Лабораторная работа №3	M3138	2022
ISA	Селезнев Дмитрий Александрович	

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

Инструментарий и требования к работе: Java(javac 17.0.4.1)

Системы кодирования команд RISC-V

RISC-V – открытая Reg-Reg ISA на основе концепции RISC. Reg-Reg архитектура означает, что только команды STORE и LOAD умеют общаться с памятью, остальные только с регистрами. Как и любое другое ISA она описывает способ представления команд процессора в памяти. Архитектура использует только модель little-endian.

В RISC-V существует несколько наборов команд для 32 и 64 битном формате, причем все команды, которые есть в 32 битном, есть и в 64 битном. Из всех наборов выделяют базовые: RV32I и их стандартные расширения: Zifencei, Zicsr, RV32M, RV32A, RV32F, RV32D, RV32Q. Также все наборы есть в 64 битном формате. Каждая команда кодируется 32 битами. Немного подробнее о каждом наборе в таблице 1.

RV32I	Базовый набор с целочисленными операциями, 32-битный
RV32M	Целочисленное умножение и деление
RV32A	Атомарные операции
RV32F	Арифметические операции с плавающей запятой над числами одинарной точности
RV32D	Арифметические операции с плавающей запятой над числами двойной точности
RV32Q	Арифметические операции с плавающей запятой над числами четверной точности
Zifencei	Инструкции синхронизации потоков команд и данных
Zicsr	Инструкции для работы с контрольными и статусными регистрами

Таблица 1 – наборы команд[5]

Теперь немного подробнее о наборах RV32I и RV32M.

Для начала рассмотрим, как кодируются константы, то есть immediate.

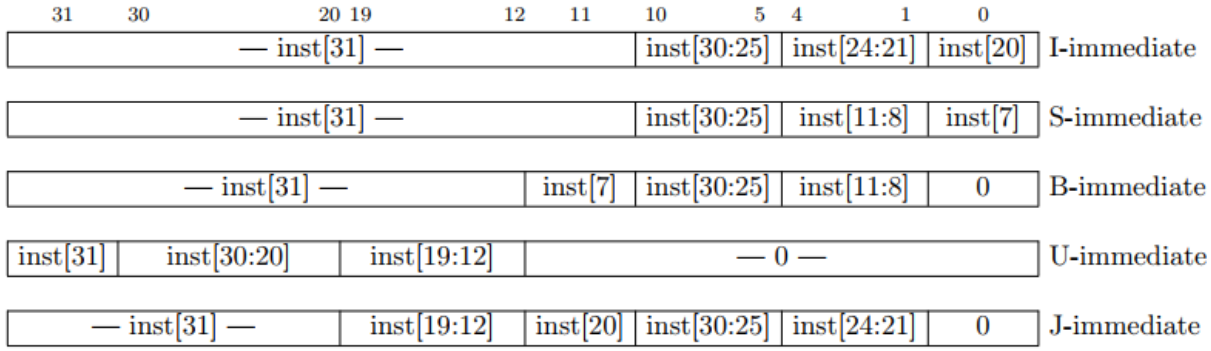

Рисунок 1 – Формат констант[6]

Константы бывают 5 типов, они отличаются количеством бит, которое они занимают в команде и в каком порядке нужно расположить биты, чтобы корректно её декодировать. На рисунке выше представлена таблица, как нужно декодировать каждую константу. Далее поговорим о типах команд.

Всего существует 6 типов команд, они приведены в таблице ниже (рисунок 2).

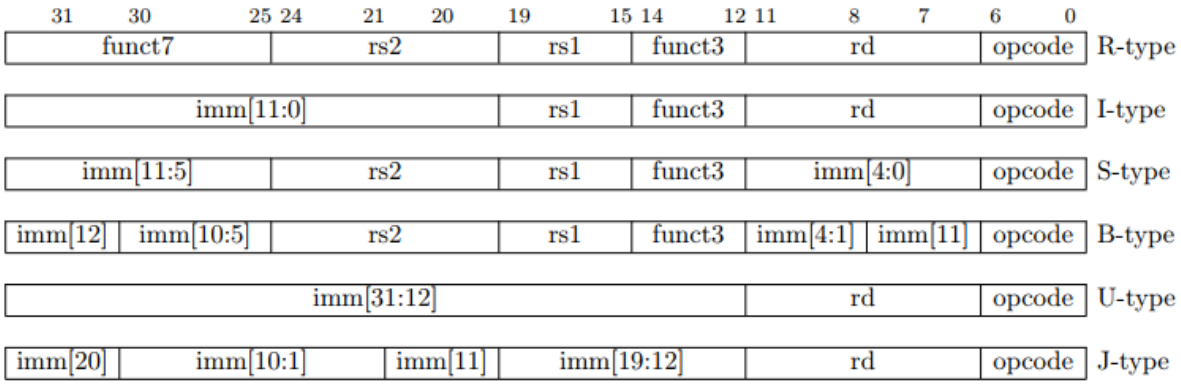

Рисунок 2 – Типы команд[6]

Opcode содержит в себе тип команды или группы команд, именно на него в первую очередь я смотрел при парсинге. Буква типа соответствует типу константы. Команды с одинаковым opcode, как правило, нужно различать по func3, если не написано обратное.

R-type – Register-Register тип команд. Все операции считывают регистры rs1 и rs2 в качестве исходных и записывают результат в регистр rd. В полях funct7 и funct3 выбирается тип операции.

I-type – Register-Immediate тип команд. Практически то же самое, что и предыдущий, только вместо получения из регистра, второе значение передается как константа.

S-type – Сохранение. Как ясно из названия этот тип команд сохраняет значение (из регистра rs2) в память. Адрес нужной ячейки вычисляется как сумма immediate и регистра rs1.

B-type - Conditional Branches. По сути, самые обычные условные переходы, которые происходят в зависимости от значений регистров rs1, rs2. Адрес перехода вычисляется, как сумма текущего адреса (то есть адрес команды) и immediate.

U-type – Upper immediate. Из данных нам наборов команд только две будут иметь такой тип, так что сразу их рассмотрим.

LUI (load upper immediate) используется для построения 32-разрядных констант. Помещает значение immediate в верхние 20 битов регистра назначения rd, заполняя нижние 12 бит с нулями.

AUIPC (add upper immediate to pc) используется для построения относительных адресов. Помещает сумму текущего адреса к immediate сдвинутом на 12 влево в регистр rd.

J-type – Unconditional Jumps. Безусловные переходы. Из данных нам команд есть только одна с таким типом.

