САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

«ISA. Ассемблер, дизассемблер»

Выполнил: Панюхин Никита Константинович

Номер ИСУ: 334964

студ. гр. М3138

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

Задание состоит из двух частей: парсера ELF-файла и дизассемблера поднабора команд RISC-V, а именно RV32I, RV32M, RVC.

Формат ELF описывает двоичный файл с данными. Структура файла состоит из нескольких частей, не обязательно стоящих в заранее известном порядке, на каждую из которых ссылается хотя бы одна предыдущая часть. Так, например, в начале файла, помимо его типа и других опознавательных символов записана таблица, которая называется заголовком ELF-файла (ELF header, File header). В основном в ней хранится информация, относящаяся непосредственно к самому файлу в целом, а также ссылки на другие части, следующие за заголовком. Вторым по порядку разбора объектом в ELF-файле являются программные заголовки (Program header table), которые в свою очередь задают заголовки секций (Section header). Секции – области в файле, которые хранят основную информацию, в том числе программный код. Таким образом, чтобы получить нужную нам информацию из ELF-файла необходимо сначала распарсить заголовок файла, затем заголовки программ, после чего перейти к заголовкам секции и только после этого нам будут доступны адреса (offset) данных в каждой из секций файла. Отмечу, что в реализации оказалось не обязательным парсить заголовки программ, так как начало таблицы заголовков секции также закодировано в заголовке самого ELF-файла.

По заданию нам необходимы лишь две секции – .text и .symtab, однако для получения текстовых значений также пригодится секция .strtab. Описание формата кодирования полей в заголовках секции здесь приведено не будет, его легко найти в интернете. После получения всех трёх таблиц и преобразования их в удобно читаемые и используемые объекты ЯП, можно вывести таблицу .symtab в требуемом

формате и приступить к выполнению второй части задания – декодированию секции .text, содержащую команды RISC-V.

Для данной работы была использована спецификация RISC-V версии 2.2, в которой описаны все необходимые инструкции, а именно как они хранятся и кодируются. Например, на странице 104 указаны инструкции набора RV32I. Каждая такая инструкция имеет фиксированную длину (32 бита для RV32 и 16 бит для RVC) и состоит из блоков, каждый из которых подробно описан в спецификации. Например, блоки “rd” или “rsN” обозначают регистры, а блоки “imm” и “nzimm” – константы. Константы в инструкциях могут быть записаны не одной последовательностью бит или биты могут стоять не в нужном порядке. В таком случае в спецификации указано, как нужно переставить биты местами, чтобы получилось нужное число. Например, imm[20|10:1|11|19:12] означает, что данные кодируют сначала 20 бит числа, зачем с 10 по 1, затем 11 и так далее.

После расшифровки инструкций по условию задания создаются недостающие в .symtab метки и выводится результат в указанном формате.

Практическая часть

Программа-транслятор была написана на языке Python 3.

В основном процесс написания программы повторяет указанные в теоретической части и спецификации действия. Чтобы сократить количество кода программы сделать её более простой, все инструкции были поделены на несколько типов. Не обязательно советующих указанным в спецификации типам. Таким каждая инструкция принадлежит одной из групп, а внутри одной группы инструкции в основном не различаются. Группы и инструкции хранятся в текстовых файлах RV32.txt и RVC.txt для соответствующих поднаборов RISC-V из условия. При инициализации программа читает эти файлы, обрабатывает их и сохраняет в удобном для себя формате структур ЯП. Каждая группа инструкций имеет свои правила вывода и обработки параметров, например знаковых или беззнаковых целых чисел, имена регистров и т.п.

