

ФИО	Группа	Работа
Пластинин Алексей Александрович	37	ISA

<https://github.com/skkv-itmo2/itmo-comp-arch-2023-riscv-GooddiLK>

C++20 g++

Вывод на тестовых данных из репозитория и гугл диска совпадает с точностью до символа '\t' перед каждым переводом строки.

```

1      .text
2
3      00010074      <main>:
4          10074:      ff010113          addi sp, sp, -16
5          10078:      00112623          sw ra, 12(sp)
6          1007c:      030000ef          jal ra, 0x100ac <mmul>
7          10080:      00c12083          lw ra, 12(sp)
8          10084:      00000513          addi a0, zero, 0
9          10088:      01010113          addi sp, sp, 16
10         1008c:      00008067          jalr zero, 0(ra)
11         10090:      00000013          addi zero, zero, 0
12         10094:      00100137          lui sp, 0x100
13         10098:      fddff0ef          jal ra, 0x10074 <main>
14         1009c:      00050593          addi a1, a0, 0
15         100a0:      00a00893          addi a7, zero, 10
16         100a4:      0ff0000f          fence iorw, iorw
17         100a8:      00000073          ecall
18
19      000100ac      <mmul>:
20          100ac:      00011f37          lui t5, 0x11
21          100b0:      124f0513          addi a0, t5, 292
22          100b4:      65450513          addi a0, a0, 1620
23          100b8:      124f0f13          addi t5, t5, 292
24          100bc:      e4018293          addi t0, gp, -448
25          100c0:      fd018f93          addi t6, gp, -48
26          100c4:      02800e93          addi t4, zero, 40
27
28      000100c8      <L2>:
29          100c8:      fec50e13          addi t3, a0, -20
30          100cc:      000f0313          addi t1, t5, 0
31          100d0:      000f8893          addi a7, t6, 0
32          100d4:      00000813          addi a6, zero, 0
33
34      000100d8      <L1>:
35          100d8:      00088693          addi a3, a7, 0
36          100dc:      000e0793          addi a5, t3, 0
37          100e0:      00000613          addi a2, zero, 0

```

```

38
39      000100e4      <L0>:
40      100e4:      00078703          lb a4, 0(a5)
41      100e8:      00069583          lh a1, 0(a3)
42      100ec:      00178793          addi a5, a5, 1
43      100f0:      02868693          addi a3, a3, 40
44      100f4:      02b70733          mul a4, a4, a1
45      100f8:      00e60633          add a2, a2, a4
46      100fc:      fea794e3          bne a5, a0, 0x100e4, <L0>
47      10100:      00c32023          sw a2, 0(t1)
48      10104:      00280813          addi a6, a6, 2
49      10108:      00430313          addi t1, t1, 4
50      1010c:      00288893          addi a7, a7, 2
51      10110:      fdd814e3          bne a6, t4, 0x100d8, <L1>
52      10114:      050f0f13          addi t5, t5, 80
53      10118:      01478513          addi a0, a5, 20
54      1011c:      fa5f16e3          bne t5, t0, 0x100c8, <L2>
55      10120:      00008067          jalr zero, 0(ra)
56
57

```

```

58      .symtab

```

```

59

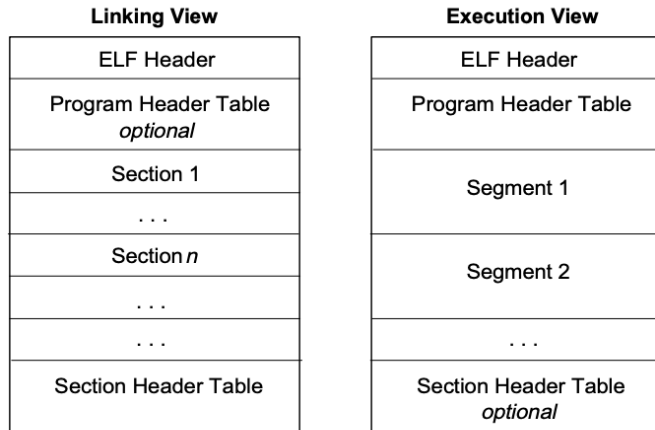
```

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

Теория:

Структура ELF файла (источник 4 страница 15) здесь и далее страница указана от начала файла, а не по существующей нумерации:

Figure 1-1. Object File Format



OSD1980

Далее будем опираться на Linked View. ELF Header содержит информацию о организации файла. Section Header Table содержит информацию о секциях в файле. Секции содержат информацию необходимую для сборки и исполнения программы.

