Мини-отчёт

Рудаков Максим, M3137

30 декабря 2023 г.

https://github.com/skkv-itmo2/itmo-comp-arch-2023-riscv-Massering

Инструментарий: Python 3.11.5.

1 Результат работы написанной программы

Реализованы и 32I, и 32M.
Пример из дополнительных тестов test2.disasm.txt:

```
.text

00010074  <register_fini>:
   10074: 00000793    addi a5, zero, 0
   10078: 00078863     beq a5, zero, 0x10088, <L0>
   1007c: 00011537     lui a0, 0x11
   10080: 94850513    addi a0, a0, -1720
   10084: 5200006f     jal zero, 0x105a4 <atexit>

00010088  <L0>:
   10088: 00008067    jalr zero, 0(ra)

0001008c  <_start>:
   1008c: 00005197   auipc gp, 0x5
   10090: dfc18193    addi gp, gp, -516
   10094: 04418513    addi a0, gp, 68
   10098: 09c18613    addi a2, gp, 156
   1009c: 40a60633     sub a2, a2, a0
   100a0: 00000593    addi a1, zero, 0
   100a4: 1a0000ef     jal ra, 0x10244 <memset>
   100a8: 00000517   auipc a0, 0x0
   100ac: 4fc50513    addi a0, a0, 1276
   100b0: 00050863     beq a0, zero, 0x100c0, <L1>
   100b4: 00001517   auipc a0, 0x1
   100b8: 89450513    addi a0, a0, -1900
   100bc: 4e8000ef     jal ra, 0x105a4 <atexit>

...
```

```
.symtab

Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x13660               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x14674               0 SECTION  LOCAL    DEFAULT       3
[   4] 0x14678               0 SECTION  LOCAL    DEFAULT       4
[   5] 0x14680               0 SECTION  LOCAL    DEFAULT       5
[   6] 0x14688               0 SECTION  LOCAL    DEFAULT       6
[   7] 0x14EB8               0 SECTION  LOCAL    DEFAULT       7
[   8] 0x14ECC               0 SECTION  LOCAL    DEFAULT       8
[   9] 0x14EE0               0 SECTION  LOCAL    DEFAULT       9
[  10] 0x0                   0 SECTION  LOCAL    DEFAULT      10
[  11] 0x0                   0 SECTION  LOCAL    DEFAULT      11
[  12] 0x0                   0 FILE     LOCAL    DEFAULT     ABS __call_atexit.c
[  13] 0x10074              24 FUNC     LOCAL    DEFAULT       1 register_fini
[  14] 0x0                   0 FILE     LOCAL    DEFAULT     ABS crtstuff.c
[  15] 0x14674               0 OBJECT   LOCAL    DEFAULT       3
[  16] 0x100D8               0 FUNC     LOCAL    DEFAULT       1 __do_global_dtors_aux
[  17] 0x14EE0               1 OBJECT   LOCAL    DEFAULT       9 completed.1
```

Вывод полностью совпадает с приведённым в файле.

2 Работа над кодом

Я по какому-то тайному наитию начал с конца. Я начал писать парсинг команд. Всю информацию брал из таблицы спецификации, приведённой в условии лабораторной (RISC-V Unprivileged ISA Specification: [PDF], Chapter 28. RV32/64G Instruction Set Listings | Page 142). Сделал функцию, которая получает набор байтов, а выдаёт что это за инструкция, по какому шаблону её выводить и какие параметры (регистры, константы) где ставятся.

Пример: допустим есть команда 0x00000793. Сразу же мы берём последние 7 бит и смотрим, что это за opcode. В нашем случае это 0010011 - OP-IMM.

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (>32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	OP-V	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥80b

Table 36. RISC-V base opcode map, inst[1:0]=11

Рис. 1: Таблица opcode'ов. Источник 1, Chapter 28. RV32/64G Instruction Set Listings | Page 141.