Листинг

disassembler.py

```
#
#                                                    ┌──────────────────────────────────────┐
#                                                    │   Copyright (c) 2021 Nikita Paniukhin│
#                                                    │     Licensed under the MIT license   │
#                                                    └──────────────────────────────────────┘
#
#
#
# ============================================================================================
============================

import sys
import os

sys.path.append("src")

from instructions import Instruction
from section_constants import *
from symbol_constants import *
from asserts import *
from utils import *

# ============================================================================================
============================


class Program:
    def __init__(self, data=None):
        # Segment type
        self.p_type = data[0x00:0x04] if data is not None else None

        # Offset of the segment in the file image
        self.p_offset = bytes2int(data[0x04:0x08]) if data is not None else None

        # Virtual address of the segment in memory
        self.p_vaddr = data[0x08:0x0C] if data is not None else None

        # Segment's physical address
        self.p_paddr = data[0x0C:0x10] if data is not None else None

        # Size in bytes of the segment in the file image. May be 0
        self.p_filesz = bytes2int(data[0x10:0x14]) if data is not None else None

        # Size in bytes of the segment in memory. May be 0
        self.p_memsz = bytes2int(data[0x14:0x18]) if data is not None else None

        # Segment-dependent flags (position for 32-bit structure)
        self.p_flags = data[0x18:0x1C] if data is not None else None

        # Alignment
        self.p_align = data[0x1C:0x20] if data is not None else None


class Section:
    def __init__(self, data=None):
        # This code is PEP8 compliant but unreadable
        self.sh_name = data[0x00:0x04] if data is not None else None
        self.sh_type = data[0x04:0x08] if data is not None else None
        self.sh_flags = data[0x08:0x0C] if data is not None else None
        self.sh_addr = data[0x0C:0x10] if data is not None else None
        self.sh_offset = data[0x10:0x14] if data is not None else None
        self.sh_size = data[0x14:0x18] if data is not None else None
```

```
        self.sh_link = data[0x18:0x1C] if data is not None else None
        self.sh_info = data[0x1C:0x20] if data is not None else None
        self.sh_addralign = data[0x20:0x24] if data is not None else None
        self.sh_entsize = data[0x24:0x28] if data is not None else None

        self.sh_name = bytes2int(self.sh_name)
        self.sh_type = bytes2int(self.sh_type)
        self.sh_addr = bytes2int(self.sh_addr)
        self.sh_offset = bytes2int(self.sh_offset)
        self.sh_size = bytes2int(self.sh_size)


class Elf32_Sym:
    def __init__(self, data=None):
        self.st_name = bytes2int(data[0x00:0x04])  # Elf32_Word
        self.st_value = bytes2int(data[0x04:0x8])  # Elf32_Addr
        self.st_size = bytes2int(data[0x8:0x0C])  # Elf32_Word
        self.st_info = bytes2int(data[0x0C:0x0D])  # unsigned char
        self.st_other = bytes2int(data[0x0D:0x0E])  # unsigned char
        self.st_shndx = bytes2int(data[0x0E:0x10])  # Elf32_Half

        self.st_bind = self.st_info >> 4
        self.st_type = self.st_info & 0xF
        self.st_info = (self.st_bind << 4) + (self.st_type & 0xF)

        self.st_visibility = self.st_other & 0x3


Elf32_Sym_SIZE = 0x10

#
============================================================================================
============================


def strtab_extract(data, strtab, offset):
    string_pos = strtab.sh_offset + offset
    string = ""
    while data[string_pos + len(string)] != 0x00:
        string += chr(data[string_pos + len(string)])
    return string


def parse(input_path, output_path):
    input_path, output_path = mkpath(input_path), mkpath(output_path)

    print("Parsing \"{}\"...".format(input_path))
    assert_cond(os.path.isfile(input_path), "File not found")

    with open(input_path, 'rb') as fin:
        data = fin.read()

    # ================================================== ELF HEADER
==================================================

    assert_equal(data[0x00], 0x7F, "ELF file not detected")
    assert_equal(data[0x01], ord('E'), "ELF file not detected")
    assert_equal(data[0x02], ord('L'), "ELF file not detected")
    assert_equal(data[0x03], ord('F'), "ELF file not detected")

    # data[04] = {1: 32-bit, 2: 64-bit}
    assert_equal(data[0x04], 1, "Should be 32-bit elf file")

    # data[05] = {1: little-endian, 2: big-endian}
```

```
    assert_equal(data[0x05], 1, "Should be coded in little-endian")

    # data[06] = Version (always 1)
    assert_equal(data[0x06], 1)

    # data[07-08] = ABI
    skip(data[0x07])
    skip(data[0x08])

    # data[09-0F] = Unused, should be 0
    assert_all_equal(data[0x09:0x0F], 0)

    # data[10-11] = File type
    skip(data[0x10])
    skip(data[0x11])

    # data[12-13] = Instruction set architecture
    skip(data[0x12:0x14])

    # data[14-17] = Elf version
    e_version = data[0x14:0x18]
    # assert_equal(e_version, b'\x01\x00\x00\x00', "Warning: elf version {} !=
1".format(e_version))

    # data[18-1B] = Memory address of the entry point
    e_entry = data[0x18:0x1C]

    # data[1C-1F] = Program header offset (for 32-bit = 0x34 = 52)
    e_phoff = bytes2int(data[0x1C:0x1F])
    assert_equal(e_phoff, 52, "Warning: program header not after file header for 32-bit")

    # data[20-23] = Section header offset
    e_shoff = bytes2int(data[0x20:0x24])

    # data[24-27] = Smth, depends on the target architecture
    skip(data[0x24:0x28])

    # data[28-29] = Size of this header (for 32-bit = 0x34 = 52)
    e_ehsize = bytes2int(data[0x28:0x29])
    assert_equal(e_ehsize, 52, "Warning: elf header size normally should be 52 bytes, not
{}".format(e_ehsize))

    # data[2A-2B] = Size of a program header
    e_phentsize = bytes2int(data[0x2A:0x2C])
    assert_equal(e_phentsize, 32, "Warning: program header size normally should be 32
bytes, not {}".format(e_phentsize))

