Лабораторная работа 4	Группа 05	2023
ISA	Ратьков Андрей Игоревич	

1 Инструментарий, описание, ссылки

1.1 Инструментарий

1. VS Code / PyCharm
2. Язык Python (3.11.5)
3. Typst для написания отчёта ♡

1.2 Описание

Необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код (извлеченный из elf-файла) в текст программы на языке ассемблера. Должен поддерживаться следующий набор команд RISC-V: RV32I, RV32M.

1.3 Ссылка на репозиторий и краткое изложение того, что я сделал

https://github.com/skkv-mkn/mkn-comp-arch-2023-riscv-AndrewRatkov
Сделал всё (RV32I, RV32M, symtab).

2 Реализация дизассемблера

Для работы с elf файлом создан класс `class ElfParser`.

2.1 Парсинг elf файла

В этом мне помогла эта статья на википедии: https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format.
Если взять и просто вывести все байты содержимого файла elf, получится что-то вроде `b'\x7fELF\x01\x01\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02\x00\xf3\x00\x01\x00...'` размера 1140 байт. Первые 52 байтов этого чуда — заголовок (во всех 32-битных файлах заголовок такого размера).
Парсинг происходит в функции `def __init__(self, elf_file)` класса `class ElfParser`, который лежит в файле `ElfParser.py`.

- Следующая информация из заголовка мне понадобится в дальнейшем (помним, что кодировка у нас Little Endian):
  1. В конце elf-файла находится таблица заголовков. Она содержит информацию о сегментах на которые разделён elf-файл. Её начало (то есть смещение относительно начала файла) записано в переменной `e_shoff` (от “section headers offset”) в заголовке файла с 32 до 35 байта (включительно).
  2. Каждый заголовок занимает `e_shentsize` (от “section header entire? size”) байтов, в заголовке файла эта переменная кодируется 46 и 47 байтами.
  3. Количество заголовков лежит в переменной `e_shnum` (“segment header number”), которая занимает 48 и 49 байты в заголовке.
  4. Также нам понадобится `e_shstrndx` (“segment header string index”) — индекс записи в таблице заголовков секций, описывающей таблицу названий секций

  Инициализация этих полей происходит в начале функции `def __init__(self, elf_file)`.

- Далее в поле `self.header` побайтово записывается таблица заголовков.
  Таблица заголовков содержит по заголовку размера `e_shentsize` для каждой секции.

Как вытащить необходимую информацию из заголовка? Пусть у нас есть заголовок `section_header`. Тогда

1. `sh_name = int.from_bytes(section_header[0:4], 'little')` — смещение строки, содержащей название данной секции, относительно начала таблицы названий секций. То есть, чтобы получить название секции, нужно обратиться к `sh_name`-ому байту в таблице названий секций и читать, пока не встретим нулевой байт
2. `sh_type = int.from_bytes(section_header[4:8], 'little')` — тип заголовка
3. `sh_offset = int.from_bytes(section_header[16:20], 'little')` — смещение секции от начала файла в байтах
4. `sh_size = int.from_bytes(section_header[20:24], 'little')` — размер секции в байтах

- Затем читается секция .shstrtab — таблица названий секций (Мы знаем, что это `e_shstrndx`-ая по счёту секция, поэтому несложно найти её параметры offset и size, а потом побайтово прочитать её в поле `self.shstrtab_section`)
- После этого можно прочитать нужные нам секции .text, .symtab и .strtab. Это несложно — просто перебираем все заголовки в `self.header`, смотрим на их `sh_name, sh_offset, sh_size`, восстанавливаем название секции с помощью `self.shstrtab_section` и если оно совпадает с нужным названием — читаем нужную секцию. В итоге в `self.self.text_section, self.symtab_section, self.strtab_section` побайтово записаны соответствующие секции.

