САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 4

«ISA. Ассемблер, дизассемблер»

Выполнил: Султанов Мирзомансурхон Махсудович

Номер ИСУ: 311629

студ. гр. M3134

Санкт-Петербург

2021

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

Инструментарий и требования к работе: Java, JDK 16.0.2.7

Теоретическая часть

RISC-V – открытая и свободная ISA, основанная на концепции RISC. Система кодирования в RISC-V строится, как показано на рисунке 1. Rd (register destination) - регистр, в который направляется результат, rs (register source) – регистр, откуда берётся нужное значение, opcode (operation code) и funct определяют выполняемую операцию, imm (immediate) в терминах RISC это константа, которую мы можем получить из кода, не обращаясь к памяти.

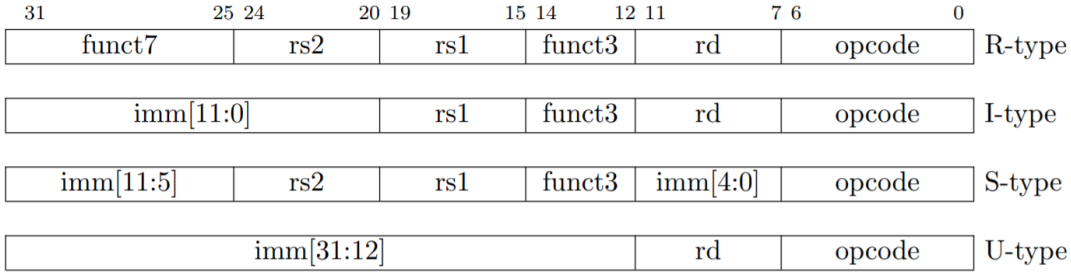

Рисунок 1 – Базовые конструкции кодировки в RISC-V

В целом, основная версия RISC (I) имеет 53 команды, но существуют и расширения. Для работы с перемножением чисел есть RISC RVM, для работы с плавающей точкой RISC RVF, для сжатых команд RISC RVC и т.д. Замечу, что RISC-V поддерживает как инструкции 32-битной длины, так и 64-битной. Но мы будем работать только с инструкциями 32-битной длины.

Инструкции бывают нескольких типов. Существуют типы R, I, S, B, U, J. К какому типу относится команда можно по opcode и по funct3. Чтобы же определить команду внутри типа для типа R ещё добавляется funct7. Для

примера, на рисунке 2 показаны все конструкции типа R, где различные значения funct7 + funct3 определяют разные команды.

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

Рисунок 2 – Примеры конструкций RV32 типа R

Так, R инструкции нужны для работы на регистрах. Мы считываем данные из rs1, rs2 и записываем результат операции в rd.

I инструкции нужны для действий, где одна из переменных берётся прямо из инструкции из imm.

S инструкции нужны для записи значений в память, но теперь imm нужен для определения дополнительного сдвига для адреса памяти.

B инструкции нужны для условных переходов. J инструкции для того, что совершить прыжок в другое место.

U инструкция нужная для записи верхних 20 бит в какой-либо регистр.

Раз уж упомянули регистры, то надо и про них расписать. В RISC-V 32 регистра, логичным образом для их кодировки потребуется 5 бит. Регистры называют x и (его номер). В unix системах есть соглашения по названию регистров, которую можно посмотреть на 1 таблице, которыми и руководствуются в RISC-V.

Таблица 1 – Соглашение об использовании регистров

Регистр	**Название**	**Смысл**
x0	Zero	Ноль
x1	Ra	Возвращаемое значение
x2	Sp	Указатель на стек
x3	Gp	Глобальный указатель
x4	Tp	Указатель потока
x5-x7	t0-t2	Временные регистры
x8-x9	s0-s1	Регистры, используемые вызывающим
x10-x17	a0-a7	Регистры аргументов
x18-x27	s2-s11	Регистры, используемые вызывающим
x28-x31	t3-t6	Временные регистры

ELF (Executable and Linkable Format) – дословно формат исполняемых и связываемых файлов. Он определяет структуру бинарных файлов, библиотек, и файлов ядра. Спецификация формата позволяет операционной системе корректно интерпретировать содержащиеся в файле машинные команды. Файл ELF, как правило, является выходным файлом компилятора или линкера и имеет двоичный формат. С помощью подходящих инструментов он может быть проанализирован и изучен.

Заголовок ELF-файла состоит из трёх частей- ELF header, Program header и Section Header.