    # data[2C-2D] = Number of entries in the program header
    e_phnum = bytes2int(data[0x2C:0x2E])

    # data[2E-2F] = Size of a section header
    e_shentsize = bytes2int(data[0x2E:0x30])
    assert_equal(e_shentsize, 0x28, "Warning: can only parse sections with size = {value2},
got size = {value1}")

    # data[30-31] = Number of entries in the section header
    e_shnum = bytes2int(data[0x30:0x32])

    # data[32-33] = Index of the section header that contains the section names
    e_shstrndx = bytes2int(data[0x32:0x34])

    # ================================================ PROGRAM HEADER
================================================
```

```
    # offset = e_phoff
    # programs = []
    # for _ in range(e_phnum):
    #     programs.append(Program(data[offset:offset + e_phentsize]))
    #     offset += e_phentsize

    # del offset

    # ================================================== SECTIONS
==================================================

    unmapped_sections = []
    offset = e_shoff
    for _ in range(e_shnum):
        unmapped_sections.append(Section(data[offset:offset + e_shentsize]))
        offset += e_shentsize

    # Finding .strtab by type (SHT_STRTAB):
    strtab = None
    for section in unmapped_sections:
        if section.sh_type == SHT_STRTAB:
            name_pos = section.sh_offset + section.sh_name
            if data[name_pos:name_pos + len(".shstrtab")] == b".shstrtab":
                strtab = section
                break

    assert_cond(strtab is not None, "Can not find .strtab")

    # Assign a name to every section:
    sections = {strtab_extract(data, strtab, section.sh_name): section for section in
unmapped_sections}

    assert_cond(".text" in sections, "Can not find .text")
    assert_cond(".strtab" in sections, "Can not find .strtab")
    assert_cond(".symtab" in sections, "Can not find .symtab")

    del unmapped_sections, offset, strtab, name_pos

    # ================================================== SYMTAB
==================================================

    symtab = sections[".symtab"]
    symbols = []
    for symbol_offset in range(symtab.sh_offset, symtab.sh_offset + symtab.sh_size,
Elf32_Sym_SIZE):
        symbol = Elf32_Sym(data[symbol_offset:symbol_offset + Elf32_Sym_SIZE])
        symbol.st_name = strtab_extract(data, sections[".strtab"], symbol.st_name)
        symbols.append(symbol)

    # ================================================== TEXT
==================================================

    labels = [0, {}]
    for symbol in symbols:
        if symbol.st_type == STT_FUNC and symbol.st_name:
            labels[1][symbol.st_value] = symbol.st_name

    instructions = []

    offset = sections[".text"].sh_offset
    while offset < sections[".text"].sh_offset + sections[".text"].sh_size:

        instruction_size = 4 if int2bits(data[offset]).endswith("11") else 2
```

```
        instruction = Instruction(
            sections[".text"].sh_addr + offset - sections[".text"].sh_offset,
            bytes2bits(data[offset:offset + instruction_size]).zfill(instruction_size * 8),
            labels  # Reference to `labels`
        )

        instructions.append(instruction)
        offset += instruction_size

    # ================================================== RESULT
==================================================

    with open(output_path, 'w', encoding="utf-8") as fout:
        # print("; формат строк указан по правилам printf (Си)", file=fout)
        print(".text", file=fout)
        # print("; строки оформляются в следующем формате", file=fout)
        # print("; с меткой: \"%08x %10s: %s %s, %s, %s\"", file=fout)
        # print("; без метки: метка является пустой строкой", file=fout)
        # print("; числа - десятичная запись", file=fout)
        # print("; load/store", file=fout)
        # print("; \"%08x %10s: %s %s, %s(%s)\"", file=fout)
        # print("; для c.addi*sp* команд sp регистр прописывается явно", file=fout)
        # print("; примеры:", file=fout)
        # print("00010078     _start: addi a0, zero, 0", file=fout)
        # print("0001007a             lui a1, 65536", file=fout)
        # print("00010080             lw a0, -24(s0)", file=fout)
        # print("00010088             c.addi4spn a0, sp, 8", file=fout)

        for instruction in instructions:
            instruction.print(file=fout)

        print(file=fout)
        # print("; между секциями text и symtab одна пустая строка", file=fout)
        print(".symtab", file=fout)
        # print("; заголовок таблицы", file=fout)
        # print("; \"%s %-15s %7s %-8s %-8s %-8s %6s %s\\n\"", file=fout)
        # print("; строки таблицы", file=fout)
        # print("; \"[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\\n\"", file=fout)
        print(
            "%s %-15s %7s %-8s %-8s %-8s %6s %s" %
            ("Symbol", "Value", "Size", "Type", "Bind", "Vis", "Index", "Name"),
            file=fout
        )
        for symbol_index, symbol in enumerate(symbols):
            print(
                "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s" %
                (
                    symbol_index, symbol.st_value, symbol.st_size,
st_type2string[symbol.st_type],
                    st_bind2string[symbol.st_bind],
st_visibility2string[symbol.st_visibility],
                    shndx2string[symbol.st_shndx] if symbol.st_shndx in shndx2string else
symbol.st_shndx,
                    symbol.st_name
                ),
                file=fout
            )

    # ================================================== CLEANUP
==================================================
    # To cleanup, make variables not unused and linter happy

    del data, e_version, e_entry, e_phoff, e_shoff, e_ehsize,
    e_phentsize, e_phnum, e_shentsize, e_shnum, e_shstrndx
```

```
def main():
    print(MSG["welcome"])

    if len(sys.argv) > 1:
        input_path = mkpath(sys.argv[1].strip())
        output_path = mkpath(sys.argv[2].strip())
        print("Detected command line arguments, running parse(\"{}\",
\"{}\")...".format(input_path, output_path), end="\n\n")
        parse(input_path, output_path)

    else:
        print()
        parse(mkpath("elfs", "test1.elf"), "result.txt")


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

utils.py

