САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

«ISA. Ассемблер, дизассемблер»

Выполнил(а): Белозоров Денис Сергеевич

Номер ИСУ: 334876

студ. гр. M3139

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

Elf (Executable and linkable file) - формат исполняемых файлов, объектных файлов, динамических библиотек. Используется во многих UNIX-подобных системах.

Elf является гибким, расширяемым и кроссплатформенным. Например, он поддерживает разные порядки следования байтов и размеры адресов. Это позволило использовать его во многих операционных системах на многих аппаратных платформах

Заголовок Elf файла определяет множество его характеристик: битность, метод кодирования (endian), начальные позиции заголовков, виртуальный адрес точки входа и т. д. (см рисунок 1)

```
#define EI_NIDENT 16

typedef struct
{
        unsigned char   e_ident[EI_NIDENT];     /* сигнатура и прочая информация                                        */
        Elf32_Half      e_type;                 /* тип объектного файла                                                 */
        Elf32_Half      e_machine;              /* архитектура аппаратной платформы                                     */
        Elf32_Word      e_version;              /* номер версии формата                                                 */
        Elf32_Addr      e_entry;                /* адрес точки входа (стартовый адрес программы)                        */
        Elf32_Off       e_phoff;                /* смещение от начала файла таблицы программных заголовков              */
        Elf32_Off       e_shoff;                /* смещение от начала файла таблицы заголовков секций                   */
        Elf32_Word      e_flags;                /* специфичные флаги процессора (не используется в архитектуре i386)    */
        Elf32_Half      e_ehsize;               /* размер ELF-заголовка файла в байтах                                  */
        Elf32_Half      e_phentsize;            /* размер записи в таблице программных заголовков                       */
        Elf32_Half      e_phnum;                /* число заголовков - количество записей в таблице программных заголовков */
        Elf32_Half      e_shentsize;            /* размер записи в таблице заголовков секций                            */
        Elf32_Half      e_shnum;                /* количество записей в таблице заголовков секций                       */
        Elf32_Half      e_shstrndx;             /* расположение сегмента, содержащего таблицу строк                     */
} Elf32_Ehdr
```

Рисунок 1 - устройство заголовка elf файла

Заголовок программы содержит необходимые для операционной системы атрибуты. Находится в позиции e_phoff. Каждому заголовку сопоставлено его содержимое в позиции p_offset. (см рисунок 2)

```
typedef struct
{
        Elf32_Word      p_type;                 /* тип сегмента                         */
        Elf32_Off       p_offset;               /* физическое смещение сегмента в файле */
        Elf32_Addr      p_vaddr;                /* виртуальный адрес начала сегмента    */
        Elf32_Addr      p_paddr;                /* физический адрес сегмента            */
        Elf32_Word      p_filesz;               /* физический размер сегмента в файле   */
        Elf32_Word      p_memsz;                /* размер сегмента в памяти             */
        Elf32_Word      p_flags;                /* флаги                                */
        Elf32_Word      p_align;                /* кратность выравнивания               */
} Elf32_Phdr
```

Рисунок 2 - устройство заголовка программы

Заголовок секции служит для описания секций. Находится в позиции e_shoff. Каждой секции сопоставлено её содержимое в позиции sh_offset. (см рисунок 3)

```
typedef struct
{
        Elf32_Word      sh_name;                /* имя секции                           */
        Elf32_Word      sh_type;                /* тип секции                           */
        Elf32_Word      sh_flags;               /* флаги секции                         */
        Elf32_Addr      sh_addr;                /* виртуальный адрес начала секции      */
        Elf32_Off       sh_offset;              /* физическое смещение секции в файле   */
        Elf32_Word      sh_size;                /* размер секции в байтах               */
        Elf32_Word      sh_link;                /* связка с другой секцией              */
        Elf32_Word      sh_info;                /* дополнительная информация о секции   */
        Elf32_Word      sh_addralign;           /* кратность выравнивания секции        */
        Elf32_Word      sh_entsize;             /* размер вложенного элемента, если есть */
} Elf32_Shdr
```

Рисунок 3 - устройство заголовка секции

Следующие секции можно выделить как необходимые для решения поставленной задачи:

Секция в позиции e_shstrndx - содержит названия остальных секций

.strtab - содержит названия записей в .symtab

.symtab - содержит символические обозначения, используемые в .text.

.text - содержит набор RISC-V команд.

RISC-V - система команд и процессорная архитектура. Поддерживает расширение команд, а его спецификация доступна для свободного и бесплатного использования.