Структура ELF Header (источник 4 страница 18):

Figure 1-3. ELF Header

```
#define EI_NIDENT          16

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;
```

Нас будут интересовать следующие данные:

`e_shnum` - количество записей в Section Header Table (SHT далее).

`e_shoff` - смещение SHT от начала файла.

e_shentsize - размер записи section header в SHT.

e_shstrndx - индекс в SHT отвечающий за section name string table (strtab).

Типы и размеры данных используемых в ELF файле (источник 4 страница 16):

Figure 1-2. 32-Bit Data Types

Name	Size	Alignment	Purpose
Elf32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
unsigned char	1	1	Unsigned small integer

Section Header Table содержит заголовки секций. Их структура (источник 4 страница 24):

Figure 1-8. Section Header

```
typedef struct {
    Elf32_Word  sh_name;
    Elf32_Word  sh_type;
    Elf32_Word  sh_flags;
    Elf32_Addr  sh_addr;
    Elf32_Off   sh_offset;
    Elf32_Word  sh_size;
    Elf32_Word  sh_link;
    Elf32_Word  sh_info;
    Elf32_Word  sh_addralign;
    Elf32_Word  sh_entsize;
} Elf32_Shdr;
```

Из них мы будем использовать:

sh_offset - смещение от начала файла до начала секции.

sh_name - индекс в section header string table с которого начинается название секции.

sh_size - размер секции.

sh_addr - индекс первого байта секции, если секция будет исполняться (отображаться в области памяти процесса).

Symbol Table section содержит информацию (таблицу символов), необходимую для определения местоположения и перемещения символических определений и ссылок программы.

Секция .symtab содержит массив следующих структур (источник 4 стр. 32):

