САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет
по домашней работе №4
«ISA. Ассемблер, дизассемблер»

Выполнил(а): Василенко Михаил Глебович

Номер ИСУ: 227231

студ. гр. M3134

Санкт-Петербург 2021

Цель работы: знакомство с архитектурой набора команд RISC-V.

Инструментарий и требования к работе: работа может быть выполнена на любом из следующих языков: C/C++, Python, Java.

Теоретическая часть

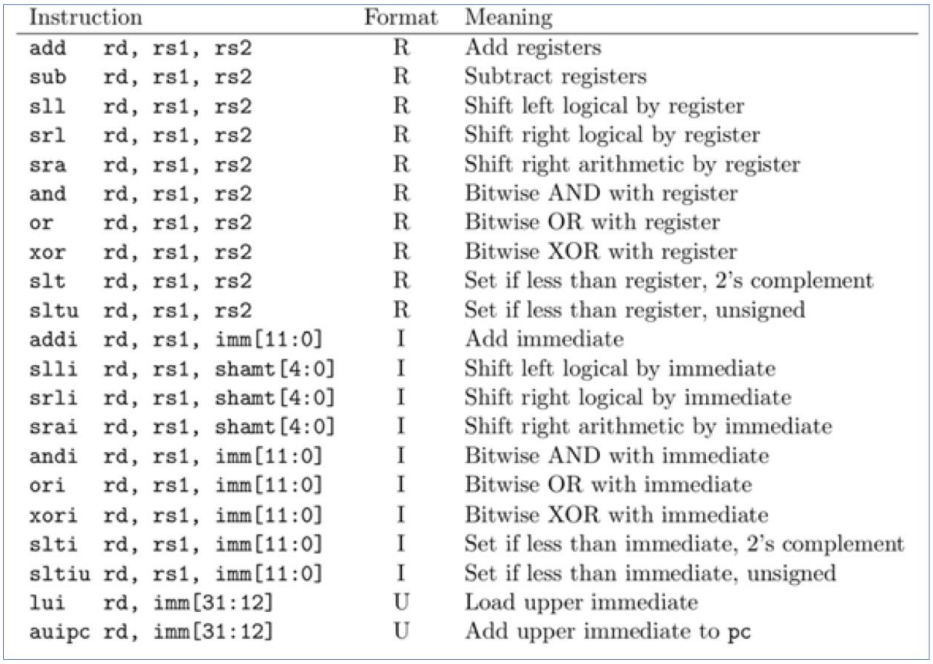
RISC-V — это свободная и открытая архитектура набора команд на основе RISC, используемая в микропроцессорах и микроконтроллерах.

Основная проблема существующих ISA заключается в том, что у них небольшой набор команд, и он не универсален. Набор команд может быть ориентирован для высокопроизводительных вычислений, требующих специальный аппаратный функционал, например криптография, многопоточность или графика... RISC-V решает эту проблему благодаря модулярной структуре команд.

Отличительные черты:

1) Результат любой операции, за исключением команд перехода всегда помещается в регистр общего назначения

2) Небольшое количество инструкций, около 50 штук, многие из которых еще существовали в RISC-I 1980 года. Стандартные расширения (M, A, F и D) расширяют набор на 53 инструкции, сжатый формат C определяет 34 команды. Используется 6 типов кодирования инструкций (форматов).



Instruction	Format	Meaning
`add rd, rs1, rs2`	R	Add registers
`sub rd, rs1, rs2`	R	Subtract registers
`sll rd, rs1, rs2`	R	Shift left logical by register
`srl rd, rs1, rs2`	R	Shift right logical by register
`sra rd, rs1, rs2`	R	Shift right arithmetic by register
`and rd, rs1, rs2`	R	Bitwise AND with register
`or rd, rs1, rs2`	R	Bitwise OR with register
`xor rd, rs1, rs2`	R	Bitwise XOR with register
`slt rd, rs1, rs2`	R	Set if less than register, 2's complement
`sltu rd, rs1, rs2`	R	Set if less than register, unsigned
`addi rd, rs1, imm[11:0]`	I	Add immediate
`slli rd, rs1, shamt[4:0]`	I	Shift left logical by immediate
`srli rd, rs1, shamt[4:0]`	I	Shift right logical by immediate
`srai rd, rs1, shamt[4:0]`	I	Shift right arithmetic by immediate
`andi rd, rs1, imm[11:0]`	I	Bitwise AND with immediate
`ori rd, rs1, imm[11:0]`	I	Bitwise OR with immediate
`xori rd, rs1, imm[11:0]`	I	Bitwise XOR with immediate
`slti rd, rs1, imm[11:0]`	I	Set if less than immediate, 2's complement
`sltiu rd, rs1, imm[11:0]`	I	Set if less than immediate, unsigned
`lui rd, imm[31:12]`	U	Load upper immediate
`auipc rd, imm[31:12]`	U	Add upper immediate to `pc`