Тогда мы получаем составляющие команды, согласно таблице: imm, rs1, func3, rd (в программе я получаю их срезами из строки-команды). Дальше в данном случае нам надо смотреть на func3, т.к. это поле определяет, какая функция перед нами. Получаем, что функция - 000 - addi. После этого можем сформировать imm_i, нужный для вывода константы этой команды.

```
imm_i = imm[0] * 20 + imm[:12]
```

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

Рис. 2: Скриншот таблицы для иллюстрации примера. Таблица RV32I Base Instruction Set. Источник 1, Chapter 28. RV32/64G Instruction Set Listings | Page 142

Теперь передаем команду дальше (с соответствующим её формату вывода шаблоном и с именами регистров из таблицы ниже) списком токенов. Её получает функция, которая дизассмеблирует все команды в секции text и кладёт в список команд. Далее этот список передаётся в функцию вывода команд. Теперь каждая команда раскладывается на шаблон вывода и аргументы, а дальше смотрится, не использует ли команда метки. Если использует и метки с этим адресом ещё нет в списке всех меток программ, создаётся новая. Иначе просто передаётся её имя.

Теперь, после того, как мы прошли все команды и собрали все метки, мы можем начать вывод в файл. Для этого проходим по списку всех команд. Если в данном адресе есть метка, выводим её. Далее в каждый шаблон вывода команды передаётся её аргументы и мы получаем строку, которую можем вывести.

Во время выполнения я начал понимать, что у меня будут какие-то проблемы с ELF-файлом, ведь не может он весь от начала до конца состоять из команд. Так и вышло. Оказалось, что важнее было распарсить не инструкции, а файл.

Тогда я занялся файлом. С чего начать я не знал и стал читать спецификацию (ссылка на источник 4) в случайном порядке. Было больно. За один вечер я понял очень мало. Тогда я пошел читать статьи (ссылки на источники 3 и 5), а также встретился с одногруппниками и послушал, что они говорят.

Тогда у меня появилось понимание, с чего начать. Я начал разбираться в header'ах, структурах и типах elf32. Постепенно до меня доходило, как связаны разные величины из header'а и сам файл, как найти начало следующей секции и её имя. Я сделал функции, которые хорошо парсят ELF файл, а именно: его заголовок, заголовки секций, секции strtab, symtab и text.

Когда я всё это соединил, наконец применил самый первый код парсинга команд, (оказалось, что я выводил аргументы не в том порядке, так что пришлось переделать).

После этого (и прошедшего дедлайна) я переделал шаблоны для printf, потому что они почему-то были неверные (я доверял вам(( ). Стало гораздо лучше. После этого я сделал парсинг имён регистров. Это оказалось очень легко, всего лишь ~~попросить chatGPT перевести табличку в словарь json~~. Табличку ABI имён регистров нашёл в спецификации 6 на странице 59.

После этого я занялся метками. Долго не понимал, откуда брать, где они находятся. В итоге стал ставить как метки все подходящие имена из symtab (погуглил ответ). Также пришлось поотлаживать известные/неизвестные метки, чтобы знать, где писать L, а где имя. В итоге всё работает стабильно.

Затем я начал делать вывод констант. Было непонятно, почему ничего не работает. Написал собственную функцию перевода из двочиной в представлении дополнения до двух. Потом нашёл графики со стрелочками, где написано, как парсить imm. Мне понравилось переносить их в код

