ЛАБОРАТОРНАЯ РАБОТА

Архитектура набора команд на примере RISC-V

Никита Поляков

17.09.2020
ChipEXPO – 2020

ВВЕДЕНИЕ

Данный материал подготовлен для мероприятия “Сколковская школа синтеза цифровых схем на Verilog” в рамках выставки «ChipEXPO – 2020».

МАТЕРИАЛЫ

1. Описание языка ассемблера RISC-V https://github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md
2. Описание системы команд RISC-V “ISA Specification Volume 1, Unprivileged Spec v. 20191213” https://github.com/riscv/riscv-isa-manual/releases/download/draft-20200727-8088ba4/riscv-spec.pdf

ПОДГОТОВКА К РАБОТЕ

Работа будет выполняться на компьютере с операционной системой Windows (XP и новее) или Linux Ubuntu. Выполнение на компьютерах с другими операционными системами также возможно, но не проверялось (при желании можете выполнить проверку самостоятельно).

Основным инструментом будет симулятор архитектуры RISC-V. Для его запуска понадобится поддержка Java (инструкции по установке см. ниже).

1. Установка Java

1. для Ubuntu: https://www.digitalocean.com/community/tutorials/java-ubuntu-apt-get-ru
2. для Windows: https://www.java.com/ru/download/help/download_options.xml#windows

2. Запуск эмулятора

1. Скачиваем репозиторий https://github.com/DigitalDesignSchool/ce2020labs либо через браузер, либо через Git утилиту.
    a. через браузер:
        i. переходим по адресу https://github.com/DigitalDesignSchool/ce2020labs
        ii. нажимаем на зеленую кнопочку Code
        iii. выбираем Download ZIP

iv. распаковываем архив в папку, в которой будем работать.

b. через командную строку Ubuntu:

i. установка Git, если его нет

sudo apt update

sudo apt install git

ii. в командной строке выполняем

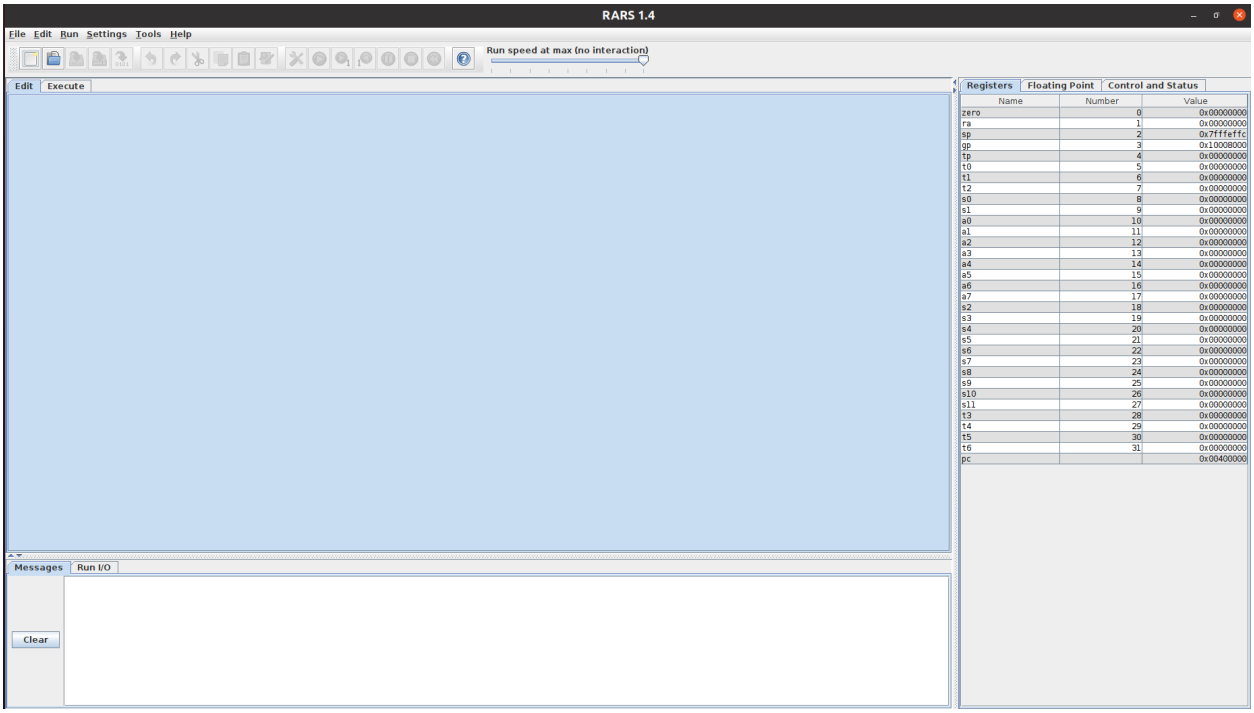
git clone https://github.com/DigitalDesignSchool/ce2020labs

создастся папка ce2020labs

2. Заходим в папку ce2020labs/day_3/arch/risc_v_lab
3. Сам запуск
   a. в Linux в командной строке выполняем

./rars.sh

   b. в Windows запускаем двумя щелчками мыши скрипт rars.bat



ПЛАН РАБОТЫ

1. Архитектура набора команд
2. Работа симулятора
3. Простейшие команды (ADD, ADDI, MV)
4. Условные операции и циклы
5. Работа с памятью

АРХИТЕКТУРА НАБОРА КОМАНД

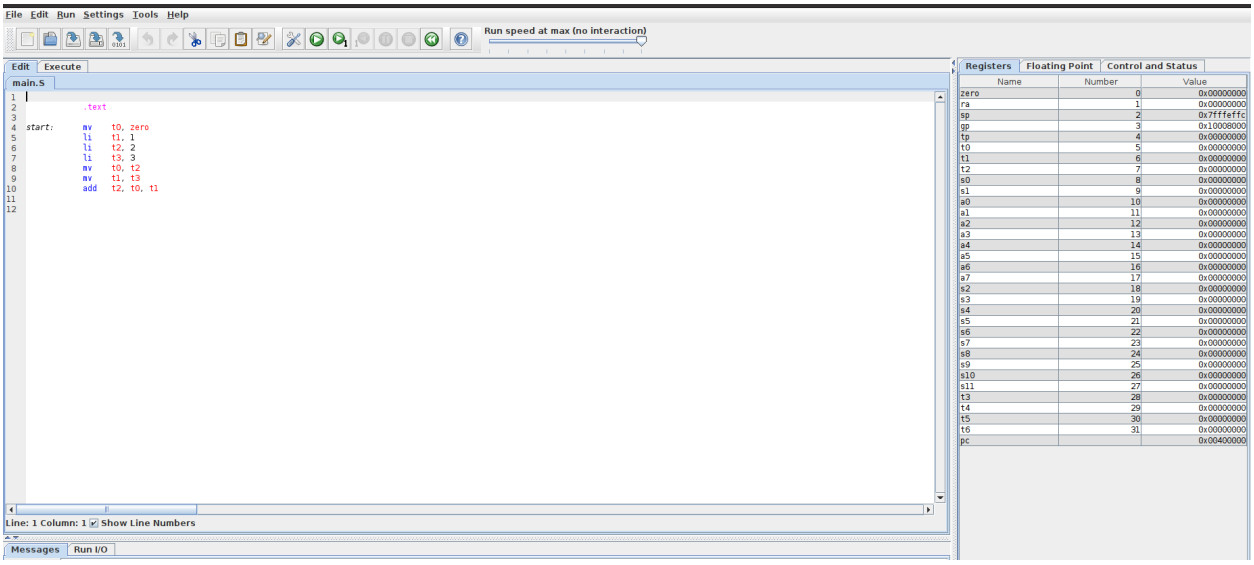
Архитектура набора команд (Instruction Set Architecture, ISA) описывает доступные команды, типы данных, модель памяти, модель ввода-вывода, а также, быть может, другие свойства абстрактной модели компьютера (вычислительного средства). Команды определяются с точностью до двоичной кодировки. Все свойства компьютера, не определенные в ISA могут быть выбраны производителем на его усмотрение (и умение) и называются ***микроархитектурой*** или реализацией (implementation) системы команд. За счет того, что двоичные кодировки команд одинаковы для разных реализаций одной и той же ISA, программа, написанная и скомпилированная (переведенная в двоичный код) для некоторой ISA, может быть исполнена на любой реализации этой ISA.