Для дебага создан метод `def get_section_info(self) -> str`.
Например, при запуске на elf-файле из условия информация о секциях такая:

```
Section number: 0
 sh_type: 0
 sh_name: 0, name:
 sh_offset: 0, sh_size: 0
Section number: 1
 sh_type: 1
 sh_name: 27, name: .text
 sh_offset: 116, sh_size: 176
Section number: 2
 sh_type: 8
 sh_name: 33, name: .bss
 sh_offset: 292, sh_size: 2800
Section number: 3
 sh_type: 1
 sh_name: 38, name: .comment
 sh_offset: 292, sh_size: 18
Section number: 4
 sh_type: 1879048195
 sh_name: 47, name: .riscv.attributes
 sh_offset: 310, sh_size: 33
Section number: 5
 sh_type: 2
 sh_name: 1, name: .symtab
 sh_offset: 344, sh_size: 304
Section number: 6
 sh_type: 3
 sh_name: 9, name: .strtab
 sh_offset: 648, sh_size: 105
Section number: 7
 sh_type: 3
 sh_name: 17, name: .shstrtab
 sh_offset: 753, sh_size: 65
```

Ещё инструкции в секции `.text` разбиты на подблоки — функции `<main>,<mmul>` и блоки `<L0>,<L1>,<L2>`. Нам понадобится (для парсинга секции `.text`) словарь, который адресу сопоставляет название функции. Это будет поле `self.value2func: dict[int, str]`, заполнение этогт словаря будет также в функции `__init__`.

2.2 Парсим секцию .symtab

По информации из прошлого пункта видим, что это 5-ая секция.
Секция разделена на блоки по `self.symtab_block_size = 16` байт (значит, в исходном примере всего $304 : 16 = 19$ блоков). Покажем, как устроен каждый блок

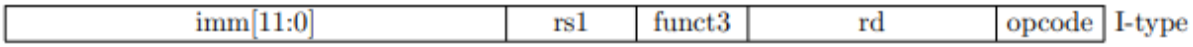
Какие байты (биты) занимает	Значению в каком столбце соответствует, смысл

0 — 3 байты	Name Смещение в таблице названий секций, откуда надо начинать читать название символа (до ближайшего нулевого байта)
4 — 7 байты	Value (число в 16-ричной системе)
8 — 11 байты	Size (целое число)
12-ый байт, первые 2 бита	Bind Восстанавливается с помощью словаря `BINDS`
12-ый байт, вторые 2 бита	Type Восстанавливается с помощью словаря `TYPES`
13-ый байт, младшие 2 бита	Vis Восстанавливается с помощью словаря `VISES`
14 — 15 байты	Index Может быть числом или, если значение особенное (см. словарь `SPECIAL_INDEXES`)

Словари `BINDS, TYPES, VISES, SPECIAL_INDEXES` лежат в файле `constants.py`.
В функции `def parse_symtab(self) -> str` распарсивается секция `.symtab`.

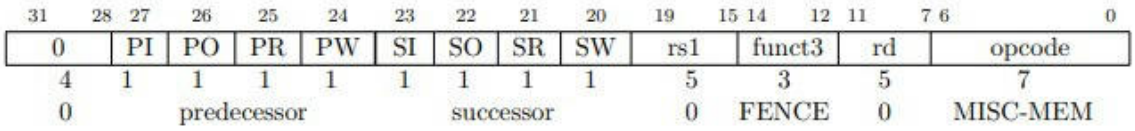
2.2.1 Парсим секцию .text

Она состоит из блоков по 4 байта (32 бита), каждый из которых кодирует какую-то команду. Адрес первой команды — `self.default_addr` — минимальный адрес среди всех блоков FUNC в таблице `.symtab`. У каждой следующей команды адрес на 4 больше. Для работы с командами создан класс `class Command32`, который лежит в файле `Command32.py`



Поскольку большинство команд, которые нам нужны, закодированы как на этой картинке, у объекта типа `Command32` заведены поля `self.opcode, self.rd, self.funct3, self.rs1, self.imm` как строчки из 0 и 1. Ещё дополнительно есть поля `self.bits` (массив из 32-ух 0 и 1 — просто код команды), и `self.reversed_bits` (тот же самый, только развёрнутый — сним иногда удобнее). Всё это определяется в методе `def __init__(self, byte_code: bytes)`.
Для дебага есть методы `def __str__(self)` и `def write_bits(self)`, они просто пишут какие-то поля на экран.
Внутри метода `def get_meaning(self) -> [str, bool, int, bool]` реализовано раскодирование (иногда частичное раскодирование) одной 32-битной команды. Выводит эта функция не `str`, а `[str, bool, int, bool]`. Это сделано потому, что некоторые команды (например `jal, bne`) вызывают какую-то функцию (адрес), но поскольку у элементов класса `Command32` нет доступа к словарю [адрес -> функция] (то, что у элементов класса `ElfParser` есть в поле `self.value2func`), то эта информация будет передана в функцию, где была вызвана эта (`get_meaning()`). Итак, на выходе — какая-то строка `str` + переменная типа `bool`, означающая, вызывается ли какой-то переменной адрес (`True`) или нет (`False`), и сам этот адрес в переменной типа `int`, если предыдущая имеет значение `True`, иначе в ней просто 0. Также ещё переменная `bool` в конце, чтобы отличать команды J и B типов (`True`, когда тип инструкции J).
Парсинг происходит довольно тупо. Просто стартуем с `offset`'а секции `.text`, берём по 4 байта, перекодируем их в команду, пользуясь табличкой с 141-144 страниц документаци для
Раскодирование команды в `self.get_meaning()` происходит так. Смотрим сначала на

self.opcode, потом на self.funct3, исходя из битов в них получаем название команды, которое записываем в переменной operation. Далее, зная, какая у нас операция, из документации можно восстановить, как в каком порядке и как лежат битики для imm, и регистров (бывают регистры rd, rs1, rs2).
Для того, чтобы по числу от 0 до 31 получить имя регистра, используется словарь reg (лежит в constants.py). В функция выводит строковое представление команды и информацию про адрес (если происходит обращение по адресу).
Практически для всех инструкций вся нужная информация находится в табличках в документации на 141-144 страницах, также я пользовался тестами, которые лежат на гугл диске и тесте из директории test_data (Главный тест, который упоминается в этом отчёте выше в качестве иллюстраций к чему-нибудь).
Для инструкции fence, насколько я понял, схема такая (первый аргумент состоит является подпоследовательностью букв “iorw”, буква “x” присутствует толгда и только тогда когда бит “PX” равен 1. Аналогично со вторым словом и битами “SX”.):



Основной парсинг текста происходит в функции def parse_text() -> str, которая просто выводит строковой представление секции .text.
В начале в переменную commands_strings записываются распарсенные команды (одна команда — одна строка). Когда распарсивается инструкция типа jal, bne и т.п. и случилось обращение по ссылке, адреса которой нет в self.value2func, в этот словарь по этому адресу добавляем метку L_i (где i — наименьшее неотрицательное, которого ещё не было). Также, в таких случаях, нужно посмотреть на 3ий аргумент, который выдала функция get_meaning(), потому что для фунций одого типа запятая нужна, а для другого нет. Так происходит парсинг команд J-типа и B-типа, в которых получается адрес: берем get_meaning() от команды, смотрим на адрес (2ой аргумент) и на тип (3 аргумент: если True, то J-тип, иначе B-тип) и просто пишем в вывод.

imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Figure 2: Команды J и B типов (первая и все кроме первой и второй соответственно) – команды, задающие сдвиг

Немного про регистры: информацию я брал из документации по ссылке:
https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf
Из следующей таблички:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 3: Регистры.

Далее, пройдясь по всем командам и записав их строковые представления в массив `commands_strings`, генерируем ответ в строке `text_string`. Перед записью каждой команды из `commands_strings` в `text_string`, предварительно проверяем, нет ли её адреса в `self.value2func` (если есть, то предварительно нужно начать новый блок, написав `text_string += "\n%08x \t<%s>\n".format(адрес, название блока)`).

2.2.2 На всякий случай, таблички со всеми распарсенными командами:

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

The RISC-V Instruction Set Manual Volume I | © RISC-V

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm	pred succ	rs1	000	rd	0001111	FENCE
1000	0011 0011	00000	000	00000	0001111	FENCE.TSO
0000	0001 0000	00000	000	00000	0001111	PAUSE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

RV32M Standard Extension						
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU

0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

2.2.3 Вывод всего этого чуда

В функции `def solve(filename) -> str` в `main.py`происходит открытие файла и вызов функций парсинга секций `.text` и `.symtab`, результаты которых склеиваются в 1 строку — ответ программы.

Основной программе остаётся только записать это в нужный выходной файл.

Figure 7: На этом пожалуй, всё. Спасибо, что дочитали до этого момента.