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

Противостояние между RISC и CISC, проходившее в конце 1990-х, уже давно прошло, 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.

Ниже приведён список некоторых из основных аспектов ISA RISC-V (instruction-set architecture) :

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

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

1. Сегодня у процессорных ядер есть сложная система прогнозирования ветвления.
2. Процессорные ядра суперскалярны, то есть выполняют множество команд параллельно.

3. Для обеспечения суперскалярности используется выполнение команд с изменением очерёдности (Out-of-Order execution).
4. 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.

Стратегия проектирования RISC-V

В этой работе будет реализовано непривелегированное RV32I подмножество RISC-V.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Структура RISC-V процессора

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

Давайте посмотрим на компоненты процессора, следуя логике выполнения инструкций.

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

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 АЛУ. Оно будет складывать, вычитать, умножать, сдвигать и т. д. на основе операции, указанной в инструкции.

6. Запись в регистры (Write)

Теперь значение результата из АЛУ 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 могут 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).

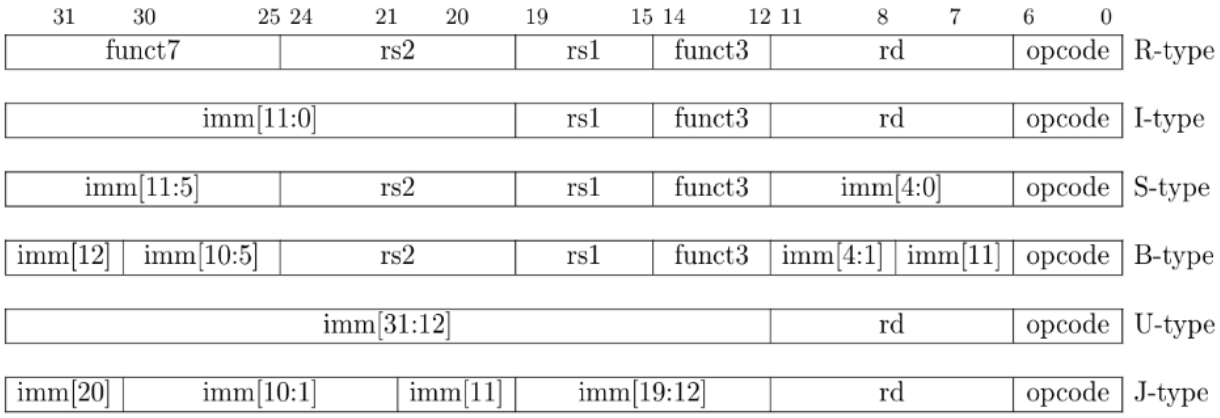
Получение инструкций осуществляется с адреса, указанного в reset_address.

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.

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 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

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

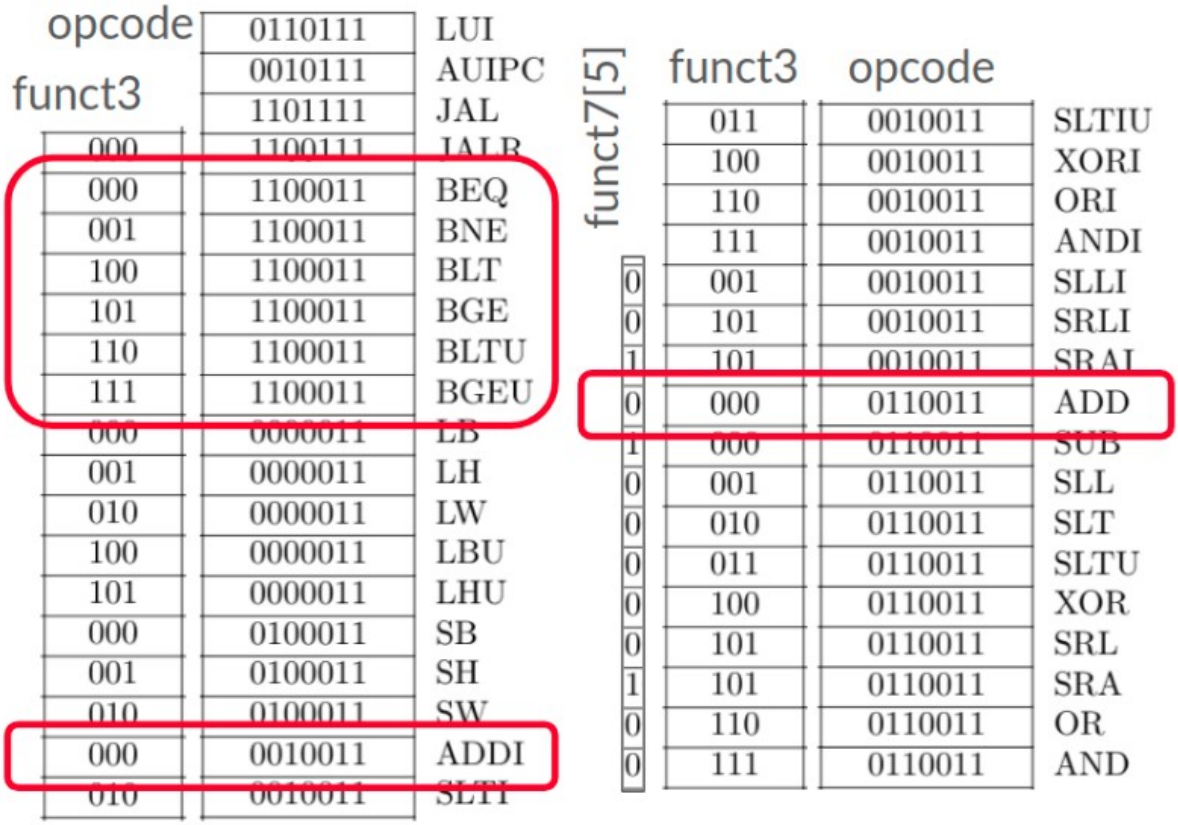
```
class Registers(Elaboratable):
    def __init__(self):
        self.mem = Memory(width = 32, depth = 32)
        self.rs1_addr = Signal(5)
        self.rs1_data = Signal(32)
        self.rs2_addr = Signal(5)
        self.rs2_data = Signal(32)
        self.rd_addr = Signal(5)
        self.rd_data = Signal(32)
        self.rd_we = Signal()
```