РАБОТА СИМУЛЯТОРА

Симулятор позволяет выполнить трансляцию программы, написанной на языке ассемблера, в машинный код и эмулировать исполнение программы.

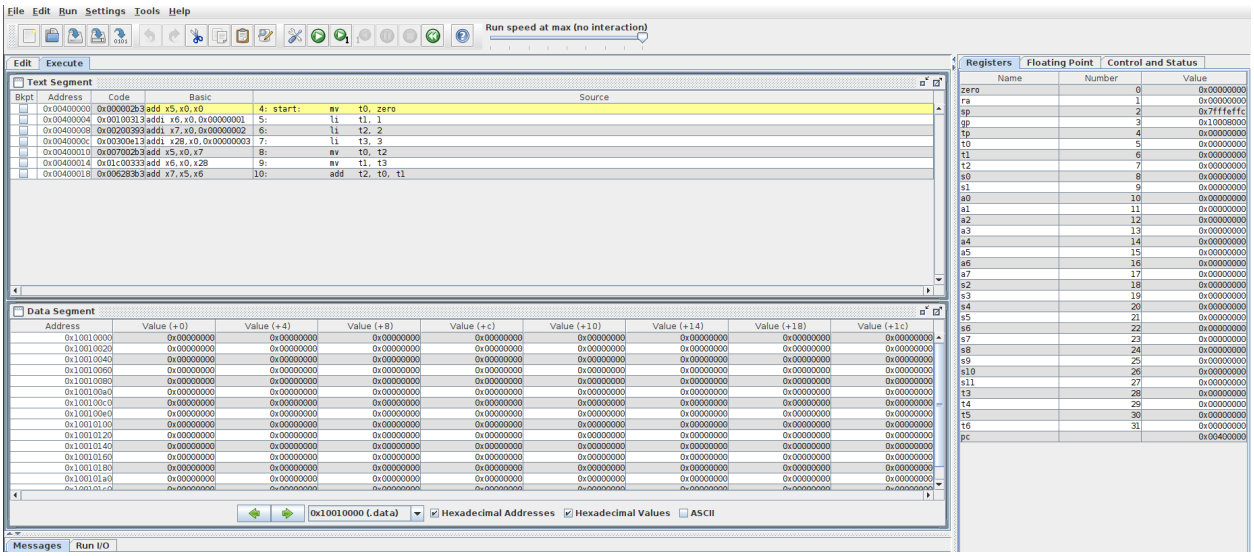
После запуска симулятора (инструкции по запуску см. выше) необходимо загрузить программу, выбрав в меню

File -> Open -> выбрать файл с расширением .S (например, ce2020labs/day_3/arch/risc_v_lab/prog/simple/main.S)



В основном окне появится код открываемой программы. В правой части отображаются все архитектурные регистры и их содержимое (значение в текущий момент).

Далее выполняем трансляцию программы в машинный код, выбрав в меню Run -> Assemble.



В открывшемся окне появится таблица, содержащая адрес инструкции в памяти, машинный код инструкции, дизассемблер инструкции (обратное представление кода на языке ассемблера) и соответствующая всему этому исходная строка программы.

Далее можно либо исполнить всю программу целиком, выбрав Run -> Go или

нажав F5, или выполнить программу по шагам, выбрав Run -> Step или нажав F7. На каждом шаге выполнения программы в правой части будут отображаться текущее содержимое регистров. Желтым выделяется следующая строка, которая будет исполнена. В нижней части отображается текущее содержимое памяти.

ПРОСТЕЙШИЕ КОМАНДЫ (ADD, ADDI, MV, LI)

Команда на языке ассемблера состоит из названия команды, а также, при необходимости, операндов и регистра результата. Рассмотрим команду целочисленного сложения ADD:

```
add rd, r1, r2
```

где rd - регистр результата, r1 - первое слагаемое, r2 - второе слагаемое, например,

```
add t2, t0, t1
```

что эквивалентно a = b + c, где a хранится в регистре t2, b в регистре t0, а c в регистре t1.

Команда ADDI выполняет сложение содержимого регистра с константой (immediate operand, т.е. непосредственное значение или литерал):

```
addi rd, r1, imm
```

где rd - регистр результата, r1 - первое слагаемое, imm - константа (второе слагаемое), например,

```
addi t2, t0, 2
```

что эквивалентно a = b + 2, где a хранится в регистре t2, b в регистре t0.

Команда MV копирует значение одного регистра в другой:

```
mv rd, r1
```

Команда LI загружает непосредственное значение в регистр:

```
li rd, imm
```

Также доступны следующие команды:

```
sub rd, rs1, rs2          вычитание rd = rs1 - rs2
```

```
and rd, rs1, rs2        побитовое И rd = rs1 & rs2
or rd, rs1, rs2         побитовое ИЛИ rd = rs1 | rs2
xor rd, rs1, rs2        побитовое исключающее ИЛИ rd = rs1 xor rs2
sll rd, rs1, rs2        сдвиг влево rd = rs1 << rs2
srl rd, rs1, rs2        сдвиг вправо rd = rs1 >> rs2
sra rd, rs1, rs2        арифметический (с учетом знака) сдвиг вправо rd = rs1 >>> rs2
```

Задание

Выполните симуляцию выполнения простейшей программы.

1. Запустите симулятор.
2. Откройте программу day_3/arch/risc_v_lab/prog/simple/main.S .
3. Выполните трансляцию программы. Сравните последние 2 столбца (Basis и Source) получившейся таблицы. Во всех ли строках команды совпадают?
4. Выполните пошаговую симуляцию выполнения программы, отслеживая на каждом шагу значения регистров.
5. Модифицируйте программу так, чтобы в конце программы в регистре t2 была сумма чисел 2 и 3, но используя минимальное количество команд. Попробуйте также минимизировать количество используемых регистров.

УСЛОВНЫЕ ОПЕРАЦИИ И ЦИКЛЫ

Для ветвлений и циклов в программе используются команды условных переходов BEQZ, BNEZ, BLEZ, BGEZ, BTLZ, BGTZ. Команды из этой группы выполняют переход на другой участок кода, если выполняется условие, соответствующее команде, если условие не выполняется, выполнение переходит на следующую команду. Пример команды:

```
beqz  rs, offset
```

Данная команда сравнивает значение в регистре rs с нулем. Если rs равен 0, выполняется переход на команду, расположенную в памяти со сдвигом (offset+4) байт относительно текущей команды.

В машинном коде переход выполняется по адресу команды в памяти. В данной группе команд адрес перехода указывается относительно адреса

текущей команды. На языке ассемблера для упрощения команду, на которую нужно выполнить переход, помечают некоторой меткой, а в команде перехода указывают эту метку. В простейшем случае метка должна быть текстовой:

```
bnez t0, label

.....

label: <команда, на которую нужно выполнить переход>
```

Значение команд:

beqz rs, label - переход, если rs == 0

bnez rs, label - переход, если rs != 0

blez rs, label - переход, если rs <= 0

bgez rs, label - переход, если rs >= 0

bltz rs, label - переход, если rs < 0

bgtz rs, label - переход, если rs > 0

Для организации цикла на заранее известное количество итераций достаточно загрузить в некоторый регистр количество итераций, уменьшать счетчик каждую итерацию и сравнивать с 0 оставшееся количество итераций.

Так как команд прямого сравнения двух чисел нет, для организации сравнения нужно сначала вычислить разность двух чисел при помощи команды SUB.

Задание

1. Запустите симулятор или закройте предыдущую программу (File -> Close).
2. Откройте программу day_3/arch/risc_v_lab/prog/fibonacci/main.S . Программа вычисляет числа Фибоначчи. Количество чисел за вычетом 2 указано в регистре t2.
3. Изучите программу, выполните трансляцию и симуляцию. В каких регистрах хранятся полученные на каждом шаге числа?
4. Модифицируйте программу - измените количество чисел Фибоначчи на 7.
5. Модифицируйте программу так, чтобы числа вычислялись бесконечно.

РАБОТА С ПАМЯТЬЮ

Память (или оперативная память) - необходимая составляющая любой современной вычислительной системы. RISC-V является load-store

архитектурой, что означает разделение команд на арифметические команды и команды доступа в память, т.е. арифметические команды всегда выполняются только с операндами, находящимися в регистрах. Память разделяют на области, каждая из которых имеет свое назначение, например область кода (команд), область данных, область стека и т.д. В RISC-V работа с памятью ведется при помощи команд считывания из памяти в регистр (load) и записи из регистра в память (store).

Команда записи

RISC-V имеет несколько видов команд записи в память, но мы будем использовать команды с косвенной абсолютной адресацией:

```
sw rs, (rd)
```

где rs - регистр, содержащий записываемые данные, rd - регистр, содержащий адрес записи в байтах. Данная команда выполняет запись 4-байт (слово, word, отсюда и название команды sw = store word) в память. Также доступны инструкции sh (store halfword) для записи 2 байт и sb (store byte) для записи 1 байта. Кроме того, данной команде можно задать ещё и смещение относительно адреса в регистре, например:

```
sw rs, -8(rd)
```

Данная команда будет записывать данные по адресу rd - 0x8.

В данной лабораторной работе мы будет работать с областью данных (.data), которая в используемом симуляторе начинается с адреса 0x10010000.

Задание

1. Запустите симулятор или закройте предыдущую программу (File -> Close).
2. Откройте программу day_3/arch/risc_v_lab/prog/fibonacci_mem/main.S . Программа вычисляет числа Фибоначчи и записывает их в память. Количество чисел за вычетом 2 указано в регистре t2.
3. Изучите программу, выполните трансляцию и симуляцию. В каких адресах хранятся полученные на каждом шаге числа?
4. Модифицируйте программу так, чтобы числа записывались не в каждую 4-байтовую ячейку памяти, а через одну.
5. Модифицируйте программу так, чтобы числа записывались в ячейки памяти по 2 байта.
6. Разработайте программу, записывающую в память числа следующих последовательностей:
    a. 00000000 00000002 00000004 00000006 00000008 ... 000001FE 00000200