Figure 1-15. Symbol Table Entry

```
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	This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names.
st_value	This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so on; details appear below.
st_size	Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.
st_info	This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.
st_other	This member currently holds 0 and has no defined meaning.
st_shndx	Every symbol table entry is "defined" in relation to some section; this member holds the relevant section header table index. As Figure 1-7 and the related text describe, some section indexes indicate special meanings.

Секция .text содержит информацию о исполняемых инструкциях программы. Размер каждой команды 4 байта. Реализован разбор команд из наборов RV32I, RV32M, RV32A. Команды в них принадлежат одному из 6 следующих типов (источник 1 страница 30):

31	30	25	24	21	20	19	15	14	12	11	8	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]	imm[10:5]			rs2		rs1	funct3		imm[4:1]	imm[11]	opcode		B-type		
imm[31:12]										rd			opcode		U-type
imm[20]		imm[10:1]			imm[11]		imm[19:12]			rd			opcode		J-type

Opcode, funct3, funct7 используются для определения типа команды. rd - регистр назначения, rs1, rs2 - регистры источника. Imm определяют смещение / адрес назначения.

Информация о отображении регистров (источник 2 страница 155):

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	t0	Temporary/alternate link register	Caller
x6–7	t1–2	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
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Информация о определении значения Imm (источник 1 страница 31):

31	30	20	19	12	11	10	5	4	1	0	
— inst[31] —						inst[30:25]	inst[24:21]	inst[20]	I-immediate		
— inst[31] —						inst[30:25]	inst[11:8]	inst[7]	S-immediate		
— inst[31] —					inst[7]	inst[30:25]	inst[11:8]	0	B-immediate		
inst[31]	inst[30:20]			inst[19:12]		— 0 —					U-immediate
— inst[31] —				inst[19:12]		inst[20]	inst[30:25]	inst[24:21]	0	J-immediate	

Более наглядный пример для определения J-Imm:

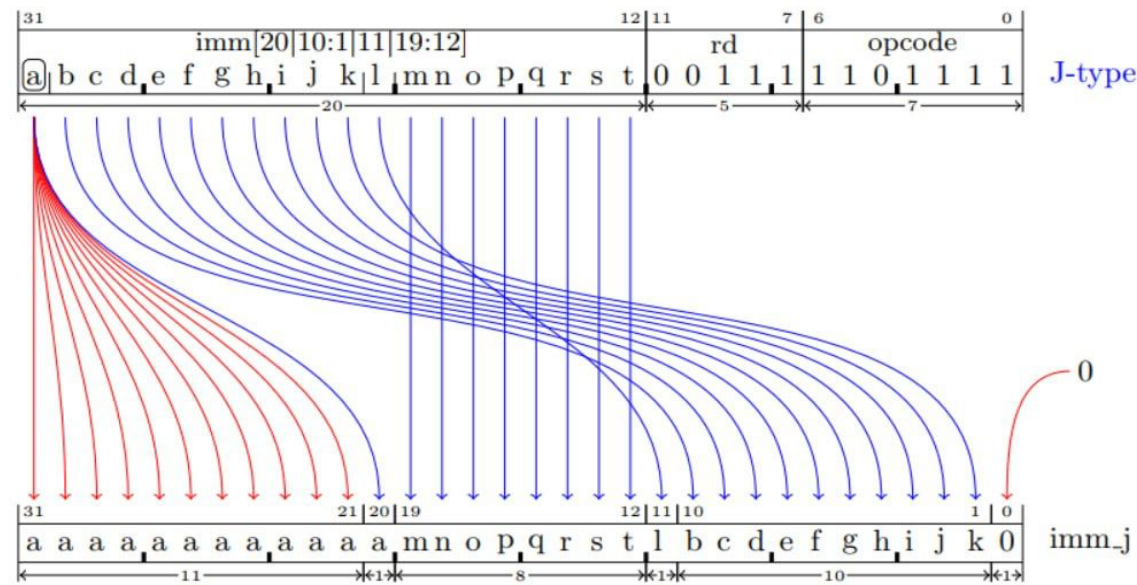


Figure 5.7: Decoding a J-type instruction.

Реализованны команды (источник 2 страница 148):

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	
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
000000000000		00000	000	00000	1110011	ECALL	
000000000001		00000	000	00000	1110011	EBREAK	

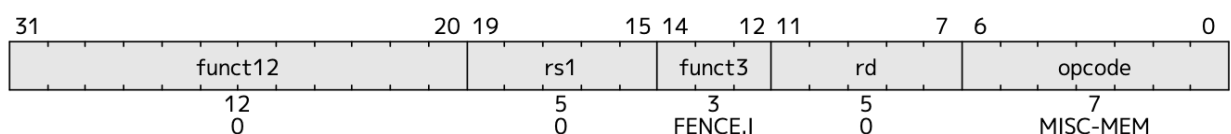
RV32M Standard Extension

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

RV32A Standard Extension

00010	aq	rl	00000	rs1	010	rd	0101111	LR.W
00011	aq	rl	rs2	rs1	010	rd	0101111	SC.W
00001	aq	rl	rs2	rs1	010	rd	0101111	AMOSWAP.W
00000	aq	rl	rs2	rs1	010	rd	0101111	AMOADD.W
00100	aq	rl	rs2	rs1	010	rd	0101111	AMOXOR.W
01100	aq	rl	rs2	rs1	010	rd	0101111	AMOAND.W
01000	aq	rl	rs2	rs1	010	rd	0101111	AMOOR.W
10000	aq	rl	rs2	rs1	010	rd	0101111	AMOMIN.W
10100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAX.W
11000	aq	rl	rs2	rs1	010	rd	0101111	AMOMINU.W
11100	aq	rl	rs2	rs1	010	rd	0101111	AMOMAXU.W

Расширение Zifencei (источник 1 страница 46) включает в себя инструкцию fence.i:



Расширение Zihintpause (источник 1 страница 52) включает в себя специальное действие, если части fence инструкции принимают следующие значения: pred = W, succ = 0, fm = 0, rd = x0, и rs1 = x0.

Детали реализации:

Весь файл записывается в `vector<int>`.

В начале парсится header (parse_header.cpp).

```

3  void read_header(std::vector<int>& file, Header& header) {
4      int ptr = 0;
5      for (int i = 0; i < EI_NIDENT; i++) {
6          header.e_ident[i] = file[ptr++];
7      }
8      header.e_type = read_half(file, ptr);
9      header.e_machine = read_half(file, ptr);
10     header.e_version = read_half(file, ptr);
11     header.e_entry = read_half(file, ptr);
12     header.e_phoff = read_half(file, ptr);
13     header.e_shoff = read_half(file, ptr);
14     header.e_flags = read_half(file, ptr);
15     header.e_ehsize = read_half(file, ptr);
16     header.e_phentsize = read_half(file, ptr);
17     header.e_phnum = read_half(file, ptr);
18     header.e_shentsize = read_half(file, ptr);
19     header.e_shnum = read_half(file, ptr);
20     header.e_shstrndx = read_half(file, ptr);
21 }
```


Здесь и далее для чтения команд/данных из нескольких байтов используются функции `read_half` и `read_whole` для данных порциями по 2 и 4 байта соответственно.