Рисунок №1 – Вычислительные инструкции RISC-V. Целочисленные инструкции из базового набора RISC-V.

Архитектура RISC-V:

RV[xxx][abc.....xyz], где [xxx] – 32, 64 или 128, разрядность регистров и адресного пространства, [abc...xyz] – набор расширений архитектуры

Например: RV32I, RV32GC, RV128I

В этой архитектуре предусмотрено 32 целочисленных регистра, с номерами от 0 до 31 и 32 опциональных вещественных регистра.

Сокращение	Наименование	Версия	Статус
	Базовые наборы		
RV32I	Базовый набор с целочисленными операциями, 32-битный	2.1	Ratified
RV32E	Базовый набор с целочисленными операциями для встраиваемых систем, 32-битный, 16 регистров	1.9	Draft
RV64I	Базовый набор с целочисленными операциями, 64-битный	2.1	Ratified
RV128I	Базовый набор с целочисленными операциями, 128-битный	1.7	Draft
	Стандартные расширенные наборы		
M	Целочисленное умножение и деление (Integer Multiplication and Division)	2.0	Ratified
A	Атомарные операции (Atomic Instructions)	2.1	Ratified
F	Арифметические операции с плавающей запятой над числами одинарной точности (Single-Precision Floating-Point)	2.2	Ratified
D	Арифметические операции с плавающей запятой над числами двойной точности (Double-Precision Floating-Point)	2.2	Ratified
G	Сокращеное обозначение для комплекта из базового и стандартного наборов команд	н/д	н/д
Q	Арифметические операции с плавающей запятой над числами четверной точности	2.2	Ratified
L	Арифметические операции над десятичными числами с плавающей запятой (Decimal Floating-Point)	0.0	Open
C	Сокращённые имена для команд (Compressed Instructions)	2.2	Ratified
B	Битовые операции (Bit Manipulation)	0.36	Open
J	Двоичная трансляция и поддержка динамической компиляции (Dynamically Translated Languages)	0.0	Open
T	Транзакционная память (Transactional Memory)	0.0	Open
P	Короткие SIMD-операции (Packed-SIMD Instructions)	0.1	Open
V	Векторные расширения (Vector Operations)	1.0-draft	Open
N	Инструкции прерывания (User-Level Interrupts)	1.1	Open

Рисунок №2 – Список наборов команд

В 32-битных микроконтроллерах и для других встраиваемых применений используется набор RV32EC. В 64-битных процессорах может быть набор групп RV64GC, то же самое в полной записи — RV64IMAFDC.

Тип	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Регистр/регистр	funct7							rs2					rs1					funct3			rd					код операции					1	1
С операндом	±	imm[10:0]											rs1					funct3			rd					код операции					1	1
С длинным операндом	±	imm[30:12]																			rd					код операции					1	1
Сохранение	±	imm[10:5]						rs2					rs1					funct3			imm[4:0]					код операции					1	1
Ветвление	±	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	код операции					1	1
Переход	±	imm[10:1]										[11]	imm[19:12]								rd					код операции					1	1

Рисунок №3 – Форматы машинных команд

Признак формата 32-битной команды – младшие биты равны 11 и 2-4 биты != 111. rs1 – номер регистра, в котором находится первый операнд, rs2 – номер регистра, в котором находится второй операнд, rd – номер регистра, в который записан результат операции, opcode – определяет тип, funct3/funct7 определяют команду, imm – значение приоритета команды.

Существует несколько типов команд, например: R-type (add, or, and, xor …) – это простые арифметические/логические операции с двумя

операндами, результат записывается в регистр S. L-type – команды store и load (считать значение и записать). B-type – branch комадны, описывающие циклы. J-type – команды перехода.

Структура **Elf**-файла:

ELF — (формат исполняемых и связываемых файлов) определяет структуру бинарных файлов, библиотек, и файлов ядра (core files). Спецификация формата позволяет операционной системе корректно интерпретировать содержащиеся в файле машинные команды. Файл ELF, как правило, является выходным файлом компилятора или линкера и имеет двоичный формат.

ELF-файл состоит из:

1. Заголовка ELF

2. Таблицы заголовков программы

3. Таблицы заголовков секций

4. Данных секций

1) Заголовок программы содержит всю необходимую информацию для размещения исполняемых данных в памяти компьютера. Заголовок зафиксирован в начале файла, занимает ровно 52 байта для 32 битных файлов и 64 байта для 64 битных файлов. Содержит тип файла (32 битный или 64 битный), метод кодирования (по условию это little endian), тип файла (исполняемый, перемещаемый и т.д.), архитектуру процессора, где начинаются остальные части файла (количество заголовков, таблица заголовков программы, где находится первая исполняемая команда и т.д.)

2) Таблица заголовков секций содержит информацию о каждой секции. Каждый заголовок секции занимает 40 байт. Из заголовков секций можно

узнать смещение секции от начала файла, тип секции, флаги этой секции и т.д.

3) Секции содержат информацию, характерную для этой секции. В секции “.text” содержатся закодированные команды. “.symtab” также является одной из секций и содержит таблицу символов.

Практическая часть

Считываем входной elf файл, в заголовке которого находим endian (по условию у нас всегда little endian) и смещение таблицы заголовков секции. e_shoff — смещение таблицы заголовков секций в байтах от начала файла (если у файла отсутсвуюет смещение заголовков секций, то это значение равно 0). Рассмотрим таблицу заголовков секций, каждое поле содержит 40 байт. Текст имеет значение progbits, секция содержит информацию, определенную программой, ее формат и значение определяется программой единолично. После того как нашли заголовок такой секции, смотрим какие байты отвечают за смещение от начала файла и размер (по документации elf). Находим текст, переводим его в двоичную систему, а потом в команды RISV-V, тк у нас little endian, то кодировка в обратном порядке.

Результат работы программы

```
00000000:<main>  addi      a2, a2, -32
00000004:        sw        a2, 28(a1)
00000008:        sw        a2, 24(a8)
0000000c:        addi      a8, a2, 32
00000010:        addi      a10, zero, 0
00000014:        sw        a8, -12(a10)
00000018:        addi      a11, zero, 64
```

```
0000001c:        sw        a8, -16(a11)
00000020:        sw        a8, -20(a10)
00000024:        addi      a10, zero, 1
00000028:        sw        a8, -24(a10)
0000002c:        jal       zero, 0
00000030:<.LBB0_1>lw        a10, -24(a8)
00000034:        lw        a11, -16(a8)
00000038:        bge       a0, a10, 0
0000003c:        jal       zero, 0
00000040:<.LBB0_2>lw        a10, -24(a8)
00000044:        mul       a10, a10, a10
00000048:        lw        a11, -20(a8)
0000004c:        add       a10, a11, a10
00000050:        sw        a8, -20(a10)
00000054:        jal       zero, 0
00000058:<.LBB0_3>lw        a10, -24(a8)
0000005c:        addi      a10, a10, 1
00000060:        sw        a8, -24(a10)
00000064:        jal       zero, 0
00000068:<.LBB0_4>lw        a10, -20(a8)
0000006c:        lw        a8, 24(a2)
00000070:        lw        a1, 28(a2)
00000074:        addi      a2, a2, 32
00000078:        jalr      a0, a1, 0
```

Листинг

Язык — Python 3

Main.py

```
from operations import *


class RuntimeException(Exception):
    def __init__(self, txt):
        self.txt = txt


def main():
    inputFile = open('./venv/bin/test_elf (1).file', 'rb')
    try:
        buffer = inputFile.read()
    except IOError:
        raise RuntimeException('Файл не найден')
    inputFile.close()
    # проверка на 64-битные файлы
    if hex(buffer[4]) != '0x1':
        raise RuntimeException('не поддерживается')
    littleEndian = isLittleEndian(buffer)
    tableAddress = getTableAddress(buffer, littleEndian)
    textAddress, textSize = getText(buffer, tableAddress, littleEndian)
    text = binText(buffer, textSize, textAddress, littleEndian)
    handler = OperationsHandler()
    for word in text:
        handler.apply(word)


if __name__ == '__main__':
    main()
```

operations.py

```
import string


OperationCodes = {'0110011': 'r',
                  '0010011': 'i',
                  '0100011': 's',
                  '1100011': 'b',
                  '0000011': 'l',
                  '1101111': 'j',
                  '1100111': 'ja'}


RInst = {('0000000', '000'): 'add   ',
         ('0100000', '000'): 'sub   ',
         ('0000000', '001'): 'sll   ',
         ('0000000', '010'): 'slt   ',
         ('0000000', '011'): 'sltu  ',
         ('0000000', '100'): 'xor   ',
         ('0000000', '101'): 'srl   ',
         ('0100000', '101'): 'sra   ',
         ('0000000', '110'): 'or    ',
         ('0000000', '111'): 'and   ',
         ('0000001', '000'): 'mul   ',
         ('0000001', '001'): 'mulh  ',
         ('0000001', '010'): 'mulhsu',
         ('0000001', '011'): 'mulhu ',
         ('0000001', '100'): 'div   ',
         ('0000001', '101'): 'divu  ',
         ('0000001', '110'): 'rem   ',
         ('0000001', '111'): 'remu  '}
```

```
IInst = {'000': 'addi  ',
         '001': 'slli  ',
         '010': 'slti  ',
         '011': 'sltiu ',
         '100': 'xori  ',
         '101': 'sr_   ',
         '110': 'ori   ',
         '111': 'andi  '}


SInst = {'000': 'sb    ',
         '001': 'sh    ',
         '010': 'sw    '}


BInst = {'000': 'beq   ',
         '001': 'bne   ',
         '100': 'blt   ',
         '101': 'bge   ',
         '110': 'bltu  ',
         '111': 'bgeu  '}


LInst = {'000': 'lb    ',
         '001': 'lh    ',
         '010': 'lw    ',
         '100': 'lbu   ',
         '101': 'lhu   '}




def reverse(value: string) -> int:
```

```
    return value[6:8] + value[4:6] + value[2:4] + value[0:2]




def fillWithZeroes(index: string) -> string:
    return '0' * (8 - len(index)) + index




def translate(a):
    if a[0] == '1':
        return int(a[1:], 2) - int(a[:1]) * (2 ** (len(a) - 1))
    return int(a, 2)




def getTableAddress(buffer, isLittle):
    """ смещение заголовков секций от начала файла или адрес в файле """
    address = ''
    for byte in buffer[32:34]:
        if isLittle:
            #  кодирование от младшего к старшему при little
            address = hex(byte)[2::] + address
        else:
            # наоборот, если big
            address += hex(byte)[2::]
    return int(address, 16)
```

```
def isLittleEndian(buffer):
    """
    :param buffer:
    :return: little endian или big endian
    """
    return hex(buffer[5]) == '0x1'




def binText(buffer, textSize, textAddress, littleEndian):
    """
    :param littleEndian:
    :param buffer:
    :param textSize:
    :param textAddress:
    :return: text в двоичной системе
    """
    command = ''
    textBin = list()
    ind = 1
    index = 0
    for byte in buffer[textAddress: textAddress + textSize]:
        now = bin(byte)[2:]
        while len(now) < 8:
            now = '0' + now
        if littleEndian:
            command = now + command
        else:
            command += now
        if ind % 4 == 0:
            textBin.append([command, index])
```

```
            command = ''
            index += 4
        ind += 1
    return textBin



def getText(buffer, tableAddress, littleEndian):
    """ находим адрес .text и .symtab
        создаем строки для считывания """
    ind = 1
    textAddress = textSize = section = ''
    for byte in buffer[tableAddress::]:
        now = hex(byte)[2::]
        while len(now) < 2:
            now = '0' + now
        section += now
        if ind % 40 == 0:
            sectionType = section[8:16]
            sectionFlags = section[16:24]
            if littleEndian:
                sectionType = reverse(sectionType)
                sectionFlags = reverse(sectionFlags)
            sectionType = int(sectionType, 16)
            sectionFlags = int(sectionFlags, 16)
            if sectionType == 1 and sectionFlags == 6:
                textAddress = section[32:40]
                textSize = section[40:48]
            section = ''
        ind += 1
    if littleEndian:
```

```
        textAddress = reverse(textAddress)
        textSize = reverse(textSize)
    textAddress = int(textAddress, 16)
    textSize = int(textSize, 16)
    return textAddress, textSize




class OperationsHandler:
    def __init__(self):
        self.n_number = 0
        self.p_number = -1


   def apply(self, word) -> None:
        instruction = word[0]
        index = word[1]
        index = fillWithZeroes(hex(index)[2::])
        opCd = instruction[25:32]
        rs1 = translate(instruction[12:17])
        rs2 = translate(instruction[7:12])
        rd = translate(instruction[20:25])
        if OperationCodes[opCd] == 'r':
            func = (instruction[0:7], instruction[17:20])
            print(index, ':', ' ' * 8, RInst[func],
                  ' ' * 4, 'a', rd, ', ', 'a',
                  rs1, ', ', 'a', rs2,
                  sep='')
        elif OperationCodes[opCd] == 'i':
            imm = translate(instruction[0:12])
            func = instruction[17:20]
            zero = 'zero' if rs1 == 0 else 'a' + str(rs1)
```

```
    if self.p_number == -1:
        print(index, ':', '<main>  ', IInst[func],
              ' ' * 4, 'a', rd, ', ',
              zero, ', ', imm,
              sep='')
        self.p_number = 0
    else:
        print(index, ':', ' ' * 8, IInst[func],
              ' ' * 4, 'a', rd, ', ',
              zero, ', ',
              imm,
              sep='')
elif OperationCodes[opCd] == 's':
    imm = translate(instruction[0:7] + instruction[20:25])
    func = instruction[17:20]
    print(index, ':', ' ' * 8, SInst[func],
          ' ' * 4, 'a', rs1, ', ',
          imm, '(a', rs2, ')',
          sep='')
elif OperationCodes[opCd] == 'j':
    imm = instruction[0:20]
    imm = translate(imm)
    zero = 'zero' if rd == 0 else 'a' + str(rd)
    print(index, ':', ' ' * 8, 'jal   ',
          ' ' * 4, zero, ', ',
          imm,
          sep='')
    self.n_number += 1
elif OperationCodes[opCd] == 'ja':
    imm = instruction[0:12]
    imm = translate(imm)
    print(index, ':', ' ' * 8, 'jalr  ',
```

```
            ' ' * 4, 'a', rd, ', ', 'a',
            rs1, ', ', imm,
            sep='')
    elif OperationCodes[opCd] == 'l':
        imm = instruction[0:12]
        imm = translate(imm)
        func = instruction[17:20]
        if LInst[func] == 'lw    ' and (self.n_number != self.p_number):
            print(index, ':', '<.LBB0_',
                  self.n_number, '>', LInst[func],
                  ' ' * 4, 'a', rd, ', ',
                  imm, '(a', rs1, ')',
                  sep='')
            self.p_number = self.n_number
        else:
            print(index, ':', ' ' * 8, LInst[func],
                  ' ' * 4, 'a', rd, ', ', imm,
                  '(a', rs1, ')',
                  sep='')
    elif OperationCodes[opCd] == 'b':
        func = instruction[17:20]
        imm = instruction[0:7] + instruction[20:25]
        imm = translate(imm)
        zero = 'zero' if rs1 == 0 else 'a' + str(rs1)
        print(index, ':', ' ' * 8, BInst[func],
              ' ' * 4, 'a', rd, ', ',
              zero, ', ', imm,
              sep='')
```