b. 00010000 00030002 00050004 00070006 00090008 ... 00FD00FC 00FF00FE
c. 03020100 07060504 0B0A0908 0F0E0D0C 13121110 ... FBFAF9F8 FFFEFDFC
d. 01010101 02020202 03030303 04040404 05050505 ... EEEEEEEE FFFFFFFF
e. 11111111 22222222 33333333 44444444 55555555 ... EEEEEEEE FFFFFFFF
f. 01000100 02000200 03000300 04000400 05000500 ... FE00FE00 FF00FF00
g. 00000001 00000002 00000004 00000008 00000010 ... 40000000 80000000
h. 00020001 00080004 00200010 00800040 02000100 ... 20001000 80004000
i. 12345678 01234567 00123456 00012345 00001234 ... 00000012 00000001
j. 12345678 23456780 34567800 45678000 56780000 ... 78000000 80000000
k. 12345678 81234567 78123456 67812345 56781234 ... 23456781 12345678
l. 12345678 23456781 34567812 45678123 56781234 ... 81234567 12345678
m. 00000001 00000003 00000009 OOOOOO1B 00000051 ...
n. 00030001 001B0009 00F30051 088B02D9 4CE319A1 ...
o. 00000001 00000004 00000009 00000010 00000019 ...
p. ABCDEF78 0ABCDEF7 00ABCDEF 000ABCDE 0000ABCD ... 000000AB 0000000A
q. ABCDEF78 BCDEF780 CDEF7800 DEF78000 EF780000 ... 78000000 80000000
r. ABCDEF78 8ABCDEF7 78ABCDEF F78ABCDE EF78ABCD ... BCDEF78A ABCDEF78
s. ABCDEF78 BCDEF78A CDEF78AB DEF78ABC EF78ABCD ... 8ABCDEF7 ABCDEF78
t. 00010001 00030002 00050003 00070004 00090005 ... 00FD007F 00FF0080
u. 000000FF 000000FE 000000FD 000000FC 000000FB ... 00000001 00000000
v. 00FE00FF 00FC00FD 00FA00FB 00F800F9 00F600F7 ... 00010001 00000000
w. FCFDFEFF F8F9FAFB F4F5F6F7 F0F1F2F3 ECEDEEEF ... 07060504 03020100
x. FFFFFFFF EEEEEEEE DDDDDDDD CCCCCCCC BBBBBBBB ... 11111111 00000000
y. FFFFFFFF FFFFFFFF EEEEEEEE EEEEEEEE DDDDDDDD ... 00000000 00000000
z. 12345678 12345670 12345600 12345000 12340000 ... 10000000 00000000
aa. 12345678 1234567F 123456FF 12345FFF 1234FFFF ... 1FFFFFFF FFFFFFFF
bb. 12345678 02345678 00345678 00045678 00005678 ... 00000008

00000000
cc. 12345678 F2345678 FF345678 FFF45678 FFFF5678 ... FFFFFFF8 FFFFFFFF
dd. 12345678 F234567F FF3456FF FFF45FFF FFFFFFFF ... (повторить шаблон).

Команда считывания

Как и в случае с командами записи будем использовать команды считывания с абсолютной косвенной адресацией

```
lw    rd, offset(rs)
```

Данная команда считывает в регистр rd содержимое памяти по адресу rs + offset, где offset - константа.

Задание

1. Запустите симулятор или закройте предыдущую программу (File -> Close).
2. Откройте программу day_3/arch/risc_v_lab/prog/fibonacci_load/main.S . Программа вычисляет числа Фибоначчи и записывает их в память. В каждом цикле вычисления предыдущие 2 числа сначала считываются из памяти.
3. Изучите программу, выполните трансляцию и симуляцию.
4. Разработайте программу, которая записывает в память числа от 0x0 до 0xf (используйте циклы), а затем 15 раз инкрементирует (т.е. прибавляет 1) каждое из них в памяти. Какие значения чисел получились после выполнения программы?

Заключение

В рамках данной лабораторной работы были изучены основные команды системы команд RISC-V. Для более подробного изучения можно обратиться к полному описанию системы команд и языка ассемблера (см. раздел Материалы).

Необходимо отметить, что подавляющее большинство программ разрабатывается на языках высокого уровня (например, C/C++) и далее компилируется в двоичный (или машинный) код, который уже может быть исполнен процессором (или симулятором). Т.е. для разработки язык ассемблера практически не используется. Однако знание языка ассемблера необходимо и программистам и инженерам в следующих случаях:

1) иногда некоторые участки кода программы необходимо написать на языке ассемблера, т.е. внутри кода на C/C++ делаются ***ассемблерные вставки,***
2) при разработке процессора язык ассемблера помогает как написать простейшие программы для тестирования, так и отлаживать разрабатываемый процессор.