Лабораторная работа №3	группа 05	2022
ISA	Москаленко Т.Д.	

Цель работы: знакомство с архитектурой набора команд RISC-V.

Инструментарий и требования к работе: работа выполнена на C++. Компилятор, на котором работал:

```
Apple clang version 14.0.0 (clang-1400.0.29.202)
```

Описание системы кодирования команд RISC-V

RISC-V — открытая и свободная система команд и процессорная архитектура на основе концепции **RISC** для микропроцессоров и микроконтроллеров. Большая часть инструкций занимает 4 байта, но есть и 2 байтовые сжатые инструкции в одном из наборов команд.

Существуют разные наборы команд **RISC-V.** Я перечислю только некоторые из них (подробнее можно посмотреть в спецификации)

Название	Описание
RV32I	Базовый набор с целочисленными операциями, 32-битный
RV64I	Базовый набор с целочисленными операциями, 64-битный
RV32E	Базовый набор с целочисленными операциями для встраиваемых систем, 32-битный, 16 регистров
RV128I	Базовый набор с целочисленными операциями, 128-битный
RV32/64M	Целочисленное умножение и деление
RV32/64A	Атомарные операции
RV32/64F	Арифметические операции с плавающей запятой над числами одинарной точности
RV32/64D	Арифметические операции с плавающей запятой над числами двойной точности
RV32/64Q	Арифметические операции с плавающей запятой над числами четверной точности
RV32/64C	Сокращённые имена для команд

Каждый набор состоит из инструкций, которые объединены в группы (рис. 1).

31 27	26 25	24 20	19 15	14 12	11 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|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Рис. 1. Группы команд

1. **R-type** (Register) — операция с регистрами rs1 и rs2, запись в rd.

2. **I-type** (Immediate) — операция с регистром rs1 и константой imm, запись в rd.

3. **S-type** (Store) — операция сохранения в память работающая с регистрами rs1 и rs2.

4. **B-type** (Branch) — операция проверки условного выражения на значения в rs1 и rs2 и изменение указателя на команду, в соответствии с результатом.

5. **U-type** (Upper-Immediate) — операция с imm и rd.

6. **J-type** (Jump) — операция, которая использует сдвига указателя на команду.

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Рис. 2. RV32I

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
`mul`	MUL	R	`0110011`	`0x0`	`0x01`	`rd = (rs1 * rs2)[31:0]`
`mulh`	MUL High	R	`0110011`	`0x1`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulsu`	MUL High (S) (U)	R	`0110011`	`0x2`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`mulu`	MUL High (U)	R	`0110011`	`0x3`	`0x01`	`rd = (rs1 * rs2)[63:32]`
`div`	DIV	R	`0110011`	`0x4`	`0x01`	`rd = rs1 / rs2`
`divu`	DIV (U)	R	`0110011`	`0x5`	`0x01`	`rd = rs1 / rs2`
`rem`	Remainder	R	`0110011`	`0x6`	`0x01`	`rd = rs1 % rs2`
`remu`	Remainder (U)	R	`0110011`	`0x7`	`0x01`	`rd = rs1 % rs2`

Рис. 3. RV32M

По значениях в opcode, funct3 и funct7 исполнитель однозначно определяет, какую команду надо выполнить. Остальные поля это параметры данной команды. rs1 и rs2 это номера регистров, над которыми осуществляется операция. rd это номер регистра, в который мы записываем результат. imm это константа

В данной работе мы подробно разбираем 2 набора команд **RV32I** и **RV32M**. Их расшифровку можно подробнее рассмотреть в таблицах 2 и 3 или в оригинальном источнике

Описание структуры файла ELF

ELF файл — формат исполняемых двоичных файлов, используемый во многих современных UNIX-подобных операционных системах.

Каждый ELF файл состоит из нескольких частей:

Заголовок файла

Заголовок файла всегда расположен в начале файла и содержит общее описание структуры файла и его основные характеристики:

1. Общая характеристика файла — `e_ident`
2. Тип — `e_type`
3. Версия — `e_version`
4. Архитектура процессора — `e_machine`
5. Виртуальный адрес точки входа — `e_entry`
6. Размеры и смещения остальных частей файла — `e_phoff`, `e_shoff`, `e_flags`, `e_ehsize`, `e_phentsize`, `e_phnum`, `e_shentsize`, `e_shnum`, `e_shstrndx`

Заголовок имеет размер 52 байта для 32-битных файлов или 64 для 64-битных. Размеры различны потому, что заголовки хранят указатели, которые занимают разное количество байт в разных архитектурах.

Таблица заголовков программы

Таблица заголовков программы содержит заголовки, каждый из которых описывает отдельный сегмент программы. Указатель на данную таблицу смещён от начала файла на значение переменной `e_phoff` заголовка ELF. В этой лабораторной данная часть файла нас не интересует.

Таблица заголовков секций

Таблица заголовков секций содержит всю информацию о секцях файла, их основные характеристики.

1. Смещение строки, содержащей название данной секции, относительно начала таблицы названий секций — `sh_name`
2. Тип заголовка — `sh_type`
3. Атрибуты секции — `sh_flags`
4. Если секция должна быть загружена в память при загрузке объектного файла, это поле указывает адрес, начиная с которого секция будет загружена — `sh_addr`
5. Смещение секции от начала файла в байтах — `sh_offset`
6. Размер секции в файле, может быть нулевым — `sh_size`
7. Индекс ассоциированной секции. Данное поле может иметь различное предназначение в зависимости от типа заголовка. — `sh_link`
8. Дополнительная информация о секции — `sh_info`
9. Необходимое выравнивание секции — `sh_addralign`
10. Размер в байтах каждой записи — `sh_entsize`

Содержимое секций

Здесь привожу содержимое секций файла, которые используются в программе.

1. `.text` — Секция, в которой хранится исполняемый код данного файла. Её нам и нужно «дизассемблировать».
2. `.symtab` — Секция, в которой хранится таблица символов: названия файлов, названия переменных, служебные символы и так далее;
3. `.strtab` — Секция, в которой хранятся имена всех символов из таблицы символов;

4. `.shstrtab` — Секция, в которой хранятся имена всех секций файла.

Описание работы написанного кода

Рассмотрим начало функции `int main`

```
1   if (argc != 3) {
2      cerr << "incorrect␣arg␣count\n";
3      return 1;
4   }
5
6   FILE *input = fopen(argv[1], "rb");
7   size_t file_len;
8
9   if (input) {
10     fseek(input, 0, SEEK_END);
11     file_len = ftell(input);
12     fseek(input, 0, SEEK_SET);
13  } else {
14     cerr << "can't␣open␣input␣file\n";
15     return 1;
16  }
17
18  byte *bytes = (byte *) malloc(file_len);
19  fread(bytes, 1, file_len, input);
20  fclose(input);
```

Мы делаем проверку на количество аргументов и проверям открытие входного файла, дальшем мы хотим полжить весь файл на heap и работать с массивом байтов этого файла (работать с указателем внутри файла сложнее).

Рассмотрим структуру `Header`, которая потребуется для хранения заголовка elf файла в программе.

```
1  size_t bytes_to_int(byte bytes[], size_t start, size_t end) {
2     size_t ans = 0;
3     for (size_t i = start; i < end; i++) {
4        ans += ((size_t) bytes[i]) << (8 * (i - start));
5     }
6     return ans;
7  }
8
9  struct Header {
10    size_t EI_MAG;
11    size_t EI_CLASS;
12    <...>
13    size_t e_shnum;
14    size_t e_shstrndx;
15 };
16
17 bool check_header(Header header) {
18    if ((header.EI_MAG != 0x464c457f) || (header.EI_CLASS != 0x01) ||
      (header.EI_DATA != 0x01) ||
```

```
    (header.EI_VERSION != 0x01) || (header.e_machine != 0xf3) || (header.e_version
    != 0x01) ||
    (header.e_shentsize != 0x28) || (header.e_ehsize != 0x34) ||
    (header.e_shstrndx == 0)) {
        cerr << "Incorrect input file\n";
        return false;
    }
    return true;
}

Header make_header(byte bytes[]) {
    Header header{};
    header.EI_MAG = bytes_to_int(bytes, 0, 4);
    header.EI_CLASS = bytes_to_int(bytes, 4, 5);
    <...>
    header.e_shnum = bytes_to_int(bytes, 48, 50);
    header.e_shstrndx = bytes_to_int(bytes, 50, 52);
    return header;
}
```

Функция `bytes_to_int` по указателю на начало массива байт, и сдвигам на начало и конец переделывает полученные байты в `int`, порядок байт little endian из условия учитывается именно здесь

Дальше идёт структура `Header` со всеми полями, функция `check_header`,которая проверяет, что полученный `Header` действительно поддерживается нашей программой и `make_header`, которая является конструктором заголовка файла. Получает на вход указатель на начало массива байт файла.

В функции `main` происходит инициализация.

```
    Header header = make_header(bytes);
    if (!checkHeader(header)) {
        cerr << "file is not supported\n";
        return 1;
    }
```

Далее рассмотрим структуру секции файла

```
struct Section {
    string name;
    size_t sh_name{};

    size_t sh_type{};
    size_t sh_flags{};
    <...>
    size_t sh_addralign{};
    size_t sh_entsize{};
};
```

```
Section make_section(byte bytes[], size_t offset, size_t names_offset) {
    Section section{};
    section.sh_name = bytes_to_int(bytes + offset, 0, 4);
```

```
4     section.name = "";
5     for (int i = 0; (int) bytes[names_offset + section.sh_name + i] != 0; i++) {
6        section.name += (char) bytes[names_offset + section.sh_name + i];
7     }
8     section.sh_type = bytes_to_int(bytes + offset, 4, 8);
9     section.sh_flags = bytes_to_int(bytes + offset, 8, 12);
10    section.sh_addr = bytes_to_int(bytes + offset, 12, 16);
11    section.sh_offset = bytes_to_int(bytes + offset, 16, 20);
12    section.sh_size = bytes_to_int(bytes + offset, 20, 24);
13    section.sh_link = bytes_to_int(bytes + offset, 24, 28);
14    section.sh_info = bytes_to_int(bytes + offset, 28, 32);
15    section.sh_addralign = bytes_to_int(bytes + offset, 32, 36);
16    section.sh_entsize = bytes_to_int(bytes + offset, 36, 40);
17    return section;
18 }
```

Секция очень похожа на заголовок. Мы создаём структуру `Section` со всеми полями секции. Далее идёт функция `make_section` — конструктор одной секции. Он получает массив с байтами исходного файла, `offset` с которого начинается заголовок данной секции, `names_offset` — число, которое отвечает за сдвиг секции с именами, из которой нужно брать имя данной секции.

`sh_name` отвечает за позицию внутри этой секции. В строчках 5-7 реализуется инициализация имени секции.

Посмотрим, как это имплементировано в `main`

```
1     size_t names_pointer = header.e_shoff + header.e_shentsize * header.e_shstrndx;
2     size_t names_offset = bytes_to_int(bytes, names_pointer + 16, names_pointer +
      20);
3
4     vector<Section> sections(header.e_shnum);
5     Section *symtab_section;
6     Section *strtab_section;
7     Section *text_section;
8
9     for (int i = 1; i < header.e_shnum; i++) {
10       sections[i] = make_section(bytes, header.e_shoff + i * header.e_shentsize,
      names_offset);
11       if (sections[i].name == ".symtab") {
12          symtab_section = &sections[i];
13       }
14       if (sections[i].name == ".strtab") {
15          strtab_section = &sections[i];
16       }
17       if (sections[i].name == ".text") {
18          text_section = &sections[i];
19       }
20    }
```

Создаём вектор секций `sections` размера `header.e_shnum`. Далее заметим, что указатель на `i`-тую секцию в файле находится на `header.e_shoff + i *`

header.e_shentsize. Секция с именами имеет номер header.e_shstrndx, а значит указатель на неё в файле это

header.e_shoff + header.e_shentsize * header.e_shstrndx

Далее мы в for создаём секции, и запоминаем указатели на специальные секции ".symtab ".strtab"и ".text". Они нам пригодятся позднее.

Рассмотрим теперь структуру связанную с символами.

```
struct Symbol {
   string name;
   string type;
   string bind;
   string vis;
   string index;

   size_t st_name{};
   size_t st_value{};
   size_t st_size{};
   size_t st_info{};
   size_t st_other{};
   size_t st_shndx{};
};

void make_symbol(Symbol *symbol, byte bytes[], size_t offset, size_t
    names_offset) {
   symbol->st_name = bytes_to_int(bytes + offset, 0, 4);
   symbol->name = "";
   for (int i = 0; (int) bytes[names_offset + symbol->st_name + i] != 0; i++) {
      symbol->name += (char) bytes[names_offset + symbol->st_name + i];
   }
   symbol->st_value = bytes_to_int(bytes + offset, 4, 8);
   symbol->st_size = bytes_to_int(bytes + offset, 8, 12);
   symbol->st_info = bytes_to_int(bytes + offset, 12, 13);
   symbol->st_other = bytes_to_int(bytes + offset, 13, 14);
   symbol->st_shndx = bytes_to_int(bytes + offset, 14, 16);

   symbol->type = symbol_types[symbol->st_info & 0xf];
   symbol->bind = symbol_binds[symbol->st_info >> 4];
   if (symbol_idx.count(symbol->st_shndx))
   symbol->index = symbol_idx[symbol->st_shndx];
   else
   symbol->index = to_string(symbol->st_shndx);
   symbol->vis = symbol_vises[symbol->st_other & 0x3];
}
```

Символ делается по аналогии с предыдущими структурами, но у нас добавляются новые строковые поля, которые будут выводиться в консоль в конце файла. Расшифровка данных хранится в начале файла в структурах map. Информация о том, как именно это делается была взята с сайта orcale.

```
map<size_t, string> symbol_types{
```

```
    {0,  "NOTYPE"},
    {1,  "OBJECT"},
    {2,  "FUNC"},
    {3,  "SECTION"},
    {4,  "FILE"},
    {5,  "COMMON"},
    {6,  "TLS"},
    {10, "LOOS"},
    {12, "HIOS"},
    {13, "LOPROC"},
    {14, "SPARC_REGISTER"},
    {15, "HIPROC"}
};

map<size_t, string> symbol_binds{
    {0,  "LOCAL"},
    {1,  "GLOBAL"},
    {2,  "WEAK"},
    {10, "LOOS"},
    {12, "HIOS"},
    {13, "LOPROC"},
    {15, "HIPROC"}
};

map<size_t, string> symbol_vises{
    {0, "DEFAULT"},
    {1, "INTERNAL"},
    {2, "HIDDEN"},
    {3, "PROTECTED"},
    {4, "EXPORTED"},
    {5, "SINGLETON"},
    {6, "ELIMINATE"}
};

map<size_t, string> symbol_idx{
    {0,      "UNDEF"},
    {0xff00, "BEFORE"},
    {0xff01, "AFTER"},
    {0xff02, "AMD64_LCOMMON"},
    {0xff1f, "HIPROC"},
    {0xff20, "LOOS"},
    {0xff3f, "HIOS"},
    {0xfff1, "ABS"},
    {0xfff2, "COMMON"},
    {0xffff, "XINDEX"}
};
```

Рассмотрим имплементацию в функции `main`

```
vector<Symbol> symbols(symtab_section->sh_size);
for (int i = 0; i * symtab_section->sh_entsize < symtab_section->sh_size; i++) {
    make_symbol(&symbols[i], bytes, symtab_section->sh_offset + i *
```

```
    symtab_section->sh_entsize,
    strtab_section->sh_offset);
    if (symbols[i].type == "OBJECT" || symbols[i].type == "FUNC") {
        symbol_dict[symbols[i].st_value] = symbols[i].name;
    }
}
```

Далее рассмотрим структуру которая содержит всю информацию о каждой инструкции:

```
struct Instruction {
    string op_name = "unknown instruction";
    string rs1_name;
    string rs2_name;
    string rd_name;
    instruction_type type = unknown_instr;

    size_t imm{};
    size_t shamt{};

    size_t data{};
    size_t opcode{};
    size_t rd{};
    size_t funct3{};
    size_t rs1{};
    size_t rs2{};
    size_t funct7{};
};
```

Поля типа `size_t` содержат в себе численное значение, поля типа `string` содержат в себе расшифровку параметров команд, чтобы потом их выводить в дизассемблере.

Инициализация всех полей происходит в функции `make_instruction`

```
Instruction make_instruction(size_t data) {
    Instruction cur;
    cur.data = data;
    cur.opcode = data & 0b1111111;
    cur.rd = (data >> 7) & 0b11111;
    cur.funct3 = (data >> 12) & 0b111;
    cur.rs1 = (data >> 15) & 0b11111;
    cur.rs2 = (data >> 20) & 0b11111;
    cur.funct7 = (data >> 25) & 0b1111111;
    cur.rd_name = registers[cur.rd];
    cur.rs1_name = registers[cur.rs1];
    cur.rs2_name = registers[cur.rs2];

    switch (cur.opcode) {
        case 0b0110111:
            cur.op_name = "lui";
            cur.imm = cur.data >> 12;
            cur.type = argdm;
```

```
19          break;
20       case 0b0010111:
21          cur.op_name = "auipc";
22          cur.imm = cur.data >> 12;
23          cur.type = argdm;
24          break;
25       case 0b1101111:
26          cur.op_name = "jal";
27          cur.imm = ((cur.data & (1 << 31)) >> 11) + (cur.rs1 << 15) + (cur.funct3
    << 12) +
28          ((cur.rs2 & 1) << 11) + ((cur.rs2 >> 1) << 1) + ((cur.funct7 & 0b111111)
    << 5);
29          if (cur.data & (1 << 31)) {
30             cur.imm |= (int) 0xfffc0000;
31          }
32          cur.type = jal;
33          break;
34       case 0b1100111:
35          cur.op_name = "jalr";
36          cur.imm = cur.data >> 19;
37          cur.type = lsdm1;
38          break;
39       case 0b1100011:
40          cur.op_name = b_type[cur.funct3];
41          cur.imm = (int) (((cur.funct7 & 0b1000000) << 5) + ((cur.rd & 1) << 10)
    + ((cur.funct7 & 0b111111) << 4) +
42          (cur.rd >> 1)) << 1;
43          if (cur.funct7 & 0b1000000) {
44             cur.imm |= (int) 0xffffe000;
45          }
46          cur.type = arg12ml;
47          break;
48       case 0b0000011:
49          cur.op_name = l_type[cur.funct3];
50          cur.imm = cur.data >> 20;
51          cur.type = lsdm1;
52          break;
53       case 0b0100011:
54          cur.op_name = s_type[cur.funct3];
55          cur.imm = (cur.funct7 << 5) + cur.rd;
56          cur.type = ls2m1;
57          break;
58       case 0b0010011:
59          if (cur.funct3 == 0b101) {
60             cur.op_name = cur.funct7 ? "srai" : "srli";
61             cur.type = argd1s;
62             cur.shamt = cur.rs2;
63          } else if (cur.funct3 == 0b001) {
64             cur.op_name = "slli";
65             cur.type = argd1s;
66             cur.shamt = cur.rs2;
67          } else {
```

```
68              cur.op_name = i_type[cur.funct3];
69              cur.type = argd1m;
70              cur.imm = cur.data >> 20;
71              if (cur.imm & 2048) {
72                 cur.imm |= 0xfffff000;
73              }
74           }
75           break;
76        case 0b0110011:
77           cur.op_name = r_type[{cur.funct7, cur.funct3}];
78           cur.type = argd12;
79           break;
80        case 0b0001111:
81           cur.op_name = "unknown_instruction";
82           cur.type = fence;
83           break;
84        case 0b1110011:
85           if (cur.funct7 == 0) {
86              cur.op_name = "ecall";
87           } else {
88              cur.op_name = "ebreak";
89           }
90           cur.type = empty_instr;
91           break;
92     }
93     return cur;
94 }
```

Изначально мы инициализируем все поля типа `size_t` и поля, отвечающие за имена регистров (некоторые из них могут не использоваться). Для этого мы используем заранее сделанную `map` по информации из документации на странице 137.

```
map<size_t, string> registers{
   {0,  "zero"},
   {1,  "ra"},
   {2,  "sp"},
   <...>
   {30, "t5"},
   {31, "t6"}
};
```

Дальше мы разбираем операции согласно 130-131 странице той же документации. Разбор происходит по переменной `opcode`. Программа делает `switch, case` и в каждом из них выдаёт нужное имя команде и некоторые дополнительные параметры.

Имя можно найти по заранее заданным `map`, которые по `funct3` выдают имя переменной данного типа

```
map<size_t, string> b_type{
   {0, "beq"},
```

```
    {1, "bne"},
    {4, "blt"},
    {5, "bge"},
    {6, "bltu"},
    {7, "bgeu"},
};

map<size_t, string> l_type{
    {0, "lb"},
    {1, "lh"},
    {2, "lw"},
    {4, "lbu"},
    {5, "lhu"},
};

map<size_t, string> s_type{
    {0, "sb"},
    {1, "sh"},
    {2, "sw"},
};

map<size_t, string> i_type{
    {0, "addi"},
    {2, "slti"},
    {3, "sltiu"},
    {4, "xori"},
    {6, "ori"},
    {7, "andi"},
};
```

Для команд типа r чтобы однозначно узнать команду, нужно знать ещё `funct7`, поэтому соответствующий словарь от двух переменных (`funct7`, `funct3`)

```
map<pair<size_t, size_t>, string> r_type{
    {{0,  0}, "add"},
    {{32, 0}, "sub"},
    {{0,  1}, "sll"},
    {{0,  2}, "slt"},
    <...>
    {{1,  0}, "mul"},
    {{1,  1}, "mulh"},
    <...>
    {{1,  7}, "remu"},
};
```

У некоторых команд мы считаем `imm`. Это делается довольно громоздко, но идея в том, чтобы переставить биты в соответствии с тем, как это написано в спецификации. Затем добавить несколько 1 в начало, если число отрицательное.

Рассмотрим поле `instruction_type`

```
enum instruction_type {
    unknown_instr,
```

```
    jal,
    arg1d,
    argdm,
    argd1m,
    arg12ml,
    argd12,
    argd1s,
    lsdm1,
    ls2m1,
    fence,
    empty_instr,
};
```

Это `enum`, который показывает, как надо делать `print` инструкции.

- `arg` — команды, которые выводятся без скобок,
- `ls` — команды, которые выводятся со скобками,
- `unknown_instr` — неизвестная инструкция,
- `fence` — инструкция `fence`,
- `jal` — инструкция `jal`,
- `empty_instr` — инструкция, которая не имеет аргументов для вывода,
- `1, 2, d` — имя соответствующего регистра,
- `s` — `shamt`,
- `m` — указатель на элемент в файле,
- `l` — ссылка в треугольных скобках.

Рассмотрим имплементацию в функции `main`

```
    for (int i = 0; i * 4 < text_section->sh_size; i++) {
        size_t data = bytes_to_int(bytes + text_section->sh_offset + i * 4, 0, 4);
        Instruction instruction = make_instruction(data);
        set_new_marker(instruction, text_section->sh_addr + i * 4);
    }
```

Мы проходим через все инструкции в секции `".text"` и вызываем функцию `set_new_marker`, которая добавляет метку, если инструкция может делать `jump`. Реализация представлена ниже.

```
map<size_t, string> symbol_dict;
int marker_idx = 0;

void set_new_marker(const Instruction &instruction, const size_t shift) {
    if (instruction.type == arg12ml || instruction.type == jal) {
```

```
        if (!symbol_dict.count(instruction.imm + shift)) {
            symbol_dict[instruction.imm + shift] = "L" + to_string(marker_idx++);
        }
    }
}
```

Рассмотрим код функции main дальше

```
    if (!freopen(argv[2], "wt", stdout)) {
        free(bytes);
        cerr << "can't␣open␣output␣file\n";
        return 1;
    }
    cout << ".text\n";
    for (int i = 0; i * 4 < text_section->sh_size; i++) {
        size_t data = bytes_to_int(bytes + text_section->sh_offset + i * 4, 0, 4);
        Instruction instruction = make_instruction(data);
        print_instruction(instruction, text_section->sh_addr + i * 4);
    }
```

Мы заменяем поток вывода на данный нам файл на входе. И начинаем выводить код блока `.text`. Рассмотрим реализацию функции `print_instruction`

```
void print_instruction(const Instruction &instruction, const size_t shift) {
    if (symbol_dict.count(shift)) {
        printf("%08zx␣␣␣<%s>:\n", shift, symbol_dict[shift].c_str());
    }
    printf("␣␣␣%05zx:\t%08zx\t%7s\t", shift, instruction.data,
    instruction.op_name.c_str());
    if (instruction.type == arg1d) {
        printf("%s,␣%s\n",
        instruction.rd_name.c_str(),
        instruction.rs1_name.c_str());
    } else if (instruction.type == argdm) {
        printf("%s,␣0x%zx\n",
        instruction.rd_name.c_str(),
        instruction.imm);
    } else if (instruction.type == argd1m) {
        printf("%s,␣%s,␣%d\n",
        instruction.rd_name.c_str(),
        instruction.rs1_name.c_str(),
        (int) instruction.imm);
    } else if (instruction.type == argd1s) {
        printf("%s,␣%s,␣%zd\n",
        instruction.rd_name.c_str(),
        instruction.rs1_name.c_str(),
        instruction.shamt);
    } else if (instruction.type == argd12) {
        printf("%s,␣%s,␣%s\n",
        instruction.rd_name.c_str(),
        instruction.rs1_name.c_str(),
        instruction.rs2_name.c_str());
    } else if (instruction.type == arg12ml) { // b-type
```

```
30          printf("%s␣%s␣0x%zx␣<%s>\n",
31          instruction.rs1_name.c_str(),
32          instruction.rs2_name.c_str(),
33          instruction.imm + shift,
34          symbol_dict[instruction.imm + shift].c_str());
35      } else if (instruction.type == lsdm1) {
36          printf("%s␣%zd(%s)\n",
37          instruction.rd_name.c_str(),
38          instruction.imm,
39          instruction.rs1_name.c_str());
40      } else if (instruction.type == ls2m1) {
41          printf("%s␣%zd(%s)\n",
42          instruction.rs2_name.c_str(),
43          instruction.imm,
44          instruction.rs1_name.c_str());
45      } else if (instruction.type == jal) {
46          printf("%s␣0x%zx␣<%s>\n",
47          instruction.rd_name.c_str(),
48          instruction.imm + shift,
49          symbol_dict[instruction.imm + shift].c_str());
50      } else {
51          printf("\n");
52      }
53  }
```

Сначала выводим строчку с названием секции, затем общие данные для всех команд, дальше действуем для каждой команды индивидуально, в зависимости от того, какого она типа.

Рассмотрим код в `main`, который выводит символы:

```
1   cout << "\n.symtab\n";
2   printf("%6s␣%-17s␣%5s␣%-8s␣%-8s␣%-8s␣%6s␣%s\n", "Symbol", "Value", "Size",
    "Type", "Bind", "Vis", "Index", "Name");
3   for (int i = 0; i * symtab_section->sh_entsize < symtab_section->sh_size; i++)
    {
4       print_symbol(symbols, i);
5   }
```

Функция `print_symbol` выводит значение символа в соответствии с заданием.

```
1  void print_symbol(vector<Symbol> symbols, const int i) {
2      printf("[%4i]␣0x%-15zX␣%5zu␣%-8s␣%-8s␣%-8s␣%6s␣%s\n", i,
3      symbols[i].st_value,
4      symbols[i].st_size,
5      symbols[i].type.c_str(),
6      symbols[i].bind.c_str(),
7      symbols[i].vis.c_str(),
8      symbols[i].index.c_str(),
9      symbols[i].name.c_str());
10 }
```

Дальше делаем чистку в функции `main`.

```
    free(bytes);
    fclose(stdout);
    return 0;
```

Результат работы написанной программы на приложенном к заданию файле

Дизассемблер

```
.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 unknown_instruction
   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)
```

Таблица символов

```
.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
```

Список источников

ELF wiki
Формат файлов ELF
Описание структуры файла ELF
ELF habr
RISC-V с нуля habr
Oracle symbol
Oracle shndx
Имена регистров RISC-V
Красивая табличка команд RISC-V
Документация RISC-V

Листинг кода

Листинг 1. acos.cpp

```
#include <iostream>
#include <cstdio>
#include <fstream>
#include <vector>
#include <map>

using namespace std;

map<size_t, string> registers{
    {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"}
};

map<size_t, string> b_type{
    {0, "beq"},
    {1, "bne"},
    {4, "blt"},
    {5, "bge"},
```

```
    {6, "bltu"},
    {7, "bgeu"},
};

map<size_t, string> l_type{
    {0, "lb"},
    {1, "lh"},
    {2, "lw"},
    {4, "lbu"},
    {5, "lhu"},
};

map<size_t, string> s_type{
    {0, "sb"},
    {1, "sh"},
    {2, "sw"},
};


map<size_t, string> i_type{
    {0, "addi"},
    {2, "slti"},
    {3, "sltiu"},
    {4, "xori"},
    {6, "ori"},
    {7, "andi"},
};

map<pair<size_t, size_t>, string> r_type{
    {{0,  0}, "add"},
    {{32, 0}, "sub"},
    {{0,  1}, "sll"},
    {{0,  2}, "slt"},
    {{0,  3}, "sltu"},
    {{0,  4}, "xor"},
    {{0,  5}, "srl"},
    {{32, 5}, "sra"},
    {{0,  6}, "or"},
    {{0,  7}, "and"},
    {{1,  0}, "mul"},
    {{1,  1}, "mulh"},
    {{1,  2}, "mulhsu"},
    {{1,  3}, "mulhu"},
    {{1,  4}, "div"},
    {{1,  5}, "divu"},
    {{1,  6}, "rem"},
    {{1,  7}, "remu"},
};

map<size_t, string> symbol_types{
    {0,  "NOTYPE"},
    {1,  "OBJECT"},
```

```
    {2,  "FUNC"},
    {3,  "SECTION"},
    {4,  "FILE"},
    {5,  "COMMON"},
    {6,  "TLS"},
    {10, "LOOS"},
    {12, "HIOS"},
    {13, "LOPROC"},
    {14, "SPARC_REGISTER"},
    {15, "HIPROC"}
};

map<size_t, string> symbol_binds{
    {0,  "LOCAL"},
    {1,  "GLOBAL"},
    {2,  "WEAK"},
    {10, "LOOS"},
    {12, "HIOS"},
    {13, "LOPROC"},
    {15, "HIPROC"}
};

map<size_t, string> symbol_vises{
    {0, "DEFAULT"},
    {1, "INTERNAL"},
    {2, "HIDDEN"},
    {3, "PROTECTED"},
    {4, "EXPORTED"},
    {5, "SINGLETON"},
    {6, "ELIMINATE"}
};

map<size_t, string> symbol_idx{
    {0,      "UNDEF"},
    {0xff00, "BEFORE"},
    {0xff01, "AFTER"},
    {0xff02, "AMD64_LCOMMON"},
    {0xff1f, "HIPROC"},
    {0xff20, "LOOS"},
    {0xff3f, "HIOS"},
    {0xfff1, "ABS"},
    {0xfff2, "COMMON"},
    {0xffff, "XINDEX"}
};

size_t bytes_to_int(byte bytes[], size_t start, size_t end) {
    size_t ans = 0;
    for (size_t i = start; i < end; i++) {
        ans += ((size_t) bytes[i]) << (8 * (i - start));
    }
    return ans;
}
```

```
153
154 struct Header {
155     size_t EI_MAG;
156     size_t EI_CLASS;
157     size_t EI_DATA;
158     size_t EI_VERSION;
159     size_t EI_OSABI;
160     size_t EI_ABIVERSION;
161
162     size_t e_type;
163     size_t e_machine;
164     size_t e_version;
165     size_t e_entry;
166     size_t e_phoff;
167     size_t e_shoff;
168     size_t e_flags;
169     size_t e_ehsize;
170     size_t e_phentsize;
171     size_t e_phnum;
172     size_t e_shentsize;
173     size_t e_shnum;
174     size_t e_shstrndx;
175 };
176
177 bool check_header(Header header) {
178     if ((header.EI_MAG != 0x464c457f) || (header.EI_CLASS != 0x01) ||
        (header.EI_DATA != 0x01) ||
179     (header.EI_VERSION != 0x01) || (header.e_machine != 0xf3) || (header.e_version
        != 0x01) ||
180     (header.e_shentsize != 0x28) || (header.e_ehsize != 0x34) ||
        (header.e_shstrndx == 0)) {
181         cerr << "Incorrect␣input␣file\n";
182         return false;
183     }
184     return true;
185 }
186
187 Header make_header(byte bytes[]) {
188     Header header{};
189     header.EI_MAG = bytes_to_int(bytes, 0, 4);
190     header.EI_CLASS = bytes_to_int(bytes, 4, 5);
191     header.EI_DATA = bytes_to_int(bytes, 5, 6);
192     header.EI_VERSION = bytes_to_int(bytes, 6, 7);
193     header.EI_OSABI = bytes_to_int(bytes, 7, 8);
194     header.EI_ABIVERSION = bytes_to_int(bytes, 8, 9);
195
196     header.e_type = bytes_to_int(bytes, 16, 18);
197     header.e_machine = bytes_to_int(bytes, 18, 20);
198     header.e_version = bytes_to_int(bytes, 20, 24);
199     header.e_entry = bytes_to_int(bytes, 24, 28);
200     header.e_phoff = bytes_to_int(bytes, 28, 32);
201     header.e_shoff = bytes_to_int(bytes, 32, 36);
```

```
202     header.e_flags = bytes_to_int(bytes, 36, 40);
203     header.e_ehsize = bytes_to_int(bytes, 40, 42);
204     header.e_phentsize = bytes_to_int(bytes, 42, 44);
205     header.e_phnum = bytes_to_int(bytes, 44, 46);
206     header.e_shentsize = bytes_to_int(bytes, 46, 48);
207     header.e_shnum = bytes_to_int(bytes, 48, 50);
208     header.e_shstrndx = bytes_to_int(bytes, 50, 52);
209     return header;
210 }
211
212 struct Section {
213     string name;
214     size_t sh_name{};
215
216     size_t sh_type{};
217     size_t sh_flags{};
218     size_t sh_addr{};
219     size_t sh_offset{};
220     size_t sh_size{};
221     size_t sh_link{};
222     size_t sh_info{};
223     size_t sh_addralign{};
224     size_t sh_entsize{};
225 };
226
227 Section make_section(byte bytes[], size_t offset, size_t names_offset) {
228     Section section{};
229     section.sh_name = bytes_to_int(bytes + offset, 0, 4);
230     section.name = "";
231     for (int i = 0; (int) bytes[names_offset + section.sh_name + i] != 0; i++) {
232         section.name += (char) bytes[names_offset + section.sh_name + i];
233     }
234     section.sh_type = bytes_to_int(bytes + offset, 4, 8);
235     section.sh_flags = bytes_to_int(bytes + offset, 8, 12);
236     section.sh_addr = bytes_to_int(bytes + offset, 12, 16);
237     section.sh_offset = bytes_to_int(bytes + offset, 16, 20);
238     section.sh_size = bytes_to_int(bytes + offset, 20, 24);
239     section.sh_link = bytes_to_int(bytes + offset, 24, 28);
240     section.sh_info = bytes_to_int(bytes + offset, 28, 32);
241     section.sh_addralign = bytes_to_int(bytes + offset, 32, 36);
242     section.sh_entsize = bytes_to_int(bytes + offset, 36, 40);
243     return section;
244 }
245
246 struct Symbol {
247     string name;
248     string type;
249     string bind;
250     string vis;
251     string index;
252
253     size_t st_name{};
```

```
254     size_t st_value{};
255     size_t st_size{};
256     size_t st_info{};
257     size_t st_other{};
258     size_t st_shndx{};
259 };
260 
261 void make_symbol(Symbol *symbol, byte bytes[], size_t offset, size_t
        names_offset) {
262     symbol->st_name = bytes_to_int(bytes + offset, 0, 4);
263     symbol->name = "";
264     for (int i = 0; (int) bytes[names_offset + symbol->st_name + i] != 0; i++) {
265         symbol->name += (char) bytes[names_offset + symbol->st_name + i];
266     }
267     symbol->st_value = bytes_to_int(bytes + offset, 4, 8);
268     symbol->st_size = bytes_to_int(bytes + offset, 8, 12);
269     symbol->st_info = bytes_to_int(bytes + offset, 12, 13);
270     symbol->st_other = bytes_to_int(bytes + offset, 13, 14);
271     symbol->st_shndx = bytes_to_int(bytes + offset, 14, 16);
272 
273     symbol->type = symbol_types[symbol->st_info & 0xf];
274     symbol->bind = symbol_binds[symbol->st_info >> 4];
275     if (symbol_idx.count(symbol->st_shndx))
276     symbol->index = symbol_idx[symbol->st_shndx];
277     else
278     symbol->index = to_string(symbol->st_shndx);
279     symbol->vis = symbol_vises[symbol->st_other & 0x3];
280 }
281 
282 enum instruction_type {
283     unknown_instr,
284     jal,
285     arg1d,
286     argdm,
287     argd1m,
288     arg12ml,
289     argd12,
290     argd1s,
291     lsdm1,
292     ls2m1,
293     fence,
294     empty_instr,
295 };
296 
297 struct Instruction {
298     string op_name = "unknown␣instruction";
299     string rs1_name;
300     string rs2_name;
301     string rd_name;
302     instruction_type type = unknown_instr;
303 
304     size_t imm{};
```

```
305     size_t shamt{};
306 
307     size_t data{};
308     size_t opcode{};
309     size_t rd{};
310     size_t funct3{};
311     size_t rs1{};
312     size_t rs2{};
313     size_t funct7{};
314 };
315 
316 Instruction make_instruction(size_t data) {
317     Instruction cur;
318     cur.data = data;
319     cur.opcode = data & 0b1111111;
320     cur.rd = (data >> 7) & 0b11111;
321     cur.funct3 = (data >> 12) & 0b111;
322     cur.rs1 = (data >> 15) & 0b11111;
323     cur.rs2 = (data >> 20) & 0b11111;
324     cur.funct7 = (data >> 25) & 0b1111111;
325 
326 
327     cur.rd_name = registers[cur.rd];
328     cur.rs1_name = registers[cur.rs1];
329     cur.rs2_name = registers[cur.rs2];
330 
331     switch (cur.opcode) {
332         case 0b0110111:
333             cur.op_name = "lui";
334             cur.imm = cur.data >> 12;
335             cur.type = argdm;
336             break;
337         case 0b0010111:
338             cur.op_name = "auipc";
339             cur.imm = cur.data >> 12;
340             cur.type = argdm;
341             break;
342         case 0b1101111:
343             cur.op_name = "jal";
344             cur.imm = ((cur.data & (1 << 31)) >> 11) + (cur.rs1 << 15) + (cur.funct3
    << 12) +
345                 ((cur.rs2 & 1) << 11) + ((cur.rs2 >> 1) << 1) + ((cur.funct7 & 0b111111)
    << 5);
346             if (cur.data & (1 << 31)) {
347                 cur.imm |= (int) 0xfffc0000;
348             }
349             cur.type = jal;
350             break;
351         case 0b1100111:
352             cur.op_name = "jalr";
353             cur.imm = cur.data >> 19;
354             cur.type = lsdm1;
```

```
355         break;
356      case 0b1100011:
357         cur.op_name = b_type[cur.funct3];
358         cur.imm = (int) (((cur.funct7 & 0b1000000) << 5) + ((cur.rd & 1) << 10)
    + ((cur.funct7 & 0b111111) << 4) +
359         (cur.rd >> 1)) << 1;
360         if (cur.funct7 & 0b1000000) {
361            cur.imm |= (int) 0xfffe000;
362         }
363         cur.type = arg12ml;
364         break;
365      case 0b0000011:
366         cur.op_name = l_type[cur.funct3];
367         cur.imm = cur.data >> 20;
368         cur.type = lsdm1;
369         break;
370      case 0b0100011:
371         cur.op_name = s_type[cur.funct3];
372         cur.imm = (cur.funct7 << 5) + cur.rd;
373         cur.type = ls2m1;
374         break;
375      case 0b0010011:
376         if (cur.funct3 == 0b101) {
377            cur.op_name = cur.funct7 ? "srai" : "srli";
378            cur.type = argd1s;
379            cur.shamt = cur.rs2;
380         } else if (cur.funct3 == 0b001) {
381            cur.op_name = "slli";
382            cur.type = argd1s;
383            cur.shamt = cur.rs2;
384         } else {
385            cur.op_name = i_type[cur.funct3];
386            cur.type = argd1m;
387            cur.imm = cur.data >> 20;
388            if (cur.imm & 2048) {
389               cur.imm |= 0xfffff000;
390            }
391         }
392         break;
393      case 0b0110011:
394         cur.op_name = r_type[{cur.funct7, cur.funct3}];
395         cur.type = argd12;
396         break;
397      case 0b0001111:
398         cur.op_name = "unknown_instruction";
399         cur.type = fence;
400         break;
401      case 0b1110011:
402         if (cur.funct7 == 0) {
403            cur.op_name = "ecall";
404         } else {
405            cur.op_name = "ebreak";
```

```
406             }
407             cur.type = empty_instr;
408         break;
409     }
410     return cur;
411 }
412
413 map<size_t, string> symbol_dict;
414 int marker_idx = 0;
415
416 void set_new_marker(const Instruction &instruction, const size_t shift) {
417     if (instruction.type == arg12ml || instruction.type == jal) {
418         if (!symbol_dict.count(instruction.imm + shift)) {
419             symbol_dict[instruction.imm + shift] = "L" + to_string(marker_idx++);
420         }
421     }
422 }
423
424 void print_instruction(const Instruction &instruction, const size_t shift) {
425     if (symbol_dict.count(shift)) {
426         printf("%08zx␣␣␣<%s>:\n", shift, symbol_dict[shift].c_str());
427     }
428     printf("␣␣␣%05zx:\t%08zx\t%7s\t", shift, instruction.data,
      instruction.op_name.c_str());
429     if (instruction.type == arg1d) {
430         printf("%s,␣%s\n",
431         instruction.rd_name.c_str(),
432         instruction.rs1_name.c_str());
433     } else if (instruction.type == argdm) {
434         printf("%s,␣0x%zx\n",
435         instruction.rd_name.c_str(),
436         instruction.imm);
437     } else if (instruction.type == argd1m) {
438         printf("%s,␣%s,␣%d\n",
439         instruction.rd_name.c_str(),
440         instruction.rs1_name.c_str(),
441         (int) instruction.imm);
442     } else if (instruction.type == argd1s) {
443         printf("%s,␣%s,␣%zd\n",
444         instruction.rd_name.c_str(),
445         instruction.rs1_name.c_str(),
446         instruction.shamt);
447     } else if (instruction.type == argd12) {
448         printf("%s,␣%s,␣%s\n",
449         instruction.rd_name.c_str(),
450         instruction.rs1_name.c_str(),
451         instruction.rs2_name.c_str());
452     } else if (instruction.type == arg12ml) { // b-type
453         printf("%s,␣%s,␣0x%zx␣<%s>\n",
454         instruction.rs1_name.c_str(),
455         instruction.rs2_name.c_str(),
456         instruction.imm + shift,
```

```
457         symbol_dict[instruction.imm + shift].c_str());
458     } else if (instruction.type == lsdm1) {
459         printf("%s, %zd(%s)\n",
460         instruction.rd_name.c_str(),
461         instruction.imm,
462         instruction.rs1_name.c_str());
463     } else if (instruction.type == ls2m1) {
464         printf("%s, %zd(%s)\n",
465         instruction.rs2_name.c_str(),
466         instruction.imm,
467         instruction.rs1_name.c_str());
468     } else if (instruction.type == jal) {
469         printf("%s, 0x%zx <%s>\n",
470         instruction.rd_name.c_str(),
471         instruction.imm + shift,
472         symbol_dict[instruction.imm + shift].c_str());
473     } else {
474         printf("\n");
475     }
476 }
477
478 void print_symbol(vector<Symbol> symbols, const int i) {
479     printf("[%4i] 0x%-15zX %5zu %-8s %-8s %-8s %6s %s\n", i,
480     symbols[i].st_value,
481     symbols[i].st_size,
482     symbols[i].type.c_str(),
483     symbols[i].bind.c_str(),
484     symbols[i].vis.c_str(),
485     symbols[i].index.c_str(),
486     symbols[i].name.c_str());
487
488 }
489
490 int main(int argc, char **argv) {
491     if (argc != 3) {
492         cerr << "incorrect arg count\n";
493         return 1;
494     }
495
496     FILE *input = fopen(argv[1], "rb");
497     size_t file_len;
498
499     if (input) {
500         fseek(input, 0, SEEK_END);
501         file_len = ftell(input);
502         fseek(input, 0, SEEK_SET);
503     } else {
504         cerr << "can't open input file\n";
505         return 1;
506     }
507
508     byte *bytes = (byte *) malloc(file_len);
```

```
509     fread(bytes, 1, file_len, input);
510     fclose(input);
511 
512     Header header = make_header(bytes);
513     if (!check_header(header)) {
514        free(bytes);
515        cerr << "file␣is␣not␣supported\n";
516        return 1;
517     }
518 
519     size_t names_pointer = header.e_shoff + header.e_shentsize * header.e_shstrndx;
520     size_t names_offset = bytes_to_int(bytes, names_pointer + 16, names_pointer +
        20);
521 
522     vector<Section> sections(header.e_shnum);
523     Section *symtab_section;
524     Section *strtab_section;
525     Section *text_section;
526 
527     for (int i = 1; i < header.e_shnum; i++) {
528        sections[i] = make_section(bytes, header.e_shoff + i * header.e_shentsize,
        names_offset);
529        if (sections[i].name == ".symtab") {
530           symtab_section = &sections[i];
531        }
532        if (sections[i].name == ".strtab") {
533           strtab_section = &sections[i];
534        }
535        if (sections[i].name == ".text") {
536           text_section = &sections[i];
537        }
538     }
539 
540     vector<Symbol> symbols(symtab_section->sh_size);
541     for (int i = 0; i * symtab_section->sh_entsize < symtab_section->sh_size; i++)
        {
542        make_symbol(&symbols[i], bytes, symtab_section->sh_offset + i *
        symtab_section->sh_entsize,
543        strtab_section->sh_offset);
544        if (symbols[i].type == "OBJECT" || symbols[i].type == "FUNC") {
545           symbol_dict[symbols[i].st_value] = symbols[i].name;
546        }
547     }
548 
549     for (int i = 0; i * 4 < text_section->sh_size; i++) {
550        size_t data = bytes_to_int(bytes + text_section->sh_offset + i * 4, 0, 4);
551        Instruction instruction = make_instruction(data);
552        set_new_marker(instruction, text_section->sh_addr + i * 4);
553     }
554 
555     if (!freopen(argv[2], "wt", stdout)) {
556        free(bytes);
```

```
557         cerr << "can't␣open␣output␣file\n";
558         return 1;
559     }
560 
561     cout << ".text\n";
562     for (int i = 0; i * 4 < text_section->sh_size; i++) {
563         size_t data = bytes_to_int(bytes + text_section->sh_offset + i * 4, 0, 4);
564         Instruction instruction = make_instruction(data);
565         print_instruction(instruction, text_section->sh_addr + i * 4);
566     }
567     cout << "\n.symtab\n";
568     printf("%6s␣%-17s␣%5s␣%-8s␣%-8s␣%-8s␣%6s␣%s\n", "Symbol", "Value", "Size",
    "Type", "Bind", "Vis", "Index", "Name");
569     for (int i = 0; i * symtab_section->sh_entsize < symtab_section->sh_size; i++)
    {
570         print_symbol(symbols, i);
571     }
572     free(bytes);
573     fclose(stdout);
574     return 0;
575 }
```