```
from json import load as json_load
import os


def mkpath(*paths):
    return os.path.normpath(os.path.join(*paths))


def int2byte(a):
    return bytes([a])


def int2hex(a):
    return hex(a)[2:]


def hex2int(a):
    return int(a, 16)


def bytes2int(a):
    return int.from_bytes(a, "little")


def bytes2hex(a):
    return int2hex(bytes2int(a))


def bytes2string(a):
    return a.decode("utf-8")


def int2bits(a):
    return bin(a)[2:]


def bytes2bits(a):
    return int2bits(bytes2int(a))


def bits2int(a):
    return int(a, 2)
```

```
def skip(*args, **kwargs):
    pass


with open("msg.json" if os.path.isfile("msg.json") else mkpath("src", "msg.json"), 'r',
encoding="utf-8") as file:
    MSG = json_load(file)
```

symbol_constants.py

```
STT_NOTYPE = 0
STT_OBJECT = 1
STT_FUNC = 2
STT_SECTION = 3
STT_FILE = 4
STT_COMMON = 5
STT_TLS = 6
STT_LOOS = 10
STT_HIOS = 12
STT_LOPROC = 13
STT_SPARC_REGISTER = 13
STT_HIPROC = 15

st_type2string = {
    STT_NOTYPE: "NOTYPE",
    STT_OBJECT: "OBJECT",
    STT_FUNC: "FUNC",
    STT_SECTION: "SECTION",
    STT_FILE: "FILE",
    STT_COMMON: "COMMON",
    STT_TLS: "TLS",
    STT_LOOS: "LOOS",
    STT_HIOS: "HIOS",
    STT_LOPROC: "LOPROC",
    STT_SPARC_REGISTER: "SPARC_REGISTER",
    STT_HIPROC: "HIPROC"
}


#
============================================================================================
============================

STB_LOCAL = 0
STB_GLOBAL = 1
STB_WEAK = 2
STB_LOOS = 10
STB_HIOS = 12
STB_LOPROC = 13
STB_HIPROC = 15

st_bind2string = {
    STB_LOCAL: "LOCAL",
    STB_GLOBAL: "GLOBAL",
    STB_WEAK: "WEAK",
    STB_LOOS: "LOOS",
    STB_HIOS: "HIOS",
    STB_LOPROC: "LOPROC",
    STB_HIPROC: "HIPROC"
}
```

```
#
============================================================================================
============================

STV_DEFAULT = 0
STV_INTERNAL = 1
STV_HIDDEN = 2
STV_PROTECTED = 3
STV_EXPORTED = 4
STV_SINGLETON = 5
STV_ELIMINATE = 6

st_visibility2string = {
    STV_DEFAULT: "DEFAULT",
    STV_INTERNAL: "INTERNAL",
    STV_HIDDEN: "HIDDEN",
    STV_PROTECTED: "PROTECTED",
    STV_EXPORTED: "EXPORTED",
    STV_SINGLETON: "SINGLETON",
    STV_ELIMINATE: "ELIMINATE"
}
```

section_constants.py

```
SHT_NULL = 0x0
SHT_PROGBITS = 0x1
SHT_SYMTAB = 0x2
SHT_STRTAB = 0x3
SHT_RELA = 0x4
SHT_HASH = 0x5
SHT_DYNAMIC = 0x6
SHT_NOTE = 0x7
SHT_NOBITS = 0x8
SHT_REL = 0x9
SHT_SHLIB = 0x0A
SHT_DYNSYM = 0x0B
SHT_INIT_ARRAY = 0x0E
SHT_FINI_ARRAY = 0x0F
SHT_PREINIT_ARRAY = 0x10
SHT_GROUP = 0x11
SHT_SYMTAB_SHNDX = 0x12
SHT_NUM = 0x13
# SHT_LOOS = 0x60000000


#
============================================================================================
============================

SHN_UNDEF = 0
SHN_LORESERVE = 0xff00
SHN_LOPROC = 0xff00
SHN_BEFORE = 0xff00
SHN_AFTER = 0xff01
SHN_AMD64_LCOMMON = 0xff02
SHN_HIPROC = 0xff1f
SHN_LOOS = 0xff20
SHN_LOSUNW = 0xff3f
SHN_SUNW_IGNORE = 0xff3f
SHN_HISUNW = 0xff3f
SHN_HIOS = 0xff3f
SHN_ABS = 0xfff1
SHN_COMMON = 0xfff2
SHN_XINDEX = 0xffff
```

```
SHN_HIRESERVE = 0xffff

shndx2string = {
    SHN_UNDEF: "UNDEF",
    SHN_LORESERVE: "LORESERVE",
    SHN_LOPROC: "LOPROC",
    SHN_BEFORE: "BEFORE",
    SHN_AFTER: "AFTER",
    SHN_AMD64_LCOMMON: "AMD64_LCOMMON",
    SHN_HIPROC: "HIPROC",
    SHN_LOOS: "LOOS",
    SHN_LOSUNW: "LOSUNW",
    SHN_SUNW_IGNORE: "SUNW_IGNORE",
    SHN_HISUNW: "HISUNW",
    SHN_HIOS: "HIOS",
    SHN_ABS: "ABS",
    SHN_COMMON: "COMMON",
    SHN_XINDEX: "XINDEX",
    SHN_HIRESERVE: "HIRESERVE"
}
```

registors.py

```
REGS_BIT4_HUMAN = {
    0: "zero",
    1: "ra",
    2: "sp",
    3: "gp",
    4: "tp",
    5: "t0",
    6: "t1",
    7: "t2",
    8: "s0",
    9: "s1",
    10: "a0",
    11: "a1",
    12: "a2",
    13: "a3",
    14: "a4",
    15: "a5",
    16: "a6",
    17: "a7",
    18: "s2",
    19: "s3",
    20: "s4",
    21: "s5",
    22: "s6",
    23: "s7",
    24: "s8",
    25: "s9",
    26: "s10",
    27: "s11",
    28: "t3",
    29: "t4",
    30: "t5",
    31: "t6"
}


