САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

ISA. Ассемблер, дизассемблер

Выполнил(а): Левицкий Иван Михайлович

Номер ИСУ: 334916

студ. гр. M3135

Санкт-Петербург

2021

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

Инструментарий и требования к работе: работа может быть выполнена на любом из следующих языков: C/C++, Python, Java (выполнил на Java).

Теоретическая часть

1) ELF файл и его структура:

ELF-файл (Executable and Linkable Format) - формат двоичного файла, используемого во многих современных UNIX-подобных системах; обычно является выходным файлом компилятора или линкера и хранит в себе ряд подряд идущих байтов. Очень гибкий в плане настройки двоичный формат, позволяющий устанавливать позиции конкретных разделов в нужные места.

В самом начале ELF-файла расположен строго регламентированный заголовок (ELF-header, размером 52 байта для 32-битного формата и 64 для 64-битного). В нем на заранее известных позициях расположена различная информация о типе файла и характеристиках системы для которой он предназначен (является ли файл ELF-ом, битность (32/64), тип кодировки(Little/BigEndian), тип процессора(x86, AMD64, RISC-V, и т.д.), тип ОС, тип ELF-файла (исполняемый/перемещаемый/ и т.п.) и т.д.), а также, что самое главное, необходимые ссылки (e_phoff, e_shoff, и т.д.) на следующие важные части ELF-файла (таблицу заголовков программы (не будем разбираться подробнее, т.к. это не важно для наших задач, но она позволяет создать образ процесса) и таблицу заголовков секций), а также ссылки на некоторые подчасти этих частей и их размеры. (Под ссылкой я

здесь имею ввиду так называемый offset - позицию первого байта нужного объекта относительно начала программы или какого-либо другого байта).

Таблица заголовков секций - это в каком-то порядке подряд идущие заголовки (.text, .symtab, .strtab, .rela, и т.д.) с информацией о соответствующих секциях (размер каждого заголовка и их количество установлены в заголовке файла), а порядок самих “полей” заголовков заранее известен (name (ссылка на имя секции в секции имен секций), type (тип секции), flags, address, offset (ссылка на саму секцию к которой предназначен этот заголовок), size (размер секции), link, info и т.д.), а значит из них можно извлекать нужную информацию и переходить к работе с основной частью ELF-файла - секциями.

Каждая из секций (размер и координаты начала которой известны из ее заголовка) хранит в себе сами данные (например, .text хранит в себе исполняемый код в виде подряд идущих двоичных команд; .symtab хранит данные о метках по 16 байт (в порядке: 0 - 3 байты - offset имени метки в секции .strtab; 4 - 7 байты - value метки, 8 - 11 байты - size метки, 12 - bind и type метки, 13 - vis метки, 14 - 15 байты - индекс метки); .shstrtab хранит имена секций; .strtab хранит имена меток из .symtab).

2) RISC-V:

RISC-V - открытая и свободная система команд и процессорная архитектура, построенная на концепции RISC. Основная часть ISA содержит всего 53 команды (32-битные), но имеет множество расширений. Например: расширение M - операции умножения и деления, расширение A - атомарные операции, расширение F (float) - различные операции для

чисел с плавающей точкой, расширение C (RVC) - сжатые команды (16-битные), расширения D, Q и так далее. Основными регистрами RISC-V называются 32 регистра (x0 - x31), переименованные по соглашению в UNIX системах соответственно по порядку на:

(x0) zero - всегда 0; (x1) ra - адрес возврата, (x2) sp - указатель стэка;

(x3, x4) gp и tp - глобальный и поточный указатели (обычно регистры для компилятора);

(x5 - x7, x28 - x31) t0 - t2, t3 - t6 - временные регистры;

(x8 - x9, x18 - x27) s0 - s1, s2 - s11 - рабочие регистры;

(x10 - x17) a0 - a7 - регистры аргументов.

Но так же в RISC-V есть и другие регистры, например 310 машинных CSR-регистров, а также есть, например, перенумерация обычных регистров для их сокращенных версий.

Из всей спецификации RISC-V для нашей задачи нам нужно только соответствие между двоичным представлением команд и их видом в ассемблере (конкретно нам нужны 32-битные команды RV32I (с подразделом машинных команд Zicsr и без FENCE команд) RV32M и сокращенные 16-битные команды RVC (аналоги которых есть в RV32I)). А также полный список имен всех нужных для этих команд регистров (32 обычных регистра, машинные CSR-регистры и регистры сокращенных команд Integer Register ABI Name, кодирующиеся по 3 бита). Всю нужную информацию находим из 2-ух томной спецификаци Volume 1, Unprivileged

Spec v.20191213 и Volume 2, Privileged Spec v.20211203, взятой с официального сайта RICS-V (https://riscv.org/technical/specifications).

Разберемся с типами тех команд, которые мы хотим дизассемблировать, сразу упомянув, что сокращенные команды можно отличать от обычных по opcode, у 32-битных команд последние 2 бита opcode всегда “11”, а у сокращенных любая другая комбинация.

RV32I Base Instruction содержит 40 32-битных команд разделенных на несколько типов (R, I, B, S, J, U), тип которых определяется по opcode.

Таблица 1 - Конструкция различных инструкций RV32I

R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
B	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
J	imm[20|10:1|11|19:12]				rd	opcode
U	imm[31:12]				rd	opcode

R инструкции производят арифметико-логические операции только между двумя регистрами (например ADD, SUB) и содержат 2 указателя (funct7, funct3) на определения операции и 3 указателя на регистры (r1, r2 - регистры откуда читаются значения, rd - куда сохраняются после операции) (см. Таблицу 1)

I инструкции производят арифметико-логические операции между регистром и константой (imm[] - константа в дополнении до 2, а значения в квадратных скобках это соответственные номера битов в порядке

кодирования инструкции, если же какие-то биты не отображены, то они заполняются нулями), содержат imm, rs1 и rd и funct3 для определения операции. (Пример: ADDI)

B инструкции предназначены для условных переходов на (addres + imm). При дизассемблировании справа от них нужно указывать метку (и на месте прыжка также нужно указывать ту же метку).

S инструкции (см. Таблицу 1) предназначены для записи значений в память.

J инструкции (см. Таблицу 1) предназначены для обычных прыжков в другое место (также нужно указывать метки).

U инструкции (см. Таблицу 1) предназначены для записи верхних 20 бит в значение регистра (например используя LUI вместе с ADDI, очевидно, можно получить любое значение регистра).

RV32I Zicsr (control and status registers) - специальные команды (CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, CSRRCI), работающие на своих специально зарезервированных машинных регистрах. Первые 3 команды вида rd csr rs1 (пишут из в rs1 в csr и из scr в rd), а вторые 3 вида rd csr uimm (пишут только из csr в rd).

RV32M по структуре полностью схожи с инструкциями RV32I типа R.

RVC - сокращенные инструкции из различных расширений. Кодируются по 16 бит и имеют несколько особенностей: часто имеют условие на корректность константы или регистра (nzimm - not zero, rd != 2 и т.п.), часто хранят константу в обычном коде, а не в дополнении до двух (uimm) и часто хранят ссылку не сразу на обычные регистры, а на сокращенные

(rs1’, rs2’, rd’), кодирующиеся 3 битами (то есть те же самые обычные, но по другому нумерованные).

Практическая часть

Язык: java 17.

Приведем алгоритм работы получившегося дизассемблера и продемонстрируем пример его работы (на секциях .text и .symtab).

Алгоритм работы (все исходники кода должны лежать в пакете hw4):

1) Дизассемблеру передается ELF-файл и файл куда нужно написать дизассемблированные .text и .symtab через командную строку при запуске в виде: java hw4.Dis <входной ELF-файл> <выходной файл>.

2) Считываем весь файл, проверяем его на корректность и обрабатываем возможные ошибки если он некорректен (нашелся ли файл, читаем ли он, является ли тип файла ELF, битность, тип кодировки, тип машины).

3) Далее, если он с виду корректен начинаем обрабатывать заголовок ELF-файла (но все равно продолжаем ловить некоторые ошибки если файл некорректен), вытаскиваем из него нужные ссылки (e_shoff - ссылка на таблицу заголовков секций, e_shentsize - длина одного заголовка секции, e_shnum - кол-во секций, e_shstrndx - номер заголовка секции с именами).

4) Далее, с помощью секции имен понимаем, где у нас находятся заголовки секций .text(ProgBits), .symtab и .strtab, заходим в них, проверяя поле type на корректность (type_ProgBits = 0x1, type_Symtab = 0x02, type_Strtab =

0x03), и считываем offset и size. тем самым понимая, где расположены сами секции (в ProgBits считываем еще address, для .

5) Разбираемся с командами (.text(ProgBits)), начинаем считывать их по 4 или 2 байта в зависимости от того какие значения у последних двух битов первого байта (первого т.к. opcode в нем, т.к. у нас LittleEndian) и сразу переводим их в двоичные строки (так удобнее с ними работать, а время и память у нас не были ограничены условиями). Далее передаем полученный “лист” уже разделенных друг от друга команд в класс ProcCommand, который, собственно, и будет их дизассемблировать.

6) Дизассемблируем каждую команду и попутно записываем в две хэшмапы новые “метки прыжков” (в виде LOC_<адрес места куда совершается прыжок в нужном по условию формате>) (в одну записываем место “откуда” совершается прыжок и метку, а в другую “куда” совершается прыжок).

7) Также разбираемся с .symtab, бьем все на блоки по 16 байт и передаем информацию в класс ParseSymTab, там уже переводим двоичную информацию в обычную информацию о метке (value, size, type, bind, vis, index, name), если у метки нет имени - оставляем поле пустым, если index имени отсутствует в зарезервированной таблице именных индексов (UNDEF, BEFORE, AFTER, ABS, COMMON, XINDEX, OS, PROC) пишем в поле просто значение индекса десятичным числом, если значение не идентифицируется в полях bind, type или vis, то пишем в соответствующем поле Unknown. Если метка типа FUNC то сохраняем ее в соответствующую мапу под ее value, потом отметим ее при выводе команд перед командой.

8) Выводим сначала обработанные команды, если они совершают прыжок или на них совершают прыжок соответственно справа или слева ставим нужную метку (свою если на этом месте нет FUNC, а если есть, то эту FUNC), затем выводим обработанную секцию .symtab. Таким образом мы проходимся по каждой команде дважды, при обработке и при выводе (при выводе смотрим на мапы прыжков).

Крайние случаи и обработка возможных “разночтений”:

1) Все имена команд и характеристики меток выводятся большими буквами (т.к. так удобнее видеть тип команды, а Assembler-ы все равно обычно регистронезависимы).

2) Все сокращенные команды RVC задействующие регистр sp (C.ADDI4SPN, C.ADDI16SP, C.LWSP, C.SWSP) явно указывают его в своих аргументах.

3) Обработка крайних случаев в виде некорректных (HINT) сокращенных RVC команд не совершается.

4) Если обнаруживаются неидентифицируемые регистры (в csr), то вся инструкция выводится как “unknown_command” (как и любая некорректная инструкция).

5) В командах типа U RV32I (LUI и AUIPC) и их сокращенных версиях в RVC (C.LUI) константа записывается без младших 12 нулей (не умноженной на 2^12) в целях упрощения записи (то есть это рассчитано на то, что в команду заложено, то, что она работает с 20 старшими битами регистра и переданную ей константу нужно умножать на 2^12). То есть, например, вместо LUI a0, 65536 пишется LUI a0, 16

6) Если инструкция совершает прыжок то, место прыжка указывается справа (без комментария, а просто так) в виде метки (если на функцию то метка с именем из symtab, если нет, то своя вида LOC_%5x).

7) Кодировка читается как дефолтная в джаве (так как нет необходимости указывать, т.к. парсим только английские буквы и цифры).

Пример результата работы программы

Пример .text

```
00010098            C.LI a1, 0
0001009a            C.JAL 290 memset
0001009c            AUIPC a0, 0
000100a0            ADDI a0, a0, 702
000100a4            C.BEQZ a0, 12 LOC_100b0
000100a6            AUIPC a0, 0
000100aa            ADDI a0, a0, 636
000100ae            C.JAL 684 atexit
000100b0  LOC_100b0: C.JAL 162 __libc_init_array
000100b2            C.LWSP a0, 0(sp)
000100b4            C.ADDI4SPN a1, sp, 4
000100b6            C.LI a2, 0
000100b8            C.JAL 84 main
000100ba            C.J 126 exit
000100bc __do_global_dtors_aux: C.ADDI sp, -16
000100be            C.SWSP s0, 8(sp)
000100c0            ADDI s0, gp, -964
000100c4            LBU a5, 0(s0)
000100c8            C.SWSP ra, 12(sp)
000100ca            C.BNEZ a5, 30 LOC_100e8
000100cc            ADDI a5, zero, 0
000100d0            C.BEQZ a5, 18 LOC_100e2
```

```
000100d2            AUIPC a0, 1
000100d6            ADDI a0, a0, 806
000100da            AUIPC ra, 0
000100de            JALR ra, 0(zero)
000100e2  LOC_100e2: C.LI a5, 1
```

Пример .symtab

```
Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x113F8               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x113FC               0 SECTION  LOCAL    DEFAULT       3
[   4] 0x11404               0 SECTION  LOCAL    DEFAULT       4
[   5] 0x11408               0 SECTION  LOCAL    DEFAULT       5
[   6] 0x11838               0 SECTION  LOCAL    DEFAULT       6
[   7] 0x1184C               0 SECTION  LOCAL    DEFAULT       7
[   8] 0x0                   0 SECTION  LOCAL    DEFAULT       8
[   9] 0x0                   0 SECTION  LOCAL    DEFAULT       9
[  10] 0x0                   0 FILE     LOCAL    DEFAULT     ABS __call_atexit.c
[  11] 0x10074              18 FUNC     LOCAL    DEFAULT       1 register_fini
[  12] 0x0                   0 FILE     LOCAL    DEFAULT     ABS crtstuff.c
[  13] 0x113F8               0 OBJECT   LOCAL    DEFAULT       2
```

Листинг

hw4/Dis.java

```java
package hw4;


import java.io.*;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.nio.file.*;
import java.util.ArrayList;
import java.util.HashMap;
```

```
public class Dis {


   public static void main(String[] args) {


       long e_Shoff, e_shentsize, e_shnum; // оффсет на таблицу заголовков секций,
длина секции, кол-во секций
       String TEXT = ".text", SYM_TAB = ".symtab", STR_TAB = ".strtab"; // имена
соответствующих секций в e_shstrndx
       long addresProg_Bits = -1, offsetProg_Bits = -1, sizeProg_Bits = -1;
       long offsetSymTab = -1, sizeSymTab = -1;
       long offsetStrTab = -1, sizeStrTab = -1;
       long e_shstrndx, offsetE_shstrndx, nameE_shstrndx;


       byte[] progBits;
       ArrayList<String> binaryCommands = new ArrayList<>();
       String[] comands;
       String[] marks;
       ArrayList<byte[]> binaryMark = new ArrayList<>();
       byte[] symtabBits;
       byte[] strtabBits;


       HashMap<Long, String> mapFuncMarks;
       HashMap<Long, String> jumps;
       HashMap<Long, Long> placesJumps;


       try {
           Path nameFileInput = FileSystems.getDefault().getPath(args[0]);
           String nameFileOutput = args[1];
           byte[] in;
           try {
               in = Files.readAllBytes(nameFileInput); // считывание всего файла,
и проверка его корректности
               if (in[0] != 0x7f || in[1] != 0x45 || in[2] != 0x4c || in[3] !=
0x46) {
                   System.out.println("This file isn't ELF");
```

```
                    System.out.println("Input file must be correct 32-bits
LittleEndian ELF on RISC-V");
                    return;
                }
                if (in[4] != 0x01) {
                    System.out.println("This file is ELF, but it isn't 32-bits");
                    System.out.println("Input file must be correct 32-bits
LittleEndian ELF on RISC-V");
                    return;
                }
                if (in[5] != 0x01) {
                    System.out.println("This file is 32-bits ELF, but it isn't
LittleEndian");
                    System.out.println("Input file must be correct 32-bits
LittleEndian ELF on RISC-V");
                    return;
                }
                if (Byte.toUnsignedInt(in[18]) != 0xF3) {
                    System.out.println("This file is 32-bits LittleEndian ELF, but
it isn't RISC-V");
                    System.out.println("Input file must be correct 32-bits
LittleEndian ELF on RISC-V");
                    return;
                }
                e_shentsize = Byte.toUnsignedInt(in[0x2E]) +
Byte.toUnsignedInt(in[0x2F]) * 256; //считываем константы
                e_Shoff = Byte.toUnsignedInt(in[32]) + Byte.toUnsignedInt(in[33]) *
256 +
                        Byte.toUnsignedInt(in[34]) * 256 * 256 + (long)
Byte.toUnsignedInt(in[35]) * 256 * 256 * 256;
                e_shnum = Byte.toUnsignedInt(in[0x30]) +
Byte.toUnsignedInt(in[0x31]) * 256;
                e_shstrndx = (Byte.toUnsignedInt(in[50]) +
Byte.toUnsignedInt(in[51]) * 256);
                nameE_shstrndx = e_Shoff + (e_shstrndx) * e_shentsize;

                offsetE_shstrndx = Byte.toUnsignedInt(in[(int) nameE_shstrndx +
16]) +
                        Byte.toUnsignedInt(in[(int) nameE_shstrndx + 17])* 256 +
```

```
                        Byte.toUnsignedInt(in[(int) nameE_shstrndx + 18]) * 256 *
256 +
                        (long) Byte.toUnsignedInt(in[(int) nameE_shstrndx + 19]) *
256 * 256 * 256;


                for (long i = e_Shoff; i < e_Shoff + e_shentsize * e_shnum; i +=
e_shentsize) { // считываем координаты секций
                    long name = Byte.toUnsignedInt(in[(int) i]) +
Byte.toUnsignedInt(in[(int) i + 1]) * 256 +
                            Byte.toUnsignedInt(in[(int) i + 2]) * 256 * 256 +
(long) Byte.toUnsignedInt(in[(int) i + 3])
                            * 256 * 256 * 256;
                    long type = Byte.toUnsignedInt(in[(int) i + 4]) +
Byte.toUnsignedInt(in[(int) i + 5]) * 256 +
                            Byte.toUnsignedInt(in[(int) i + 6]) * 256 * 256 +
(long) Byte.toUnsignedInt(in[(int) i + 7])
                            * 256 * 256 * 256;


                    ArrayList<Byte> chars = new ArrayList<>(); //проверка имени
секции
                    String stringName = "";
                    long thisSectionOffset = offsetE_shstrndx + name;
                    while (in[(int) thisSectionOffset] != 0) {
                        chars.add(in[(int) thisSectionOffset]);
                        thisSectionOffset++;
                    }
                    if (chars.size() != 0) {
                        byte[] charsStringName = new byte[chars.size()];
                        for (int j = 0; j < chars.size(); j++) {
                            charsStringName[j] = chars.get(j);
                        }
                        stringName = new String(charsStringName);
                    }
                    long addres = Byte.toUnsignedInt(in[(int) i + 12]) +
Byte.toUnsignedInt(in[(int) i + 13])
                            * 256 + Byte.toUnsignedInt(in[(int) i + 14]) * 256 *
256 +
```

```
                            (long) Byte.toUnsignedInt(in[(int) i + 15]) * 256 * 256
* 256;
                    long offset = Byte.toUnsignedInt(in[(int) i + 16]) +
Byte.toUnsignedInt(in[(int) i + 17])
                            * 256 + Byte.toUnsignedInt(in[(int) i + 18]) * 256 *
256 +
                            (long) Byte.toUnsignedInt(in[(int) i + 19]) * 256 * 256
* 256;
                    long size = Byte.toUnsignedInt(in[(int) i + 20]) +
Byte.toUnsignedInt(in[(int) i + 21])
                            * 256 + Byte.toUnsignedInt(in[(int) i + 22]) * 256 *
256 +
                            (long) Byte.toUnsignedInt(in[(int) i + 23]) * 256 * 256
* 256;
                    if (type == 0x1 && stringName.equals(TEXT)) {
                        addresProg_Bits = addres;
                        offsetProg_Bits = offset;
                        sizeProg_Bits = size;
                    } else if (type == 0x02 && stringName.equals(SYM_TAB)) {
                        offsetSymTab = offset;
                        sizeSymTab = size;
                    } else if (type == 0x03 && stringName.equals(STR_TAB)) {
                        offsetStrTab = offset;
                        sizeStrTab = size;
                    }
                }

                progBits = new byte[(int) sizeProg_Bits];
                for (long i = offsetProg_Bits; i < offsetProg_Bits + sizeProg_Bits;
i++) {
                    progBits[(int) (i - offsetProg_Bits)] = in[(int) i];
                }
                for (int i = 0; i < progBits.length; i += 2) { //перевод команд в
двоичные строки
                    String lastB = "0" + "0" +
Integer.toBinaryString(Byte.toUnsignedInt(progBits[i]));
                    StringBuilder sb = new StringBuilder();
```

```
                    if (lastB.substring(lastB.length() - 2,
lastB.length()).equals("11")) { //RVC  or RVI
                        for (int j = i + 3; j >= i; j--) { //RVI
                            String binary =
Integer.toBinaryString(Byte.toUnsignedInt(progBits[j]));
                            if (binary.length() < 8) {
                                sb.append("0".repeat(8 - binary.length()));
                            }
                            sb.append(binary);
                        }
                        binaryCommands.add(sb.toString());
                        i += 2;
                    } else {
                        for (int j = i + 1; j >= i; j--) { //RVC
                            String binary =
Integer.toBinaryString(Byte.toUnsignedInt(progBits[j]));
                            if (binary.length() < 8) {
                                sb.append("0".repeat(8 - binary.length()));
                            }
                            sb.append(binary);
                        }
                        binaryCommands.add(sb.toString());
                    }
                }
                ProcCommand procCommand = new ProcCommand(binaryCommands,
addresProg_Bits); //обработка команд
                comands =  procCommand.disCommand();
                jumps = procCommand.getJumps(); //передача мапы прыжков
                placesJumps = procCommand.getPlacesJumps();//

                symtabBits = new byte[(int) sizeSymTab];
                for (long i = offsetSymTab; i < offsetSymTab + sizeSymTab; i++) {
                    symtabBits[(int) (i - offsetSymTab)] = in[(int) i];
                }
                for (int i = 0; i < symtabBits.length; i += 16) {
                    byte[] mark = new byte[16];
```

```
                    for (int j = i; j < i + 16; j++) {
                        mark[j - i] = symtabBits[j];
                    }
                    binaryMark.add(mark);
                }
                strtabBits = new byte[(int) sizeStrTab];
                for (long i = offsetStrTab; i < offsetStrTab + sizeStrTab; i++) {
                    strtabBits[(int) (i - offsetStrTab)] = in[(int) i];
                }
                ParseSymTab parseSymTab = new ParseSymTab(binaryMark, strtabBits);
// обработка симтаба
                marks = parseSymTab.disSymTab();
                mapFuncMarks = parseSymTab.getMapFuncMarks(); //передача мапы
функций
            } catch (IOException e) {
                System.out.println("Cannot read: " + e.getMessage());
                System.out.println("You must enter the data in the format: " +
                        "hw4.Dis <имя_входного_elf_файла> <имя_выходного_файла>");
                System.out.println("And yor file must be correct 32-bits
LittleEndian ELF on RISC-V");
                return;
            }
            try { //вывод (сначала команды потом симтаб)
                try (BufferedWriter output = new BufferedWriter(
                        new OutputStreamWriter(
                                new FileOutputStream(nameFileOutput)
                        )
                )) {
                    output.write(String.format("%s\n", TEXT));
                    long counter = addresProg_Bits;
                    for (int i = 0; i < comands.length; i++) { // вывод команд
                        if (mapFuncMarks.containsKey(counter)) { // добавление
меток функций
                            String func = mapFuncMarks.get(counter);
                            if (placesJumps.containsKey(counter)) { //добавление
метки того "куда попадает прыжок" (если команда прыгает)
```

```
                                if
(mapFuncMarks.containsKey(placesJumps.get(counter))) { // если место прыжка - FUNC
                                    output.write(String.format("%08x %10s: %s
%s\n", counter, func, comands[i],

mapFuncMarks.get(placesJumps.get(counter))));
                                } else { // если место прыжка не FUNC
                                    output.write(String.format("%08x %10s: %s
LOC_%05x\n", counter, func, comands[i],
                                            placesJumps.get(counter)));
                                }
                            } else {
                                output.write(String.format("%08x %10s: %s\n",
counter, func, comands[i]));
                            }
                        } else if (jumps.containsKey(counter)) { // добавление
новых меток "на прыжки"
                            String ourMark = jumps.get(counter);
                            if (placesJumps.containsKey(counter)) {
                                if
(mapFuncMarks.containsKey(placesJumps.get(counter))) {
                                    output.write(String.format("%08x %10s: %s
%s\n", counter, ourMark, comands[i],

mapFuncMarks.get(placesJumps.get(counter))));
                                } else {
                                    output.write(String.format("%08x %10s: %s
LOC_%05x\n", counter, ourMark, comands[i],
                                            placesJumps.get(counter)));
                                }
                            } else {
                                output.write(String.format("%08x %10s: %s\n",
counter, ourMark, comands[i]));
                            }
                        } else { //вывод если прыжков и функций нет
                            if (placesJumps.containsKey(counter)) {
                                if
(mapFuncMarks.containsKey(placesJumps.get(counter))) {
```

```
                                output.write(String.format("%08x %10s %s %s\n",
counter, "", comands[i],

mapFuncMarks.get(placesJumps.get(counter))));
                            } else {
                                output.write(String.format("%08x %10s %s
LOC_%05x\n", counter, "", comands[i],
                                        placesJumps.get(counter)));
                            }
                        } else {
                            output.write(String.format("%08x %10s %s\n",
counter, "", comands[i]));
                        }
                    }
                    if (comands[i].charAt(0) == 'C') {
                        counter += 2; // если 16-битная команда
                    } else {
                        counter += 4; // если 32-битная
                    }
                }
                output.newLine();


                output.write(String.format("%s\n", SYM_TAB));
                output.write(String.format("%s %-15s %7s %-8s %-8s %-8s %6s
%s\n", "Symbol", "Value",
                        "Size", "Type", "Bind", "Vis", "Index", "Name"));
                for (int i = 0; i < binaryMark.size(); i++) {
                    output.write(marks[i]);
                }
                output.close();
            }
        } catch (FileNotFoundException e) {
            System.out.println(nameFileOutput + "not found" + e.getMessage());
            System.out.println("You must enter the data in the format: " +
                    "hw4.Dis <имя_входного_elf_файла> <имя_выходного_файла>");
            return;
```

```
            } catch (IOException e) {
                System.out.println("Cannot write in output file: " +
e.getMessage());
                System.out.println("You must enter the data in the format: " +
                        "hw4.Dis <имя_входного_elf_файла> <имя_выходного_файла>");
                return;
            }
        } catch (IllegalArgumentException e) {
            System.out.println("File input incorrect" + e.getMessage());
            System.out.println("Input file must be correct 32-bits LittleEndian ELF
on RISC-V");
            return;
        } catch (IndexOutOfBoundsException e) {
            System.out.println("File input incorrect" + e.getMessage());
            System.out.println("Input file must be correct 32-bits LittleEndian ELF
on RISC-V");
        }
    }
}
```

hw4/ProcCommand.java

```
package hw4;


import java.util.ArrayList;
import java.util.HashMap;


public class ProcCommand {
    private long thisAddresProg_Bits;


    private final ArrayList<String> binaryCommands;
    private final HashMap<Long, String> jumps = new HashMap<>();
    private final HashMap<Long, Long> placesJumps = new HashMap<>();


    final HashMap<String, String> registerNames = new HashMap<>() {{ // обычные
регистры
```

```
        put("00000", "zero"); put("00001", "ra"); put("00010", "sp");
        put("00011", "gp"); put("00100", "tp"); put("00101", "t0");
        put("00110", "t1"); put("00111", "t2"); put("01000", "s0");
        put("01001", "s1"); put("01010", "a0"); put("01011", "a1");
        put("01100", "a2"); put("01101", "a3"); put("01110", "a4");
        put("01111", "a5"); put("10000", "a6"); put("10001", "a7");
        put("10010", "s2"); put("10011", "s3"); put("10100", "s4");
        put("10101", "s5"); put("10110", "s6"); put("10111", "s7");
        put("11000", "s8"); put("11001", "s9"); put("11010", "s10");
        put("11011", "s11"); put("11100", "t3"); put("11101", "t4");
        put("11110", "t5"); put("11111", "t6");
    }};


    final HashMap<Integer, String> registersCSR = new HashMap<>() {{ //машинные
регистры
        put(0x001, "fflags"); put(0x002, "frm"); put(0x003, "fcsr");
        put(0xC00, "cycle"); put(0xC01, "time"); put(0xC02, "instret");
        put(0xC80, "cycleh"); put(0xC81, "timeh"); put(0xC82, "instreth");
        put(0x100, "sstatus"); put(0x104, "sie"); put(0x105, "stvec");
        put(0x106, "scounteren"); put(0x10A, "senvcfg"); put(0x140, "sscratch");
        put(0x141, "sepc"); put(0x142, "scause"); put(0x143, "stval");
        put(0x144, "sip"); put(0x180, "satp"); put(0x5A8, "scontext");
        put(0x600, "hstatus"); put(0x602, "hedeleg"); put(0x603, "hideleg");
        put(0x604, "hie"); put(0x606, "hcounteren"); put(0x607, "hgeie");
        put(0x643, "htval"); put(0x644, "hip"); put(0x645, "hvip");
        put(0x64A, "htinst"); put(0xE12, "hgeip"); put(0x60A, "henvcfg");
        put(0x61A, "henvcfgh"); put(0x680, "hgatp"); put(0x6A8, "hcontext");
        put(0x605, "htimedelta"); put(0x615, "htimedeltah"); put(0x200,
"vsstatus");
        put(0x204, "vsie"); put(0x205, "vstvec"); put(0x240, "vsscratch");
        put(0x241, "vsepc"); put(0x242, "vscause"); put(0x243, "vstval");
        put(0x244, "vsip"); put(0x280, "vsatp"); put(0xF11, "mvendorid");
        put(0xF12, "marchid"); put(0xF13, "mimpid"); put(0xF14, "mhartid");
        put(0xF15, "mconfigptr"); put(0x300, "mstatus"); put(0x301, "misa");
        put(0x302, "medeleg"); put(0x303, "mideleg"); put(0x304, "mie");
```

```
        put(0x305, "mtvec"); put(0x306, "mcounteren"); put(0x310, "mstatush");
        put(0x340, "mscratch"); put(0x341, "mepc"); put(0x342, "mcause");
        put(0x343, "mtval"); put(0x344, "mip"); put(0x34A, "mtinst");
        put(0x34B, "mtval2"); put(0x30A, "menvcfg"); put(0x31A, "menvcfgh");
        put(0x747, "mseccfg"); put(0x757, "mseccfgh"); put(0xB00, "mcycle");
        put(0xB02, "minstret"); put(0xB80, "mcycleh"); put(0xB82, "minstreth");
        put(0x320, "mcountinhibit"); put(0x7A0, "tselect"); put(0x7A8, "mcontext");
        put(0x7B0, "dcsr"); put(0x7B1, "dpc"); put(0x7B2, "dscratch0"); put(0x7B3,
"dscratch1");
        put(0x7A1, "tdata1"); put(0x7A2, "tdata2"); put(0x7A3, "tdata3");
    }};


    final HashMap<String, String> registersRVC = new HashMap<>() {{ //сокращенные
регистры
        put("000", "s0"); put("001", "s1"); put("010", "a0");
        put("011", "a1"); put("100", "a2"); put("101", "a3");
        put("110", "a4"); put("111", "a5");
    }};


    final HashMap<String, String> typeR_and_RV32M = new HashMap<>() {{ //далее мапы
разбиты на похожие типы команд
        put("0000000000", "ADD"); put("0100000000", "SUB");
        put("0000000001", "SLL"); put("0000000010", "SLT");
        put("0000000011", "SLTU");put("0000000100", "XOR");
        put("0000000101", "SRL"); put("0100000101", "SRA");
        put("0000000110", "OR");  put("0000000111", "AND");
        put("0000001000", "MUL"); put("0000001001", "MULH");
        put("0000001010", "MULHSU"); put("0000001011", "MULHU");
        put("0000001100", "DIV"); put("0000001101", "DIVU");
        put("0000001110", "REM"); put("0000001111", "REMU");
    }};


    final HashMap<String, String> typeI = new HashMap<>() {{
        put("000", "ADDI"); put("001", "SLTI");
        put("011", "SLTIU"); put("100", "XORI");
```

```
        put("110", "ORI"); put("111", "ANDI");
        put("0000000001", "SLLI"); put("0000000101", "SRLI");
        put("0100000101", "SRLI");
    }};

    final HashMap<String, String> typeS = new HashMap<>() {{
        put("000", "SB"); put("001", "SH"); put("010", "SW");
    }};

    final HashMap<String, String> typeL = new HashMap<>() {{
        put("000", "LB"); put("001", "LH"); put("010", "LW");
        put("100", "LBU"); put("101", "LHU");
    }};

    final HashMap<String, String> typeB = new HashMap<>() {{
        put("000", "BEQ"); put("001", "BNE"); put("100", "BLT");
        put("101", "BGE"); put("110", "BLTU"); put("111", "BGEU");
    }};

    final HashMap<String, String> typeCSR = new HashMap<>() {{
        put("001", "CSRRW"); put("010", "CSRRS"); put("011", "CSRRC");
        put("101", "CSRRWI"); put("110", "CSRRSI"); put("111", "CSRRCI");
    }};




    public ProcCommand(ArrayList<String> binaryCommands, long addresProg_Bits) {
        this.binaryCommands = binaryCommands;
        thisAddresProg_Bits = addresProg_Bits;
    }

    private int parseInAdditionTo2(String binaryString) { //перевод числа в
дополнении до 2
        return Integer.parseUnsignedInt(binaryString.substring(1,
binaryString.length()), 2) -
```

```
                (1 << (binaryString.length() - 1)) *
Integer.parseInt(Character.toString(binaryString.charAt(0)));

    }


    private String convertR_andR32M(String com) { //далее ф-ции вида convert
обрабатывают типы команд

        StringBuilder sb = new StringBuilder();

        sb.append(typeR_and_RV32M.get(com.substring(0, 7) + com.substring(17,
20))).append(" ");

        sb.append(registerNames.get(com.substring(20, 25))).append(",").append("
");

        sb.append(registerNames.get(com.substring(12, 17))).append(",").append("
");

        sb.append(registerNames.get(com.substring(7, 12)));

        if (!typeR_and_RV32M.containsKey(com.substring(0, 7) + com.substring(17,
20))) {

            return "unknown_command";

        }

        return sb.toString();

    }

//

    private String convertI(String com) {

        StringBuilder sb = new StringBuilder();

        String str = com.substring(0, 7) + com.substring(17, 20);

        if (str.equals("0000000001") || str.equals("0000000101") ||
str.equals("0100000101")) {

            sb.append(typeI.get(str)).append(" ");

            sb.append(registerNames.get(com.substring(20,
25))).append(",").append(" ");

            sb.append(registerNames.get(com.substring(12,
17))).append(",").append(" ");

            sb.append(Integer.parseUnsignedInt(com.substring(7, 12), 2));

        } else {

            if (!typeI.containsKey(com.substring(17, 20))) {

                return "unknown_command";

            }

            sb.append(typeI.get(com.substring(17, 20))).append(" ");
```

```
            sb.append(registerNames.get(com.substring(20,
25))).append(",").append(" ");
            sb.append(registerNames.get(com.substring(12,
17))).append(",").append(" ");
            sb.append(parseInAdditionTo2(com.substring(0, 12)));
        }
        return sb.toString();
    }
    private String convertS(String com) {
        if (!typeS.containsKey(com.substring(17, 20))) {
            return "unknown_command";
        }
        return typeS.get(com.substring(17, 20)) + "," + " " +
                registerNames.get(com.substring(7, 12)) + "," + " " +
                parseInAdditionTo2(com.substring(0, 7) + com.substring(20, 25)) +
                "(" + registerNames.get(com.substring(12, 17)) + ")";
    }

    private String convertL(String com) {
        if (!typeL.containsKey(com.substring(17, 20))) {
            return "unknown_command";
        }
        return typeL.get(com.substring(17, 20)) + " " +
                registerNames.get(com.substring(20, 25)) + "," + " " +
                parseInAdditionTo2(com.substring(0, 12)) + "(" +
                registerNames.get(com.substring(12, 17)) + ")";
    }

    private String conwertB(String com) {
        if (!typeB.containsKey(com.substring(17, 20))) {
            return "unknown_command";
        }
        long jump = parseInAdditionTo2(com.substring(0, 1) + com.substring(24, 25)
+ com.substring(1, 7) +
                com.substring(20, 24) + "0");
```

```
        jumps.put(thisAddresProg_Bits + jump, String.format("LOC_%05x",
thisAddresProg_Bits + jump));
        placesJumps.put(thisAddresProg_Bits, thisAddresProg_Bits + jump);
        return typeB.get(com.substring(17, 20)) + " " +
                registerNames.get(com.substring(12, 17)) + "," + " " +
                registerNames.get(com.substring(7, 12)) + "," + " " + jump;
    }



    private String conwertCSR(String com) {
        StringBuilder sb = new StringBuilder();
        int number = Integer.parseUnsignedInt(com.substring(0, 12));
        sb.append(typeCSR.get(com.substring(17, 20))).append(" ");
        sb.append(registerNames.get(com.substring(20, 25))).append(",").append("
");
        if (registersCSR.containsKey(number)) {
            sb.append(registersCSR.get(number));
        } else {
            if (number <= 0xC1F && number >= 0xC03) {
                sb.append("hpmcounter").append(number - 0xC00);
            } else if (number <= 0xC9F && number >= 0xC83) {
                sb.append("hpmcounter").append(number - 0xC80).append("h");
            } else if (number <= 0x3AF && number >= 0x3A0) {
                sb.append("pmpcfg").append(number - 0x3A0);
            } else if (number <= 0x3EF && number >= 0x3B0) {
                sb.append("pmpaddr").append(number - 0x3B0);
            } else if (number <= 0xB1F && number >= 0xB03) {
                sb.append("mhpmcounter").append(number - 0xB00);
            } else if (number <= 0xB9F && number >= 0xB83) {
                sb.append("mhpmcounter").append(number - 0xB80).append("h");
            } else if (number <= 0x33F && number >= 0x323) {
                sb.append("mhpmevent").append(number - 0x320);
            } else {
```

```
                return "unknown_command";
            }
        }
        sb.append(",").append(" ");
        if (com.charAt(17) == '0') {
            sb.append(registerNames.get(com.substring(12, 17)));
        } else {
            sb.append(Integer.parseUnsignedInt(com.substring(12, 17)));
        }
        return sb.toString();
    }


    private String conwertRVC10001(String com) {
        switch (com.substring(4, 6)) {
            case "00":
                return "C.SRLI" + " " + registersRVC.get(com.substring(6, 9)) + ","
+ " " +
                        Integer.parseUnsignedInt(com.charAt(3) + com.substring(9,
14), 2);
            case "01":
                return "C.SRAI" + " " + registersRVC.get(com.substring(6, 9)) + ","
+ " " +
                        Integer.parseUnsignedInt(com.charAt(3) + com.substring(9,
14), 2);
            case "10":
                return "C.ANDI" + " " + registersRVC.get(com.substring(6, 9)) + ","
+ " " +
                        parseInAdditionTo2(com.charAt(3) + com.substring(9, 14));
            case "11":
                if (com.charAt(3) == '0') {
                    StringBuilder sb = new StringBuilder();
                    switch (com.substring(9, 11)) {
                        case "00":
                            sb.append("C.SUB");
                            break;
```

```
                    case "01":
                        sb.append("C.XOR");
                        break;
                    case "10":
                        sb.append("C.OR");
                        break;
                    case "11":
                        sb.append("C.AND");
                        break;
                }
                sb.append(" ").append(registersRVC.get(com.substring(6,
9))).append(",").append(" ");
                sb.append(registersRVC.get(com.substring(11, 14)));
                return sb.toString();
            } else {
                break;
            }
        }
        return "unknown_command";
    }

    private String conwertRVC10010(String com) {
        if (com.charAt(3) == '0') {
            if (com.substring(9, 14).equals("00000")) {
                return "C.JR" + " " + registerNames.get(com.substring(4, 9));
            } else {
                return "C.MV" + " " + registerNames.get(com.substring(4, 9)) + ","
+ " " +
                        registerNames.get(com.substring(9, 14));
            }
        } else {
            if (com.substring(9, 14).equals("00000")) {
                if (com.substring(4, 9).equals("00000")) {
                    return "C.EBREAK";
                } else {
```

```
                    return "C.JALR" + " " + registerNames.get(com.substring(4, 9));
                }
            } else {
                return "C.ADD" + " " + registerNames.get(com.substring(4, 9)) + ","
+ " " +
                        registerNames.get(com.substring(9, 14));
            }
        }
    }


    public String[] disCommand() throws IndexOutOfBoundsException { //обработка
всех команд
        String[] disassemblersCom = new String[binaryCommands.size()];
        for (int i = 0; i < binaryCommands.size(); i++) {
            String command = binaryCommands.get(i);
            if (command.length() == 32) { //обработка 32-битных
                switch (command.substring(25, 32)) {
                    case "0110011":
                        disassemblersCom[i] = convertR_andR32M(command); //R32IR
R32M
                        break;
                    case "0010011":
                        disassemblersCom[i] = convertI(command); //I
                        break;
                    case "0100011":
                        disassemblersCom[i] = convertS(command); // S
                        break;
                    case "0000011":
                        disassemblersCom[i] = convertL(command); // LI
                        break;
                    case "1100011":
                        disassemblersCom[i] = conwertB(command); //B
                        break;
                    case "1110011":
```

```
                        if (command.substring(0, 25).equals("0".repeat(25))) {
//Ebreak, Ecall
                            disassemblersCom[i] = "ECALL";
                        } else if (command.substring(0, 25).equals("0".repeat(11) +
"1" + "0".repeat(13))) {
                            disassemblersCom[i] = "EBREAK";
                        } else {
                            disassemblersCom[i] = conwertCSR(command); //CSR Zicsr
                        }
                        break;
                    case "1100111":
                        disassemblersCom[i] = "JALR" + " " +
registerNames.get(command.substring(20, 25)) + "," + " " +
                                parseInAdditionTo2(command.substring(0, 12)) + "("
+
                                registerNames.get(command.substring(12, 17)) + ")";
                        break;
                    case "1101111":
                        long jump = parseInAdditionTo2(command.charAt(0) +
command.substring(12, 20) +
                                command.charAt(11) + command.substring(1, 11) +
"0");
                        disassemblersCom[i] = "JAL" + " " +
registerNames.get(command.substring(20, 25)) + "," + " " +
                                jump;
                        jumps.put(thisAddresProg_Bits + jump,
String.format("LOC_%05x", thisAddresProg_Bits + jump));
                        placesJumps.put(thisAddresProg_Bits, thisAddresProg_Bits +
jump);
                        break;
                    case "0110111":
                        disassemblersCom[i] = "LUI" + " " +
registerNames.get(command.substring(20, 25)) + "," + " " +
                                parseInAdditionTo2(command.substring(0, 20)); // не
дописываем нули
                        break;
                    case "0010111":
                        disassemblersCom[i] = "AUIPC" + " " +
registerNames.get(command.substring(20, 25)) + "," + " " +
```

```
                                parseInAdditionTo2(command.substring(0, 20)); //не
дописываем нули
                        break;
                    default:
                        disassemblersCom[i] = "unknown_command";
                        break;
                }
                thisAddresProg_Bits += 4;
            } else { //обработка 16-битных
                switch (command.substring(0, 3) + command.substring(14, 16)) {
                    case "10001":
                        disassemblersCom[i] = conwertRVC10001(command);
                        break;
                    case "00000":
                        disassemblersCom[i] = "C.ADDI4SPN" + " " +
registersRVC.get(command.substring(11, 14)) + "," +
                                " " + "sp" + "," + " " +
Integer.parseUnsignedInt(command.substring(5, 9) +
                                command.substring(3, 5) + command.charAt(10) +
command.charAt(9) +
                                "0" + "0", 2);
                        break;
                    case "01000":
                        disassemblersCom[i] = "C.LW" + " " +
registersRVC.get(command.substring(11, 14)) + "," + " " +
                                Integer.parseUnsignedInt(command.charAt(10) +
command.substring(3, 6) +
                                        command.charAt(9) + "0" + "0", 2) + "(" +
registersRVC.get(
                                                command.substring(6, 9)) + ")";
                        break;
                    case "11000":
                        disassemblersCom[i] = "C.SW" + " " +
registersRVC.get(command.substring(11, 14)) + "," + " " +
                                Integer.parseUnsignedInt(command.charAt(10) +
command.substring(3, 6) +
                                        command.charAt(9)+ "0" + "0", 2) + "(" +
                                registersRVC.get(command.substring(6, 9)) + ")";
```

```
                        break;
                    case "00001":
                        if (command.substring(4, 9).equals("00000")) {
                            disassemblersCom[i] = "C.NOP" + " "; ///???
                        } else {
                            disassemblersCom[i] = "C.ADDI" + " " +
registerNames.get(command.substring(4, 9)) + "," +
                                    " ";
                        }
                        disassemblersCom[i] += parseInAdditionTo2(command.charAt(3)
+
                                command.substring(9, 14));
                        break;
                    case "00101":
                        long jump = parseInAdditionTo2(command.charAt(3) +
                                command.substring(7, 8) + command.substring(5, 7) +
command.charAt(9) +
                                command.charAt(8) + command.charAt(13) +
command.charAt(4) +
                                command.substring(10, 13) + "0");
                        disassemblersCom[i] = "C.JAL" + " " + jump;
                        jumps.put(thisAddresProg_Bits + jump,
String.format("LOC_%05x", thisAddresProg_Bits + jump));
                        placesJumps.put(thisAddresProg_Bits, thisAddresProg_Bits +
jump);
                        break;
                    case "01001":
                        disassemblersCom[i] = "C.LI" + " " +
registerNames.get(command.substring(4, 9)) + "," + " " +
                                parseInAdditionTo2(command.charAt(3) +
command.substring(9, 14));
                        break;
                    case "01101":
                        if (registerNames.get(command.substring(4,
9)).equals("sp")) {
                            disassemblersCom[i] = "C.ADDI16SP" + " " + "sp" + "," +
" " + parseInAdditionTo2(
                                    command.charAt(3) + command.substring(11, 13) +
```

```
                                        command.charAt(10) + command.charAt(13)
                                        + command.charAt(9) + "0".repeat(4));
                    } else {
                        disassemblersCom[i] = "C.LUI" + " " +
registerNames.get(command.substring(4, 9)) + "," + " "
                                + parseInAdditionTo2(command.charAt(3) +
command.substring(9, 14));
                    }
                    break;
                case "10101":
                    jump = parseInAdditionTo2(command.charAt(3) +
                            command.substring(7, 8) + command.substring(5, 7) +
command.charAt(9) +
                            command.charAt(8) + command.charAt(13) +
command.charAt(4) +
                            command.substring(10, 13) + "0");
                    disassemblersCom[i] = "C.J" + " " + jump;
                    jumps.put(thisAddresProg_Bits + jump,
String.format("LOC_%05x", thisAddresProg_Bits + jump));
                    placesJumps.put(thisAddresProg_Bits, thisAddresProg_Bits +
jump);
                    break;
                case "11001":
                case "11101":
                    if (command.substring(0, 3).equals("110")) {
                        disassemblersCom[i] = "C.BEQZ";
                    } else {
                        disassemblersCom[i] = "C.BNEZ";
                    }
                    jump = parseInAdditionTo2(command.charAt(3) +
command.substring(9, 11) +
                            command.charAt(13) + command.substring(4, 6) +
                            command.substring(11, 13) + "0");
                    disassemblersCom[i] += " " +
registersRVC.get(command.substring(6, 9)) + "," + " " + jump;
                    jumps.put(thisAddresProg_Bits + jump,
String.format("LOC_%05x", thisAddresProg_Bits + jump));
```

```
                        placesJumps.put(thisAddresProg_Bits, thisAddresProg_Bits +
jump);
                        break;
                    case "00010":
                        disassemblersCom[i] = "C.SLLI" + " " +
registerNames.get(command.substring(4, 9)) + "," +
                                " " + Integer.parseUnsignedInt(command.charAt(3) +
command.substring(9, 14),
                                2);
                        break;
                    case "01010":
                        disassemblersCom[i] = "C.LWSP" + " " +
registerNames.get(command.substring(4, 9)) + "," + " " +
                                Integer.parseUnsignedInt(command.substring(12, 14)
+ command.charAt(3) +
                                        command.substring(9, 12) + "0" + "0", 2) +
"(sp)";
                        break;
                    case "10010":
                        disassemblersCom[i] = conwertRVC10010(command);
                        break;
                    case "11010":
                        disassemblersCom[i] =  "C.SWSP" + " " +
registerNames.get(command.substring(9, 14)) + ","
                                + " " +
Integer.parseUnsignedInt(command.substring(7, 9) + command.substring(3, 7) +
                                "0" + "0", 2) + "(sp)";
                        break;
                    default:
                        disassemblersCom[i] = "unknown_command";
                        break;
                }
                thisAddresProg_Bits += 2;
            }
        }
        return disassemblersCom;
    }
```

```
    public HashMap<Long, String> getJumps() { //возвращаем мапу прыжков (ставим если на команду есть прыжок)
        return jumps;
    }

    public HashMap<Long, Long> getPlacesJumps() { //(ставим если с команды есть прыжок и куда он)
        return placesJumps;
    }
}
```

hw4/ParseSymTab.java

```
package hw4;

import java.util.ArrayList;
import java.util.HashMap;

public class ParseSymTab {
    private final ArrayList<byte[]> binaryMarks;
    private final byte[] strTabBits;
    private final HashMap<Long, String> mapFuncMarks = new HashMap<>();


    public ParseSymTab(ArrayList<byte[]> binaryMarks, byte[] strTabBits) {
        this.binaryMarks = binaryMarks;
        this.strTabBits = strTabBits;
    }

    private String getName(long offset) { //обрабатываем имя метки
        String name = "";
        int j = (int) offset;
        ArrayList<Byte> bytesforName = new ArrayList<>();
        while (Byte.toUnsignedInt(strTabBits[j]) != 0) { //считываем имя метки
```

```
            bytesforName.add(strTabBits[j]);
            j++;
        }
        if (bytesforName.size() != 0) { //переводим имя метки в символ
            byte[] stringName = new byte[bytesforName.size()];
            for (int k = 0; k < stringName.length; k++) {
                stringName[k] = bytesforName.get(k);
            }
            name = new String(stringName);
        }
        return name;
    }
```

```
    public String[] disSymTab() throws IndexOutOfBoundsException { // обрабатываем
все части меток
        String[] disassemblersMarks = new String[binaryMarks.size()];

        int count = 0;
        for (int i = 0; i < binaryMarks.size(); i++) {
            byte[] mark = binaryMarks.get(i);
            long offset = Byte.toUnsignedInt(mark[0]) + Byte.toUnsignedInt(mark[1])
* 256 +
                    Byte.toUnsignedInt(mark[2]) * 256 * 256 + (long)
Byte.toUnsignedInt(mark[3]) * 256 * 256 * 256;
            String name = getName(offset);
            long value = Byte.toUnsignedInt(mark[4]) + Byte.toUnsignedInt(mark[5])
* 256 +
                    Byte.toUnsignedInt(mark[6]) * 256 * 256 + (long)
Byte.toUnsignedInt(mark[7]) * 256 * 256 * 256;
            long size = Byte.toUnsignedInt(mark[8]) + Byte.toUnsignedInt(mark[9]) *
256 +
                    Byte.toUnsignedInt(mark[10]) * 256 * 256 + (long)
Byte.toUnsignedInt(mark[11]) * 256 * 256 * 256;
            int typeMark = Byte.toUnsignedInt(mark[12]);
            String bind = "", type = "", vis = "";
```

```
switch (typeMark / 16) { // bind
    case 0:
        bind = "LOCAL";
        break;
    case 1:
        bind = "GLOBAL";
        break;
    case 2:
        bind = "WEAK";
        break;
    case 10:
    case 11:
    case 12:
        bind = "OS";
        break;
    case 13:
    case 14:
    case 15:
        bind = "PROC";
        break;
    default:
        bind = "Unknown";
}

switch (typeMark % 16) { //type
    case 0:
        type = "NOTYPE";
        break;
    case 1:
        type = "OBJECT";
        break;
    case 2:
        type = "FUNC";
        break;
```

```
        case 3:
            type = "SECTION";
            break;
        case 4:
            type = "FILE";
            break;
        case 5:
            type = "COMMON";
            break;
        case 6:
            type = "TLS";
            break;
        case 10:
        case 11:
        case 12:
            type = "OS";
            break;
        case 13:
            type = "_SPARC_REGISTER";
            break;
        case 14:
        case 15:
            type = "PROC";
            break;
        default:
            type = "Unknown";
    }
    int visibility = Byte.toUnsignedInt(mark[13]);
    switch (visibility) { //vis
        case 0:
            vis = "DEFAULT";
            break;
        case 1:
            vis = "INTERNAL";
```

```
                    break;
                case 2:
                    vis = "HIDDEN";
                    break;
                case 3:
                    vis = "PROTECTED";
                    break;
                default:
                    vis = "Unknown";
            }
            long index = Byte.toUnsignedInt(mark[14]) +
Byte.toUnsignedInt(mark[15]) * 256;
            String sectionOnIndex = "";
            if (index == 0) { // соответствие секции по индексу (вложенные
промежутки названы одним именем)
                sectionOnIndex = "UNDEF";
            } else if (index == 0xff00) {
                sectionOnIndex = "BEFORE";
            } else if (index == 0xff01) {
                sectionOnIndex = "AFTER";
            } else if (index == 0xfff1) {
                sectionOnIndex = "ABS";
            } else if (index == 0xfff2) {
                sectionOnIndex = "COMMON";
            } else if (index == 0xffff) {
                sectionOnIndex = "XINDEX";
            } else if (index <= 0xff3f && index >= 0xff20) {
                sectionOnIndex = "OS";
            } else if (index <= 0xff1f && index >= 0xff00) {
                sectionOnIndex = "PROC";
            } else if (index <= 0xffff && index >= 0xff00) {
                sectionOnIndex = "RESERVE";
            }
            if (sectionOnIndex.isEmpty()) {
                sectionOnIndex = Long.toString(index);
```

```
            }
            if (type.equals("FUNC")) { //запись в мапу меток, чтобы потом приписать
в вывод команд
                if (name.isEmpty()) {
                    mapFuncMarks.put(value, String.format("LOC_%05x", value));
                } else {
                    mapFuncMarks.put(value, name);
                }
            }
            disassemblersMarks[i] = String.format("[%4d] 0x%-15X %5d %-8s %-8s %-8s
%6s %s\n", count, value, size,
                    type, bind, vis, sectionOnIndex, name);
            count++;
        }
        return disassemblersMarks;
    }


    public HashMap<Long, String> getMapFuncMarks() throws IndexOutOfBoundsException
{ //передаем мапу меток-функций
        return mapFuncMarks;
    }
}
```