```
23  ✓ Elf32_Half read_half(std::vector<int>& file, int& ptr) {
24      Elf32_Half read_el = 0;
25      for(int i = 1; i >= 0; i--) {
26          read_el *= 16*16;
27          read_el += file[ptr + i];
28      }
29      ptr += 2;
30      return read_el;
31  }
32
33  ✓ uint32_t read_whole(std::vector<int>& file, int& ptr) {
34      uint32_t read_el = 0;
35      for(int i = 3; i >= 0; i--) {
36          read_el *= 16*16;
37          read_el += file[ptr + i];
38      }
39      ptr += 4;
40      return read_el;
41  }
```

Формат хранения данных `littel endian` из-за чего перед прочтением следующего байта уже имеющаяся часть сдвигается на 8 бит (умножается на 16^2).

Далее парсится Section Header Table (`parse_sh.cpp`). Информация собирается в структуры `Elf32_Shdr` (источник 4 страница 24) и помещается в массив.

```
3  ✓ void read_sh(std::vector<int>& file, Elf32_Shdr* sh, Header& header) {
4      for (int i = 0; i < header.e_shnum; i++) {
5          int ptr = header.e_shoff + i * header.e_shentsize;
6          Elf32_Shdr tmp;
7          tmp.sh_name = read_whole(file, ptr);
8          tmp.sh_type = read_whole(file, ptr);
9          tmp.sh_flags = read_whole(file, ptr);
10         tmp.sh_addr = read_whole(file, ptr);
11         tmp.sh_offset = read_whole(file, ptr);
12         tmp.sh_size = read_whole(file, ptr);
13         tmp.sh_link = read_whole(file, ptr);
14         tmp.sh_info = read_whole(file, ptr);
15         tmp.sh_addralign = read_whole(file, ptr);
16         tmp.sh_entsize = read_whole(file, ptr);
17         sh[i] = (tmp);
18     }
19 }
```

Так как имена секций хранятся в `strtab`, далее записываем имена секций.

```
27 void read_real_names(std::vector<int>& file, std::vector<std::vector<char>>& sh_names, Header& header, Elf32_Shdr* sh) {
28     int strtab_idx = sh[header.e_shstrndx].sh_offset;
29     for(int i = 0; i < header.e_shnum; i++) {
30         std::vector<char> tek_name;
31         int j = 0;
32         while(file[strtab_idx + sh[i].sh_name + j]) {
33             tek_name.push_back(file[strtab_idx + sh[i].sh_name + j]);
34             j++;
35         }
36         sh_names[i] = tek_name;
37         //ar_copy(sh_names, tek_name, i);
38     }
39 }
```

Далее рассмотрим разбор `symtab`, данные из него потребуются для корректного вывода `text`.

```
3 void parse_symtab(std::vector<int>& file, Elf32_Shdr* sh, std::vector<std::vector<char>>& sh_names,
4     int strtab_idx = find_strtab(sh_names, header);
5     if (strtab_idx == -1) {
6         return;
7     }
8     int names_start_ptr = sh[strtab_idx].sh_offset;
9     for (int i = 0; i < header.e_shnum; i++) {
10         if (is_symtab(sh_names[i])) {
11             int sym_num = sh[i].sh_size / SYMTAB_SIZE;
12             Elf32_Sym* symtab_header = new Elf32_Sym[sym_num];
13             read_symtab_header(file, symtab_header, sh[i].sh_offset, sym_num);
14             final_print_symtab(file, symtab_header, sym_num, names_start_ptr, fp);
15             delete[] symtab_header;
16         }
17     }
18 }
```

Ищем `strtab` (просто потому что можем) далее определяем кол-во записей в `strtab`. `SYMTAB` посчитали как сумму размеров всех переменных в `Elf32_Sym` (скрин был приложен выше).