REGS_BIT2_HUMAN = {
    0: "s0",
    1: "s1",
    2: "a0",
    3: "a1",
```

```
    4: "a2",
    5: "a3",
    6: "a4",
    7: "a5"
}


REGS_BIT4 = list(REGS_BIT4_HUMAN.values())
REGS_BIT2 = list(REGS_BIT2_HUMAN.values())
```

instructions.py

```
from instruction_formatter import format_instruction
from traceback import print_exc
from os.path import isfile
from asserts import *
from utils import *

from registors import REGS_BIT4, REGS_BIT2
from CSR import csr2string


RV32_PATH = "RV32.txt" if isfile("RV32.txt") else mkpath("src", "RV32.txt")
RVC_PATH = "RVC.txt" if isfile("RVC.txt") else mkpath("src", "RVC.txt")

with open(RV32_PATH, 'r', encoding="utf-8") as file:
    RV32 = [line.strip().split() for line in file if not line.startswith('#') and
line.strip()]
with open(RVC_PATH, 'r', encoding="utf-8") as file:
    RVC = [line.strip().split() for line in file if not line.startswith('#') and
line.strip()]


def remove_comments(data):
    for line_index, line in enumerate(data):
        for i in range(len(line)):
            if line[i].startswith('#'):
                del line[i:]
                break


def gen_reg_checks(source):
    checks = []

    if "!=" in source:
        i = source.find("!=") + 2
        while i < len(source) and ('0' <= source[i] <= '9' or source[i] in "{, }"):
            i += 1

        right_operand = source[source.find('!=') + 2:i]

        if right_operand.startswith('{'):
            right_operand = tuple(map(int,
right_operand.lstrip('{').rstrip('}').split(',')))
            checks.append(lambda x, right_operand=right_operand: x not in right_operand)
        else:
            right_operand = int(right_operand)
            checks.append(lambda x, right_operand=right_operand: x != int(right_operand))

    return tuple(checks)


def preprocess_templates(templates):
    for template_instruction in templates:
```

```
        for i, item in enumerate(template_instruction):
            if any(item.startswith(x) for x in ("imm", "uimm", "nzimm", "nzuimm")):
                imm_type, item = item.split('[', 1)
                imm = tuple(
                    tuple(map(int, x.split(':'))) if ':' in x else int(x)
                    for x in item.rstrip("]").split('|')
                )

                imm_length = 0
                for x in imm:
                    if isinstance(x, int):
                        imm_length += 1
                    else:
                        imm_length += abs(x[0] - x[1]) + 1

                template_instruction[i] = (imm, imm_length, imm_type)

            elif item.startswith("rd") or item.startswith("rs"):
                reg_size = 3 if "’" in item else 5
                checks = gen_reg_checks(item)
                template_instruction[i] = ("REG", reg_size, checks)

            elif item.startswith("int"):
                num, size = map(int, item.lstrip("int").lstrip('(').rstrip(')').split(','))
                template_instruction[i] = (
                    "INT",
                    int(size),
                    tuple([lambda x, num=num: x == int(num)])
                )


remove_comments(RV32)
preprocess_templates(RV32)

remove_comments(RVC)
preprocess_templates(RVC)


class Instruction:
    def __init__(self, addr, source, labels):
        '''
        Params:
            addr, source, labels (lables should be passed by-reference)

        Attributes:
            addr   - [int] instruction address
            source - [str] source bits
            labels
            uknown - [bool] if command is unknown
            data
            name   - [str] command name
            type   - [str] command type
        '''

        self.addr = addr
        self.labels = labels
        self.source = source

        if source.endswith("11"):
            instructions = RV32
            self.command_size = 4
        else:
            instructions = RVC
            self.command_size = 2
```

```
        self.unknown = True
        last_match_template = None
        for template_instruction in instructions:
            try:
                match = self.parse(source, template_instruction)
            except Exception as e:
                print_exc()
                skip(e)
                continue

            if match is not None:
                assert_cond(
                    self.unknown,
                    "One instruction (#{}) ({}) refers to multiple templates: {} and
{}".format(
                        int2hex(self.addr).zfill(8), source, last_match_template,
template_instruction
                    )
                )
                self.unknown = False
                last_match_template = template_instruction
                self.data, self.name, self.type = match

        if self.unknown:
            print("Instruction(#{:08x}) does not match any template: {}".format(self.addr,
source))

    def parse(self, source, template):
        type = template[-1]
        name = template[-2]
        data = []

        imm_parts = []
        imm_size = 0
        imm_type = None

        cur_pos = 0
        for item in template[:-2]:

            # REG
            if isinstance(item, tuple) and item[0] == "REG":
                _, reg_size, checks = item

                reg = bits2int(source[cur_pos:cur_pos + reg_size])
                if not all(check(reg) for check in checks):
                    return None

                data.append(REGS_BIT2[reg] if reg_size == 3 else REGS_BIT4[reg])
                cur_pos += reg_size

            # REG
            elif isinstance(item, tuple) and item[0] == "INT":
                _, int_size, checks = item

                num = bits2int(source[cur_pos:cur_pos + int_size])
                if not all(check(num) for check in checks):
                    return None

                data.append(num)
                cur_pos += int_size

            # Imm
            elif isinstance(item, tuple):
```

```
            imm, cur_imm_length, imm_type = item

            for x in imm:
                if isinstance(x, int):
                    imm_size = max(imm_size, x)
                else:
                    imm_size = max(imm_size, *map(int, x))

            imm_parts.append((imm, source[cur_pos:cur_pos + cur_imm_length]))
            data.append("imm")
            cur_pos += cur_imm_length

        # Const
        elif all('0' <= i <= '9' for i in item):
            if item != source[cur_pos:cur_pos + len(item)]:
                return None

            data.append(item)
            cur_pos += len(item)

        # Unsigned imm
        elif item == "zimm":
            data.append(bits2int(source[cur_pos:cur_pos + 5]))
            cur_pos += 5

        # Const
        elif item == "shamt":
            data.append(bits2int(source[cur_pos:cur_pos + 5]))
            cur_pos += 5

        # CSR const
        elif item == "csr":
            if bits2int(source[cur_pos:cur_pos + 12]) not in csr2string:
                return None
            data.append(csr2string[bits2int(source[cur_pos:cur_pos + 12])])
            cur_pos += 12

    if imm_parts:
        imm_size += 1
        imm = [0] * imm_size

        for imm_template, imm_source in imm_parts:
            cur_pos = 0
            for x in imm_template:
                if isinstance(x, int):
                    imm[-x - 1] = imm_source[cur_pos]
                    cur_pos += 1
                else:
                    for i in range(x[0], x[1] - 1, -1):
                        imm[-i - 1] = imm_source[cur_pos]
                        cur_pos += 1

        imm = "".join(map(str, imm))

        if 'u' not in imm_type and imm[0] == '1':  # Negative integers
            imm = int(imm, 2) - (1 << imm_size)
        else:
            imm = int(imm, 2)

        if type in ("J", "CJ", "B", "CB"):
            imm += self.addr

        for i, item in enumerate(data):
            if item == "imm":
```

```
                    data[i] = imm

        if type in ("J", "B"):
            if data[0] not in self.labels[1]:
                self.labels[1][data[0]] = "LOC_%05x" % (self.labels[0])
                data[0] = "LOC_%05x" % (self.labels[0])
                self.labels[0] += 1

        elif type in ("CB", "CJ"):
            if data[1] not in self.labels[1]:
                self.labels[1][data[1]] = "LOC_%05x" % (self.labels[0])
                data[1] = "LOC_%05x" % (self.labels[0])
                self.labels[0] += 1

        return data, name, type

    def __str__(self):
        return format_instruction(self)

    def print(self, *args, **kwargs):
        return print(self, *args, **kwargs)
```

instruction_formatter.py

```
unmatched_label_count = 0


def format_instruction(instruction):
    label = find_label(instruction, instruction.addr)

    if instruction.unknown:
        return "%08x %11s %s" % (
            instruction.addr,
            label + ':' if label else '',
            "unknown_command"
        )

    if instruction.type == "I":
        return "%08x %11s %s %s, %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[3],
            instruction.data[1],
            instruction.data[0]
        )

    if instruction.type == "J":
        target_label = find_label(instruction, instruction.data[0])
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[1],
            instruction.data[0] if target_label is None else target_label
        )

    if instruction.type in ("JR", "I-load/store"):
        return "%08x %11s %s %s, %s(%s)" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[3],
            instruction.data[0],
```

```
            instruction.data[1]
        )

    if instruction.type == "U":
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[1],
            instruction.data[0]
        )

    if instruction.type == "S-load/store":
        return "%08x %11s %s %s, %s(%s)" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[1],
            instruction.data[0],
            instruction.data[2]
        )

    if instruction.type == "R":
        return "%08x %11s %s %s, %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[4],
            instruction.data[2],
            instruction.data[1]
        )

    if instruction.type == "B":
        target_label = find_label(instruction, instruction.data[0])
        return "%08x %11s %s %s, %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[1],
            instruction.data[0] if target_label is None else target_label
        )

    if instruction.type in "C":
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[1]
        )

    if instruction.type in "CB":
        target_label = find_label(instruction, instruction.data[1])
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[1] if target_label is None else target_label
        )

    if instruction.type == "CSP":
        return "%08x %11s %s %s, %s(sp)" % (
```

```
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[1]
        )

    if instruction.type == "CSP2":
        return "%08x %11s %s sp, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[1]
        )

    if instruction.type in ("CSNG", "System"):
        return "%08x %11s %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower()
        )

    if instruction.type == "CJR":
        return "%08x %11s %s %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2]
        )

    if instruction.type == "CI2R":
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[3]
        )

    if instruction.type == "CS":
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[3],
            instruction.data[5]
        )

    if instruction.type == "CJ":
        target_label = find_label(instruction, instruction.data[1])
        return "%08x %11s %s %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[1] if target_label is None else target_label
        )

    if instruction.type == "CIW":
        return "%08x %11s %s %s, sp, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[2],
            instruction.data[1]
```

```
        )

    if instruction.type == "CI2":
        return "%08x %11s %s %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[3],
            instruction.data[1]
        )

    if instruction.type == "CL":
        return "%08x %11s %s %s, %s(%s)" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[4],
            instruction.data[1],
            instruction.data[2]
        )

    if instruction.type == "CSR":
        return "%08x %11s %s %s, %s, %s" % (
            instruction.addr,
            label + ':' if label else '',
            instruction.name.lower(),
            instruction.data[3],
            instruction.data[0],
            instruction.data[1]
        )

    return "%08x" % (instruction.addr)


def find_label(instruction, address):
    if address in instruction.labels[1]:
        return instruction.labels[1][address]
    return None
```

CSR.py

```
csr2string = {
    0x001: "fflags",
    0x002: "frm",
    0x003: "fcsr",
    0xC00: "cycle",
    0xC01: "time",
    0xC02: "instret",
    0xC80: "cycleh",
    0xC81: "timeh",
    0xC82: "instreth"
}
```

asserts.py

```
from types import GeneratorType

from utils import MSG


def assert_cond(value, error_message=MSG["assert_cond"]):
    if not value:
        exit(error_message.format(value=value))
```

```
def assert_equal(value1, value2, error_message=MSG["assert_equal"]):
    if isinstance(value2, tuple) or isinstance(value2, list) or isinstance(value2,
GeneratorType):
        assert_cond(
            all(value1 != check_value2 for check_value2 in value2),
            error_message.format(value1=value1, value2=value2)
        )
    else:
        assert_cond(value1 == value2, error_message.format(value1=value1, value2=value2))

    return value1


def assert_not_equal(value1, value2, error_message=MSG["assert_equal"]):
    if isinstance(value2, tuple) or isinstance(value2, list) or isinstance(value2,
GeneratorType):
        assert_cond(
            all(value1 == check_value2 for check_value2 in value2),
            error_message.format(value1=value1, value2=value2)
        )
    else:
        assert_cond(value1 != value2, error_message.format(value1=value1, value2=value2))

    return value1


def assert_all_equal(value1, value2, error_message=MSG["assert_equal"]):
    for check_value1 in value1:
        assert_equal(check_value1, value2, error_message)

    return value1
```

MSG.json

```
{
      "assert_equal": "Assertion error: {value1} != {value2}",
      "assert_not_equal": "Assertion error: {value1} == {value2}",
      "assert_cond": "Assertion error",

      "welcome": "Welcome to ELF disassembler. Feel free to disable any warning if needed -
just comment them."
}
```

RV32.txt

```
# RV32I Base Instruction Set                          TYPE

imm[31:12] rd 0110111 LUI                             U
imm[31:12] rd 0010111 AUIPC                           U
imm[20|10:1|11|19:12] rd 1101111 JAL                  J
imm[11:0] rs1 000 rd 1100111 JALR                     JR
imm[12|10:5] rs2 rs1 000 imm[4:1|11] 1100011 BEQ      B
imm[12|10:5] rs2 rs1 001 imm[4:1|11] 1100011 BNE      B
imm[12|10:5] rs2 rs1 100 imm[4:1|11] 1100011 BLT      B
imm[12|10:5] rs2 rs1 101 imm[4:1|11] 1100011 BGE      B
imm[12|10:5] rs2 rs1 110 imm[4:1|11] 1100011 BLTU     B
imm[12|10:5] rs2 rs1 111 imm[4:1|11] 1100011 BGEU     B
imm[11:0] rs1 000 rd 0000011 LB                       I-load/store
imm[11:0] rs1 001 rd 0000011 LH                       I-load/store
imm[11:0] rs1 010 rd 0000011 LW                       I-load/store
imm[11:0] rs1 100 rd 0000011 LBU                      I-load/store
imm[11:0] rs1 101 rd 0000011 LHU                      I-load/store
imm[11:5] rs2 rs1 000 imm[4:0] 0100011 SB             S-load/store
imm[11:5] rs2 rs1 001 imm[4:0] 0100011 SH             S-load/store
```

```
imm[11:5] rs2 rs1 010 imm[4:0] 0100011 SW           S-load/store
imm[11:0] rs1 000 rd 0010011 ADDI                   I
imm[11:0] rs1 010 rd 0010011 SLTI                   I
imm[11:0] rs1 011 rd 0010011 SLTIU                  I
imm[11:0] rs1 100 rd 0010011 XORI                   I
imm[11:0] rs1 110 rd 0010011 ORI                    I
imm[11:0] rs1 111 rd 0010011 ANDI                   I
0000000 shamt rs1 001 rd 0010011 SLLI               R
0000000 shamt rs1 101 rd 0010011 SRLI               R
0100000 shamt rs1 101 rd 0010011 SRAI               R
0000000 rs2 rs1 000 rd 0110011 ADD                  R
0100000 rs2 rs1 000 rd 0110011 SUB                  R
0000000 rs2 rs1 001 rd 0110011 SLL                  R
0000000 rs2 rs1 010 rd 0110011 SLT                  R
0000000 rs2 rs1 011 rd 0110011 SLTU                 R
0000000 rs2 rs1 100 rd 0110011 XOR                  R
0000000 rs2 rs1 101 rd 0110011 SRL                  R
0100000 rs2 rs1 101 rd 0110011 SRA                  R
0000000 rs2 rs1 110 rd 0110011 OR                   R
0000000 rs2 rs1 111 rd 0110011 AND                  R
0000 pred succ 00000 000 00000 0001111 FENCE        FENCE
0000 0000 0000 00000 001 00000 0001111 FENCE.I      FENCE
000000000000 00000 000 00000 1110011 ECALL          System
000000000001 00000 000 00000 1110011 EBREAK         System
csr rs1 001 rd 1110011 CSRRW                        CSR
csr rs1 010 rd 1110011 CSRRS                        CSR
csr rs1 011 rd 1110011 CSRRC                        CSR
csr zimm 101 rd 1110011 CSRRWI                      CSR
csr zimm 110 rd 1110011 CSRRSI                      CSR
csr zimm 111 rd 1110011 CSRRCI                      CSR



