Лабораторная работа №4

ISA

1. Бобовский Кирилл Алексеевич, M3138, ISA
2. https://github.com/skkv-itmo/itmo-comp-arch-2023-riscv-WannaSleep43/tree/main
3. Python 3.11.5
4.

.text

```
00010074 <main>:
   10074: ff010113    addi    sp, sp, -16
   10078: 00112623      sw    ra, 12(sp)
   1007c: 030000ef     jal    ra, 0x100ac <mmul>
   10080: 00c12083      lw    ra, 12(sp)
   10084: 00000513    addi    a0, zero, 0
   10088: 01010113    addi    sp, sp, 16
   1008c: 00008067    jalr    zero, 0(ra)
   10090: 00000013    addi    zero, zero, 0
   10094: 00100137     lui    sp, 0x100
   10098: fddff0ef     jal    ra, 0x10074 <main>
   1009c: 00050593    addi    a1, a0, 0
   100a0: 00a00893    addi    a7, zero, 10
   100a4: 0ff0000f    fence   iorw, iorw
   100a8: 00000073    ecall

000100ac <mmul>:
   100ac: 00011f37     lui    t5, 0x11
   100b0: 124f0513    addi    a0, t5, 292
   100b4: 65450513    addi    a0, a0, 1620
   100b8: 124f0f13    addi    t5, t5, 292
   100bc: e4018293    addi    t0, gp, -448
   100c0: fd018f93    addi    t6, gp, -48
   100c4: 02800e93    addi    t4, zero, 40

000100c8 <L2>:
   100c8: fec50e13    addi    t3, a0, -20
   100cc: 000f0313    addi    t1, t5, 0
```

```
   100d0: 000f8893   addi     a7, t6, 0
   100d4: 00000813              addi     a6, zero, 0

000100d8            <L1>:
   100d8: 00088693              addi     a3, a7, 0
   100dc: 000e0793              addi     a5, t3, 0
   100e0: 00000613              addi     a2, zero, 0

000100e4            <L0>:
   100e4: 00078703                lb     a4, 0(a5)
   100e8: 00069583                lh     a1, 0(a3)
   100ec: 00178793              addi     a5, a5, 1
   100f0: 02868693              addi     a3, a3, 40
   100f4: 02b70733                mul    a4, a4, a1
   100f8: 00e60633                add    a2, a2, a4
   100fc: fea794e3    bne      a5, a0, 0x100e4, <L0>
   10100: 00c32023                sw     a2, 0(t1)
   10104: 00280813              addi     a6, a6, 2
   10108: 00430313              addi     t1, t1, 4
   1010c: 00288893              addi     a7, a7, 2
   10110: fdd814e3    bne      a6, t4, 0x100d8, <L1>
   10114: 050f0f13   addi      t5, t5, 80
   10118: 01478513              addi     a0, a5, 20
   1011c: fa5f16e3    bne      t5, t0, 0x100c8, <L2>
   10120: 00008067              jalr     zero, 0(ra)
```

.symtab

```
Symbol Value            Size Type     Bind     Vis       Index Name
[   0] 0x0              0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074            0 SECTION  LOCAL    DEFAULT      1
[   2] 0x11124            0 SECTION  LOCAL    DEFAULT      2
[   3] 0x0              0 SECTION  LOCAL    DEFAULT      3
[   4] 0x0              0 SECTION  LOCAL    DEFAULT      4
[   5] 0x0              0 FILE     LOCAL    DEFAULT     ABS test.c
[   6] 0x11924            0 NOTYPE   GLOBAL   DEFAULT     ABS __global_pointer$
```

```
[  7] 0x118F4          800 OBJECT   GLOBAL   DEFAULT      2 b
[  8] 0x11124            0 NOTYPE   GLOBAL   DEFAULT      1 __SDATA_BEGIN__
[  9] 0x100AC          120 FUNC     GLOBAL   DEFAULT      1 mmul
[ 10] 0x0                0 NOTYPE   GLOBAL   DEFAULT   UNDEF _start
[ 11] 0x11124         1600 OBJECT   GLOBAL   DEFAULT      2 c
[ 12] 0x11C14            0 NOTYPE   GLOBAL   DEFAULT      2 __BSS_END__
[ 13] 0x11124            0 NOTYPE   GLOBAL   DEFAULT      2 __bss_start
[ 14] 0x10074           28 FUNC     GLOBAL   DEFAULT      1 main
[ 15] 0x11124            0 NOTYPE   GLOBAL   DEFAULT      1 __DATA_BEGIN__
[ 16] 0x11124            0 NOTYPE   GLOBAL   DEFAULT      1 _edata
[ 17] 0x11C14            0 NOTYPE   GLOBAL   DEFAULT      2 _end
[ 18] 0x11764          400 OBJECT   GLOBAL   DEFAULT      2 a
```

5. Был реализован RV32I, RV32M и simtab

Следующая информация и скриншоты про разбор ELF-файла взята с первого источника(ejudge).