В нашем случае используются наборы инструкций RV32I(обязательное для реализации стандартное множество команд), RV32M(поддерживает целочисленное умножение и деление) и RVC(сжатые инструкции - два байта вместо четырех).

Чтобы отличить 32х-битную инструкцию от 16ти-битной, следует посмотреть на её первые два бита. Если они оба равны 1, то инструкция 32х битная, и 16-ти битная соответственно иначе.

Будем последовательно идти по данным .text и декодировать полученные инструкции в соответствии с таблицами в спецификации RISC-V (см таблица 1.)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Таблица 1 - разделение RISC-V инструкций на классы.

Здесь opcode, funct7, funct3 позволяют определить инструкцию, rs2, rs1, rd - регистры.

Imm (Immutable) - некоторые данные, представленные целым числом со знаком (бит под знак), причем бит знака всегда находится в последнем бите инструкции (в 12м бите для RVC)

Uimm - беззнаковая реализация.

Nzuimm/Nzimm - ненулевое значение.

Каждый блок imm разбит на части вида [a, b], причем сумма всех a - b + 1 равна длине блока, то есть используется каждый бит и только один раз. соответственно берутся последовательные a - b + 1 бит, начиная с конца блока, затем со сдвигом b добавляются к ответу с помощью побитового или:

answer |= getSlice(a, b, start) << b

Практическая часть

Результатом является код на языке Python версии 3.9.

Описание порядка работы программы:

1) чтение заголовка elf файла
2) чтение секции с названиями секций
3) чтение секции .strtab
4) чтение секции .text
5) чтение и вывод секции .symtab
6) Последовательный просмотр команд для записи меток вида LOC_XXXXX
7) Последовательный разбор команд и их вывод

Запуск python solve.py #input# #output#

utils.py

```
def convBytes(data):
    return int.from_bytes(data, byteorder='little')
```

```
def convBits(data):
    return int(data, 2)

def getRvcRegister(val):
    if val <= 1:
        return 's' + str(val)
    else:
        return 'a' + str(val - 2)

def getSlice(val, start, end):
    return (val >> start) & ((1 << (end - start + 1)) - 1)

def calcImm(val, end, start, last):
    return getSlice(val,last - (end - start),last) << (start)

def getRegister(val):
    if val == 0:
        return "zero"
    elif val == 1:
        return "ra"
    elif val == 2:
        return "sp"
    elif val == 3:
        return "gp"
    elif val == 4:
        return "tp"
    elif 5 <= val <= 7:
        return "t" + str(val - 5)
    elif val == 8:
        return "s0"
```

```
    elif val == 9:
        return "s1"
    elif 10 <= val <= 17:
        return "a" + str(val - 10)
    elif 18 <= val <= 27:
        return "s" + str(val - 16)
    elif 28 <= val <= 31:
        return "t" + str(val - 25)
```