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.



opcode		
	0110111	LUI
	0010111	AUIPC
	1101111	JAL
funct3		
000	1100111	JALR
000	1100011	BEQ
001	1100011	BNE
100	1100011	BLT
101	1100011	BGE
110	1100011	BLTU
111	1100011	BGEU
000	0000011	LB
001	0000011	LH
010	0000011	LW
100	0000011	LBU
101	0000011	LHU
000	0100011	SB
001	0100011	SH
010	0100011	SW
000	0010011	ADDI
010	0010011	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

Instruction decode table, with needed instructions circled

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 .

Table: Condition expressions for each conditional branch instruction

Instruction	Meaning	Condition Expression
BEQ	Branch if equal	`x1 == x2`
BNE	Branch if not equal	`x1 != x2`
BLT	Branch if less than	`(x1 < x2) ^ (x1[31] != x2[31])`
BGE	Branch if greater than or equal	`(x1 >= x2) ^ (x1[31] != x2[31])`
BLTU	Branch if less than, unsigned	`x1 < x2`
BGEU	Branch if greater than or equal, unsigned	`x1 >= x2`

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 АЛУ, процессор определяет, 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.

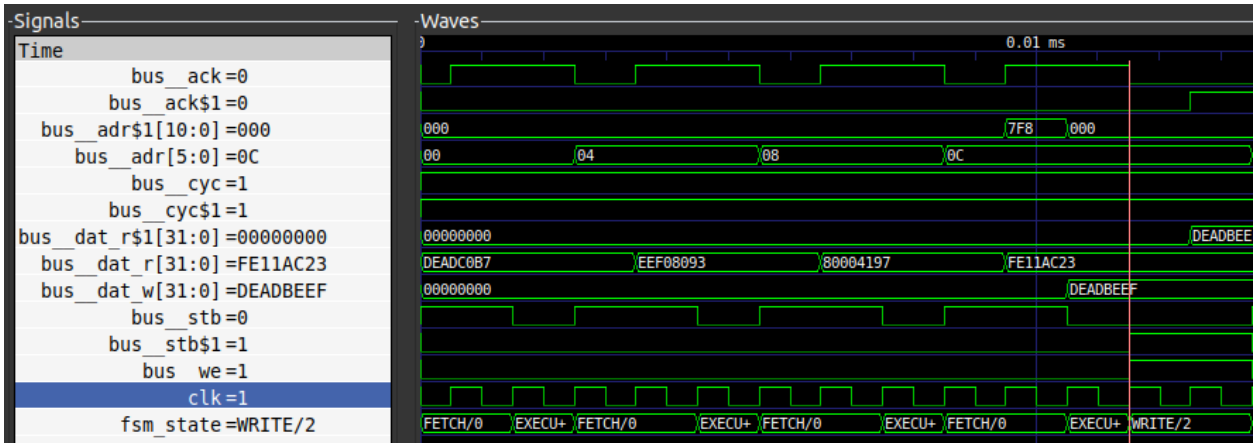
Запись в память

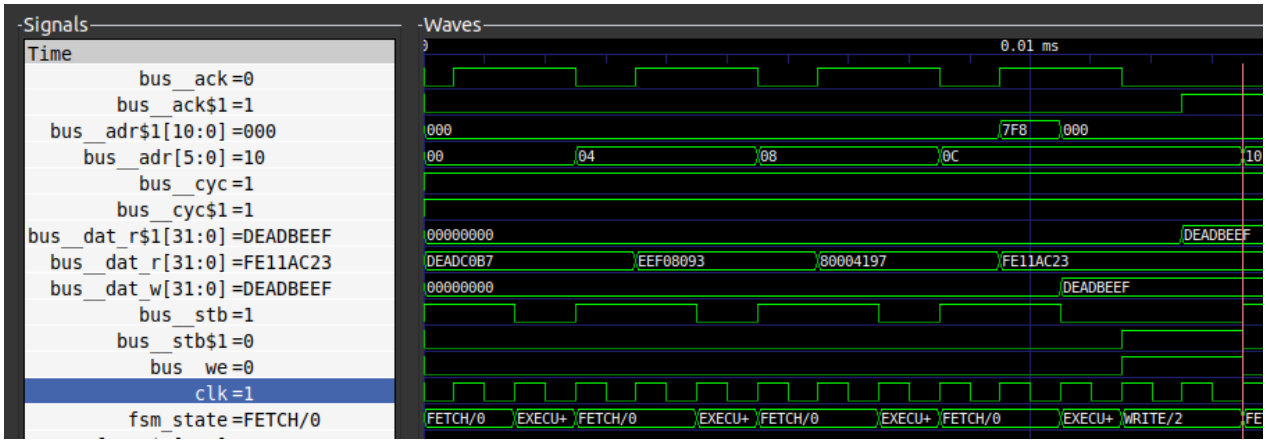
Процессор должен иметь возможность запрашивать новые операции чтения и ждать завершения модуля памяти, не зная заранее, сколько времени займет доступ к памяти. Это распространенная проблема, для которой уже разработаны стандарты, поэтому я собираюсь использовать бесплатный стандарт Wishbone Bus для передачи данных. nMigen включает реализацию такого типа шины в репозиторий nmigen-soc, поэтому его удобно и легко использовать в проекте. Добавив в модуль памяти атрибут

nmigen_soc.wishbone.Interface с желаемым адресом и шириной мы можем получить доступ к сигналам шины с помощью self.arb.bus:

- ack: сигнал «подтверждения», который выдается дочерним элементом по завершении транзакции.
- cyc: сигнал «цикла», который заявляется родителем, когда транзакция шины продолжается. Дочерний элемент должен игнорировать любые входные данные и избегать утверждения каких-либо выходных данных, когда «цикл» не утвержден.
- stb: «стробоскопический» сигнал, который выдается родителем, когда продолжается цикл передачи данных по шине. Сигнал «строб» можно переключать несколько раз, в то время как сигнал «цикла» используется для выполнения нескольких передач в одной транзакции.
- adr: адрес по которому данные будут записаны/считаны.
- dat_r: буфер «чтения данных», который дочерний элемент заполняет данными для чтения родителем после получения подтверждения.
- dat_w: буфер «записи данных», который дочерний элемент заполняет данными для записи родителем после получения подтверждения.
- we: сигнал, разрешающий запись данных в dat_w по указанному адресу.

Сигналы шины до записи и после:





Проверка работоспособности полученной модели

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

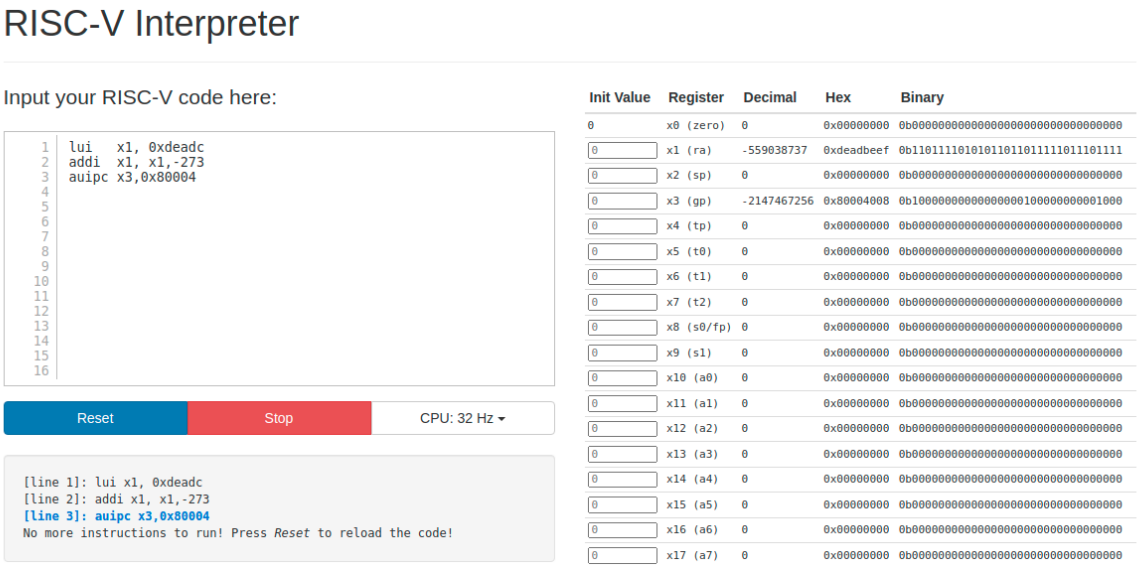
```
prog = [
    0xdead_c0b7, # lui   x1, 0xdeadc
    0xeef0_8093, # addi  x1, x1,-273
    0x8000_4197, # auipc x3,0x80004
    0xfe11_ac23, # sw    x1,-8(x3) # 0x4000
    0x8000_4117, # auipc x2,0x80004
    0xff01_2103, # lw    x2,-16(x2) # 0x4000
    0x0011_0463, # beq   x2,x1,80000020
    0x0000_0073, # ecall
    0x0000_0013, # addi x0,x0,0
]
```

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:

```
     Instr         PC   RS1 Addr   RS1 Data   RS2 Addr   RS2 Data    RD Addr    RD Data
0xdeadc0b7 0x80000000        0x0        0x0        0x0        0x0        0x1 0xdeadc000
0xeef08093 0x80000004        0x1 0xdeadc000        0x0        0x0        0x1 0xdeadbeef
0x80004197 0x80000008        0x0        0x0        0x0        0x0        0x3     0x4008
     Instr         PC   RS2 Addr   RS2 Data   Bus Addr Bus RData
0xfe11ac23 0x8000000c        0x1 0xdeadbeef        0x0 0xdeadbeef
     Instr         PC   RS1 Addr   RS1 Data   RS2 Addr   RS2 Data    RD Addr    RD Data
0x80004117 0x80000010        0x0        0x0        0x0        0x0        0x2     0x4010
0xff012103 0x80000014        0x2 0xdeadbeef        0x0 0xdeadbeef
     Instr         PC   RS2 Addr   RS2 Data   Bus Addr Bus RData
  0x110463 0x80000018        0x2 0xdeadbeef        0x1 0xdeadbeef        0x0        0x0
      0x13 0x80000020
```