Сначала я узнал, что первые 52 байта отводятся под заголовок файла. Структура заголовка выглядит так:

```
typedef struct
{
    unsigned char e_ident[16];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    uint32_t      e_entry;
    uint32_t      e_phoff;
    uint32_t      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} Elf32_Ehdr;
```

Я считал все эти поля. Из них я воспользовался:

Поле **e_shoff** задает смещение от начала файла до начала таблицы заголовков секций
Поле **e_phoff** задает смещение от начала файла до начала таблицы заголовков программы
Поле **e_phnum** хранит количество записей в таблице заголовков программы.
Поле **e_shentsize** хранит размер одной записи в таблице заголовков секций.
Поле **e_shnum** хранит количество записей в таблице заголовков секций.
Поле **e_shstrndx** хранит индекс заголовка секции, которая хранит имена всех секций.

Структура Elf32_Shdr определена следующим образом:

```
typedef struct
{
    uint32_t   sh_name;
    uint32_t   sh_type;
    uint32_t   sh_flags;
    uint32_t   sh_addr;
    uint32_t   sh_offset;
    uint32_t   sh_size;
    uint32_t   sh_link;
    uint32_t   sh_info;
    uint32_t   sh_addralign;
    uint32_t   sh_entsize;
} Elf32_Shdr;
```

Здесь мне понадобились поля sh_name (смещение имени от начала таблицы имен), чтобы найти «.symtab» и «.text». Для симтаба я так же запомнил sh_link, индекс секции «.strtab», чтобы потом в симтабе найти сами символы. Для «.text» я использовал sh_addr, чтобы узнать начало адрессов. Так же полезными были sh_offset (смещение секции от начала файла) и sh_size (размер секции в байтах).

Для начала я нашел секцию, хранящей имена всех остальных заголовков секций, ее начало находилось в e_shoff + e_shstrndx * 40.

Затем я прошел по всем заголовкам секций и для каждого нашел его имя. Имя секции хранилось в байтах начиная с headings[sh_offset] + section[sh_name] и я добавлял в имя символы, пока не встречал символ 0.

Пройдясь по всем заголовкам секций я нашел секции «.text» и «.simtab» и сохранил их.

Simtab

Информацию для парсинга симтаба, а также таблички и константы я взял из 3 источника.

Устройство таблицы симтаб. Здесь понадобились все поля.

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

st_name — смещение символа в таблице «.strtab»
st_value — адрес
st_size — размер
st_info — использовал для нахождения BIND и TYPE по следующим формулам (357 страница 3 источника):
#define ELF32_ST_BIND(i) ((i)>>4)
#define ELF32_ST_TYPE(i) ((i)&0xf)
st_other — использовал для нахождения VIS
#define ELF32_ST_VISIBILITY(o) ((o)&0x3)
st_shndx — index

Все эти поля я пропустил через словарики, написанные по документации(источник 3) и получил ответы на simtab.

Text

В тексте я реализовал RV32M и RV32I.

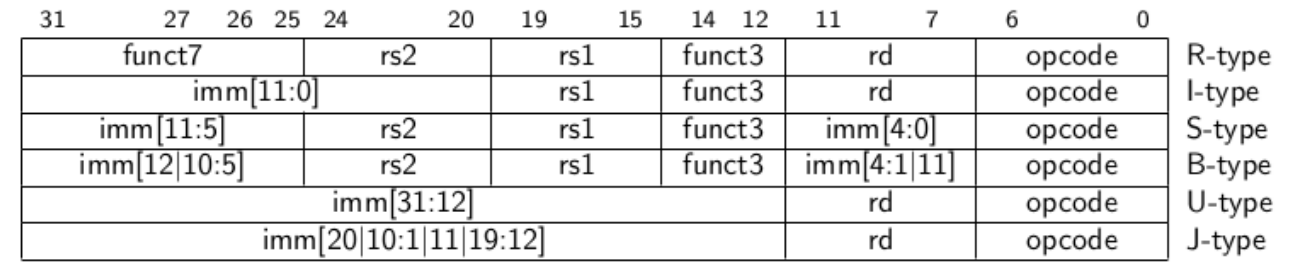
Базовая спецификация RV32I (RV — RISC-V, 32-разрядная, I означает Integer — целочисленную арифметику), содержит набор из 32 регистров и включает 39 инструкций. Используется 6 типов кодирования инструкций (форматов).

Каждая комадна в тексте, которую нужно реализовать, занимает 4 байта. Я нашел секцию text и считал все команды. Каждую команду я перевел в 32 бита и дальше работал с битами. Биты я разбил на нужные поля по документации.

За тип инструкции здесь отвечает opcode, иногда еще нужно учитывать funct3 и funct7.

Я посмотрел в спецификацию(приложение 2) и для каждой инструкции по opcode определил ее тип, а дальше я распарсил каждый тип в соответствии с документацией.

Например, мне понадобилось реализовать I-type, я просто разбил 32 бита на соответствующие им части:



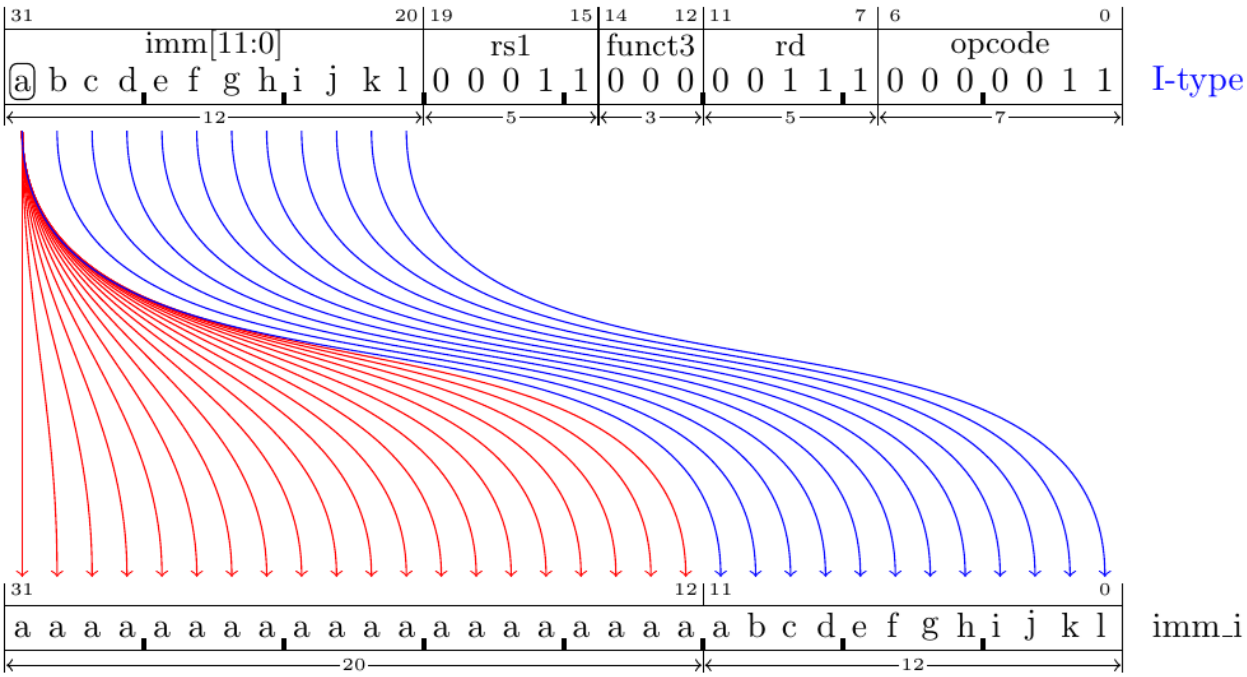
Для того, чтобы лучше понять процесс парсинга текста приведу конкретный пример. Например, нам дана команда

[1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]

Сначала я смотрю на младшие 7 бит(опкод) и по табличке из 4 источника определяю, что это инструкция I-type из RV32I. Затем я смотрю на биты с 12 по 14 (0,0,0) и понимаю, что это инструкция ADD. Дальше я разбиваю всю команду на соответствующие блоки(rd, rs1, rs2) и собираю все это в ответ.

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
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

Лучшее понимание о том, как получается imm в каждом конкретном типе я получил по четвертому источнику, например для I-type было полезным посмотреть на этот снимок:



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

Адреса в text — это sh_addr (от текста) + 4 * (индекс секции)

rd, rs1 и rs2 я так же перевел в соответствующие аргументы согласно документации.

Источники:

https://ejudge.ru/study/3sem/elf.html — Здесь я прочитал про форматы ELF файлов и то, как их нужно парсить. (первый источник)

https://five-embeddev.com/riscv-isa-manual/latest/instr-table.html — Здесь я взял основные наборы инструкций RV32I и RV32M и посмотрел, как нужно парсить отдельные типы инструкций. (второй источник)

https://docs.oracle.com/cd/E26502_01/pdf/E26507.pdf — Отсюда я взял основную информацию для разбора симтаба (третий источник)

https://refspecs.linuxfoundation.org/elf/elf.pdf - Так же здесь было достаточно много полезной информации по устройству ELF файла в целом

https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view — Здесь на 26 странице я взял информацию про то, как нужно парсить fence инструкции

https://github.com/johnwinans/rvalp/releases/download/v0.18.1/rvalp.pdf — Тут я получил какое-то понимание о том, как нужно парсить каждый конкретный тип(53 страница I type, 56 S type, 57 B type, 51 R type, 50 J type, 49 U type) (четвертый источник)