rv32_instructions.py

```
from utils import *
from enum import Enum

class RV32type(Enum):
    Rtype = 0,
    Itype = 1,
    Stype = 2,
    Btype = 3,
    Utype = 4,
    Jtype = 5

def parseRtype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['rd'] = getRegister(getSlice(data, 7, 11) )
    result['funct3'] =  getSlice(data, 12, 14)
    result['rs1'] =  getRegister(getSlice(data, 15, 19) )
    result['rs2'] =  getRegister(getSlice(data, 20, 24) )
    result['funct7'] =  getSlice(data, 25, 31)
    return result
```

```
def parseItype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['rd'] = getRegister(getSlice(data, 7, 11) )
    result['funct3'] =  getSlice(data, 12, 14)
    result['rs1'] =  getRegister(getSlice(data, 15, 19) )
    result['uimm'] =  getSlice(data, 20, 31)
    result['imm'] =  calcImm(data, 11, 0, 31)
    if getSlice(data, 31, 31):
        result['imm'] -= (1 << 12)
    return result

def parseStype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['funct3'] =  getSlice(data, 12, 14)
    result['rs1'] =  getRegister(getSlice(data, 15, 19) )
    result['rs2'] =  getRegister(getSlice(data, 20, 24) )
    result['imm'] =  calcImm(data, 11, 5, 31) | calcImm(data, 4, 0, 11)
    if getSlice(data, 31, 31):
        result['imm'] -= (1 << 12)
    return result

def parseBtype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['imm'] = calcImm(data, 12, 12, 31) | calcImm(data, 10, 5, 30) |
calcImm(data, 4, 1, 11) | calcImm(data, 11, 11, 7)
    result['funct3'] =  getSlice(data, 12, 14)
    result['rs1'] =  getRegister(getSlice(data, 15, 19) )
    result['rs2'] =  getRegister(getSlice(data, 20, 24) )
```

```
    if getSlice(data, 31, 31):
        result['imm'] -= (1 << 13)
    return result


def parseUtype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['rd'] = getRegister(getSlice(data, 7, 11))
    result['imm'] =  calcImm(data, 31, 12, 31)
    if getSlice(data, 31, 31):
        result['imm'] -= (1 << 32)
    return result

def parseJtype(data):
    result = {}
    result['opcode'] = getSlice(data, 0, 6)
    result['rd'] =  getRegister(getSlice(data, 7, 11))
    result['imm'] = calcImm(data, 20, 20, 31) | calcImm(data, 10, 1, 30) |
calcImm(data, 11, 11, 20) | calcImm(data, 19, 12, 19)
    if getSlice(data, 31, 31):
        result['imm'] -= (1 << 21)
    return result

typeByOpcode = {
    0b0110111 : RV32type.Utype,
    0b0010111 : RV32type.Utype,
    0b1101111 : RV32type.Jtype,
    0b1100111 : RV32type.Itype,
    0b1100011 : RV32type.Btype,
    0b0000011 : RV32type.Itype,
```

```
    0b0100011 : RV32type.Stype,
    0b0010011 : RV32type.Itype,
    0b0110011 : RV32type.Rtype,
    0b0001111 : RV32type.Btype,
    0b1110011 : RV32type.Itype,
    0b1110011 : RV32type.Itype
}

typeToParser = {
    RV32type.Rtype : parseRtype,
    RV32type.Itype : parseItype,
    RV32type.Stype : parseStype,
    RV32type.Btype : parseBtype,
    RV32type.Utype : parseUtype,
    RV32type.Jtype : parseJtype,
}
```

solve.py

```
import sys
from tkinter import EXCEPTION
from rv32_instructions import *
from utils import *

# Сигнатура 7f 45 4c 46
class ElfParser:
    def __init__(self, name):
        self.inp = open(name, "rb")
        if self.inp.read(4) != b'\x7fELF':
            print("Wrong file signature!")
            exit(0)
        self.EI_CLASS = self.readIntBytes(1)
        if self.EI_CLASS == 0 or not (0 <= self.EI_CLASS <= 2):
```

```
    print("Wrong class!")
    exit(0)
self.dataLength = 4 if self.EI_CLASS == 1 else 8
self.EI_DATA = self.readIntBytes(1)
if self.EI_DATA != 1:
    print("Only little endiand supported!")
    exit(0)
self.EI_VERSION = self.readIntBytes(1)
if self.EI_VERSION != 1:
    print("Wrong version!")
    exit(0)
self.EI_OSABI = self.readIntBytes(1)
self.EI_ABIVERSION = self.readIntBytes(1)
self.EI_PAD = self.inp.read(7)
self.e_type = self.readIntBytes(2)
self.e_machine = self.readIntBytes(2)
self.e_version = self.readIntBytes(4)
self.e_entry = self.readIntBytes(self.dataLength)
self.e_phoff = self.readIntBytes(self.dataLength)
self.e_shoff = self.readIntBytes(self.dataLength)
self.e_flags = self.inp.read(4)
self.e_ehsize = self.readIntBytes(2)
self.e_phentsize = self.readIntBytes(2)
self.e_phnum = self.readIntBytes(2)
self.e_shentsize = self.readIntBytes(2)
self.e_shnum = self.readIntBytes(2)
self.e_shstrndx = self.readIntBytes(2)
self.names = self.readTablesNames()
self.stringTable = self.readStringTable()
self.nameByPos = dict()
```

```
    def readIntBytes(self, ln):
        return int.from_bytes(self.inp.read(ln), byteorder='little')

    def readTablesNames(self):
        pos = self.e_shoff + self.e_shstrndx * self.e_shentsize
        self.inp.seek(pos)
        self.inp.read(4 * 2 + 2 * self.dataLength)
        addr = self.readIntBytes(self.dataLength)
        size = self.readIntBytes(self.dataLength)
        self.inp.seek(addr)
        self.rawNames = self.inp.read(size).decode("utf-8")
        return self.rawNames.split(chr(0))[1:]

    def readStringTable(self):
        data = self.readByName(".strtab")[0]
        self.rawStringTable = data.decode("utf-8")
        return self.rawStringTable.split(chr(0))[1:-1]

    def readSymTab(self):
        data = self.readByName(".symtab")[0]
        pos = 0
        cnt = 0
        print("%s %-15s %7s %-8s %-8s %-8s %6s %s" % ("Symbol", "Value",
"Size", "Type", "Bind", "Vis", "Index", "Name"))
        while pos < len(data):
            name = int.from_bytes(data[pos:4 + pos], byteorder='little')
            pos += 4
            value = int.from_bytes(data[pos:self.dataLength + pos],
byteorder='little')
            pos += self.dataLength
            size = int.from_bytes(data[pos:4 + pos], byteorder='little')
```

```
            pos += 4
            info = int.from_bytes(data[pos:1 + pos], byteorder='little')
            pos += 1
            other = int.from_bytes(data[pos:1 + pos], byteorder='little')
            pos += 1
            shndx = int.from_bytes(data[pos:2 + pos], byteorder='little')
            pos += 2
            sname = self.rawStringTable[name:]
            sname = sname[:sname.index(chr(0))] if chr(0) in sname else
'UNDEF'
            self.nameByPos[value] = sname
            bind = info >> 4
            type = info & 0xf
            vis = other & 0x3
            ind = 'ABS' if shndx == 65521 else shndx
            if ind == 0:
                ind = 'UNDEF'
            parse = {0 : 'NOTYPE', 1: 'OBJECT', 2: 'FUNC', 3: 'SECTION',
4: 'FILE', 5: 'COMMON', 6: 'TLS', 10: 'LOOS', 12: 'HIOS', 13: 'LOPROC',
15: 'HIPROC'}
            parseBind = {
                0: 'LOCAL',
                1: 'GLOBAL',
                2: 'WEAK',
                10: 'LOOS',
                12: 'HIOS',
                13: 'LOPROC',
                15: 'HIPROC'
            }
            parseVis = {
                0: 'DEFAULT',
                1: 'INTERNAL',
```

```
                2: 'HIDDEN',
                3: 'PROTECTED'
            }
            print("[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s" % (cnt, value,
size, parse[type], parseBind[bind], parseVis[vis], ind, sname))
            cnt += 1
        print()
            #print(sname if name != 0 else 'NONAME', value, size, info,
other, shndx)


    def readByName(self, value):
        for i in range(self.e_shnum):
            pos = self.e_shoff + self.e_shentsize * i
            self.inp.seek(pos)
            name = self.rawNames[self.readIntBytes(4):]
            name = name[:name.index(chr(0))] if chr(0) in name else name
            if name != value:
                continue
            self.inp.read(4 * 1 + 1 * self.dataLength)
            addr = self.readIntBytes(self.dataLength)
            offset = self.readIntBytes(self.dataLength)
            size = self.readIntBytes(self.dataLength)
            self.inp.seek(offset)
            return self.inp.read(size), addr



def generateName(pos):
    mn = pos + 1
    mnind = -1
    for key, val in parser.nameByPos.items():
        if key > pos:
```

```
            continue
        if pos - key < mn:
            mn = pos - key
            mnind = val
    return -mn + pos, mnind


def getBit(data, begin, end):
    return (((data) & ((1 << (end + 1)) - 1)) >> begin)



def readLOC(pos, data):
    loc = 0
    while(pos - start < len(data)):
        ln = 4
        val = int.from_bytes(data[pos - start:pos - start + ln],
byteorder='little')
        offset = ''
        if getSlice(val, 0, 1) != 0b11:
            ln = 2
            val = int.from_bytes(data[pos - start:pos - start + ln],
byteorder='little')
            func = getSlice(val, 13, 15)
            opcode = getSlice(val, 0, 1)


            if opcode == 0b01 and (func == 0b001 or func == 0b101):
                offset = calcImm(val, 11, 11, 12) | calcImm(val, 4, 4, 11)
| calcImm(val, 9, 8, 10) | calcImm(val, 10, 10, 8) | calcImm(val, 6, 6, 7)
| calcImm(val, 7, 7, 6) | calcImm(val, 3, 1, 5) | calcImm(val, 5, 5, 2)
                if getSlice(val, 12, 12):
                    offset -= (1 << 12)
            elif opcode == 0b01 and (func == 0b110 or func == 0b111):
```

```
                offset = calcImm(val, 8, 8, 12) | calcImm(val, 4, 3, 11) |
calcImm(val, 7, 6, 6) | calcImm(val, 2, 1, 4) | calcImm(val, 5, 5, 2)
                if getSlice(val, 12, 12):
                    offset -= (1 << 9)
        else:
            if getSlice(val, 0, 6) in typeByOpcode:
                parsed = typeToParser[typeByOpcode[getSlice(val, 0,
6)]](val)
                if parsed['opcode'] == 0b1101111 or parsed['opcode'] ==
0b1100011:
                    offset = parsed['imm']
        if offset != '':
            name = generateName(pos + offset)
            if name[0] != pos + offset:
                parser.nameByPos[pos + offset] = 'LOC_%05x' % loc
                loc += 1
        pos += ln



def printCmd16(val):
    ############ RVC
    func = getSlice(val, 13, 15)
    opcode = getSlice(val, 0, 1)
    if opcode == 0b00:
        ans = ''
        imm = -1
        rs1 = getSlice(val, 7, 9)
        rd = getSlice(val, 2, 4)
        if func == 0b000:
            ans = 'C.ADDI4SPN'
            imm = calcImm(val, 5, 4, 12) | calcImm(val, 9, 6, 10) |
calcImm(val, 2, 2, 6) | calcImm(val, 3, 3, 5)
```

```
            print('{} {},{},{}'.format(ans, getRvcRegister(getSlice(val,
2, 4)), "sp", imm))

            return

        elif func == 0b001:

            ans = 'C.FLD'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 7, 6, 6)

        elif func == 0b010:

            ans = 'C.LW'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 2, 2, 6) |
calcImm(val, 6, 6, 5)

        elif func == 0b011:

            ans = 'C.FLW'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 2, 2, 6) |
calcImm(val, 6, 6, 5)

        elif func == 0b101:

            ans = 'C.FSD'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 7, 6, 6)

        elif func == 0b110:

            ans = 'C.SW'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 2, 2, 6) |
calcImm(val, 6, 6, 5)

        elif func == 0b111:

            ans = 'C.FSW'

            imm = calcImm(val, 5, 3, 12) | calcImm(val, 2, 2, 6) |
calcImm(val, 6, 6, 5)

        print('{} {},{}({})'.format(ans,  getRvcRegister(rd), imm,
getRvcRegister(rs1)))

    elif opcode == 0b01:

        ans = ''

        if val <= 1:

            ans = 'C.NOP'

            print(ans)

            return
```

```
        elif func == 0b000:
            ans = 'C.ADDI'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
            if getSlice(val, 12, 12):
                imm -= (1 << 6)
            print('{} {},{}'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b001:
            ans = 'C.JAL'
            offset = calcImm(val, 11, 11, 12) | calcImm(val, 4, 4, 11) |
calcImm(val, 9, 8, 10) | calcImm(val, 10, 10, 8) | calcImm(val, 6, 6, 7) |
calcImm(val, 7, 7, 6) | calcImm(val, 3, 1, 5) | calcImm(val, 5, 5, 2)
            if getSlice(val, 12, 12):
                offset -= (1 << 12)
            name = generateName(pos + offset)
            print('C.JAL {:x} <{}>'.format(pos + offset, name[1]))
            return
        elif func == 0b010:
            ans = 'C.LI'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
            if getSlice(val, 12, 12):
                imm -= (1 << 6)
            print('{} {},{}'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b011:
            if getSlice(val, 7, 11) == 2:
                imm = calcImm(val, 9, 9, 12) | calcImm(val, 4, 4, 6) |
calcImm(val, 6, 6, 5) | calcImm(val, 8, 7, 4) | calcImm(val, 5, 5, 2)
                ans = 'C.ADDI16SP'
                if getSlice(val, 12, 12):
```

```
            imm -= (1 << 10)
        print('{} {},{}'.format(ans,  getRegister(2), imm))
        return
    else:
        ans = 'C.LUI'
        rd = getSlice(val, 7, 11)
        imm = calcImm(val, 17, 17, 12) | calcImm(val, 16, 12, 6)
        if getSlice(val, 12, 12):
            imm -= (1 << 18)
        print('{} {},0x{:x}'.format(ans,  getRegister(rd), imm))
        return
elif func == 0b100:
    type = getSlice(val, 10, 11)
    if type == 0b00:
        ans = 'C.SRLI'
        rd = getSlice(val, 7, 9)
        imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
        print('{} {},{}'.format(ans,  getRvcRegister(rd), imm))
        return
    elif type == 0b01:
        ans = 'C.SRAI'
        rd = getSlice(val, 7, 9)
        imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
        print('{} {},{}'.format(ans,  getRvcRegister(rd), imm))
        return
    elif type == 0b10:
        ans = 'C.ANDI'
        rd = getSlice(val, 7, 9)
        imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
        if getSlice(val, 12, 12):
            imm -= (1 << 6)
```

```
                print('{} {},{}'.format(ans,  getRvcRegister(rd), imm))
                return
            else:
                type = getSlice(val, 5, 6)
                sign = getSlice(val, 12, 12)
                rd = getSlice(val, 7, 9)
                rs2 = getSlice(val, 2, 4)
                if type == 0b00 and sign == 0:
                    ans = 'C.SUB'
                elif type == 0b01 and sign == 0:
                    ans = 'C.XOR'
                elif type == 0b10 and sign == 0:
                    ans = 'C.OR'
                elif type == 0b11 and sign == 0:
                    ans = 'C.AND'
                elif type == 0b00 and sign == 1:
                    ans = 'C.SUBW'
                elif type == 0b01 and sign == 1:
                    ans = 'C.ADDW'
                print('{} {},{}'.format(ans,  getRvcRegister(rd),
getRvcRegister(rs2)))
                return
        elif func == 0b101:
            ans = 'C.J'
            offset = calcImm(val, 11, 11, 12) | calcImm(val, 4, 4, 11) |
calcImm(val, 9, 8, 10) | calcImm(val, 10, 10, 8) | calcImm(val, 6, 6, 7) |
calcImm(val, 7, 7, 6) | calcImm(val, 3, 1, 5) | calcImm(val, 5, 5, 2)
            if getSlice(val, 12, 12):
                offset -= (1 << 12)
            name = generateName(pos + offset)
            print('C.J {:x} <{}>'.format(pos + offset, name[1]))
            return
```

```
        elif func == 0b110:
            rs = getSlice(val, 7, 9)
            offset = calcImm(val, 8, 8, 12) | calcImm(val, 4, 3, 11) |
calcImm(val, 7, 6, 6) | calcImm(val, 2, 1, 4) | calcImm(val, 5, 5, 2)
            if getSlice(val, 12, 12):
                offset -= (1 << 9)
            ans = 'C.BEQZ'
            name = generateName(pos + offset)
            if name[0] != pos + offset:
                print('{} {},{:x} <{}+0x{:x}>'.format(ans,
getRvcRegister(rs), pos + offset, name[1], pos + offset - name[0]))
            else:
                print('{} {},{:x} <{}>'.format(ans, getRvcRegister(rs),
pos + offset, name[1]))
            return
        elif func == 0b111:
            rs = getSlice(val, 7, 9)
            offset = calcImm(val, 8, 8, 12) | calcImm(val, 4, 3, 11) |
calcImm(val, 7, 6, 6) | calcImm(val, 2, 1, 4) | calcImm(val, 5, 5, 2)
            if getSlice(val, 12, 12):
                offset -= (1 << 9)
            ans = 'C.BNEZ'
            name = generateName(pos + offset)
            if name[0] != pos + offset:
                print('{} {},{:x} <{}+0x{:x}>'.format(ans,
getRvcRegister(rs), pos + offset, name[1], pos + offset - name[0]))
            else:
                print('{} {},{:x} <{}>'.format(ans, getRvcRegister(rs),
pos + offset, name[1]))
            return
        print(ans)
    elif opcode == 0b10:
        ans = ''
```

```
        if func == 0b000:
            ans = 'C.SLLI'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 0, 6)
            print('{} {},0x{:x}'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b001:
            ans = 'C.FLDSP'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 3, 6) |
calcImm(val, 8, 6, 4)
            print('{} {},{}(sp)'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b010:
            ans = 'C.LWSP'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 2, 6) |
calcImm(val, 7, 6, 3)
            print('{} {},{}(sp)'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b011:
            ans = 'C.FLWSP'
            rd = getSlice(val, 7, 11)
            imm = calcImm(val, 5, 5, 12) | calcImm(val, 4, 2, 6) |
calcImm(val, 7, 6, 3)
            print('{} {},{}(sp)'.format(ans,  getRegister(rd), imm))
            return
        elif func == 0b100:
            type = getSlice(val, 12, 12)
            v = getSlice(val, 2, 6)
            s = getSlice(val, 7, 11)
            if type == 0 and v == 0:
```

```
                ans = 'C.JR'
                rs = getSlice(val, 7, 11)
                print('{} {}'.format(ans,  getRegister(rs)))
                return
            elif type == 0:
                ans = 'C.MV'
                rd = getSlice(val, 7, 11)
                rs = getSlice(val, 2, 6)
                print('{} {},{}'.format(ans,  getRegister(rd),
getRegister(rs)))
                return
            elif type == 1 and v == 0 and s == 0:
                ans = 'C.EBREAK'
                print(ans)
                return
            elif type == 1 and v == 0:
                ans = 'C.JALR'
                rs = getSlice(val, 7, 11)
                print('{} {}'.format(ans,  getRegister(rs)))
                return
            elif type == 1:
                ans = 'C.ADD'
                rs = getSlice(val, 2, 6)
                rd = getSlice(val, 7, 11)
                print('{} {},{}'.format(ans,  getRegister(rd),
getRegister(rs)))
                return
        elif func == 0b101:
            imm = calcImm(val, 5, 3, 12) | calcImm(val, 8, 6, 9)
            rs = getSlice(val, 2, 6)
            ans = 'C.FSDSP'
```

```
            print('{} {},{}({})'.format(ans, getRegister(rs), imm, "sp"))
            return
        elif func == 0b110:
            imm = calcImm(val, 5, 2, 12) | calcImm(val, 7, 6, 8)
            rs = getSlice(val, 2, 6)
            ans = 'C.SWSP'
            print('{} {},{}({})'.format(ans, getRegister(rs), imm, "sp"))
            return
        elif func == 0b111:
            imm = calcImm(val, 5, 2, 12) | calcImm(val, 7, 6, 8)
            rs = getSlice(val, 2, 6)
            ans = 'C.FSWSP'
            print('{} {},{}({})'.format(ans, getRegister(rs), imm, "sp"))
            return
        print('unknown_command')
    ############

def printCmd32(data):
    if getSlice(data, 0, 6) not in typeByOpcode:
        print('unknown_command')
        return
    parsed = typeToParser[typeByOpcode[getSlice(data, 0, 6)]](data)
    ############ RV32I
    if parsed['opcode'] == 0b0110111:
        print('LUI {},0x{:x}'.format(parsed["rd"] , parsed["imm"]))
    elif parsed['opcode'] == 0b0010111:
        print('AUIPC {},0x{:x}'.format(parsed['rd'], parsed['imm']))
    elif parsed['opcode'] == 0b1101111:
        offset = parsed['imm']
        name = generateName(pos + offset)
```

```
        print('JAL {},{:x} <{}>'.format(parsed['rd'], pos + offset,
name[1]))
    elif parsed['opcode'] == 0b1100111:
        print('JALR {},{}({})'.format(parsed['rd'], parsed['imm'],
parsed['rs1']))
    elif parsed['opcode'] == 0b1100011:
        ans = ''
        if parsed['funct3'] == 0b000:
            ans = 'BEQ'
        elif parsed['funct3'] == 0b001:
            ans = 'BNE'
        elif parsed['funct3'] == 0b100:
            ans = 'BLT'
        elif parsed['funct3'] == 0b101:
            ans = 'BGE'
        elif parsed['funct3'] == 0b110:
            ans = 'BLTU'
        elif parsed['funct3'] == 0b111:
            ans = 'BGEU'
        if ans != '':
            offset = parsed['imm']
            name = generateName(pos + offset)
            print("{} {},{},{:x}, <{}>".format( ans, parsed['rs1'],
parsed['rs2'], pos + offset, name[1]))
    elif parsed['opcode'] == 0b0000011:
        ans = ''
        if parsed['funct3'] == 0b000:
            ans = 'LB'
        elif parsed['funct3'] == 0b001:
            ans = 'LH'
        elif parsed['funct3'] == 0b010:
            ans = 'LW'
```

```
        elif parsed['funct3'] == 0b100:
            ans ='LBU'
        elif parsed['funct3'] == 0b101:
            ans = 'LHU'
        if ans != '':
            print('{} {},{}({})'.format(ans, parsed['rd'], parsed['imm'],
parsed['rs1']))
    elif parsed['opcode'] == 0b0100011:
        ans = ''
        if parsed['funct3'] == 0b000:
            ans = 'SB'
        elif parsed['funct3'] == 0b001:
            ans = 'SH'