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



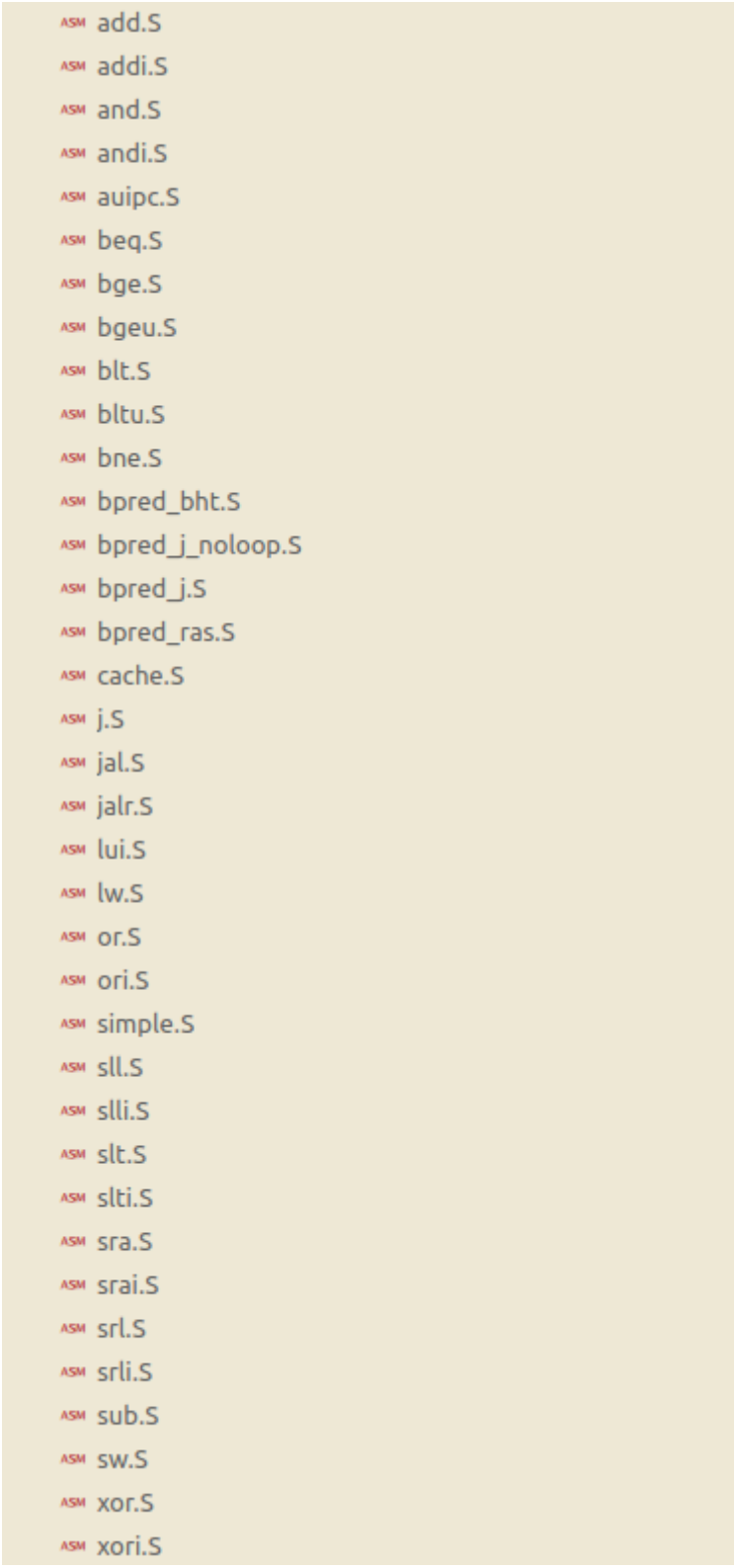
Проверка на совместимость

Для проверки полученной модели на соответствие спецификации 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, если тест завершается .

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 распространяется по лицензии GNU General Public License v3.0 и может быть расширен и модифицирован, например, для реализации других спецификаций и расширений.

ИСТОЧНИКИ:

- https://github.com/RobertBaruch/nmigen-tutorial/
- https://vivonomicon.com/2020/04/14/learning-fpga-design-with-nmigen/
- https://vivonomicon.com/2020/06/13/lets-write-a-minimal-risc-v-cpu-in-nmigen/
- https://ru.wikipedia.org/wiki/Wishbone