JAL (jump and link). Адрес перехода вычисляется так же, как и в B-type, только теперь адрес команды следующей за переходом записывается в регистр rd.

Сразу опишу другую команду похожую на эту. JALR (jump and link register). Она кодируется в I-type, поэтому адрес перехода считается несколько иначе: добавляем расширенный по знаку immediate в регистр rs1, а затем устанавливаем младший значащий бит результата равным нулю, затем записываем адрес команды следующей за переходом в регистр rd.

Далее рассмотрим поподробнее команды из набора RV32M.

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Рисунок 3 – RV32M[6]

Как написано выше это команды умножения и деления целых чисел. Их можно отличить от других команд с таким же OPCODE по старшим 7 битам. А между собой их нужно различать по func3.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Рисунок 4 – RV32I[6]

Для начала о Conditional branches. Они также отличаются от остальных по OPCODE, а между собой по func3.

Команды с LD по LHU, это команды LOAD, то есть выписывания из памяти. Они кодируются в I-type, ячейка, из которой выписывается значение,

вычисляется как сумма константы и значения в rs1, и записывают в регистр rd.

Следующе три команды это команды STORE, то есть записи в память. Кодируются в S-type. Выписывает значение из регистра rs2, в ячейку памяти, которая считается так же, как и в LOAD.

Команды с ADDI по ANDI кодируются в I-type, опять же от остальных отличаются OPCODE, а между собой по func3. Выполняют соответствующие операции над rs1 и immediate, записывая результат в регистр rd.

Команды SLLI, SRLI, SRAI кодируются в I-type, но, по сути, выглядит это как R-type, только вместо rs2, выводим shamt как число.

Команды с ADD по AND кодируются в R-type и соответствуют их правилам.

Далее команды ECALL и EBREAK, которые отличаются друг от друга по старшим 12 битам и не имеют регистров как таковых.

Также есть команда FENCE, но нам ее не нужно парсить, так что вместо нее будет выводиться unknown_instruction. Как и для любой неизвестной команды.

Описание структуры файла ELF.

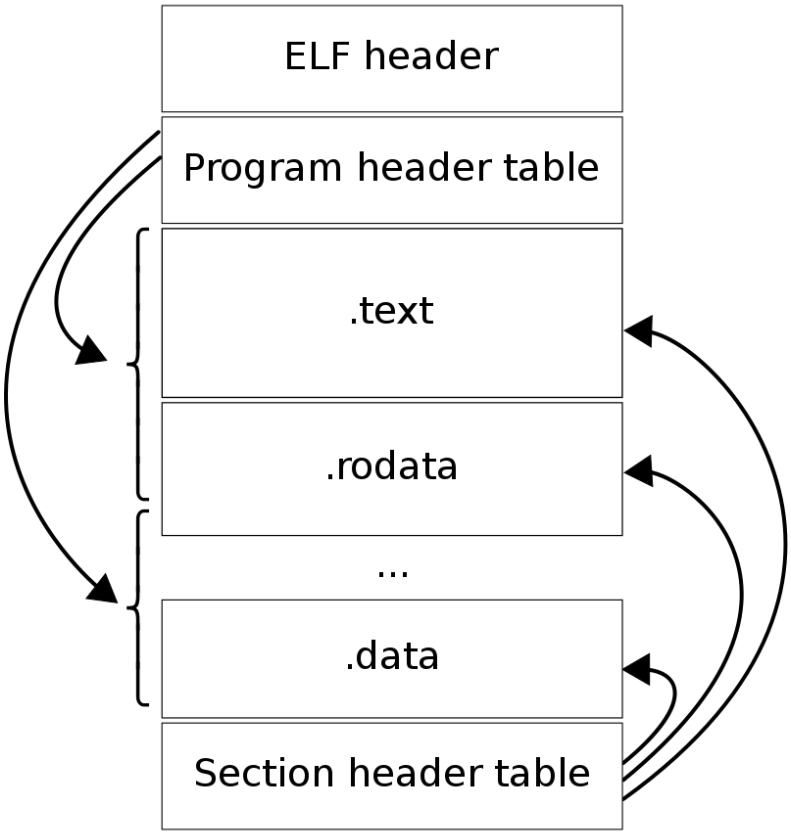

Рисунок 5 – Структура ELF файла.[4]

Как видно на рисунке 5, сначала идет ELF header, который занимает 52 байта, если формат файла 32 битный, и 64 байта, если 64 битный. Затем идет program header table, который нам с точки зрения парсинга неинтересен, занимающий 40(64, если на 64 битном формате) байт в файле. Затем идут различные разделы, а в конце лежит Section header table. Далее везде будет рассматриваться только 32 битный формат, так как только его нам нужно попарсить.

Теперь о том, как это парсить. Для начала нужно убедиться, что нам дали именно ELF файл, для этого нужно посмотреть на первые 4 байта. Они должны быть равны: $7F_{16}$, 45_{16}, $4C_{16}$, 46_{16}. Затем убедиться, что формат 32 битный, для этого в следующем байте должна лежать единица. А также убедиться, что файл имеет little endian кодировку, посмотрев, что следующий байт равен единице. Ну и наконец проверить, что вид ISA, это RISC-V, об этом можно узнать из 18 и 19 байт. Ниже представлена полная структура ELF header.

Байты	Название	Функция
0-3	EI_MAG0	Пометка, что файл ELF
4	EI_CLASS	Формат файла(32 или 64 бита, 1 или 2 соответственно)
5	EI_DATA	Кодировка файла(little or big endian, 1 или 2 соответственно)
6	EI_VERSION	Устанавливает значение 1 для оригинальной и текущей версии ELF
7	EI_OSABI	Определяет ABI целевой операционной системы
8	EI_ABIVERSION	Версия ABI
9-15	EI_PAD	Зарезервированные байты заполнения. В настоящее время не используется.
16-17	e_type	Определяет тип объектного файла.
18-19	e_machine	Определяет ISA
20-23	e_version	Устанавливает значение 1 для оригинальной версии ELF
24-27	e_entry	Это адрес памяти, с которого начинается выполнение процесса

28-31	e_phoff	Адрес начала таблицы заголовка программы
32-35	e_shoff	Адрес начала таблицы заголовков программы
36-39	e_flags	Зависит от целевой архитектуры
40-41	e_ehsize	Размер этого заголовка
42-43	e_phentsize	Размер таблицы заголовков программы
44-45	e_phnum	Количество записей в program header
46-47	e_shentsize	Размер таблицы заголовков программы
48-49	e_shnum	Количество заголовков в таблице заголовков
50-51	e_shstrndx	Индекс заголовка shstrtab(то есть таблицы имен заголовков) в таблице заголовков

Таблица 2 – Структура ELF header[4]

Для того чтобы попарсить интересующие нас разделы, нам необходимы лишь несколько из приведенных полей, а именно: e_shoff(32-35), e_shnum(48-49) и e_shstrndx(50-51). С помощью первого мы найдем таблицу заголовков, вторая для удобства, третья, чтобы можно было узнавать имена других разделов в таблице.

Рассмотрим заголовок одного раздела.

Байты	Название	Функция
0-3	sh_name	Смещение относительно начала shstrtab, начиная с которого лежит имя раздела
4-7	sh_type	Тип заголовка
8-11	sh_flags	Атрибуты раздела
12-15	sh_addr	Виртуальный адрес в памяти
16-19	sh_offset	Адрес начала раздела в памяти
20-23	sh_size	Размер раздела
24-27	sh_link	Индекс начала связанного раздела
28-31	sh_info	Некоторая дополнительная информация о разделе
32-35	sh_addralign	Требуемое выравнивание раздела
36-39	sh_entsize	Если записи внутри раздела фиксированной длины, то здесь лежит эта длина

Таблица 3 – Заголовок раздела[4]

Для того чтобы узнать имя раздела нужно в shstrtab “собрать” его побайтово пользуясь sh_name(нужно идти пока не встретишь null, так как имена в разделе лежат разделенные null-ами). Затем можно узнать его адрес и размер из полей sh_offset и sh_size.

По заданию нам нужно попарсить .symtab и .text, для того чтобы попарсить .symtab, нам также будет необходимо найти .strtab, чтобы можно было находить символьные названия имен символов.

Все строки .symtab лежат друг за другом и занимают по 16 байт. Каждая строка соответствует структуре ниже.

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

Рисунок 6 – Структура строки .symtab[2]

Первые 4 байта это смещение относительно начала .strtab, с которого начинается имя. Следующие 4 байта это значение символа, затем еще 4 байта хранящие размер символа. Последние 2 байта это индекс специального раздела (shndx), он выбирается в соответствии с таблицей, а если его там нет, то просто записывается значение.

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_BEFORE	0xff00
SHN_AFTER	0xff01
SHN_AMD64_LCOMMON	0xff02
SHN_HIPROC	0xff1f
SHN_LOOS	0xff20
SHN_LOSUNW	0xff3f
SHN_SUNW_IGNORE	0xff3f
SHN_HISUNW	0xff3f
SHN_HIOS	0xff3f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_XINDEX	0xffff
SHN_HIRESERVE	0xffff

Рисунок 7 – Индексы специальных секций[3]

Остальные 3 параметра каждой строки вычисляются по формулам, приведенным ниже, а затем находятся соответствия в таблицах.

- bind = info >> 4 (то есть старшие 4 бита info)
- type = info & (0xf) (младшие 4 бита info)
- vis = other & (0x3)

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

Рисунок 8 – Таблица bind[2]

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

Рисунок 9 – Таблица type[2]

Name	Value
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3
STV_EXPORTED	4
STV_SINGLETON	5
STV_ELIMINATE	6

Рисунок 10 – Таблица visibility[2]

Теперь рассмотрим то, как нужно парсить .text. Внутри этого раздела команды лежат последовательно, каждая занимает по 4 байта. Все команды парсятся в соответствии с тем, как я уже писал в описании RISC-V. Однако некоторые команды нужно рассмотреть отдельно.

Например, для всех команд переходов, то есть JAL и BRANCH, нужно еще указывать на какую метку они перешли, а если ее нет, то создать метку в формате L%i. Также стоит отметить, что виртуальный адрес команд, который нам нужно вывести, не совпадает с реальным, так что нужно его достать из соответствующего заголовка (sh_addr).

Описание работы парсера

Компилить код нужно командой: javac *.java && java Main test_elf output.txt. Причем вместо test_elf и output.txt нужно указывать файлы ввода и вывода соответственно.

На вход нашей программе подается бинарник, который я считал побайтово, а результат возвращался в другой файл.

```
    public static void main(String[] args) throws Exception {
        try (
                InputStream inputStream = new FileInputStream(args[0])
        ) {
            int byteRead = inputStream.read();
            while (byteRead != -1) {
                arr.add(byteRead);
                byteRead = inputStream.read();
            }
        } catch (IOException ex) {
            throw error("IO exception: " + ex.getMessage());
        }
        try (OutputStream outputStream = new FileOutputStream(args[1])) {
            parse(outputStream);
        } catch (IOException ex) {
            throw error("IO exception: " + ex.getMessage());
        } catch (Exception e) {
            throw error(e.getMessage());
        }
    }
```

Листинг 1 – Считывание из файла, вызов основного парсера

Как видно весь парсинг происходил в функции parse, здесь же происходит считывание файла в List arr, отлавливаются некоторые ошибки, которые могут появиться при открытии файла, а также вообще все ошибки, которые нам может вернуть parse.

```
    private static void parse(OutputStream out) throws Exception {
```

```
        try {
            if (arr.size() < FILE_HEADER_SIZE + PROGRAM_HEADER_SIZE) {
                throw error("File is not full");
            }
            int shstrndx = getBytes(50, 2);
            int startOfStringTableHeader = shstrndx * SECTION_HEADER_SIZE +
getBytes(32, 2);
            if (startOfStringTableHeader > arr.size()) {
                throw error("File is not full");
            }
            if (!checkELF()) {
                throw error("This file is not ELF");
            }
            if (arr.get(4) != 1) {
                throw error("Our system are 32-bit");
            }
            if (arr.get(5) != 1) {
                throw error("It's not a little endian");
            }
            if (getBytes(18, 2) != 0xF3) {
                throw error("It's not a RISC-V");
            }
            int startOfStringTable = getBytes(startOfStringTableHeader + 16,
4);
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                String name = getName(startOfStringTable + getBytes(i, 4));
                if (name.equals(".strtab")) {
                    strtabidx = getBytes(i + 16, 4);
                    break;
                }
            }
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                int posInStringTable = getBytes(i, 4);
                String name = getName(startOfStringTable + posInStringTable);
                if (name.equals(".symtab")) {
                    parseSymTab(getBytes(i + 16, 4), getBytes(i + 20, 4));
                    break;
                }
            }
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                int posInStringTable = getBytes(i, 4);
                String name = getName(startOfStringTable + posInStringTable);
                if (name.equals(".text")) {
                    virtualAddressOfText = getBytes(i + 12, 4);
                    parseText(getBytes(i + 16, 4), getBytes(i + 20, 4), out);
                }
            }
            out.write(LINE_FEED.getBytes());
            printSymTab(out);
        } catch (Exception e) {
            throw error("Something wrong: " + e.getMessage() + " I give up");
        }
    }
```

Листинг 2 – Функция parse

Здесь происходит отлов ошибок и проверка на то, что нам передали тот файл, который мы ожидали увидеть. Затем ищется .strtab, чтобы можно было искать имена символов из .symtab. Затем ищется .symtab и происходит его парсинг, затем происходит парсинг и вывод .text, и в конце вывод .symtab. Причем весь вывод происходит как в консоль, так и в файл, для удобства дебага. Теперь рассмотрим парсинг и вывод .symtab.

```
    private static void parseSymTab(int start, int size) {
        int pos = 0;
        for (int i = start; i < start + size; i += 16) {
            int nameI = getBytes(i, 4);
            String name = nameI == 0 ? "" : getName(strtabidx + nameI);
            int value = getBytes(i + 4, 4);
            int size_ = getBytes(i + 8, 4);
            int info = getBytes(i + 12, 1);
            String bind = getBind(info >> 4);
            String type = getType(info & (0xf));
            String vis = getVis(getBytes(i + 13, 1) & (0x3));
            String ndx = getNdx(getBytes(i + 14, 2));
            symbolTable.add(new SymbolTableSegment(pos, value, size_, type,
bind, vis, ndx, name));
            pos++;
        }
    }

    private static void printSymTab(OutputStream out) throws IOException {
        out.write((".symtab:" + LINE_FEED).getBytes());
        System.out.println(".symtab:");
        out.write(HEADER_OF_SYMTAB.getBytes());
        out.write(LINE_FEED.getBytes());
        List<SymbolTableSegment> symbolTableSegments =
symbolTable.getSegments();
        for (SymbolTableSegment symbolTableSegment : symbolTableSegments) {
            out.write((symbolTableSegment.toString() +
LINE_FEED).getBytes());
        }
        System.out.println(HEADER_OF_SYMTAB);
        symbolTable.print();
    }
```

Листинг 3 – Парсинг .symtab

На вход передается начало раздела и его размер. Парсинг происходит построчно в соответствии с тем, что я уже писал выше. В выводе нет ничего интересного, просто выводятся все строки друг за другом. Все get функции возвращают именно то, что написано в их названии, в соответствии с таблицами или другими особенностями, которые я описал выше. Стоит разве что показать работу getBytes.

```
    private static int getBytes(int pos, int num) {
        int ans = 0;
        for (int i = pos + num - 1; i >= pos; i--) {
            ans <<= 8;
            ans += arr.get(i);
```

```
        }
        return ans;
    }
```

Листинг 4 – Функция getBytes

Как видно она принимает на вход позицию в исходном файле и количество байт (не более 4), которое нужно, начиная с этой позиции, вернуть.

Еще необходимо в парсинге .symtab, показать, что из себя представляют symbolTable и класс SymbolTableSegment. symbolTable это экземпляр класса SymbolTable, реализация которого приведена ниже.

```
public class SymbolTable {
    private final Set<Integer> functions = new TreeSet<>();
    private int counterMarks = 0;
    private final Map<Integer, String> names = new TreeMap<>();
    private final List<SymbolTableSegment> segments = new ArrayList<>();

    public SymbolTable() {
    }

    public void add(SymbolTableSegment segment) {
        segments.add(segment);
        String name = segment.getName();
        int value = segment.getValue();
        if (!name.isEmpty()) {
            names.put(value, name);
        }
        if ("FUNC".equals(segment.getType())) {
            functions.add(value);
        }
    }

    public String getMark(int addr) {
        if (!names.containsKey(addr)) {
            names.put(addr, "L" + (counterMarks++));
        }
        return names.get(addr);
    }

    public void print() {
        for (SymbolTableSegment segment : segments) {
            System.out.println(segment);
        }
    }

    public List<SymbolTableSegment> getSegments() {
        return segments;
    }

    public boolean checkMarks(int addr) {
        return names.containsKey(addr);
    }

    public boolean checkFunc(int addr) {
        return functions.contains(addr);
```

```
    }
}
```

Листинг 5 – Класс SymbolTable

Класс хранит в себе список из экземпляров класса SymbolTableSegment. Умеет добавлять новую строку к списку, возвращать этот список, а также возвращать метку символа, если она имеется, и создавать новую в формате L%i, если нет.

```
public class SymbolTableSegment {
    private static final String SYMTAB_FORMAT = "[%4d] 0x%-13X %5d %-7s %-6s
%-6s %5s %s";
    private final int symbol;
    private final int value;
    private final int size;
    private final String type;
    private final String bind;
    private final String vis;
    private final String index;
    private final String name;

    public SymbolTableSegment(int symbol, int value, int size,
                              String type, String bind, String vis,
                              String index, String name) {
        this.symbol = symbol;
        this.value = value;
        this.size = size;
        this.type = type;
        this.bind = bind;
        this.vis = vis;
        this.index = index;
        this.name = name;
    }

    public int getSymbol() {
        return symbol;
    }

    public int getValue() {
        return value;
    }

    public int getSize() {
        return size;
    }

    public String getType() {
        return type;
    }

    public String getBind() {
        return bind;
    }

    public String getVis() {
        return vis;
    }
```

```
    public String getIndex() {
        return index;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return String.format((SYMTAB_FORMAT), symbol, value, size, type,
bind, vis, index, name);
    }
}
```

Листинг 6 – Класс SymbolTableSegment

Хранит в себе одну строку из .symtab. Умеет красиво выводиться в соответствии с форматом.

Теперь рассмотрим парсинг .text.

```
        private static void parseText(int start, int size, OutputStream out)
throws IOException {
        System.out.println(".text:");
        out.write((".text:" + LINE_FEED).getBytes());
        for (int i = start; i < start + size; i+=4) {
            int addr = virtualAddressOfText + i - start;
            marks.add(addr);
            linesInText.add(parseOpcode(getBytes(i, 4), addr));
        }
        for (int i = 0; i < marks.size(); i++) {
            int addr = marks.get(i);
            if (symbolTable.checkMarks(addr)) {
                out.write(LINE_FEED.getBytes());
                out.write((String.format("%08x <%s>:", addr,
symbolTable.getMark(addr)) + LINE_FEED).getBytes());
                System.out.println();
                System.out.printf("%08x <%s>:%n", addr,
symbolTable.getMark(addr));
            }
            String line = linesInText.get(i);
            out.write((line + LINE_FEED).getBytes());
            System.out.println(line);
        }
```

Листинг 7 – Парсинг .text

На вход подается начало раздела и его размер. Парсинг снова происходит построчно, то есть по 1 команде, которая собирается в функции parseOpcode. Также если оказывается, что текущий виртуальный адрес имеет метку, то выводится соответствующая метка.

```
    private static String parseOpcode(int command, int idx) {
        int rd_i = (command & MASK_RD) >> 7;
        String rd = REGISTER[rd_i];
        int func3 = (command & MASK_FUNC3) >> 12;
```

```
        int rs1_i = (command & MASK_RS1) >> 15;
        String rs1 = REGISTER[rs1_i];
        int rs2_i = (command & MASK_RS2) >> 20;
        String rs2 = REGISTER[rs2_i];
        int func7 = (command & MASK_FUNC7) >> 25;
        return switch (command & (MASK_OPCODE)) {
            // LUI
            case 0b0110111 -> String.format(TWO_ARGUMENTS, idx, command,
"lui",
                    rd, Integer.toHexString(getImmediate(command, 'U')));
            // AUIPC
            case 0b0010111 -> String.format(TWO_ARGUMENTS, idx, command,
"auipc",
                    rd, Integer.toHexString(getImmediate(command, 'U')));
            // JAL
            case 0b1101111 -> {
                int addr = idx + getImmediate(command, 'J');
                yield String.format(JAL_FORMAT, idx, command, "jal",
                        rd, Integer.toHexString(addr),
symbolTable.getMark(addr));
            }
            // JALR (у константы зануляется самый младший бит в силу
особенности JALR)
            case 0b1100111 -> String.format(LOAD_STORE_JARL, idx, command,
"jalr",
                    rd, Integer.toHexString((getImmediate(command, 'I') >> 1)
<< 1), rs1);
            // BEQ, BNE, BLT, BGE, BLTU, BGEU
            case 0b1100011 -> {
                int addr = idx + getImmediate(command, 'B');
                yield String.format(BRANCH_FORMAT, idx, command, getB(func3),
                        rs1, rs2, Integer.toHexString(addr),
symbolTable.getMark(addr));
            }
            // LB, LH, LW, LBU, LHU
            case 0b0000011 -> String.format(S_AND_L_FORMAT, idx, command,
getL(func3),
                    rd, getImmediate(command, 'I'), rs1);
            // SB, SH, SW
            case 0b0100011 -> String.format(S_AND_L_FORMAT, idx, command,
getS(func3), rs2, rd_i, rs1);
            // ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
            case 0b0010011 -> {
                String name = getIType(func3, func7);
                String answer;
                if ("srai".equals(name) || "srli".equals(name) ||
"slli".equals(name)) {
                    answer = String.format(THREE_ARGUMENTS, idx, command,
getStandard(func3, func7), rd, rs1, rs2_i);
                } else {
                    answer = String.format(THREE_ARGUMENTS, idx, command,
getIType(func3, func7),
                            rd, rs1, getImmediate(command, 'I'));
                }
                yield answer;
            }
            // ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
            // MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU
            case 0b0110011 -> String.format(THREE_ARGUMENTS, idx, command,
```

```
getStandard(func3, func7), rd, rs1, rs2);
            // FENCE
            case 0b0001111 -> String.format("    %05x:\t%08x\t%-7s", idx,
command, "unknown_instruction");
            // ECALL, EBREAK
            case 0b1110011 -> {
                String name = "ebreak";
                if ((command >> 7) == 0) {
                    name = "ecall";
                }
                yield String.format("    %05x:\t%08x\t%-7s", idx, command,
name);
            }
            default -> String.format("    %05x:\t%08x\t%-7s", idx, command,
"unknown_instruction");
        };
    }
```

Листинг 8 – Парсинг строки .text

Все команды парсятся в соответствии с тем, что я писал выше. Все функции get, возвращают имя команды при у совпадающих opcode. Формат вывода команд совпадает с шаблонами из задания, то есть сначала регистр, в который записывается результат, затем из которых получается (если константа, то она выписывается в 16-ой системе). Также у каждой команды выводится ее виртуальный адрес, который передается в функцию, и ее значение. Команды ecall и ebreak не имеют регистров, так что выводятся просто как названия. Имена регистров получаются в соответствии с таблицей на рисунке ниже.

Name	x0	x1	x2	x3	x4	x5 - x7	x8 - x9	x10 - x17	x18 - x27	x28 - x31
ABI Mnem	zero	ra	sp	gp	tp	t0 - t2	s0 - s1	a0 - a7	s2 - s11	t3 - t6
Meaning	Zero	Return address	Stack pointer	Global pointer	Thread pointer	Temporary registers	Callee-saved registers	Argument registers	Callee-saved registers	Temporary registers

Рисунок 11 – Таблица имен регистров[1]

```
    private static int getBits(int num, int start, int end) {
        return (num >>> start) % (1 << (end - start + 1));
    }

    private static int createInts31(int size, int val) {
        int answer = 0;
        for (int i = 0; i < size; i++) {
            answer <<= 1;
            answer += val;
        }
        return answer;
    }

    private static int getImmediate(int a, char t) {
        return switch (Character.toUpperCase(t)) {
            case 'I' -> getBits(a, 20, 30) + (createInts31(21, getBits(a, 31,
31)) << 11);
```

```
            case 'S' -> getBits(a, 7, 11) + (getBits(a, 25, 30) << 5)
                    + (createInts31(21, getBits(a, 31, 31)) << 11);
            case 'B' -> (getBits(a, 8, 11) << 1) + (getBits(a, 25, 30) << 5)
                    + (getBits(a, 7, 7) << 11) + (createInts31(20, getBits(a,
31, 31)) << 12);
            case 'U' -> ((a >> 12));
            case 'J' -> (getBits(a, 21, 30) << 1) + (getBits(a, 20, 20) <<
11)
                    + (getBits(a, 12, 19) << 12) + (createInts31(12,
getBits(a, 31, 31)) << 20);
            default -> a;
        };
    }
```

Листинг 9 – Парсинг констант

Парсинг констант происходит в соответствии с рисунком 1. Функции getBits и createInts31 вспомогательные.

Также в коде присутствует огромное количество констант и get функции (которые я упоминал), но их листинг вставлять сюда бессмысленно, так как там нет ничего необычного или идейного.

Результат работы программы

.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

100a4:0ff0000fNOP

100a8:00000073 ecall

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

000100c8 <L2>:
   100c8:fec50e13          addi    t3,a0,-20
   100cc:000f0313          addi    t1,t5,0
   100d0:      000f8893    addi    a7,t6,0
   100d4:      00000813    addi    a6,zero,0

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

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

```
  10120:          00008067        jalr    zero,0(ra)
```

.symtab:

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

Список источников

1) https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc#register-convention
2) https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-79797.html#chapter6-tbl-21
3) https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-94076.html#chapter6-tbl-16
4) https://en.wikipedia.org/wiki/Executable_and_Linkable_Format#Section_header
5) https://en.wikipedia.org/wiki/RISC-V
6) https://riscv.org/technical/specifications/

Листинг кода

```
import java.util.*;

public class SymbolTable {
    private final Set<Integer> functions = new TreeSet<>();
    private int counterMarks = 0;
    private final Map<Integer, String> names = new TreeMap<>();
    private final List<SymbolTableSegment> segments = new ArrayList<>();

    public SymbolTable() {
    }

    public void add(SymbolTableSegment segment) {
        segments.add(segment);
        String name = segment.getName();
        int value = segment.getValue();
        if (!name.isEmpty()) {
            names.put(value, name);
        }
        if ("FUNC".equals(segment.getType())) {
            functions.add(value);
        }
    }

    public String getMark(int addr) {
        if (!names.containsKey(addr)) {
            names.put(addr, "L" + (counterMarks++));
        }
        return names.get(addr);
    }

    public void print() {
        for (SymbolTableSegment segment : segments) {
            System.out.println(segment);
        }
    }

    public List<SymbolTableSegment> getSegments() {
        return segments;
    }

    public boolean checkMarks(int addr) {
        return names.containsKey(addr);
    }

    public boolean checkFunc(int addr) {
        return functions.contains(addr);
    }
}
```

Листинг 10 – SymbolTable

```
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;

public class SymbolTableSegment {
    private static final String SYMTAB_FORMAT = "[%4d] 0x%-13X %5d %-7s %-6s
```

```
%-6s %5s %s";
    private final int symbol;
    private final int value;
    private final int size;
    private final String type;
    private final String bind;
    private final String vis;
    private final String index;
    private final String name;

    public SymbolTableSegment(int symbol, int value, int size,
                              String type, String bind, String vis,
                              String index, String name) {
        this.symbol = symbol;
        this.value = value;
        this.size = size;
        this.type = type;
        this.bind = bind;
        this.vis = vis;
        this.index = index;
        this.name = name;
    }

    public int getSymbol() {
        return symbol;
    }

    public int getValue() {
        return value;
    }

    public int getSize() {
        return size;
    }

    public String getType() {
        return type;
    }

    public String getBind() {
        return bind;
    }

    public String getVis() {
        return vis;
    }

    public String getIndex() {
        return index;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return String.format((SYMTAB_FORMAT), symbol, value, size, type,
bind, vis, index, name);
```

```
    }
}
```

Листинг 11 – SymbolTableSegment

```
import java.io.*;
import java.util.ArrayList;
import java.util.List;

public class Main {
    private static final int FILE_HEADER_SIZE = 0x34;
    private static final String LINE_FEED = System.lineSeparator();
    private static final int PROGRAM_HEADER_SIZE = 0x20;
    private static List<Integer> marks = new ArrayList<>();
    private static List<String> linesInText = new ArrayList<>();
    private static final int SECTION_HEADER_SIZE = 0x28;
    private static final SymbolTable symbolTable = new SymbolTable();
    private static int strtabidx;
    private static int virtualAddressOfText;
    private static final String HEADER_OF_SYMTAB = "Symbol Value
        Size Type    Bind   Vis        Index Name";
    private static final String THREE_ARGUMENTS = "   %05x:\t%08x\t%-
7s\t%s,%s,%s";
    private static final String BRANCH_FORMAT = "   %05x:\t%08x\t%-
7s\t%s,%s,0x%s <%s>";
    private static final String TWO_ARGUMENTS = "   %05x:\t%08x\t%-
7s\t%s,0x%s";
    private static final String JAL_FORMAT = "   %05x:\t%08x\t%-7s\t%s,0x%s
<%s>";
    private static final String LOAD_STORE_JARL = "   %05x:\t%08x\t%-
7s\t%s,%s(%s)";
    private static final String S_AND_L_FORMAT = "   %05x:\t%08x\t%-
7s\t%s,%s(%s)";
    private static final String[] REGISTER = {
            "zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2",
            "s0", "s1", "a0", "a1", "a2", "a3", "a4", "a5",
            "a6", "a7", "s2", "s3", "s4", "s5", "s6", "s7",
            "s8", "s9", "s10", "s11", "t3", "t4", "t5", "t6"
    };
    private static final List<Integer> arr = new ArrayList<>();
    private static final int MASK_OPCODE = 0x7f;
    private static final int MASK_RD = 0xf80;
    private static final int MASK_FUNC3 = 0x7000;
    private static final int MASK_RS1 = 0xf8000;
    private static final int MASK_RS2 = 0x1f00000;
    private static final int MASK_FUNC7 = 0xfe000000;
    private static final int MASK_imm20 = 0xfffff000;

    private static String getBind(int val) {
        return switch (val) {
            case 0 -> "LOCAL";
            case 1 -> "GLOBAL";
            case 2 -> "WEAK";
            case 10 -> "LOOS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> "NULL";
        };
```

```
    }

    private static String getType(int val) {
        return switch (val) {
            case 0 -> "NOTYPE";
            case 1 -> "OBJECT";
            case 2 -> "FUNC";
            case 3 -> "SECTION";
            case 4 -> "FILE";
            case 5 -> "COMMON";
            case 6 -> "TLS";
            case 10 -> "LOOS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> "NULL";
        };
    }

    private static String getVis(int val) {
        return switch (val) {
            case 0 -> "DEFAULT";
            case 1 -> "INTERNAL";
            case 2 -> "HIDDEN";
            case 3 -> "PROTECTED";
            case 4 -> "EXPORTED";
            case 5 -> "SINGLETON";
            case 6 -> "ELIMINATE";
            default -> "NULL";
        };
    }

    private static String getNdx(int val) {
        return switch (val) {
            case 0 -> "UND";
            case 0xff00 -> "LORESERVE";
            case 0xff01 -> "AFTER";
            case 0xff02 -> "AMD64_LCOMMON";
            case 0xff1f -> "HIPROC";
            case 0xff20 -> "LOOS";
            case 0xff3f -> "LOSUNW";
            case 0xfff1 -> "ABS";
            case 0xfff2 -> "COMMON";
            case 0xffff -> "XINDEX";
            default -> Integer.toString(val);
        };
    }

    private static Exception error(String message) {
        return new Exception(message);
    }

    private static boolean checkELF() {
        return arr.size() > 3 && arr.get(0) == 127 && arr.get(1) == 69 &&
arr.get(2) == 76 && arr.get(3) == 70;
    }

    private static int getBytes(int pos, int num) {
        int ans = 0;
```

```
        for (int i = pos + num - 1; i >= pos; i--) {
            ans <<= 8;
            ans += arr.get(i);
        }
        return ans;
    }

    private static String getName(int start) {
        StringBuilder sb = new StringBuilder();
        int pos = start;
        while (arr.get(pos) > 0) {
            sb.append((char)(int)arr.get(pos));
            pos++;
        }
        return sb.toString();
    }

    private static String getB(int a) {
        return switch (a) {
            case 0b000 -> "beq";
            case 0b001 -> "bne";
            case 0b100 -> "blt";
            case 0b101 -> "bge";
            case 0b110 -> "bltu";
            case 0b111 -> "bgeu";
            default -> "nop";
        };
    }

    private static String getL(int a) {
        return switch (a) {
            case 0b000 -> "lb";
            case 0b001 -> "lh";
            case 0b010 -> "lw";
            case 0b100 -> "lbu";
            case 0b101 -> "lhu";
            default -> "nop";
        };
    }

    private static String getS(int a) {
        return switch (a) {
            case 0b000 -> "sb";
            case 0b001 -> "sh";
            case 0b010 -> "sw";
            default -> "nop";
        };
    }

    private static String getIType(int a, int b) {
        return switch (a) {
            case 0b000 -> "addi";
            case 0b010 -> "slti";
            case 0b011 -> "sltiu";
            case 0b100 -> "xori";
            case 0b110 -> "ori";
            case 0b111 -> "andi";
            case 0b001 -> "slli";
            case 0b101 -> b == 0 ? "srli" : "srai";
```

```
            default -> "nop";
        };
    }

    private static String getStandard(int a, int b) {
        return switch (b) {
            case 0b0000000 -> switch (a) {
                case 0b000 -> "add";
                case 0b001 -> "sll";
                case 0b010 -> "slt";
                case 0b011 -> "sltu";
                case 0b100 -> "xor";
                case 0b101 -> "srl";
                case 0b110 -> "or";
                case 0b111 -> "and";
                default -> "nop";
            };
            case 0b0100000 -> switch (a) {
                case 0b000 -> "sub";
                case 0b101 -> "sra";
                default -> "nop";
            };
            case 0b0000001 -> switch (a) {
                case 0b000 -> "mul";
                case 0b001 -> "mulh";
                case 0b010 -> "mulhsu";
                case 0b011 -> "mulhu";
                case 0b100 -> "div";
                case 0b101 -> "divu";
                case 0b110 -> "rem";
                case 0b111 -> "remu";
                default -> "nop";
            };
            default -> "nop";
        };
    }

    private static int getBits(int num, int start, int end) {
        return (num >>> start) % (1 << (end - start + 1));
    }

    private static int createInts31(int size, int val) {
        int answer = 0;
        for (int i = 0; i < size; i++) {
            answer <<= 1;
            answer += val;
        }
        return answer;
    }

    private static int getImmediate(int a, char t) {
        return switch (Character.toUpperCase(t)) {
            case 'I' -> getBits(a, 20, 30) + (createInts31(21, getBits(a, 31,
31)) << 11);
            case 'S' -> getBits(a, 7, 11) + (getBits(a, 25, 30) << 5)
                    + (createInts31(21, getBits(a, 31, 31)) << 11);
            case 'B' -> (getBits(a, 8, 11) << 1) + (getBits(a, 25, 30) << 5)
                    + (getBits(a, 7, 7) << 11) + (createInts31(20, getBits(a,
31, 31)) << 12);
```

```
            case 'U' -> ((a >> 12));
            case 'J' -> (getBits(a, 21, 30) << 1) + (getBits(a, 20, 20) <<
11)
                    + (getBits(a, 12, 19) << 12) + (createInts31(12,
getBits(a, 31, 31)) << 20);
            default -> a;
        };
    }

    private static String parseOpcode(int command, int idx) {
        int rd_i = (command & MASK_RD) >> 7;
        String rd = REGISTER[rd_i];
        int func3 = (command & MASK_FUNC3) >> 12;
        int rs1_i = (command & MASK_RS1) >> 15;
        String rs1 = REGISTER[rs1_i];
        int rs2_i = (command & MASK_RS2) >> 20;
        String rs2 = REGISTER[rs2_i];
        int func7 = (command & MASK_FUNC7) >> 25;
        return switch (command & (MASK_OPCODE)) {
            // LUI
            case 0b0110111 -> String.format(TWO_ARGUMENTS, idx, command,
"lui",
                    rd, Integer.toHexString(getImmediate(command, 'U')));
            // AUIPC
            case 0b0010111 -> String.format(TWO_ARGUMENTS, idx, command,
"auipc",
                    rd, Integer.toHexString(getImmediate(command, 'U')));
            // JAL
            case 0b1101111 -> {
                int addr = idx + getImmediate(command, 'J');
                yield String.format(JAL_FORMAT, idx, command, "jal",
                        rd, Integer.toHexString(addr),
symbolTable.getMark(addr));
            }
            // JALR (у константы зануляется самый младший бит в силу
особенности JALR)
            case 0b1100111 -> String.format(LOAD_STORE_JARL, idx, command,
"jalr",
                    rd, Integer.toHexString((getImmediate(command, 'I') >> 1)
<< 1), rs1);
            // BEQ, BNE, BLT, BGE, BLTU, BGEU
            case 0b1100011 -> {
                int addr = idx + getImmediate(command, 'B');
                yield String.format(BRANCH_FORMAT, idx, command, getB(func3),
                        rs1, rs2, Integer.toHexString(addr),
symbolTable.getMark(addr));
            }
            // LB, LH, LW, LBU, LHU
            case 0b0000011 -> String.format(S_AND_L_FORMAT, idx, command,
getL(func3),
                    rd, getImmediate(command, 'I'), rs1);
            // SB, SH, SW
            case 0b0100011 -> String.format(S_AND_L_FORMAT, idx, command,
getS(func3), rs2, rd_i, rs1);
            // ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI
            case 0b0010011 -> {
                String name = getIType(func3, func7);
                String answer;
                if ("srai".equals(name) || "srli".equals(name) ||
```

```
"slli".equals(name)) {
                    answer = String.format(THREE_ARGUMENTS, idx, command,
getStandard(func3, func7), rd, rs1, rs2_i);
                } else {
                    answer = String.format(THREE_ARGUMENTS, idx, command,
getIType(func3, func7),
                            rd, rs1, getImmediate(command, 'I'));
                }
                yield answer;
            }
            // ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND
            // MUL, MULH, MULHSU, MULHU, DIV, DIVU, REM, REMU
            case 0b0110011 -> String.format(THREE_ARGUMENTS, idx, command,
getStandard(func3, func7), rd, rs1, rs2);
            // FENCE
            case 0b0001111 -> String.format("    %05x:\t%08x\t%-7s", idx,
command, "unknown_instruction");
            // ECALL, EBREAK
            case 0b1110011 -> {
                String name = "ebreak";
                if ((command >> 7) == 0) {
                    name = "ecall";
                }
                yield String.format("    %05x:\t%08x\t%-7s", idx, command,
name);
            }
            default -> String.format("    %05x:\t%08x\t%-7s", idx, command,
"unknown_instruction");
        };
    }

    private static void parseSymTab(int start, int size) {
        int pos = 0;
        for (int i = start; i < start + size; i += 16) {
            int nameI = getBytes(i, 4);
            String name = nameI == 0 ? "" : getName(strtabidx + nameI);
            int value = getBytes(i + 4, 4);
            int size_ = getBytes(i + 8, 4);
            int info = getBytes(i + 12, 1);
            String bind = getBind(info >> 4);
            String type = getType(info & (0xf));
            String vis = getVis(getBytes(i + 13, 1) & (0x3));
            String ndx = getNdx(getBytes(i + 14, 2));
            symbolTable.add(new SymbolTableSegment(pos, value, size_, type,
bind, vis, ndx, name));
            pos++;
        }
    }

    private static void parseText(int start, int size, OutputStream out)
throws IOException {
        System.out.println(".text:");
        out.write((".text:" + LINE_FEED).getBytes());
        for (int i = start; i < start + size; i+=4) {
            int addr = virtualAddressOfText + i - start;
            marks.add(addr);
            linesInText.add(parseOpcode(getBytes(i, 4), addr));
        }
        for (int i = 0; i < marks.size(); i++) {
```

```
            int addr = marks.get(i);
            if (symbolTable.checkMarks(addr)) {
                out.write(LINE_FEED.getBytes());
                out.write((String.format("%08x <%s>:", addr,
symbolTable.getMark(addr)) + LINE_FEED).getBytes());
                System.out.println();
                System.out.printf("%08x <%s>:%n", addr,
symbolTable.getMark(addr));
            }
            String line = linesInText.get(i);
            out.write((line + LINE_FEED).getBytes());
            System.out.println(line);
        }
    }

    private static void printSymTab(OutputStream out) throws IOException {
        out.write((".symtab:" + LINE_FEED).getBytes());
        System.out.println(".symtab:");
        out.write(HEADER_OF_SYMTAB.getBytes());
        out.write(LINE_FEED.getBytes());
        List<SymbolTableSegment> symbolTableSegments =
symbolTable.getSegments();
        for (SymbolTableSegment symbolTableSegment : symbolTableSegments) {
            out.write((symbolTableSegment.toString() +
LINE_FEED).getBytes());
        }
        System.out.println(HEADER_OF_SYMTAB);
        symbolTable.print();
    }

    private static void parse(OutputStream out) throws Exception {
        try {
            if (arr.size() < FILE_HEADER_SIZE + PROGRAM_HEADER_SIZE) {
                throw error("File is not full");
            }
            int shstrndx = getBytes(50, 2);
            int startOfStringTableHeader = shstrndx * SECTION_HEADER_SIZE +
getBytes(32, 2);
            if (startOfStringTableHeader > arr.size()) {
                throw error("File is not full");
            }
            if (!checkELF()) {
                throw error("This file is not ELF");
            }
            if (arr.get(4) != 1) {
                throw error("Our system are 32-bit");
            }
            if (arr.get(5) != 1) {
                throw error("It's not a little endian");
            }
            if (getBytes(18, 2) != 0xF3) {
                throw error("It's not a RISC-V");
            }
            int startOfStringTable = getBytes(startOfStringTableHeader + 16,
4);
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                String name = getName(startOfStringTable + getBytes(i, 4));
                if (name.equals(".strtab")) {
```

```
                    strtabidx = getBytes(i + 16, 4);
                    break;
                }
            }
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                int posInStringTable = getBytes(i, 4);
                String name = getName(startOfStringTable + posInStringTable);
                if (name.equals(".symtab")) {
                    parseSymTab(getBytes(i + 16, 4), getBytes(i + 20, 4));
                    break;
                }
            }
            for (int i = getBytes(32, 2); i < arr.size(); i +=
SECTION_HEADER_SIZE) {
                int posInStringTable = getBytes(i, 4);
                String name = getName(startOfStringTable + posInStringTable);
                if (name.equals(".text")) {
                    virtualAddressOfText = getBytes(i + 12, 4);
                    parseText(getBytes(i + 16, 4), getBytes(i + 20, 4), out);
                }
            }
            out.write(LINE_FEED.getBytes());
            printSymTab(out);
        } catch (Exception e) {
            throw error("Something wrong: " + e.getMessage() + " I give up");
        }
    }

    public static void main(String[] args) throws Exception {
        try (
                InputStream inputStream = new FileInputStream(args[0])
        ) {
            int byteRead = inputStream.read();
            while (byteRead != -1) {
                arr.add(byteRead);
                byteRead = inputStream.read();
            }
        } catch (IOException ex) {
            throw error("IO exception: " + ex.getMessage());
        }
        try (OutputStream outputStream = new FileOutputStream(args[1])) {
            parse(outputStream);
        } catch (IOException ex) {
            throw error("IO exception: " + ex.getMessage());
        } catch (Exception e) {
            throw error(e.getMessage());
        }
    }
}
```

Листинг 12 – Main