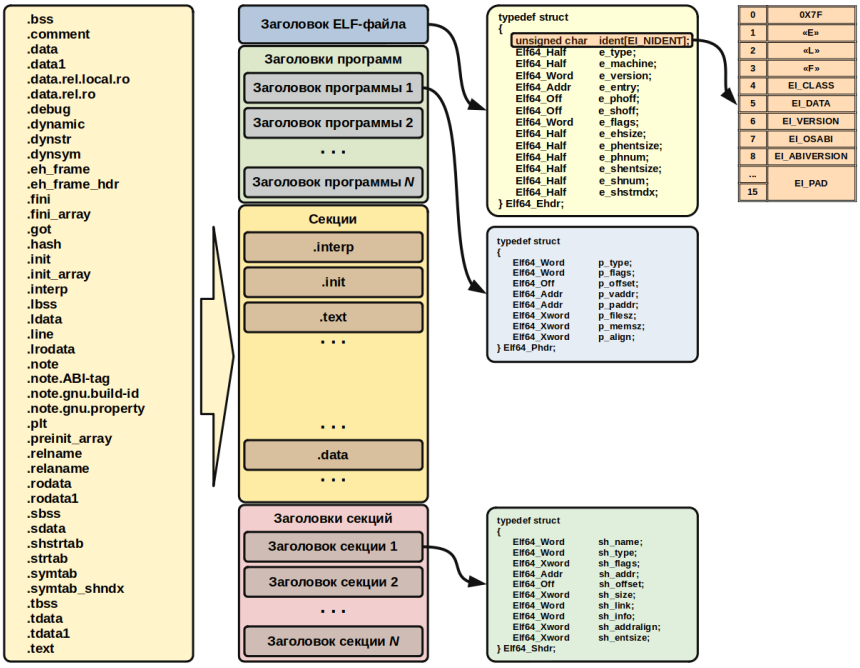

Рис. 3: Структура ELF файла, источник 3

(не считая, что я пару раз накосячил и долго дебажил). Правда непонятно, зачем это сделали так муторно (наверное потому что 32-ух бит мало и надо выкручиваться).

После этого ещё немного подформатировал вывод и у меня вывелся правильно первый файл с тестом. Тогда я скачал по ссылке ещё тестовые файлы и проверил на них. Пришлось ещё подебажить вывод констант, потому что оказалось, что imm чуть ли не в каждой команде изменяется, а не только в избранных. Также сбились метки, но это тоже наладилось с починкой констант (зря дебажил). Добавил ещё несколько команд, которых почему-то не было в спецификации в источнике 6, но были в источнике 1.

Тогда у меня стали верно выводиться все тестовые файлы.

```
typedef struct {
        unsigned char   e_ident[EI_NIDENT];
        Elf32_Half      e_type;
        Elf32_Half      e_machine;
        Elf32_Word      e_version;
        Elf32_Addr      e_entry;
        Elf32_Off       e_phoff;
        Elf32_Off       e_shoff;
        Elf32_Word      e_flags;
        Elf32_Half      e_ehsize;
        Elf32_Half      e_phentsize;
        Elf32_Half      e_phnum;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

Рис. 4: Структура заголовка ELF файла (https://refspecs.linuxbase.org/elf/gabi4+/ch4.eheader.html)

3 Ссылки на источники

1. RISC-V Unprivileged ISA Specification: [PDF] - Таблица дизассемблирования команд.
https://github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf

2. RISC-V RV32I Base Integer Instruction Set: [Webpage] - понимание, как дизассемблировать команды.
https://mpsu.github.io/APS-reborn/Other/rv32i.html

3. ELF-файлы в Linux: [Статья] - примерно понял структуру ELF файла и почему он такой. Поиграл с elfread.
https://tech-geek.ru/elf-files-linux/

4. ELF: Executable and Linkable Format: [PDF] - брал точную информацию по поводу структур заголовков, то есть какого типа каждое поле и как называются.
(https://uclibc.org/docs/elf.pdf)

5. Tool Interface Standards (TIS) Executable and Linking Format (ELF) Specification: [Webpage] - Нашёл, как парсить symtab, а именно st_info. Брал таблицы значений для symtab, которые используются в коде.
https://refspecs.linuxbase.org/elf/gabi4+/contents.html

6. RISC-V Assembly Language Programming [PDF] - самый полезный источник. Порядок аргументов команд, правила перевода imm по разным типам и какие команды какие типы используют, ABI регистры, little-endian.
https://github.com/johnwinans/rvalp

```python
def parse_elf_header(file_path: str) -> dict:
    with open(file_path, "rb") as file:
        e_ident = file.read(16)
        e_type = file.read(2)
        e_machine = file.read(2)
        e_version = file.read(4)
        e_entry = file.read(4)
        e_phoff = file.read(4)
        e_shoff = file.read(4)
        e_flags = file.read(4)
        e_ehsize = file.read(2)
        e_phentsize = file.read(2)
        e_phnum = file.read(2)
        e_shentsize = file.read(2)
        e_shnum = file.read(2)
        e_shstrndx = file.read(2)

        elf_info = {
            "e_ident": e_ident,
            "e_type": int.from_bytes(e_type, byteorder: 'little'),
            "e_machine": int.from_bytes(e_machine, byteorder: 'little'),
            "e_version": int.from_bytes(e_version, byteorder: 'little'),
            "e_entry": int.from_bytes(e_entry, byteorder: 'little'),
            "e_phoff": int.from_bytes(e_phoff, byteorder: 'little'),
            "e_shoff": int.from_bytes(e_shoff, byteorder: 'little'),
            "e_flags": int.from_bytes(e_flags, byteorder: 'little'),
            "e_ehsize": int.from_bytes(e_ehsize, byteorder: 'little'),
            "e_phentsize": int.from_bytes(e_phentsize, byteorder: 'little'),
            "e_phnum": int.from_bytes(e_phnum, byteorder: 'little'),
            "e_shentsize": int.from_bytes(e_shentsize, byteorder: 'little'),
            "e_shnum": int.from_bytes(e_shnum, byteorder: 'little'),
            "e_shstrndx": int.from_bytes(e_shstrndx, byteorder: 'little')
        }

        return elf_info
```

Рис. 5: скриншот кода, который переводит заголовок ELF файла из байт в числа по полям

Reg	ABI/Alias	Description	Saved
x0	zero	Hard-wired zero	
x1	ra	Return address	
x2	sp	Stack pointer	yes
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary/alternate link register	
x6-7	t1-2	Temporaries	
x8	s0/fp	Saved register/frame pointer	yes
x9	s1	Saved register	yes
x10-11	a0-1	Function arguments/return value	
x12-17	a2-7	Function arguments	
x18-27	s2-11	Saved registers	yes
x28-31	t3-6	Temporaries	

Рис. 6: табличка ABI имён регистров

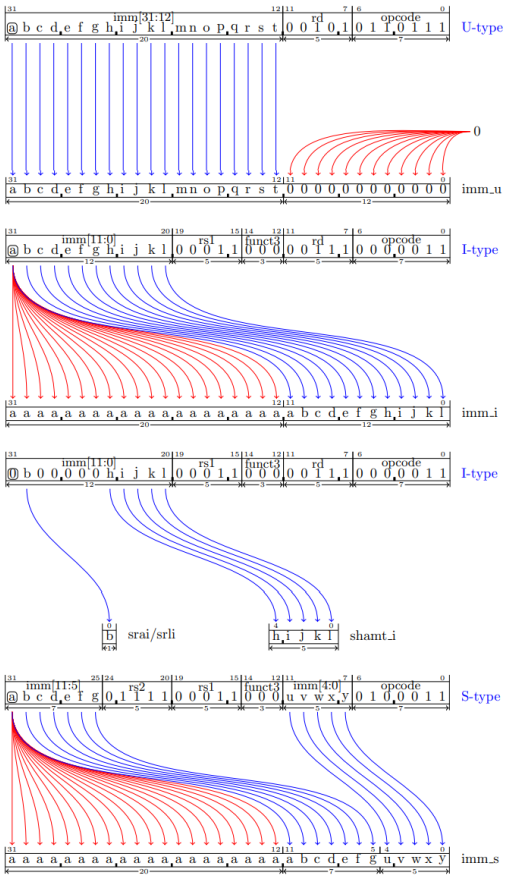

Рис. 7: графики со стрелочками, источник 6