ПРИЛОЖЕНИЕ

Листинги кода

alu.py:

```
from nmigen import *
from nmigen.sim import *


class AluFunc:
ADD = 0b0000
SUB = 0b0001
SLL = 0b0010
SLT = 0b0100
SLTU = 0b0110
XOR = 0b1000
SRL = 0b1010
SRA = 0b1011
OR = 0b1100
AND = 0b1110


class ALU(Elaboratable):

def __init__(self):
self.rs1_val = Signal(32, reset=0x0000000)
self.rs2_val = Signal(32, reset=0x0000000)
self.funct = Signal(4, reset=0b0000)
self.rd_val = Signal(32, reset=0x00000000)
self.rd_has_val = Signal(1, reset=1)

def elaborate(self, platform):
m = Module()

with m.Switch(self.funct[:4]):
with m.Case(AluFunc.ADD):
m.d.comb += self.rd_val.eq(self.rs1_val.as_signed() + self.rs2_val.as_signed())
with m.Case(AluFunc.SUB):
m.d.comb += self.rd_val.eq(self.rs1_val.as_signed() - self.rs2_val.as_signed())
```

```
with m.Case(AluFunc.AND):
m.d.comb += self.rd_val.eq(self.rs1_val & self.rs2_val)
with m.Case(AluFunc.OR):
m.d.comb += self.rd_val.eq(self.rs1_val | self.rs2_val)
with m.Case(AluFunc.XOR):
m.d.comb += self.rd_val.eq(self.rs1_val ^ self.rs2_val)
with m.Case(AluFunc.SLT):
m.d.comb += self.rd_val.eq(self.rs1_val.as_signed() < self.rs2_val.as_signed())
with m.Case(AluFunc.SLTU):
m.d.comb += self.rd_val.eq(self.rs1_val < self.rs2_val)
with m.Case(AluFunc.SLL):
m.d.comb += self.rd_val.eq(self.rs1_val << self.rs2_val[:5])
with m.Case(AluFunc.SRL):
m.d.comb += self.rd_val.eq(self.rs1_val >> self.rs2_val[:5])
with m.Case(AluFunc.SRA):
m.d.comb += self.rd_val.eq(self.rs1_val.as_signed() >> self.rs2_val[:5])
with m.Default():
m.d.comb += self.rd_has_val.eq(0)

return m
```

branch.py:

```
from nmigen import *

from nmigen.sim import *


class BRANCH:
BEQ = 0b000
BNE = 0b001
BLT = 0b100
BGE = 0b101
BLTU = 0b110
BGEU = 0b111


class Branch(Elaboratable):
def __init__(self):
self.funct = Signal(3)
self.src1 = Signal(signed(32))
self.src2 = Signal(signed(32))
self.res = Signal(1)
self.res_has_val = Signal(1, reset=1)

def elaborate(self, platform):
```

```
m = Module()

with m.Switch(self.funct):
with m.Case(BRANCH.BEQ):
m.d.comb += self.res.eq(self.src1 == self.src2)
with m.Case(BRANCH.BNE):
m.d.comb += self.res.eq(self.src1 != self.src2)
with m.Case(BRANCH.BGE):
m.d.comb += self.res.eq(self.src1 >= self.src2)
with m.Case(BRANCH.BLT):
m.d.comb += self.res.eq(self.src1 < self.src2)
with m.Case(BRANCH.BGEU):
m.d.comb += self.res.eq(self.src1.as_unsigned() >= self.src2.as_unsigned())
with m.Case(BRANCH.BLTU):
m.d.comb += self.res.eq(self.src1.as_unsigned() < self.src2.as_unsigned())
with m.Default():
m.d.comb += self.res_has_val.eq(0)

return m
```

decoder.py:

```
from nmigen import *

from nmigen.sim import *


class Opcode:
LUI = 0b0110111
AUIPC = 0b0010111
JAL = 0b1101111
JALR = 0b1100111
BRANCH = 0b1100011
LOAD = 0b0000011
STORE = 0b0100011
IMM = 0b0010011
REG = 0b0110011

class IType:
ALU = 0b000
BR = 0b001
J = 0b010
JR = 0b011
LD = 0b100
ST = 0b101

class Decoder(Elaboratable):
```

```
def __init__(self):
self.inst = Signal(32)

self.rs1 = Signal(5)
self.rs1_en = Signal()

self.rs2 = Signal(5)
self.rs2_en = Signal()

self.rd = Signal(5)
self.rd_en = Signal()

self.itype = Signal(3)
self.mem_op_en = Signal()
self.mem_op_store = Signal()
self.funct3 = Signal(3)
self.funct1 = Signal()

self.imm = Signal(32)

def elaborate(self, platform):
m = Module()
inst = self.inst
opcode = inst[:7]
rd = inst[7:12]
rs1 = inst[15:20]
rs2 = inst[20:25]
funct3 = inst[12:15]
funct7 = inst[25:32]

imm_i = Signal(32)
imm_s = Signal(32)
imm_b = Signal(32)
imm_u = Signal(32)
imm_j = Signal(32)
m.d.comb += [
imm_i.eq(Cat(self.inst[20], self.inst[21:25], self.inst[25:31], Repl(self.inst[31], 21))),
imm_s.eq(Cat(self.inst[7], self.inst[8:12], self.inst[25:31], Repl(self.inst[31], 21))),
imm_b.eq(Cat(0, self.inst[8:12], self.inst[25:31], self.inst[7], Repl(self.inst[31], 20))),
imm_u.eq(Cat(Repl(0, 12), self.inst[12:20], self.inst[20:31], self.inst[31])),
imm_j.eq(Cat(0, self.inst[21:25], self.inst[25:31], self.inst[20], self.inst[12:20],
Repl(self.inst[31], 12))),
]

funct1 = Signal()
funct1_valid = Signal()
m.d.comb += [
```

```
self.itype.eq(IType.ALU),
self.rs1.eq(Mux(self.rs1_en, rs1, 0)),
self.rs2.eq(Mux(self.rs2_en, rs2, 0)),
self.rd.eq(Mux(self.rd_en, rd, 0)),
self.funct3.eq(funct3),
]

with m.Switch(funct7):
with m.Case('0100000'):
m.d.comb += [
funct1.eq(1),
funct1_valid.eq(1),
]
with m.Case('0000000'):
m.d.comb += [
funct1.eq(0),
funct1_valid.eq(1),
]
with m.Default():
m.d.comb += [
funct1.eq(0),
funct1_valid.eq(0),
]

with m.Switch(opcode):
with m.Case(Opcode.LUI):
m.d.comb += [
self.rs1_en.eq(1),
self.rs1.eq(0),
self.rs2_en.eq(0),
self.rd_en.eq(1),
self.funct3.eq(0),
self.imm.eq(imm_u),
]
with m.Case(Opcode.AUIPC):
m.d.comb += [
self.rs1_en.eq(0),
self.rs2_en.eq(0),
self.rd_en.eq(1),
self.funct3.eq(0),
self.imm.eq(imm_u),
]
with m.Case(Opcode.JAL):
m.d.comb += [
self.rs1_en.eq(0),
self.rs2_en.eq(0),
self.rd_en.eq(1),
```

```
self.itype.eq(IType.J),
self.funct3.eq(0),
self.imm.eq(imm_j),
]
with m.Case(Opcode.JALR):
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(0),
self.rd_en.eq(1),
self.itype.eq(IType.JR),
self.funct3.eq(0),
self.imm.eq(imm_i),
]
with m.Case(Opcode.BRANCH):
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(1),
self.rd_en.eq(0),
self.itype.eq(IType.BR),
self.imm.eq(imm_b),
]
with m.Case(Opcode.LOAD):
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(0),
self.rd_en.eq(1),
self.imm.eq(imm_i),
self.mem_op_en.eq(1),
self.itype.eq(IType.LD)
]
with m.Case(Opcode.STORE):
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(1),
self.rd_en.eq(0),
self.imm.eq(imm_s),
self.mem_op_en.eq(1),
self.mem_op_store.eq(1),
self.itype.eq(IType.ST)
]
with m.Case(Opcode.IMM):
with m.Switch(Cat(funct1_valid, funct3)):
with m.Case('-011'):
m.d.comb += [
self.imm.eq(rs2),
self.funct1.eq(funct1),
]
```

```
with m.Default():
m.d.comb += self.imm.eq(imm_i)
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(0),
self.rd_en.eq(1),
]
with m.Case(Opcode.REG):
m.d.comb += [
self.rs1_en.eq(1),
self.rs2_en.eq(1),
self.rd_en.eq(1),
self.imm.eq(0),
self.funct1.eq(funct1),
]

return m
```

memory.py:

```
from math import ceil, log2

from nmigen import *
from nmigen.sim import *
from nmigen_soc.wishbone.bus import Arbiter, Interface, MemoryMap


class MemoryUnit(Elaboratable):
def __init__(self, size_words, data=[]):
self.size = size_words * 4
self.mem = Memory(
width=32,
depth=size_words,
init=data
)
self.read_port = self.mem.read_port()
self.write_port = self.mem.write_port()
self.arb = Arbiter(addr_width=ceil(log2(self.size + 1)),
data_width=32)
self.arb.bus.memory_map = MemoryMap(
addr_width = self.arb.bus.addr_width,
data_width = self.arb.bus.data_width,
alignment = 0 )

def new_bus( self ):
bus = Interface( addr_width = self.arb.bus.addr_width,
data_width = self.arb.bus.data_width )
```

```
bus.memory_map = MemoryMap( addr_width = bus.addr_width,
data_width = bus.data_width,
alignment = 0 )
self.arb.add( bus )
return bus

def elaborate(self, platform):
m = Module()

m.submodules.read_port = self.read_port
m.submodules.write_port = self.write_port
m.submodules.arb = self.arb

m.d.sync += self.arb.bus.ack.eq(0)
with m.If(self.arb.bus.cyc):
m.d.sync += self.arb.bus.ack.eq(self.arb.bus.stb)
m.d.comb += [
self.read_port.addr.eq(self.arb.bus.adr >> 2),
self.write_port.addr.eq(self.arb.bus.adr >> 2),
self.arb.bus.dat_r.eq(self.read_port.data),
self.write_port.data.eq(self.arb.bus.dat_w),
self.write_port.en.eq(self.arb.bus.we),
]

return m
```

registers.py:

```
from nmigen import *

from nmigen.sim import *

class Registers(Elaboratable):
def __init__(self):
self.mem = Memory(width = 32, depth = 32)
self.rs1_addr = Signal(5)
self.rs1_data = Signal(32)
self.rs2_addr = Signal(5)
self.rs2_data = Signal(32)
self.rd_addr = Signal(5)
self.rd_data = Signal(32)
self.rd_we = Signal()

def elaborate(self, platform):
m = Module()
```

```
rs1 = m.submodules.rs1 = self.mem.read_port()
rs2 = m.submodules.rs2 = self.mem.read_port()
rd = m.submodules.rd = self.mem.write_port()

m.d.comb += [
rs1.addr.eq(self.rs1_addr),
self.rs1_data.eq(rs1.data),
rs2.addr.eq(self.rs2_addr),
self.rs2_data.eq(rs2.data),
]

with m.If(self.rd_addr != 0):
m.d.comb += [
rd.addr.eq(self.rd_addr),
rd.data.eq(self.rd_data),
rd.en.eq(self.rd_we),
]

return m
```

cpu.py:

```
from nmigen import *

from nmigen.hdl.rec import *
from nmigen.sim import *
from nmigen.back import rtlil
from alu import ALU
from branch import Branch
from decoder import Decoder, IType
from registers import Registers
from memory import MemoryUnit
# from core.alu import ALU
# from core.branch import Branch
# from core.decoder import Decoder, IType
# from core.registers import Registers
# from core.memory import MemoryUnit


class CPU(Elaboratable):
def __init__(self, reset_address=0x0000_0000, data=[]):
self.reset_address = reset_address

self.ram = MemoryUnit(256)
self.rom = MemoryUnit(len(data), data=data)
self.ibus = self.rom.new_bus()
```

```
self.dbus = self.ram.new_bus()
self.pc = Signal(32, reset=reset_address)
self.instruction = Signal(32)
self.decoder = Decoder()
self.regs = Registers()
self.alu = ALU()
self.branch = Branch()
self.valid = Signal(1, reset=0)

def elaborate(self, platform):
m = Module()

m.submodules.decoder = decoder = self.decoder
m.submodules.regs = regs = self.regs
m.submodules.alu = alu = self.alu
m.submodules.branch = branch = self.branch
m.submodules.ram = self.ram
m.submodules.rom = self.rom

pc_next = Signal(32)
pc_next_temp = Signal(32)
pc_4 = Signal(32)
inst = self.instruction
m.d.comb += pc_4.eq(self.pc + 4)
rs1_en = Signal()
rs2_en = Signal()

valid = self.valid

m.d.comb += [
self.ibus.adr.eq(self.pc),
self.ibus.cyc.eq(1)
]

m.d.comb += decoder.inst.eq(self.ibus.dat_r),

m.d.comb += [
regs.rs1_addr.eq(decoder.rs1),
regs.rs2_addr.eq(decoder.rs2),
regs.rd_addr.eq(decoder.rd),
]

m.d.comb += [
alu.rs1_val.eq(Mux(rs1_en, regs.rs1_data, self.pc)),
alu.rs2_val.eq(Mux(rs2_en, regs.rs2_data, decoder.imm)),
]
```

```
m.d.comb += [
branch.funct.eq(decoder.funct3),
branch.src1.eq(regs.rs1_data),
branch.src2.eq(regs.rs2_data),
]

m.d.comb += [
self.dbus.adr.eq(Mux(decoder.mem_op_en, alu.rd_val, 0)),
self.dbus.dat_w.eq(Mux(decoder.mem_op_store, regs.rs2_data, self.dbus.dat_r)),
self.dbus.cyc.eq(1),
pc_next.eq(pc_next_temp),
]

with m.Switch(decoder.itype):
with m.Case(IType.ALU):
m.d.comb += [
alu.funct.eq(Cat(decoder.funct1, decoder.funct3)),
rs1_en.eq(decoder.rs1_en),
rs2_en.eq(decoder.rs2_en),
pc_next_temp.eq(pc_4),
regs.rd_data.eq(alu.rd_val),
]
with m.If(decoder.mem_op_en):
m.d.comb += [
alu.funct.eq(0),
rs2_en.eq(0),
]
with m.Case(IType.J):
m.d.comb += [
rs1_en.eq(0),
rs2_en.eq(0),
pc_next_temp.eq(alu.rd_val),
regs.rd_data.eq(pc_4),
]
with m.Case(IType.JR):
m.d.comb += [
rs1_en.eq(1),
rs2_en.eq(0),
pc_next_temp.eq(alu.rd_val),
regs.rd_data.eq(pc_4),
]
with m.Case(IType.BR):
m.d.comb += [
alu.funct.eq(0),
rs1_en.eq(0),
rs2_en.eq(0),
pc_next_temp.eq(Mux(branch.res, alu.rd_val, pc_4)),
```

```
]
with m.Case(IType.ST):
m.d.comb += [
alu.funct.eq(0),
rs1_en.eq(1),
rs2_en.eq(0),
pc_next_temp.eq(pc_4)
]
with m.Case(IType.LD):
m.d.comb += [
alu.funct.eq(0),
rs1_en.eq(1),
rs2_en.eq(0),
pc_next_temp.eq(pc_4),
]

with m.FSM():
with m.State('FETCH'):
m.d.comb += self.ibus.stb.eq(1)
with m.If(self.ibus.ack):
m.next = 'EXECUTE'
m.d.sync += inst.eq(self.ibus.dat_r)
m.d.comb += decoder.inst.eq(self.ibus.dat_r)
with m.State('EXECUTE'):
m.d.comb += decoder.inst.eq(inst)
with m.If(decoder.mem_op_en):
m.next = 'WRITE'
with m.Else():
m.next = 'FETCH'
m.d.comb += [
regs.rd_we.eq(1),
valid.eq(1),
]
m.d.sync += self.pc.eq(pc_next)
with m.State('WRITE'):
m.d.comb += [
decoder.inst.eq(inst),
self.dbus.we.eq(decoder.mem_op_store),
self.dbus.stb.eq(1),
]
with m.If(self.dbus.ack):
m.next = 'FETCH'
m.d.comb += [
valid.eq(1),
regs.rd_we.eq(~decoder.mem_op_store),
regs.rd_data.eq(self.dbus.dat_r),
]
```

```
m.d.sync += self.pc.eq(pc_next)

return m


if __name__ == '__main__':
prog = [
0xdead_c0b7, # lui x1, 0xdeadc
0xeef0_8093, # addi x1, x1,-273
0x8000_4197, # auipc x3,0x80004
0xfe11_ac23, # sw x1,-8(x3) # 0x4000
0x8000_4117, # auipc x2,0x80004
0xff01_2103, # lw x2,-16(x2) # 0x4000
0x0011_0463, # beq x2,x1,80000020
0x0000_0073, # ecall
0x0000_0013, # addi x0,x0,0
]

cpu = CPU(reset_address=0x8000_0000, data=prog)
sim = Simulator(cpu)
def step():
clock = 0
yield Tick()
yield Settle()
while (yield cpu.valid) != 1 and clock < 5:
clock += 1
yield Tick()