```
50 void final_print_symtab(std::vector<int>& file, Elf32_Sym* symtab_header, int sym_num, int names_start_ptr, FILE *fp) {
51     fprintf(fp, "\n.symtab\n");
52     fprintf(fp, "\nSymbol Value          Size Type      Bind      Vis      Index Name\n");
53     for(int i = 0; i < sym_num; i++) {
54         std::string type = symtab_type(symtab_header, i);
55         std::string bind = symtab_bind(symtab_header, i);
56         std::string vis = symtab_vis(symtab_header, i);
57         std::string idx = symtab_index(symtab_header, i);
58         std::vector<char> name;
59         symtab_name(file, names_start_ptr, symtab_header, i, name);
60         std::string out_str (name.begin(), name.end());
61         if (out_str.size()) {
62             fprintf(fp, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %-8s %6s %s\n", i, symtab_header[i].st_value, symtab_header[i].st_size,
63                 type.c_str(), bind.c_str(), vis.c_str(), idx.c_str(), out_str.c_str());
64         } else {
65             fprintf(fp, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %-8s %6s %s\n", i, symtab_header[i].st_value, symtab_header[i].st_size,
66                 type.c_str(), bind.c_str(), vis.c_str(), idx.c_str(), "");
67         }
68     }
69 }
```

Информации о том, как отображать `type`, `bind`, `vis` и `idx` взята из (источки 3 страница 359, 357, 360, 307 соответственно). Имена считываем из `strtab` начиная с индекса, хранимого в переменной `st_name` каждой структуры до нуля.

Разбор text (скриншоты не приложены в силу большого объема кода): Для корректного отображения меток необходимо иметь информацию о блоках кода, описанных в symtab, а также знать метки, создаваемые инструкциями J и B типа. В начале перебираем все инструкции и, если они принадлежат B или J типу, а также адрес назначения еще не известен, создаем новую метку L<x> и сохраняем ее в вектор Ls для будущего использования (create_labels и функции, вызываемые ею).

```
3  void parse_text(std::vector<int>& file, Elf32_Shdr* sh, std::vector<std::vector<char>>& sh_names, Header& header, FILE* fp) {
4      int symtab_idx = find_symtab(sh_names, header.e_shnum);
5      int names_start_ptr = sh[find_strtab(sh_names, header)].sh_offset;
6      int sym_num = sh[symtab_idx].sh_size / SYMTAB_SIZE;
7
8      Elf32_Sym* symtab_header = new Elf32_Sym[sym_num];
9      read_symtab_header(file, symtab_header, sh[symtab_idx].sh_offset, sym_num);
10
11      std::vector<unsigned int> Ls;
12
13      for (int i = 0; i < header.e_shnum; i++) {
14          if (is_text(sh_names[i])) {
15              create_labels(file, sh, i, symtab_header, names_start_ptr, sym_num, Ls);
16          }
17      }
18
19      for (int i = 0; i < header.e_shnum; i++) {
20          if (is_text(sh_names[i])) {
21              final_print_text(file, sh, i, fp, symtab_header, names_start_ptr, sym_num, Ls);
22          }
23      }
24
25      delete[] symtab_header;
26  }
27
```

Имея данные о всех метках, создаваемых в процессе исполнения программы и данных нам в секции symtab можно выводить инструкции (final_print_text и функции, вызываемые ею). На основе последних семи битов инструкции и, при необходимости, 13-15 битах определяем инструкцию и тип, к которому она принадлежит.

Информация, необходимая для вывода инструкций была приложена выше.

Комментарии относительно авторского понимания RV32A и расширений:

Атомарные операции имеют два дополнительных поля aq, rl отвечающие за доступ инструкции к памяти (источник 1 страница 64). При выводе инструкции, если биты, отвечающие за aq и/или rl равны 1, к имени команды добавляется .aq/.rl/.aqrl.

Расширение Zihintpause: fence с пустым полем источника и направления ничего не делает и используется для эффективного простаивания процессора и/или выравнивания других инструкций. Вывод ничем не отличается от обычной fence инструкции кроме отсутствия ', '.

Расширение Zifencei: Вывод fence.i инструкции совпадает с обычными l-type инструкциями. (я тут не очень понял)

Источники:

- 1: <https://github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf>
- 2: <https://refspecs.linuxfoundation.org/elf/elf.pdf>
- 3: https://docs.oracle.com/cd/E26502_01/pdf/E26507.pdf
- 4: <https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf>