Лабораторная работа №3	M3139	2022
ISA	Матвеев Андрей Денисович	

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

Инструментарий и требования к работе: Язык C++, компилятор GCC 8.1.0

Описание

Необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Вариант 76

1 Система кодирования команд RISC-V.

1.1 Виды RISC-V

Первым делом надо сказать, что RISC-V представляет собой семейство родственных ISA. Каждый базовый набор команд характеризуется шириной регистров, соответствующим размером адресного пространства и количеством регистров.

Система команд состоит из базового набора команд и расширений. Предусмотрены 32-, 64- и 128-битные реализации RISC-V. Для 32- и 64 битных версий определены следующие подмножества системы команд:

1) RV32I (базовый набор) — содержит команды целочисленной арифметики, ветвления, доступа к памяти и системных вызовов, общие для 32- и 64- битной версий;
2) RV64I (дополнение к RV32I) — содержит дополнительные команды целочисленной арифметики и доступа к памяти для 64-битной версии;
3) RV32M (стандартное расширение) — содержит команды целочисленного умножения и деления, общие для 32- и 64-битной версий;
4) RV64M (дополнение к RV32M) — содержит дополнительные команды целочисленного умножения и деления для 64-битной версии;
5) RV32A (стандартное расширение) — содержит команды атомарных операций над данными из памяти для 32- и 64-битной версий;
6) RV64A (дополнение к RV32A) — содержит дополнительные команды атомарных операций над данными из памяти для 64-битной версии.

7) RV32F (стандартное расширение) — содержит команды арифметики с плавающей точкой одинарной точности и команды конвертации в целочисленный формат, общие для 32- и 64-битной версий.

8) RV64F (дополнение к RV32F) — содержит дополнительные команды конвертации для 64-битной версии.

9) RV32D (стандартное расширение) — содержит команды арифметики с плавающей точкой двойной точности и команды конвертации в целочисленный формат, общие для 32- и 64-битной версий.

10) RV64D (дополнение к RV32D) — содержит дополнительные команды конвертации для 64-битной версии.

11) RV32C (стандартное расширение) — содержит сжатые (16-битные) команды, общие для 32- и 64- битной версий.

12) RV64C (дополнение к RV32C) — содержит дополнительные сжатые (16-битные) команды для 64- битной версии.

В нашей лабораторной работе мы будем подробно рассматривать RV32I и RV32M, поэтому сфокусируемся на этих наборах команд..

1.2 Регистры

RISC-V использует 32 регистра названных x0-x31 (рисунок 1).

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

Рисунок 1 – Регистры в RISC-V

Первый регистр, x0 имеет специальное назначение, он содержит 0. Вне зависимости от того, какое значение вы в него записываете, при чтении из этого регистра вы всегда получите 0. Это может показаться странным, но это очень практично. Существует множество операций, для которых вам нужен ноль в качестве операнда. Если у вас есть регистр, равный нулю, вам не нужно копировать ноль в регистр.

Второй регистр, x1 имеет специальное название и обозначение (ra = return address) он используется для записи адреса возврата перед вызовом подпрограммы.

Третий регистр, x2 (sp = stack pointer) – указатель на положение в стеке

Четвёртый регистр, x3 (gp = global pointer) – глобальный указатель

Пятый регистр, x4 (tp = thread pointer) – указатель потока

Регистры x5-x7, x28-x31 (t0-t6 = temporary registers) - регистры временных переменных. Подпрограммы не обязаны их сохранять.

Регистры x8, x9, x18-x27 (s0-s11 = saved registers) - сохраняемые регистры. Подпрограммы обязаны сохранять их состояние.

Регистры x10-x17 (a0-a7 = function registers) - аргументы функций. Перед вызовом подпрограммы вы передаёте аргументы в эти регистры.

Есть еще один регистр, видимый пользователю: pc (program counter) содержит адрес текущей инструкции

1.3 Общий вид инструкции

Инструкции в RISC-V имеют длину 32 бита, они бывают шести видов (рисунок 2) - Register/Immediate/Store/Upper immediate/Branch/Jump

Format	Bit																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

Рисунок 2 – Виды инструкций RISC-V и их формат

В opcode, funct3 и funct7 хранятся информация о типе инструкции. rs1 и rs2 определяют регистры, в которых хранятся аргументы инструкции, а в rd – регистр, в который записывается ответ. Поля imm (immediates) нужны для констант. Давайте для каждого вида рассмотрим основные инструкции.

1.4 Register-Immediate инструкции

Register-Immediate инструкции имеют следующий вид (рисунок 3)

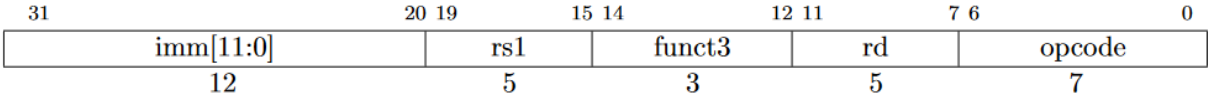

Рисунок 3 – Общий вид Register-Immediate инструкции

1) ADDI - прибавляет 12-битную константу к регистру rs1.

Пример: ADDI rd, rs1, CONST

2) SLTI - помещает значение 1 в регистр rd, если регистр rs1 меньше константы, иначе в rd записывается 0.

Пример: SLTI rd, rs1, CONST

3) ANDI, ORI, XORI — это логические операции, которые выполняют побитовые AND, OR и XOR с регистром rs1 и 12-битной константой. Результат помещается в rd.

Пример: XORI rd, rs1, CONST

4) SLLI – Логический сдвиг влево.

5) SRLI, SRAI – соответственно логический и арифметический сдвиг вправо.

6) LUI (load upper immediate) используется чтобы записать 32-битную константу в регистр rd. Из рисунка №4 видно, что мы можем влиять только на биты в промежутке 31-12, поэтому младшие 12 битов заполняются нулями.

7) AUIPC (add upper immediate to pc) – добавляет такую же, как и lui (12 младших битов заполняются нулями), константу к старшим битам PC

Вообще две последние инструкции выделяются в отдельную группу Upper-Immediate, но я решил упомянуть их сразу со всеми immediate инструкциями.

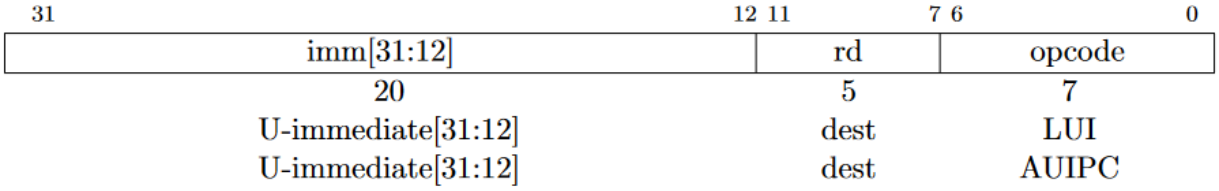

Рисунок 4 – Upper-Immediate инструкции

1.5 Register-Register инструкции

Register-Register инструкции имеют следующий вид (рисунок 5)

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Рисунок 5 – Общий вид reg-reg инструкции

1) ADD, SUB, AND, OR, SLT, SLTU, XOR, SLL, SRL, SRA – инструкции, работающие так же как и свои register-immediate аналоги, но их аргументы – два регистра.

1.6 Jump инструкции

1) JALR (jump and link register) – инструкция, предназначенная для перехода к какому-либо адресу. Регистр rd используется для хранения адреса инструкции, к которой надо будет вернуться (pc + 4). Адрес, на который надо перейти, вычисляется, как сумма immediate[11:0] и rs1, с последующем занулением младшего бита (рисунок 6). Обычно, в качестве rd используют x1. Регистр x0 используется, если нам не важно, куда мы вернёмся.

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

Рисунок 6 – схема кодирования JALR

2) JAL (jump and link) –работает, как и предыдущая инструкция, только она по-другому декодируется (рисунок 7)

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
offset[20:1]				dest	JAL

Рисунок 7 – схема кодирования JAL.

Как мы видим здесь младший бит не заполняется, он автоматически считается равным нулю, также не используется переменный адрес rs1. За счёт этого достигается большой разброс значений прыжка (+-1 мегабайт)

1.7 Branch инструкции

Branch инструкции имеют следующий вид (рисунок 8)

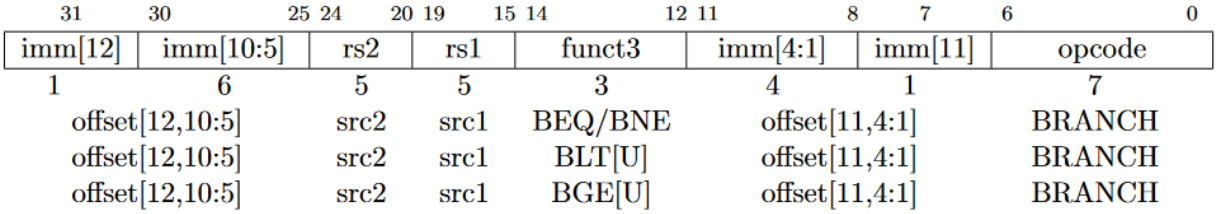

Рисунок 8 – общий вид branch инструкции

1) BEQ, BNE (branch equal/not equal) – эти инструкции сравнивают значения в регистрах rs1, rs2 и прибавляет к pc некоторое константное число, если rs1 равен/не равен rs2.

2) BLT, BLTU – аналогично, но переход происходит, если rs1 меньше rs2. BLTU нужно, чтобы сравнивать unsigned числа.

Также есть BGE, BGEU, BLE, BLEU – они работают аналогично, но условия перехода немного отличаются.

1.8 Load инструкции

Load инструкции имеют следующий вид (рисунок 9)

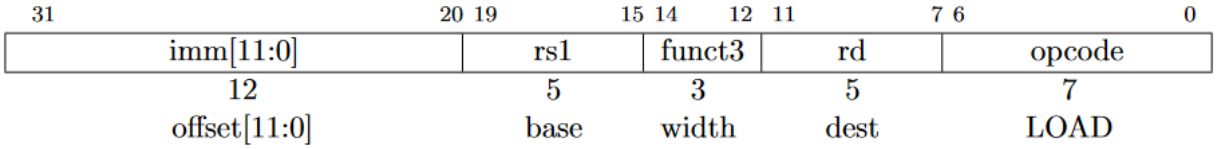

Рисунок 9 – общий вид load инструкции

1) LW, LB (load word, load byte) – Используются для загрузки из памяти в регистр слова(4 байта) или байта, в зависимости от инструкции. В регистр rd записывается значение из памяти по адресу imm[11:0] + rs1.

1.9 Store инструкции

Store инструкции имеют следующий вид (рисунок 10)

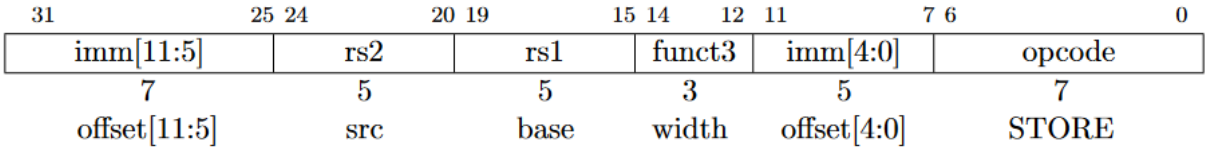

Рисунок 10 – общий вид store инструкции

1) SW, SB (store word, store byte) – Предназначены для загрузки в память значения из регистра. Значение в регистре rs2 копируется и записывается по адресу immediate + rs1.

2-3. Структура ELF-файла + описание работы кода.

Структура ELF-файла

ELF-файл – исполняемый файл для UNIX'овых систем. Этот файл содержит много различного контента, но мы будем говорить только о том, что нужно для дизассемблера.

Сначала идёт заголовок ELF-файла (рисунок 11). Давайте рассмотрим все поля этого заголовка, которые пригодятся нам.

```
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;
```

Рисунок 11 – Структура заголовка elf-файла

Первые 4 байта - сигнатура файла (магическое число). Они определяют является ли этот файл elf-файлом или нет. Эти 4 байта должны быть равны 7f 45 4c 46 (7f ELF).

Следующий байт определяет битность elf-файла. 1/2 значит, что файл 32/64 битный. (Далее всегда будет подразумеваться 32-битный elf-файл)

Следующий байт определяет формат записи чисел. 1/2 значит, что формат little endian/big endian

Рассмотрим байт с offset'ом 0x12, он определяет ISA. В нашем случае этот байт равен 0xf3, что соответствует RISC-V.

Следующее важное поле имеет offset 0x20, оно имеет размер 4 байта. Это поле имеет название e_shoff (section header offset), оно содержит отступ до section header.

Также, прочитаем поле e_shnum, это поле содержит количество элементов section header'a.

Следующий код проверяет первые четыре поля на соответствие (в случае ошибки выводится сообщение, и программа завершается). Также читается e_shoff и e_shnum

```
    void check_is_elf() {
        if (buf[0] != 0x7f || buf[1] != 0x45 || buf[2] !=
0x4c || buf[3] != 0x46) {
            error_exit("Input file is not correct elf file");
        }
    }

    void check_bit_depth() {
        if (buf[E_OFFSET_BIT_DEPTH] != 1) {
            error_exit("Only 32-bit elf files are
supported");
        }
    }

    void check_endianness() {
        if (buf[E_OFFSET_ENDIANNESS] != 1) {
            error_exit("Only little-endian elf files are
supported");
        }
    }

    void check_isa() {
        if (buf[E_OFFSET_ISA] != RISCV_ID) {
            error_exit("Only RISC-V ISA is supported");
        }
    }
```

```
check_is_elf();
check_bit_depth();
check_endianness();
check_isa();

uint32_t e_shoff = get_section_offset();
uint16_t e_shnum = get_eshnum();
```

С заголовком ELF-файла разобрались. Наша задача распарсить две секции: symtab и text. Но сейчас мы распарсим strtab, т. к. он нам понадобится. По адресу e_shoff находится таблица заголовков, из неё мы и достанем информацию о strtab, symtab и text. Таблица заголовков представляет из себя массив структур размера 26 байт. Давайте перебирать эти структуры и проверять, какая из структур отвечает за strtab (проверка делается с помощью поля sh_type). Количество strtab'ов может быть больше одного, поэтому для каждого strtab'а в массив strtab_offset запишем его адрес начала. Оффсет данных заголовка можно достать из поля sh_offset. Следующий листинг кода показывает, как заполняется массив strtab_offset. Функция is_strtab просто проверяет поле sh_type.

```
    for (uint32_t i = e_shoff, j = 0;i < file_length - 
ELF32_SHDR_SIZE; i += ELF32_SHDR_SIZE) {
            if (is_strtab(i)) {
                strtab_offset[j] = get_sh_offset(i);
                j++;
            }
        }
```

Теперь в таблице заголовков найдём symtab. Единственные поля, которые нам нужны: sh_size, sh_offset. sh_offset нам указывает реальный адрес symtab, а в sh_size записан размер symtab в байтах. Теперь, зная sh_offset можно распарсить symtab.

Symtab представляет из себя массив структур размера 16 байт (в 32-битном ELF-файле).

```
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;
```

Рисунок 12 – Структура symtab

Давайте перебирать эти структуры. Вся информация, кроме имени хранится в структуре. Имя же лежит по адресу strtab_offset[sh_name] + st_name, где st_name достаётся из структуры. Symtab хранит информацию о функциях, используемых в программе, чтобы отличить функцию от чего-то другого надо проверить поле st_info. Если текущая структура оказалась функцией, то запишем в functions_mapping map<int, string> пару [адрес ф-ии, имя ф-ии], также, если имя функции - “main”, то запишем адрес этой функции в отдельную переменную main_address. Следующий листинг кода делает эту работу.

```
    for (uint32_t i = e_shoff; (i - e_shoff) /
ELF32_SHDR_SIZE < e_shnum; i += ELF32_SHDR_SIZE) {
            uint32_t sh_name = get_sh_name(i);
            if (is_symtab(i)) {
                uint32_t sh_size = get_sh_size(i);
                uint32_t sh_offset = get_sh_offset(i);

                int32_t cnt = 0;

                for (uint32_t j = sh_offset; cnt < sh_size /
SYMTAB_SIZE; j += SYMTAB_SIZE, cnt++) {
                    uint32_t st_name = read32(j);
                    if (st_name > 0) {
                        std::string name =
get_name(strtab_offset[sh_name] + st_name);
                        if (id_to_type[buf[j + 12] & 0xf] ==
"FUNC") {
```

```
                            function_mapping[read32(j + 4)] =
name;
                            if (name == "main") {
                                main_address = read32(j + 4);
                            }
                        }
                    }
                }
            }
        }
```

Теперь, надо распарсить секцию text. Опять же пробегаемся по всем структурам в таблице заголовков и ищем структуру, соответствующую секции text. У каждой структуры будем смотреть поле sh_type и sh_offset. sh_type должен быть равен 0x1 (program data), а sh_offset должен быть равен main_address, т. к. main – это точка входа программы. Теперь достанем из структуры поле sh_size и пойдём парсить text секцию, зная её размер и адрес начала.

Для парсинга нам потребуется ещё “functions mapping” – map, который каждому адресу функции ставит в соответствие её имя. Его мы построили, когда парсили symtab.

Следующий листинг кода ищет text секцию

```
for (uint32_t i = e_shoff;i < file_length - ELF32_SHDR_SIZE;
i += ELF32_SHDR_SIZE) {
        if (is_progbits(i)) {
            disassembly(i, function_mapping,
get_sh_size(i));
            break;
        }
    }
```

is_progbits как раз проверяет sh_offset и sh_type на соответствие.

В программе дизассемблирование осуществляет функция disassembly, которая принимает адрес main’а, маппинг [адрес-имя функции] и размер самой секции. Листинг кода функции disassembly:

```
    void disassembly(uint32_t offset, std::map<uint32_t,
std::string> functions_mapping, uint32_t code_size) {
        printf(".text\n");
        uint32_t sh_offset = get_sh_offset(offset);
```

```
        uint32_t sh_size = get_sh_size(offset);
        uint32_t cur = main_address;
        uint32_t mark_index = 0;
        for (uint32_t j = sh_offset; j < sh_offset +
code_size; j += sizeof(uint32_t)) {
            add_mark(read32(j), cur, mark_index,
functions_mapping);
            cur += sizeof(uint32_t);
        }
        cur = main_address;
        for (uint32_t j = sh_offset; j < sh_offset +
code_size; j += sizeof(uint32_t)) {
            if (functions_mapping.count(cur) > 0) {
                printf("%08x   <%s>:\n", cur,
functions_mapping[cur].c_str());
            }
            std::pair<std::string, std::string> instruction =
parse_instruction(read32(j), cur, functions_mapping);
            printf("   %05x:\t%08x\t%7s\t%s\n", cur,
read32(j), instruction.first.c_str(),
instruction.second.c_str());
            cur += sizeof(int32_t);
        }
        printf("\n");
    }
```

Первый “for” перебирает инструкции и вызывает функцию add_mark, которая смотрит на инструкцию и проверяет, ссылается ли эта инструкция на какую-то метку, которой в маппинге нет. Если ссылается, то в маппинг добавляется пара [адрес, “L%i”].

Теперь второй “for”. Итерируясь по блокам из 4 байтов, я паршу каждую инструкцию отдельно в функции parse_instruction, и вывожу её. Давайте рассмотрим функцию parse_instruction. Для начала, из 4-х байтового значения мы делаем структуру “instruction” (код снизу), в которой есть данные, использующиеся во многих инструкциях (rs1, rs2, rd, funct3, funct7).

```
    struct instruction {
        uint32_t value;

        uint32_t opcode;
        uint32_t rd;
        uint32_t funct3;
        uint32_t rs1;
        uint32_t rs2;
        uint32_t funct7;
```

```
    instruction(uint32_t x) : value(x) {
        opcode = x & ((1 << OPCODE_LEN) - 1);
        x >>= OPCODE_LEN;
        rd = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        funct3 = x & ((1 << FUNCT3_LEN) - 1);
        x >>= FUNCT3_LEN;
        rs1 = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        rs2 = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        funct7 = x;
    }

    uint16_t immediate12() {
        return value >> (32 - 12) & ((1 << 12) - 1);
    }

    uint32_t immediate20() {
        return value >> (32 - 20) & ((1 << 20) - 1);
    }

    uint32_t immediate7() {
      return value >> (32 - 7) & ((1 << 7) - 1);
    }
};
```

Как мы знаем, инструкции делятся на группы, в своём дизассемблере я тоже поделил инструкции на группы.

Группа REG_REG соответствует своему аналогу описаному в документации risc-v. Группу Immediate мы поделим на две подгруппы IMMEDIATE_ARITHMETIC и IMMEDIATE_MEMORY. Группы BRANCH и STORE никак не изменены. Остальные команды мы вынесем в отдельные группы из одной инструкции, такими командами оказались: lui, auipc, jal, jalr, ecall, ebreak. В самой функции parse_instruction просто делается switch по группе инструкции (по opcode'у) и уже в зависимости от группы парсится интсрукция. Здесь уже нет никакой умственной работы, мне пришлось захардкодить map<int, string> отображение, которое в соответствие funct3 или funct3 | funct7 << 3 ставит строковое представление команды. Рассмотрим несколько интересных инструкций, которые не так уж и просто было реализовать. Команда jal запоминает адрес следующей инструкции и прыгает на другую инструкцию по заданному адресу, т. е. по сути – это вызов функции. В ТЗ просилось выводить имя функции, которую мы вызываем, вот тут-то и

пригодилась мапа functions_mapping, с помощью неё я вывожу имя функции. Листинг кода, для парсинга jal:

```
    case JAL: {
            uint32_t address_to_jump = cur_address +
get_jal_offset(instr.value);
            command_name = "jal";
            arguments = reg_mapping(instr.rd) + ", " +
to_hex(address_to_jump, -1) + " <" +
functions_mapping[address_to_jump] + ">";
            break;
        }
```

Также рассмотрим семейство инструкций BRANCH. Они тоже используют function_mapping, т. к. для этих инструкций требуется вывод адреса выбираемой ветки в таком формате: “имя_функции + offset”. Плюсом при парсинге BRANCH инструкции используется метод functions_mapping.upper_bound(), эта функция нужна для нахождения функции, в которую мы перейдём, если условие branch выполнится. Листинг кода, для парсинга команд вида BRANCH:

```
    case BRANCH: {
            int32_t offset = get_branch_offset(instr.value);
            uint32_t address_to_jump = cur_address + offset;
            command_name = branch_mapping[instr.funct3];
            arguments = reg_mapping(instr.rs1) + ", " +
reg_mapping(instr.rs2) + ", " + to_hex(address_to_jump, -1)
+ " <" + functions_mapping[address_to_jump] + ">";
            break;
        }
```

Осталось только перевести всё в строковое представление. Это не сложная, но достаточно муторная работа.

Наконец-то вернёмся к symtab и выведем всю эту таблицу. От предыдущего парсинга symtab код будет отличаться только тем, что надо достать все поля из структуры и вывести их. Также, заполнять map<int, string> function_mapping – не надо.

Листинг кода:

```
for (uint32_t i = e_shoff; (i - e_shoff) / ELF32_SHDR_SIZE <
e_shnum; i += ELF32_SHDR_SIZE) {

        uint32_t sh_name = get_sh_name(i);

        if (is_symtab(i)) {

            uint32_t sh_size = get_sh_size(i);
```

```
            uint32_t sh_offset = get_sh_offset(i);
            printf(".symtab\nSymbol Value
Size Type     Bind     Vis         Index Name\n");


            int32_t cnt = 0;


            for (uint32_t j = sh_offset; cnt < sh_size /
SYMTAB_SIZE; j += SYMTAB_SIZE, cnt++) {
                printf("[%4i] 0x%-15X %5i %-8s %-8s %-8s ",
cnt, read32(j + 4),
                 read32(j + 8), id_to_type[buf[j + 12] &
0xf].c_str(), id_to_bind[buf[j + 12] >> 4].c_str(),
                 id_to_visibility[buf[j + 13]].c_str());
                uint32_t ndx_bytes = read16(j + 14);
                std::string ndx = std::to_string(ndx_bytes);
                if (id_to_ndx.find(ndx_bytes) !=
id_to_ndx.end()) {
                    ndx = id_to_ndx[ndx_bytes];
                }
                printf("%6s ", ndx.c_str());
                uint32_t st_name = read32(j);
                if (st_name > 0) {
                    std::string name =
get_name(strtab_offset[sh_name] + st_name);
                    printf("%s", name.c_str());
                }
                printf("\n");
            }
        }
    }
```

4. Результат работы программы

```
.text
00010074   <main>:
   10074:ff010113    addi  sp, sp, -16
   10078:00112623      sw  ra, 12(sp)
   1007c:030000ef     jal  ra, 100ac <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, 0x100000
   10098:fddff0ef     jal  ra, 10074 <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, 0x11000
   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, 100e4 <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, 100d8 <L1>
   10114:050f0f13    addi  t5, t5, 80
   10118:01478513    addi  a0, a5, 20
   1011c:fa5f16e3     bne  t5, t0, 100c8 <L2>
   10120:00008067    jalr  zero, 0(ra)

.symtab
Symbol Value              Size Type     Bind     Vis
Index Name
[   0] 0x0                    0 NOTYPE   LOCAL    DEFAULT
UND
```

```
[   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
UND _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
```

Источники информации:

https://www.researchgate.net/publication/328184731_MicroTESK-Based_Test_Program_Generator_for_the_RISC-V_Architecture

https://en.wikipedia.org/wiki/RISC-V#Design

(volume 1): https://riscv.org/technical/specifications/

https://habr.com/ru/post/558706/

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-79797.html

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

https://ejudge.ru/study/3sem/elf.html

https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-73709.html

Листинг кода

main.cpp

```
#include <iostream>
#include <algorithm>
#include <map>
#include <string>

uint32_t file_length;

const uint32_t MAX_SIZE = 1e5;
const uint32_t MAX_STRTAB_COUNT = 1e4;
uint8_t* buf;
uint32_t strtab_offset[MAX_STRTAB_COUNT];
uint32_t main_address = 0;

const uint8_t RISCV_ID = 0xf3;
const uint8_t SYMTAB_ID = 0x02;
const uint8_t STRTAB_ID = 0x03;
const uint8_t PROGBITS_ID = 0x01;

const uint32_t E_OFFSET_BIT_DEPTH = 0x04;
const uint32_t E_OFFSET_ENDIANNESS = 0x05;
const uint32_t E_OFFSET_ISA = 0x12;
const uint32_t E_OFFSET_SECTION_HEADER = 0x20;
const uint32_t E_OFFSET_SECTION_HEADER_TABLE_INDEX =
0x32;
const uint32_t E_OFFSET_ESHNUM = 0x30;
const uint32_t SH_OFFSET_TYPE = 0x04;
const uint32_t SH_OFFSET_OFFSET = 0x10;
```

```
    const uint32_t SH_OFFSET_SIZE = 0x14;


    const uint32_t ELF32_SHDR_SIZE = sizeof(int32_t) * 10; //
10 - количество полей в одном элементе массива
    const uint32_t SYMTAB_SIZE = 4 * 3 + 2 + 1 + 1;


    std::map<uint8_t, std::string> id_to_type = {{0,
"NOTYPE"}, {1, "OBJECT"}, {2, "FUNC"},
     {3, "SECTION"}, {4, "FILE"}, {5, "COMMON"},
      {6, "TLS"}, {10, "LOOS"}, {12, "HIOS"}, {13, "LOPROC"},
{15, "HIPROC"}};
    std::map<uint8_t, std::string> id_to_bind = {{0,
"LOCAL"}, {1, "GLOBAL"}, {2, "WEAK"},
       {10, "LOOS"}, {12, "LOPROC"}, {15, "HIPROC"}};
    std::map<uint8_t, std::string> id_to_visibility = {{0,
"DEFAULT"}, {1, "INTERNAL"}, {2, "HIDDEN"},
       {3, "PROTECTED"}, {4, "EXPORTED"}, {5, "SINGLETON"},
{6, "ELIMINATE"}};
    std::map<uint16_t, std::string> id_to_ndx = {{0, "UND"},
{0xff00, "LORESERVE"},
    {0xff01, "AFTER"}, {0xff02, "AMD64_LCOMMON"}, {0xff1f,
"HIPROC"}, {0xff20, "LOOS"}, {0xff3f, "LOSUNW"},
    {0xfff1, "ABS"}, {0xfff2, "COMMON"}, {0xffff, "XINDEX"}};


    void error_exit(std::string message) {
        std::cerr << message << std::endl;
        exit(1);
    }


    std::string to_hex(int n, int len) {
        if (n == 0 && len == -1) return "0";
```

```
        std::string res = "";
        for (int i = 0; i < len || (len == -1 && n > 0); i++)
{
            if ((n & 15) < 10) {
                res.push_back('0' + (n & 15));
            } else {
                res.push_back('a' + (n & 15) - 10);
            }
            n >>= 4;
        }
        reverse(res.begin(), res.end());
        return res;
    }

    FILE* open_file(const char* name, char* mode) {
        FILE* file = fopen(name, mode);
        if (file == nullptr) {
            error_exit("Couldn't open file: ");
        }
        return file;
    }

    void check_is_elf() {
        if (buf[0] != 0x7f || buf[1] != 0x45 || buf[2] !=
0x4c || buf[3] != 0x46) {
            error_exit("Input file is not correct elf file");
        }
    }
```

```
    void check_bit_depth() {
        if (buf[E_OFFSET_BIT_DEPTH] != 1) {
            error_exit("Only 32-bit elf files are
supported");
        }
    }

    void check_endianness() {
        if (buf[E_OFFSET_ENDIANNESS] != 1) {
            error_exit("Only little-endian elf files are
supported");
        }
    }

    void check_isa() {
        if (buf[E_OFFSET_ISA] != RISCV_ID) {
            error_exit("Only RISC-V ISA is supported");
        }
    }

    uint32_t read32(uint32_t address) {
        return buf[address] | buf[address + 1] << 8 |
buf[address + 2] << 16 | buf[address + 3] << 24;
    }

    uint16_t read16(uint32_t address) {
        return buf[address] | buf[address + 1] << 8;
    }
```

```
    uint32_t get_section_offset() {
        return read32(E_OFFSET_SECTION_HEADER);
    }

    uint16_t get_shstrndx() {
        return read16(E_OFFSET_SECTION_HEADER_TABLE_INDEX);
    }

    uint16_t get_eshnum() {
        return read16(E_OFFSET_ESHNUM);
    }

    bool is_strtab(uint32_t address) {
        return buf[address + SH_OFFSET_TYPE] == STRTAB_ID;
    }

    bool is_symtab(uint32_t address) {
        return buf[address + SH_OFFSET_TYPE] == SYMTAB_ID;
    }

    uint32_t get_sh_addr(uint32_t address) {
        return read32(address + 3 * 4);
    }

    bool is_progbits(uint32_t address) {
        return buf[address + SH_OFFSET_TYPE] == PROGBITS_ID
&& get_sh_addr(address) == main_address;
    }
```

```
    uint32_t get_sh_offset(uint32_t address) {
        return read32(address + SH_OFFSET_OFFSET);
    }

    uint32_t get_sh_name(uint32_t address) {
        return read32(address);
    }

    uint32_t get_sh_size(uint32_t address) {
        return read32(address + SH_OFFSET_SIZE);
    }

    std::string get_name(int address) {
        std::string result = "";
        for (uint32_t i = address; i < file_length && buf[i]
!= 0; i++) {
            result.push_back((char)buf[i]);
        }
        return result;
    }

    const uint8_t OPCODE_LEN = 7;
    const uint8_t REG_SIZE = 5;
    const uint8_t FUNCT3_LEN = 3;



    const uint8_t REG_REG = 0b0110011;
```

```
const uint8_t ADD_SUB = 0b000;

const uint8_t ADD = 0b0000000;
const uint8_t SUB = 0b0100000;

const uint8_t SLL = 0b001;
const uint8_t SLT = 0b010;
const uint8_t SLTU = 0b011;
const uint8_t XOR = 0b100;
const uint8_t SR = 0b101;

const uint8_t SRL = 0b0000000;
const uint8_t SRA = 0b0100000;

const uint8_t OR = 0b110;
const uint8_t AND = 0b111;
const uint8_t MUL = 0b000;
const uint8_t MULH = 0b001;
const uint8_t MULHSU = 0b010;
const uint8_t MULHU = 0b011;
const uint8_t DIV = 0b100;
const uint8_t DIVU = 0b101;
const uint8_t REM = 0b110;
const uint8_t REMU = 0b111;

const uint8_t RV32M_FUNCT7 = 1;

std::map<uint16_t, std::string> reg_reg_mapping = {
```

```
    {ADD << FUNCT3_LEN, "add"},
    {SUB << FUNCT3_LEN, "sub"},
    {SLL, "sll"},
    {SLT, "slt"},
    {SLTU, "sltu"},
    {XOR, "xor"},
    {SR | SRL << FUNCT3_LEN, "srl"},
    {SR | SRA << FUNCT3_LEN, "sra"},
    {OR, "or"},
    {AND, "and"},
    {MUL | RV32M_FUNCT7 << FUNCT3_LEN, "mul"},
    {MULH | RV32M_FUNCT7 << FUNCT3_LEN, "mulh"},
    {MULHSU | RV32M_FUNCT7 << FUNCT3_LEN, "mulhsu"},
    {MULHU | RV32M_FUNCT7 << FUNCT3_LEN, "mulhu"},
    {DIV | RV32M_FUNCT7 << FUNCT3_LEN, "div"},
    {DIVU | RV32M_FUNCT7 << FUNCT3_LEN, "divu"},
    {REM | RV32M_FUNCT7 << FUNCT3_LEN, "rem"},
    {REMU | RV32M_FUNCT7 << FUNCT3_LEN, "remu"}
};

const uint8_t IMMEDIATE_ARITHMETIC = 0b0010011;
const uint8_t ADDI = 0b000;
const uint8_t SLTI = 0b010;
const uint8_t SLTIU = 0b011;
const uint8_t XORI = 0b100;
const uint8_t ORI = 0b110;
const uint8_t ANDI = 0b111;
const uint8_t SLLI = 0b001;
```

```
    const uint8_t SRI = 0b101;


    std::map<uint8_t, std::string>
immediate_arithmetic_mapping = {
        {ADDI, "addi"},
        {SLTI, "slti"},
        {SLTIU, "sltiu"},
        {XORI, "xori"},
        {ORI, "ori"},
        {ANDI, "andi"},
        {SLLI, "slli"},
        {SRI, "sri"}
    };



    const uint8_t IMMEDIATE_MEMORY = 0b0000011;
    const uint8_t LB = 0b000;
    const uint8_t LH = 0b001;
    const uint8_t LW = 0b010;
    const uint8_t LBU = 0b100;
    const uint8_t LHU = 0b101;


    std::map<uint8_t, std::string> immediate_memory_mapping =
{
        {LB, "lb"},
        {LH, "lh"},
        {LW, "lw"},
        {LBU, "lbu"},
        {LHU, "lhu"}
```

```
};

const uint8_t LUI = 0b0110111;
const uint8_t AUIPC = 0b0010111;
const uint8_t JAL = 0b1101111;

const uint8_t JALR = 0b1100111;
const uint8_t E = 0b1110011;
const uint16_t ECALL = 0b000000000000;
const uint16_t EBREAK = 0b000000000001;

const uint8_t BRANCH = 0b1100011;
const uint8_t BEQ = 0b000;
const uint8_t BNE = 0b01;
const uint8_t BLT = 0b100;
const uint8_t BGE = 0b101;
const uint8_t BLTU = 0b110;
const uint8_t BGEU = 0b111;

std::map<uint8_t, std::string> branch_mapping = {
    {BEQ, "beq"},
    {BNE, "bne"},
    {BLT, "blt"},
    {BGE, "bge"},
    {BLTU, "bltu"},
    {BGEU, "bgeu"}
};
```

```
    const uint8_t STORE = 0b0100011;
    const uint8_t SB = 0b000;
    const uint8_t SH = 0b001;
    const uint8_t SW = 0b010;

    std::map<uint8_t, std::string> store_mapping = {
        {SB, "sb"},
        {SH, "sh"},
        {SW, "sw"}
    };

    std::string reg_mapping(uint8_t x) {
        if (x == 0) return "zero";
        else if (x == 1) return "ra";
        else if (x == 2) return "sp";
        else if (x == 3) return "gp";
        else if (x == 4) return "tp";
        else if (x >= 5 && x <= 7) return "t" +
std::to_string(x - 5);
        else if (x >= 8 && x <= 9) return "s" +
std::to_string(x - 8);
        else if (x >= 10 && x <= 17) return 'a' +
std::to_string(x - 10);
        else if (x >= 18 && x <= 27) return "s" +
std::to_string(x - 18 + 2);
        else if (x >= 28 && x <= 31) return "t" +
std::to_string(x - 28 + 3);
    }
```

```
struct instruction {
    uint32_t value;

    uint32_t opcode;
    uint32_t rd;
    uint32_t funct3;
    uint32_t rs1;
    uint32_t rs2;
    uint32_t funct7;

    instruction(uint32_t x) : value(x) {
        opcode = x & ((1 << OPCODE_LEN) - 1);
        x >>= OPCODE_LEN;
        rd = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        funct3 = x & ((1 << FUNCT3_LEN) - 1);
        x >>= FUNCT3_LEN;
        rs1 = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        rs2 = x & ((1 << REG_SIZE) - 1);
        x >>= REG_SIZE;
        funct7 = x;
    }

    uint16_t immediate12() {
        return value >> (32 - 12) & ((1 << 12) - 1);
    }
```

```
    uint32_t immediate20() {
        return value >> (32 - 20) & ((1 << 20) - 1);
    }

    uint32_t immediate7() {
      return value >> (32 - 7) & ((1 << 7) - 1);
    }
};

int32_t get_jal_offset(uint32_t value) {
    value >>= 12;
    int32_t result = 0;
    result |= (value & ((1 << 8) - 1)) << 12;
    value >>= 8;
    result |= (value & 1) << 11;
    value >>= 1;
    result |= (value & ((1 << 10) - 1)) << 1;
    value >>= 10;
    result |= (value & 1) << 20;
    value >>= 1;
    result -= (1 << 21) * (result >> 20);
    return result;
}

int32_t get_branch_offset(uint32_t value) {
    value >>= 7;
    int32_t result = 0;
    result |= (value & 1) << 11;
```

```
        value >>= 1;
        result |= (value & ((1 << 4) - 1)) << 1;
        value >>= 4 + 3 + 5 + 5;
        result |= (value & ((1 << 6) - 1)) << 5;
        value >>= 6;
        result |= (value & 1) << 12;
        result -= (1 << 13) * (result >> 12);
        return result;
    }

    std::pair<std::string, std::string>
parse_instruction(uint32_t x, uint32_t cur_address,
std::map<uint32_t, std::string> functions_mapping) {
        std::string command_name = "";
        std::string arguments = "";
        instruction instr = instruction(x);
        int16_t local_immediate12 = instr.immediate12();
        local_immediate12 -= (1 << 12) * (local_immediate12
>> 11);
        switch (instr.opcode) {
        case REG_REG: {
            command_name = reg_reg_mapping[instr.funct3 |
(instr.funct7 << FUNCT3_LEN)];
            arguments = reg_mapping(instr.rd) + ", " +
reg_mapping(instr.rs1) + ", " + reg_mapping(instr.rs2);
            break;
        }
        case IMMEDIATE_ARITHMETIC: {
            command_name =
immediate_arithmetic_mapping[instr.funct3];
```

```
            arguments = reg_mapping(instr.rd) + ", " +
reg_mapping(instr.rs1) + ", " +
std::to_string(local_immediate12);
            break;
        }
        case IMMEDIATE_MEMORY: {
            command_name =
immediate_memory_mapping[instr.funct3];
            arguments = reg_mapping(instr.rd) + ", " +
std::to_string(local_immediate12) + "(" +
reg_mapping(instr.rs1) + ")";
            break;
        }
        case BRANCH: {
            int32_t offset = get_branch_offset(instr.value);
            uint32_t address_to_jump = cur_address + offset;
            command_name = branch_mapping[instr.funct3];
            arguments = reg_mapping(instr.rs1) + ", " +
reg_mapping(instr.rs2) + ", " + to_hex(address_to_jump, -1)
+ " <" + functions_mapping[address_to_jump] + ">";
            break;
        }
        case STORE: {
            command_name = store_mapping[instr.funct3];
            arguments = reg_mapping(instr.rs2) + ", " +
std::to_string(instr.rd | instr.immediate7() << 5) + "(" +
reg_mapping(instr.rs1) + ")";
            break;
        }
        case LUI: {
            command_name = "lui";
```

```
            arguments = reg_mapping(instr.rd) + ", 0x" +
to_hex(instr.immediate20() << 12, -1);
            break;
        }
        case AUIPC: {
            command_name = "auipc";
            arguments = reg_mapping(instr.rd) + ", " +
std::to_string(instr.immediate20() << 12);
            break;
        }
        case JAL: {
            uint32_t address_to_jump = cur_address +
get_jal_offset(instr.value);
            command_name = "jal";
            arguments = reg_mapping(instr.rd) + ", " +
to_hex(address_to_jump, -1) + " <" +
functions_mapping[address_to_jump] + ">";
            break;
        }
        case JALR: {
            command_name = "jalr";
            arguments = reg_mapping(instr.rd) + ", " +
to_hex(instr.immediate12(), -1) + "(" +
reg_mapping(instr.rs1) + ")";
            break;
        }
        case E: {
            if (instr.immediate12() == ECALL) {
                command_name = "ecall";
            } else {
```

```
                command_name = "ebreak";
            }
            break;
        }
        default:
            return std::make_pair("unknown_instruction", "");
        }
        return std::make_pair(command_name, arguments);
    }


    void add_mark(uint32_t x, uint32_t cur_address, uint32_t&
mark_index, std::map<uint32_t, std::string>&
functions_mapping) {
        std::string command_name = "";
        std::string arguments = "";
        instruction instr = instruction(x);
        int16_t local_immediate12 = instr.immediate12();
        local_immediate12 -= (1 << 12) * (local_immediate12
>> 11);
        if (instr.opcode == BRANCH) {
            int32_t offset = get_branch_offset(instr.value);
            uint32_t address_to_jump = cur_address + offset;
            if (functions_mapping.count(address_to_jump) ==
0) {
                functions_mapping[address_to_jump] = "L" +
std::to_string(mark_index);
                mark_index++;
            }
        } else if (instr.opcode == JAL) {
```

```
            uint32_t address_to_jump = cur_address +
get_jal_offset(instr.value);
            if (functions_mapping.count(address_to_jump) ==
0) {
                functions_mapping[address_to_jump] = "L" +
std::to_string(mark_index);
                mark_index++;
            }
        }
    }


    void disassembly(uint32_t offset, std::map<uint32_t,
std::string> functions_mapping, uint32_t code_size) {
        printf(".text\n");
        uint32_t sh_offset = get_sh_offset(offset);
        uint32_t sh_size = get_sh_size(offset);
        uint32_t cur = main_address;
        uint32_t mark_index = 0;
        for (uint32_t j = sh_offset; j < sh_offset +
code_size; j += sizeof(uint32_t)) {
            add_mark(read32(j), cur, mark_index,
functions_mapping);
            cur += sizeof(uint32_t);
        }
        cur = main_address;
        for (uint32_t j = sh_offset; j < sh_offset +
code_size; j += sizeof(uint32_t)) {
            if (functions_mapping.count(cur) > 0) {
                printf("%08x   <%s>:\n", cur,
functions_mapping[cur].c_str());
            }
```

```
            std::pair<std::string, std::string> instruction =
parse_instruction(read32(j), cur, functions_mapping);
            printf("   %05x:\t%08x\t%7s\t%s\n", cur,
read32(j), instruction.first.c_str(),
instruction.second.c_str());
            cur += sizeof(int32_t);
        }
        printf("\n");
    }

    uint32_t get_file_size(std::string filename) {
        FILE* fd = open_file(filename.c_str(), "rb");
        fseek(fd, 0, SEEK_END);
     uint32_t file_size = ftello(fd);
     fclose(fd);
        return file_size;
    }

    int main(int argc, char** argv) {
         if (argc < 3) {
             error_exit("Expected at least 3 arguments");
         }

        file_length = get_file_size(argv[1]);
        buf = (uint8_t*)malloc(file_length);

        FILE* input_file = open_file(argv[1], "rb");
        FILE* output_file = open_file(argv[2], "w");
        freopen(argv[2], "w", stdout);
```

```
        fread(buf, sizeof(uint8_t), file_length, input_file);

        check_is_elf();
        check_bit_depth();
        check_endianness();
        check_isa();

        uint32_t e_shoff = get_section_offset();
        uint16_t e_shnum = get_eshnum();

        std::map<uint32_t, std::string> function_mapping;

        for (uint32_t i = e_shoff, j = 0;i < file_length -
ELF32_SHDR_SIZE; i += ELF32_SHDR_SIZE) {
            if (is_strtab(i)) {
                strtab_offset[j] = get_sh_offset(i);
                j++;
            }
        }

        for (uint32_t i = e_shoff; (i - e_shoff) /
ELF32_SHDR_SIZE < e_shnum; i += ELF32_SHDR_SIZE) {
            uint32_t sh_name = get_sh_name(i);
            if (is_symtab(i)) {
                uint32_t sh_size = get_sh_size(i);
                uint32_t sh_offset = get_sh_offset(i);

                int32_t cnt = 0;
```

```
                for (uint32_t j = sh_offset; cnt < sh_size /
SYMTAB_SIZE; j += SYMTAB_SIZE, cnt++) {
                    uint32_t st_name = read32(j);
                    if (st_name > 0) {
                        std::string name =
get_name(strtab_offset[sh_name] + st_name);
                        if (id_to_type[buf[j + 12] & 0xf] ==
"FUNC") {
                            function_mapping[read32(j + 4)] =
name;
                            if (name == "main") {
                                main_address = read32(j + 4);
                            }
                        }
                    }
                }
            }
        }

        for (uint32_t i = e_shoff;i < file_length -
ELF32_SHDR_SIZE; i += ELF32_SHDR_SIZE) {
            if (is_progbits(i)) {
                disassembly(i, function_mapping,
get_sh_size(i));
                break;
            }
        }
```

```
        for (uint32_t i = e_shoff; (i - e_shoff) /
ELF32_SHDR_SIZE < e_shnum; i += ELF32_SHDR_SIZE) {
            uint32_t sh_name = get_sh_name(i);
            if (is_symtab(i)) {
                uint32_t sh_size = get_sh_size(i);
                uint32_t sh_offset = get_sh_offset(i);
                printf(".symtab\nSymbol Value
Size Type     Bind     Vis          Index Name\n");


                int32_t cnt = 0;


                for (uint32_t j = sh_offset; cnt < sh_size /
SYMTAB_SIZE; j += SYMTAB_SIZE, cnt++) {
                    printf("[%4i] 0x%-15X %5i %-8s %-8s %-8s
", cnt, read32(j + 4),
                     read32(j + 8), id_to_type[buf[j + 12] &
0xf].c_str(), id_to_bind[buf[j + 12] >> 4].c_str(),
                     id_to_visibility[buf[j + 13]].c_str());
                    uint32_t ndx_bytes = read16(j + 14);
                    std::string ndx =
std::to_string(ndx_bytes);
                    if (id_to_ndx.find(ndx_bytes) !=
id_to_ndx.end()) {
                        ndx = id_to_ndx[ndx_bytes];
                    }
                    printf("%6s ", ndx.c_str());
                    uint32_t st_name = read32(j);
                    if (st_name > 0) {
                        std::string name =
get_name(strtab_offset[sh_name] + st_name);
                        printf("%s", name.c_str());
```

```
                }
                printf("\n");
            }
        }
    }

    free(buf);
    fclose(input_file);

    return 0;
}
```