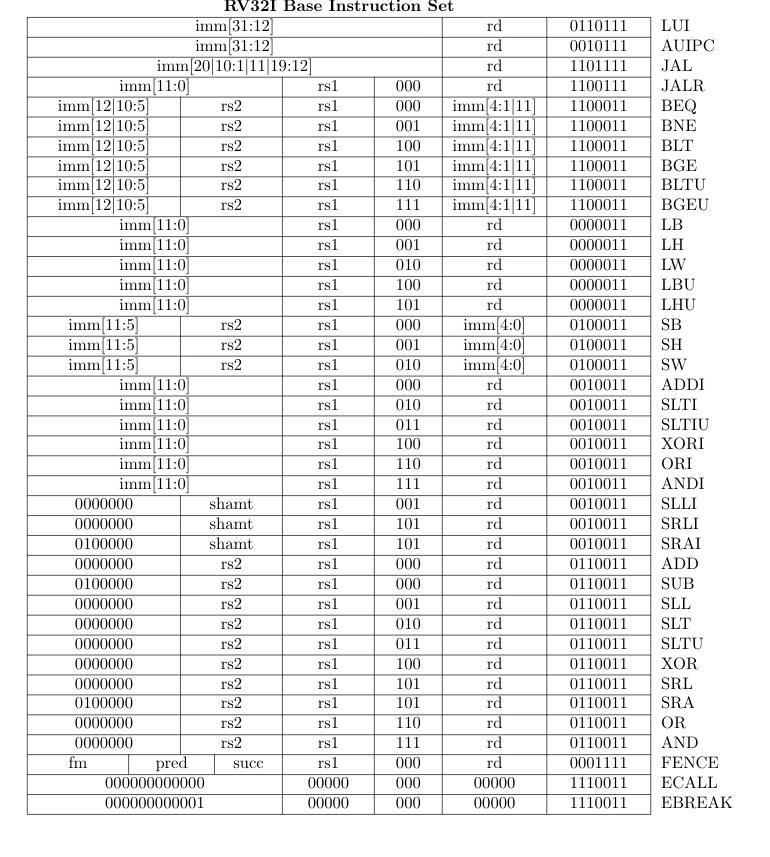
Другие библиотеки nMigen построены на FHDL и предоставляют различные инструменты и логические ядра. nMigen также содержит симулятор, который позволяет эмулировать работу программы на Python.

nMigen основан на Migen, похожем HDL(hardware description language) на основе Python. Хотя Migen очень хорошо работает в производственной среде, его дизайн может быть улучшен многими фундаментальными способами, и для этого nMigen заново реализует концепции Migen с нуля. nMigen также предоставляет обширный уровень совместимости, который позволяет создавать и моделировать большинство проектов Migen без изменений, а также интегрировать модули, написанные для Migen и nMigen.

RISC-V

ешё несколько слов о риске

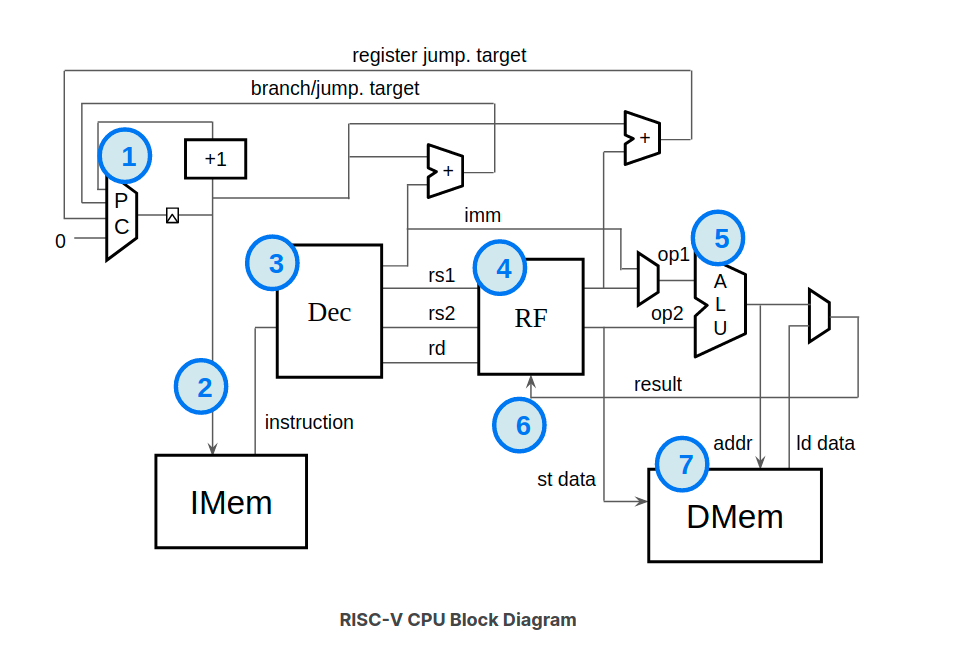
# Стратегия проектирования RISC-V



RISC-V взял всё, что мы знаем сегодня о современных процессорах, и использовал эти знания в проектировании процессоров ISA. Например, мы знаем, что:

* Сегодня у процессорных ядер есть сложная система прогнозирования ветвления.
* Процессорные ядра суперскалярны, то есть выполняют множество команд параллельно.
* Для обеспечения суперскалярности используется выполнение команд с изменением очерёдности (Out-of-Order execution).
* Они имеют конвейеры.

Это означает, что такие особенности, как поддерживаемое ARM условное выполнение, больше не требуется. Поддержка этой функции в ARM отъедает биты от формата команд. RISC-V может сэкономить эти биты.  
  
 Изначально условное выполнение создавалось для того, чтобы избегать ветвлений, потому что они плохо влияют на конвейеры. Для ускорения работы процессора он обычно заранее получает следующие команды, чтобы сразу после выполнения предыдущей на первой стадии процессора можно было подхватить следующую.  
  
 При условном ветвлении мы не можем заранее знать, где будет следующая команда, когда начинаем заполнять конвейер. Однако суперскалярный процессор может просто выполнять обе ветви параллельно.  
  
 Именно из-за этого RISC-V не имеет и регистров состояния, ведь они создают зависимости между командами. Чем более независима каждая команда, тем проще выполнять её параллельно с другой командой.  
  
 По сути, стратегия RISC-V заключается в том, что мы можем сделать ISA как можно более простым, а минимальную реализацию процессора RISC-V как можно более простой без необходимости принятия конструкторских решений, из-за которых невозможно будет создать высокопроизводительный процессор.



Структура RISC-V процессора

Процессор будет полностью выполнять одну инструкцию с каждым новым тактовым циклом.

Давайте посмотрим на компоненты нашего ЦП, следуя логике выполнения инструкций. Это также примерно порядок, в котором мы будем реализовывать логику.

1. Логика PC(program counter)

Эта логика отвечает за счетчик программ (PC). PC идентифицирует инструкцию, которую наш процессор выполнит следующей. Большинство инструкций выполняются последовательно, что означает, что по умолчанию PC выполняет приращение до следующей инструкции каждый такт. Однако инструкции условного и безусловного переходов не являются последовательными. Они определяют целевую инструкцию, которую нужно выполнить следующей, и логика PC должна соответствующим образом обновить PC.

2. Получение инструкции(Fetch)

Память инструкций (Imem, ROM) содержит инструкции для выполнения. Чтобы прочитать IMem , мы просто извлекаем инструкцию, на которую указывает PC.

3. Декодирование(Execute)

Теперь, когда у нас есть инструкция для выполнения, мы должны ее интерпретировать или декодировать. Мы должны разбить её на поля в зависимости от её типа. Эти поля сообщают нам, какие регистры читать, какую операцию выполнять и т.д.

4. Чтение из регистров

Регистровый файл - это небольшое локальное хранилище значений, с которыми программа активно работает. Мы декодировали инструкцию, чтобы определить, с какими регистрами нам нужно работать. Теперь нам нужно прочитать эти регистры из файла регистров.

5. Арифметико-логический блок (АЛУ) (Execute)

Теперь, когда у нас есть значения регистров, пора поработать с ними. Это работа ALU. Он будет складывать, вычитать, умножать, сдвигать и т. д. на основе операции, указанной в инструкции.

6. Запись в регистры (Write)

Теперь значение результата из ALU можно записать обратно в регистр назначения, указанный в инструкции.

7. Запись в память(Write)

Наша тестовая программа выполняется полностью вне регистрового файла и не требует памяти данных (Dmem, RAM). Но ни один процессор не обходится без него. DMem записывается инструкциями сохранения и считывается инструкциями загрузки.

При реализации процессора используется непривилегированная спецификация. Мы игнорируем всю логику, которая может потребоваться для взаимодействия с окружающей системой, такую ​​как контроллеры ввода / вывода (I / O), логику прерываний, и т. д.

Процессор общего назначения обычно имеет большую память, содержащую как инструкции, так и данные. При любой разумной тактовой частоте для доступа к памяти потребуется много тактовых циклов. Кеши будут использоваться для хранения недавно использованных данных памяти рядом с ядром процессора, однако в этой реализации они представленны не будут.

PC

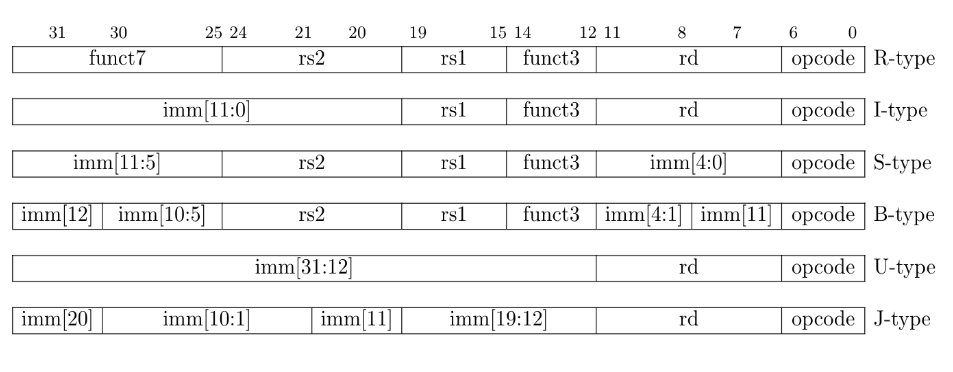
PC - это байтовый адрес, то есть он ссылается на первый байт инструкции в IMem. Инструкции имеют длину 4 байта, поэтому, хотя приращение PC обозначено как «+1» (инструкция), фактическое приращение должно быть на 4 (байта).

Получение инструкций осуществляется с адреса, указанного в reset\_address.

Обычно структура памяти, подобная нашему IMem, может быть реализована с использованием физической структуры, называемой статической памятью с произвольным доступом или SRAM. Адрес будет предоставлен в одном тактовом цикле, а данные будут считаны в следующем цикле. Однако весь наш процессор будет работать за один такт.

Decoder

Теперь, когда у нас есть инструкция, давайте разберемся, что это такое. RISC-V определяет различные типы инструкций, которые определяют структуру полей инструкции, в соответствии с этой таблицей из спецификаций RISC-V:

 Прежде чем мы сможем интерпретировать инструкцию, мы должны знать ее тип. Это определяется его кодом операции в instr [6: 0]. Фактически, instr [1: 0] должен быть 2'b11 для действительных инструкций RV32I. Мы будем считать, что все инструкции действительны, поэтому мы можем просто игнорировать эти два бита.

Теперь, в зависимости от типа инструкции, мы можем извлечь поля инструкции. Большинство полей всегда происходят из одних и тех же битов независимо от типа инструкции, но имеют значение только для определенных типов инструкций. Поле imm, «непосредственное» значение, встроенное в саму инструкцию, является исключением. Он состоит из разных битов в зависимости от типа инструкции.

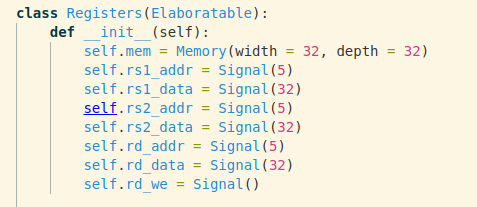
Начнем с более простых полей, которые не являются непосредственными: funct3, funct7, rs1, rs2, rd, opcode.

Непосредственное значение немного сложнее. Оно состоит из битов из разных полей в зависимости от типа.

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

Чтение из регистров

Pегистровый файл представляет собой довольно типичную структуру массива, поэтому мы можем использовать встроенный в nMigen модуль Memory:

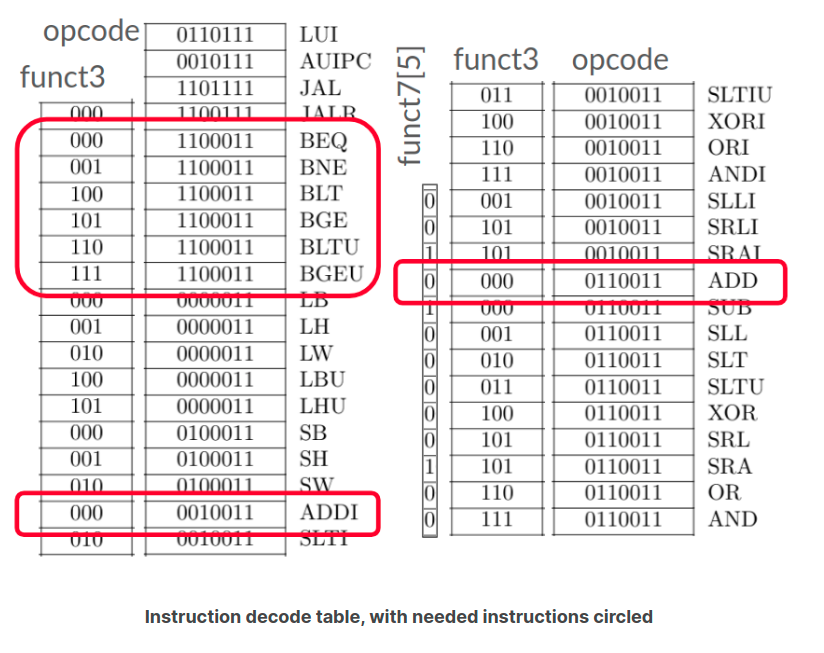


Это создаст экземпляр 32-разрядного регистрового файла с 32 записями, подключенного к заданным входным и выходным сигналам.

Логика декодирования инструкций предоставляет сигналы, необходимые для чтения файла регистров. Он определяет, в зависимости от типа инструкции, нужны ли исходные регистры. Он извлекает поля rs1 и rs2, которые предоставляют индексы для этих регистров, если они действительны.

АЛУ

Теперь, когда есть исходные значения, с которыми можно работать, выполняется блок АЛУ. Он вычисляет для каждой возможной инструкции результат, который она выдаст. Затем он выбирает, основываясь на фактических инструкциях, какой из этих результатов является правильным.

 Запись в регистры

Результат необходимо записать обратно в регистр назначения (rd) в файле регистров (если инструкция имеет регистр назначения).

В RISC-V x0 (в регистровом файле с индексом 0) всегда равен нулю. Один из способов реализовать это поведение - избегать записи x0.

Логика ветвления

Последний кусочек головоломки, необходимый для правильной работы эмулятора - это реализация инструкций ветвления.

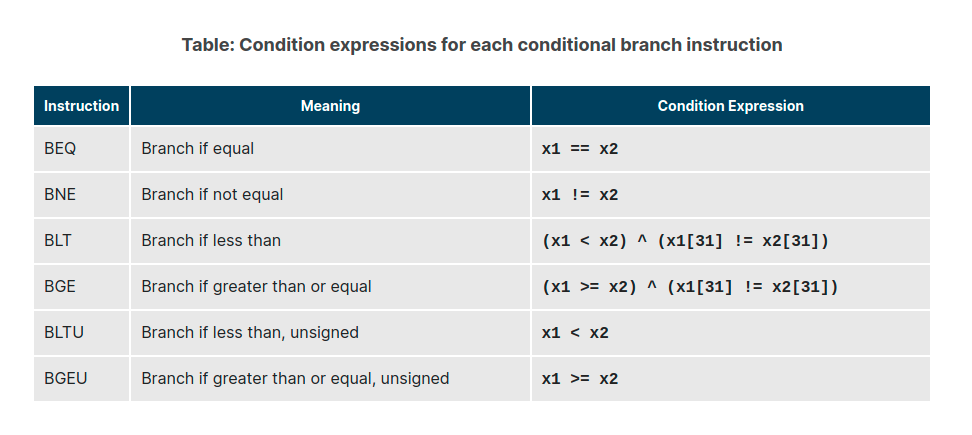
Команда условного перехода перейдет к целевому PC, если ее условие истинно. Условия представляют собой сравнение значений двух исходных регистров. Для реализации инструкций условного перехода потребуется:

▪ Определение того, является ли инструкция переходом, который выполняется .

▪ Вычисление цели перехода .

▪ Соответствующее обновление PC .

Начнем с условия ветвления . Каждая инструкция условного перехода имеет различное выражение условия, основанное на двух значениях исходного регистра (src1\_value и src2\_value, представленные как x1 и x2 ниже).



Подобно структуре ALU, вы определяете, нужно ли переходить в ветвь, выбирая соответствующий результат сравнения.

Нам также необходимо знать целевой PC инструкции перехода. Целевой PC указывается в поле сразу как относительное смещение в байтах от текущего PC. Итак, целевой PC - это PC ветви плюс его непосредственное значение.

Write to memory

#couple of screens form gtkwave

Проверка работоспособнсти полученной модели

Для проверки работоспособности полученной модели процессор исполнит ряд инструкций и выведет результаты их работы на экран:

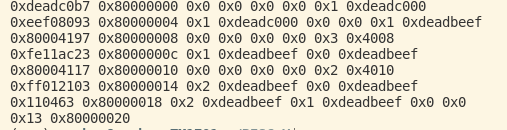
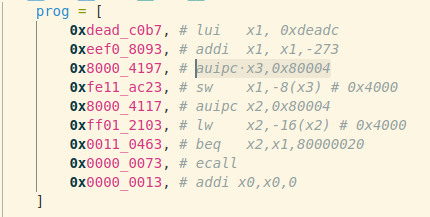
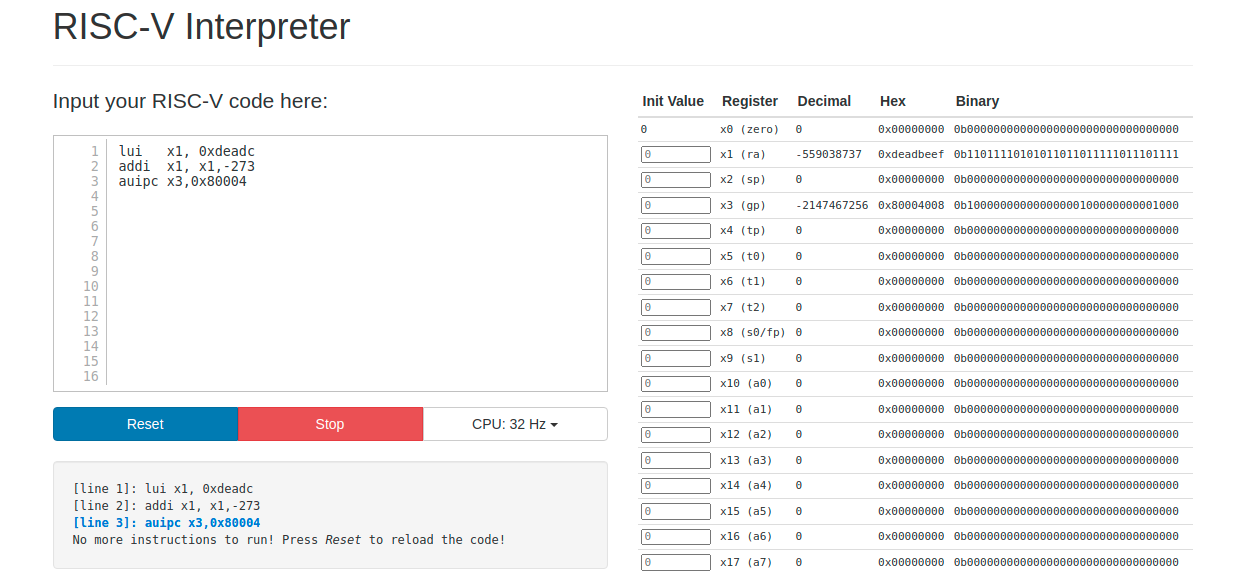
Набор этих простых команд позволяет проверить основные модули программы, в том числе декодирование, арифметические операции, операции условных и безусловных переходов, чтение/запись в память.

Результат симуляции:

Команды исполняются в правильной последовательности, что видно по их программному счётчику. Работоспособность отдельных команд можно проверить в онлайн симуляторе RISC-V:

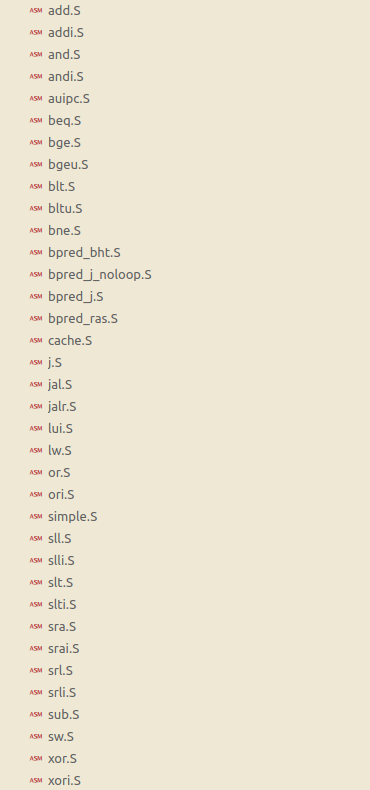
Проверка на совместимость

Для проверки полученное модели на соответствие спецификции RV32I можно использовать короткие ассемблерные тесты.

 Для сборки тестов используется технология CMake. Для генерации тестовой программы инструкции считываются из объектных файлов по 4 байта.

Корректность выполнения теста осуществлялась путём проверки регистра t3(x28), который в ходе исполнения теста устанавливается в 0.

В ходе тестирования все тесты дали положительный результат.



Заключение

В ходе курсовой работы был разработан простейший процессор RISC-V семейства RV32I используя фреймворк языка Python nMigen. Исходный код полученного симулятора загружен в репозиторий и распростроняется по лицензии GNU General Public License v3.0 и может быть расширен и модифицирован, например для реализации других спецификаций и расширений.