# RV32M Standard Extension                          TYPE

0000001 rs2 rs1 000 rd 0110011 MUL                  R
0000001 rs2 rs1 001 rd 0110011 MULH                 R
0000001 rs2 rs1 010 rd 0110011 MULHSU               R
0000001 rs2 rs1 011 rd 0110011 MULHU                R
0000001 rs2 rs1 100 rd 0110011 DIV                  R
0000001 rs2 rs1 101 rd 0110011 DIVU                 R
0000001 rs2 rs1 110 rd 0110011 REM                  R
0000001 rs2 rs1 111 rd 0110011 REMU                 R
```

RVC.txt

```
# Table 12.4: Instruction listing for RVC, Quadrant 0.            TYPE

# 000 0 0 00 Illegal instruction
000 nzuimm[5:4|9:6|2|3] rd’ 00 C.ADDI4SPN                         CIW
# 001 uimm[5:3] rs1’ uimm[7:6] rd’ 00 C.FLD
# 001 uimm[5:4|8] rs1’ uimm[7:6] rd’ 00 C.LQ
010 uimm[5:3] rs1’ uimm[2|6] rd’ 00 C.LW                          CL
# 011 uimm[5:3] rs1’ uimm[2|6] rd’ 00 C.FLW
# 011 uimm[5:3] rs1’ uimm[7:6] rd’ 00 C.LD
# 100 – 00 Reserved
# 101 uimm[5:3] rs1’ uimm[7:6] rs2’ 00 C.FSD
# 101 uimm[5:4|8] rs1’ uimm[7:6] rs2’ 00 C.SQ
110 uimm[5:3] rs1’ uimm[2|6] rs2’ 00 C.SW                         CL
# 111 uimm[5:3] rs1’ uimm[2|6] rs2’ 00 C.FSW
# 111 uimm[5:3] rs1’ uimm[7:6] rs2’ 00 C.SD
```

```
# Table 12.5: Instruction listing for RVC, Quadrant 1.            TYPE

000 0 int(0,5) int(0,5) 01 C.NOP                                   NOP
000 nzimm[5] rs1/rd!=0 nzimm[4:0] 01 C.ADDI                        C
001 imm[11|4|9:8|10|6|7|3:1|5] 01 C.JAL                            CJ
# 001 imm[5] rs1/rd!=0 imm[4:0] 01 C.ADDIW
010 imm[5] rd!=0 imm[4:0] 01 C.LI                                  C
011 nzimm[9] int(2,5) nzimm[4|6|8:7|5] 01 C.ADDI16SP               CSP2
011 nzimm[17] rd!={0,2} nzimm[16:12] 01 C.LUI                      C
100 nzuimm[5] 00 rs1’/rd’ nzuimm[4:0] 01 C.SRLI                    CI2
# 100 0 00 rs1’/rd’ 0 01 C.SRLI64
100 nzuimm[5] 01 rs1’/rd’ nzuimm[4:0] 01 C.SRAI                    CI2
# 100 0 01 rs1’/rd’ 0 01 C.SRAI64
100 imm[5] 10 rs1’/rd’ imm[4:0] 01 C.ANDI                          CI2
100 0 11 rs1’/rd’ 00 rs2’ 01 C.SUB                                 CS
100 0 11 rs1’/rd’ 01 rs2’ 01 C.XOR                                 CS
100 0 11 rs1’/rd’ 10 rs2’ 01 C.OR                                  CS
100 0 11 rs1’/rd’ 11 rs2’ 01 C.AND                                 CS
# 100 1 11 rs1’/rd’ 00 rs2’ 01 C.SUBW
# 100 1 11 rs1’/rd’ 01 rs2’ 01 C.ADDW
# 100 1 11 – 10 – 01 Reserved
# 100 1 11 – 11 – 01 Reserved
101 imm[11|4|9:8|10|6|7|3:1|5] 01 C.J                              CJ
110 imm[8|4:3] rs1’ imm[7:6|2:1|5] 01 C.BEQZ                       CB
111 imm[8|4:3] rs1’ imm[7:6|2:1|5] 01 C.BNEZ                       CB



# Table 12.6: Instruction listing for RVC, Quadrant 2.            TYPE

000 nzuimm[5] rs1/rd!=0 nzuimm[4:0] 10 C.SLLI                      C
# 000 0 rs1/rd!=0 0 10 C.SLLI64
# 001 uimm[5] rd uimm[4:3|8:6] 10 C.FLDSP
# 001 uimm[5] rd!=0 uimm[4|9:6] 10 C.LQSP
010 uimm[5] rd!=0 uimm[4:2|7:6] 10 C.LWSP                          CSP
# 011 uimm[5] rd uimm[4:2|7:6] 10 C.FLWSP
# 011 uimm[5] rd!=0 uimm[4:3|8:6] 10 C.LDSP
100 0 rs1!=0 int(0,5) 10 C.JR                                      CJR
100 0 rd!=0 rs2!=0 10 C.MV                                         CI2R
100 1 int(0,5) int(0,5) 10 C.EBREAK                                CSNG
100 1 rs1!=0 int(0,5) 10 C.JALR                                    CJR
100 1 rs1/rd!=0 rs2!=0 10 C.ADD                                    CI2R
# 101 uimm[5:3|8:6] rs2 10 C.FSDSP
# 101 uimm[5:4|9:6] rs2 10 C.SQSP
110 uimm[5:2|7:6] rs2 10 C.SWSP                                    CSP
# 111 uimm[5:2|7:6] rs2 10 C.FSWSP
# 111 uimm[5:3|8:6] rs2 10 C.SDSP
```