        elif parsed['funct3'] == 0b010:
            ans = 'SW'
        print('{} {},{}({})'.format(ans, parsed['rs2'], parsed['imm'],
parsed['rs1']))
    elif parsed['opcode'] == 0b0010011:
        ans = ''
        if parsed['funct3'] == 0b000:
            ans = 'ADDI'
        elif parsed['funct3'] == 0b010:
            ans = 'SLTI'
        elif parsed['funct3'] == 0b011:
            ans = 'SLTIU'
        elif parsed['funct3'] == 0b100:
            ans = 'XORI'
        elif parsed['funct3'] == 0b110:
            ans = 'ORI'
        elif parsed['funct3'] == 0b111:
            ans = 'ANDI'
```

```
        elif parsed['funct3'] == 0b001:
            ans = 'SLLI'
            print('{} {},{},0x{:x}'.format(ans, parsed['rd'],
parsed['rs1'], getSlice(data, 20, 24)))
            return
        elif parsed['funct3'] == 0b101:
            if getSlice(data, 25, 31) == 0b0000000:
                ans = 'SRLI'
                print('{} {},{},0x{:x}'.format(ans, parsed['rd'],
parsed['rs1'], getSlice(data, 20, 24)))
                return
            elif getSlice(data, 25, 31) == 0b0100000:
                ans = 'SRAI'
                print('{} {},{},0x{:x}'.format(ans, parsed['rd'],
parsed['rs1'], getSlice(data, 20, 24)))
                return
        if ans != '':
            print('{} {},{},{}'.format(ans, parsed['rd'], parsed['rs1'],
parsed["imm"]))
    elif parsed['opcode'] == 0b0110011:
        ans = ''
        if parsed['funct3'] == 0b000:
            if parsed['funct7'] == 0b0000000:
                ans = 'ADD'
            elif parsed['funct7'] == 0b0100000:
                ans = 'SUB'
        elif parsed['funct3'] == 0b001 and parsed['funct7'] == 0:
            ans = 'SLL'
        elif parsed['funct3'] == 0b010 and parsed['funct7'] == 0:
            ans = 'SLT'
        elif parsed['funct3'] == 0b011 and parsed['funct7'] == 0:
            ans = 'SLTU'
```

```
        elif parsed['funct3'] == 0b100 and parsed['funct7'] == 0:
            ans = 'XOR'
        elif parsed['funct3'] == 0b101:
            if parsed['funct7'] == 0:
                ans = 'SRL'
            elif parsed['funct7'] == 0b0100000:
                ans = 'SRA'
        elif parsed['funct3'] == 0b110 and parsed['funct7'] == 0:
            ans = 'OR'
        elif parsed['funct3'] == 0b111 and parsed['funct7'] == 0:
            ans = 'AND'
        if ans != '':
            print("{} {},{},{}".format(ans, parsed['rd'], parsed['rs1'],
parsed['rs2']))
    elif parsed['opcode'] == 0b0001111 and parsed['funct7'] == 0b0000000:
        print('FENCE') # Not Implemented
    elif parsed['opcode'] == 0b1110011:
        if parsed['imm'] == 0:
            print('ECALL')
        elif parsed['imm'] == 1:
            print('EBREAK')
    ############
    ############ RV32M
    if parsed['opcode'] == 0b0110011 and parsed['funct7'] == 1:
        ans = ''
        if parsed['funct3'] == 0b000:
            ans = 'MUL'
        elif parsed['funct3'] == 0b001:
            ans = 'MULH'
        elif parsed['funct3'] == 0b010:
            ans = 'MULHSU'
```

```
        elif parsed['funct3'] == 0b011:
            ans = 'MULHU'
        elif parsed['funct3'] == 0b100:
            ans = 'DIV'
        elif parsed['funct3'] == 0b101:
            ans = 'DIVU'
        elif parsed['funct3'] == 0b110:
            ans = 'REM'
        elif parsed['funct3'] == 0b111:
            ans = 'REMU'
        if ans != '':
            print("{} {},{},{}".format(ans, parsed['rd'],
parsed['rs1'],parsed['rs2']))
    ############



if __name__ == '__main__':
    i, o = sys.argv[-2], sys.argv[-1]


    try:
        sys.stdin = open(i, 'r', encoding='utf-8')
    except Exception as e:
        print('Could not open input file!')
        exit(0)


    try:
        sys.stdout = open(o, 'w', encoding='utf-8')
    except Exception as e:
        print('Could not open output file!')
        exit(0)
```

```
    parser = ElfParser(i)
    data = parser.readByName('.text')
    start = data[1]
    print('.symtab')
    parser.readSymTab()
    pos = start
    data = data[0]
    readLOC(pos, data)
    print('.text')
    while(pos - start < len(data)):
        print('%08x' % (pos), end=' ')
        print('%10s ' % (parser.nameByPos[pos] + ':' if pos in
parser.nameByPos else ''), end='')
        ln = 4
        v = int.from_bytes(data[pos - start:pos - start + ln],
byteorder='little')
        if getSlice(v, 0, 1) != 0b11:
            ln = 2
            v = int.from_bytes(data[pos - start:pos - start + ln],
byteorder='little')
            printCmd16(v)
        else:
            printCmd32(v)
        pos += ln
    parser.inp.close()
```