yield Settle()
assert(clock < 8)

def proc():
print('Instr'.rjust(10), 'PC'.rjust(10), 'RS1 Addr'.rjust(10),
'RS1 Data'.rjust(10), 'RS2 Addr'.rjust(10), 'RS2 Data'.rjust(10), 'RD Addr'.rjust(10), 'RD
Data'.rjust(10))
yield from step()
instr = yield cpu.instruction
pc = yield cpu.pc
rs1_addr = yield cpu.regs.rs1_addr
rs1_data = yield cpu.regs.rs1_data
rs2_addr = yield cpu.regs.rs2_addr
rs2_data = yield cpu.regs.rs2_data
rd_addr = yield cpu.regs.rd_addr
rd_data = yield cpu.regs.rd_data
print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs1_addr).rjust(10),
hex(rs1_data).rjust(10), hex(rs2_addr).rjust(10), hex(rs2_data).rjust(10),
hex(rd_addr).rjust(10), hex(rd_data).rjust(10),)
```

```
yield from step()
instr = yield cpu.instruction
pc = yield cpu.pc
rs1_addr = yield cpu.regs.rs1_addr
rs1_data = yield cpu.regs.rs1_data
rs2_addr = yield cpu.regs.rs2_addr
rs2_data = yield cpu.regs.rs2_data
rd_addr = yield cpu.regs.rd_addr
rd_data = yield cpu.regs.rd_data
print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs1_addr).rjust(10),
hex(rs1_data).rjust(10), hex(rs2_addr).rjust(10), hex(rs2_data).rjust(10),
hex(rd_addr).rjust(10), hex(rd_data).rjust(10),)

yield from step()
instr = yield cpu.instruction
pc = yield cpu.pc
rs1_addr = yield cpu.regs.rs1_addr
rs1_data = yield cpu.regs.rs1_data
rs2_addr = yield cpu.regs.rs2_addr
rs2_data = yield cpu.regs.rs2_data
rd_addr = yield cpu.regs.rd_addr
rd_data = yield cpu.regs.rd_data
print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs1_addr).rjust(10),
hex(rs1_data).rjust(10), hex(rs2_addr).rjust(10), hex(rs2_data).rjust(10),
hex(rd_addr).rjust(10), hex(rd_data).rjust(10),)

print('Instr'.rjust(10), 'PC'.rjust(10), 'RS2 Addr'.rjust(10), 'RS2 Data'.rjust(10), 'Bus
Addr'.rjust(10), 'Bus RData')
yield from step()
instr = yield cpu.instruction
pc = yield cpu.pc
rs2_addr = yield cpu.regs.rs2_addr
rs2_data = yield cpu.regs.rs2_data
addr = yield cpu.dbus.adr
data = yield cpu.dbus.dat_w
print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs2_addr).rjust(10),
hex(rs2_data).rjust(10),hex(addr).rjust(10), hex(data).rjust(10),)

print('Instr'.rjust(10), 'PC'.rjust(10), 'RS1 Addr'.rjust(10),
'RS1 Data'.rjust(10), 'RS2 Addr'.rjust(10), 'RS2 Data'.rjust(10), 'RD Addr'.rjust(10), 'RD
Data'.rjust(10))
yield from step()
instr = yield cpu.instruction
pc = yield cpu.pc
rs1_addr = yield cpu.regs.rs1_addr
rs1_data = yield cpu.regs.rs1_data
rs2_addr = yield cpu.regs.rs2_addr
```

```
    rs2_data = yield cpu.regs.rs2_data
    rd_addr = yield cpu.regs.rd_addr
    rd_data = yield cpu.regs.rd_data
    print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs1_addr).rjust(10),
    hex(rs1_data).rjust(10), hex(rs2_addr).rjust(10), hex(rs2_data).rjust(10),
    hex(rd_addr).rjust(10), hex(rd_data).rjust(10),)

    yield from step()
    instr = yield cpu.instruction
    pc = yield cpu.pc
    rd_addr = yield cpu.regs.rd_addr
    rd_data = yield cpu.regs.rd_data
    addr = yield cpu.dbus.adr
    data = yield cpu.dbus.dat_r
    print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rd_addr).rjust(10),
    hex(rd_data).rjust(10),hex(addr).rjust(10), hex(data).rjust(10),)

    print('Instr'.rjust(10), 'PC'.rjust(10), 'RS2 Addr'.rjust(10), 'RS2 Data'.rjust(10), 'Bus
    Addr'.rjust(10), 'Bus RData')
    yield from step()
    instr = yield cpu.instruction
    pc = yield cpu.pc
    rs1_addr = yield cpu.regs.rs1_addr
    rs1_data = yield cpu.regs.rs1_data
    rs2_addr = yield cpu.regs.rs2_addr
    rs2_data = yield cpu.regs.rs2_data
    rd_addr = yield cpu.regs.rd_addr
    rd_data = yield cpu.regs.rd_data
    print(hex(instr).rjust(10), hex(pc).rjust(10), hex(rs1_addr).rjust(10),
    hex(rs1_data).rjust(10), hex(rs2_addr).rjust(10), hex(rs2_data).rjust(10),
    hex(rd_addr).rjust(10), hex(rd_data).rjust(10),)

    yield from step()
    instr = yield cpu.instruction
    pc = yield cpu.pc
    print(hex(instr).rjust(10), hex(pc).rjust(10))

    yield Tick()
    yield Tick()
    yield Tick()
    yield Tick()
    yield Settle()

    sim.add_clock(1e-6, domain='sync')
    sim.add_sync_process(proc)
    with sim.write_vcd("cpu.vcd", "cpu.gtkw"):
    sim.run()
```