В самом начале идёт ELF header, в котором дана основная информация о файле. Первые четыре байта это магические числа самого ELF-файла – 0x7F и идущие за ними закодированные буквы ELF (45 4c 46). Дальше идёт байт формата файла. Для 1 или 2 соответствуют 32- и 64-битные форматы. За ним байт, который определяет endian. Для 1 и 2

соответствуют little и big endian. Дальше идёт байт версии ELF, всегда должно равняться 1. Затем идёт много полезной информации о файле, которую можно увидеть на 3 рисунке. Я не буду описывать здесь все из них. Отмечу лишь те, которые пригодятся в дальнейшем. e_shoff хранит в себе оффсет на начало section header, e_shentsize хранит размер одной секции из section header, e_shnum хранит количество секций в section header, e_shstrndx хранит индекс секции, которая хранит в себе имена всех секций.

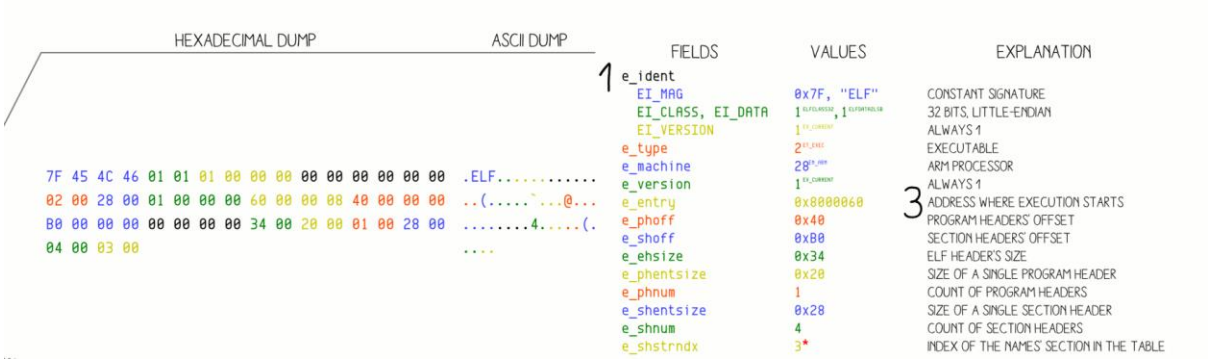

Рисунок 3 – структура ELF header

Program Header хранит много полезной информации для исполнения программы, заложенной в ELF файле. Полезна она будет именно при исполнении, а не при парсинге, поэтому вдаваться в детали устройства Program Header здесь не будем.

Section Header хранит в себе секции, на которые разбит весь код. Каждая секция имеет свою цель. Так, из тех секций, которые нам пригодятся, есть text – это основной исполняемый код, symbol table – здесь хранится вся информация, чтобы найти или переместить нужные переменные и ссылки, string table – здесь хранится информация про имена переменных, shrtrtab – здесь хранятся названия всех секций, индекс к которой нам известен из упомянутой выше sh_strndx. Вернёмся к Section Header. На его начало указывает e_shoff. Для каждой секции есть свой заголовок, которые идут друг за дружкой, начиная с оффсета e_shoff, и при этом каждый из заголовков имеет размер e_shentsize. Аналогично в

заголовке каждой секции есть немало полезной информации, но для нас будут иметь значения sh_offset – начало секции по сравнению с началом файла, sh_size – размер самой секции в байтах.

Описание работы написанного кода

1)Откроем файл для чтения.

2)Прочитаем Elf Header, попутно обрабатывая возможные ошибки и считывая все параметры Elf Header.

3)Прочитаем секцию shrtrtab, чтобы узнать имена всех секций.

4)Найти нужные секции, а именно text, symbol table и string table и распарсить их.

5)Распарсить text. А точнее, сначала надо пройтись по всем командам и запомнить, где нужно поставить метки, названия которых нет в symbol table. А затем распарсить все команды с учётом меток. К сожалению, в моём коде не реализован парсер команд.

6)Распарсить symbol table.

Результат работы

```
.symtab
Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x115CC               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x115D0               0 SECTION  LOCAL    DEFAULT       3
[   4] 0x115D8               0 SECTION  LOCAL    DEFAULT       4
[   5] 0x115E0               0 SECTION  LOCAL    DEFAULT       5
```

```
[   6] 0x11A08              0 SECTION  LOCAL    DEFAULT       6
[   7] 0x11A14              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             24 FUNC     LOCAL    DEFAULT       1 register_fini
[  12] 0x0                  0 FILE     LOCAL    DEFAULT     ABS crtstuff.c
[  13] 0x115CC              0 OBJECT   LOCAL    DEFAULT       2
[  14] 0x100D8              0 FUNC     LOCAL    DEFAULT       1
__do_global_dtors_aux
```

Листинг кода

Java 16.0.2.7 AdoptOpenJdk

Hw4.java

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.*;


public class Hw4 {
    public static void main(String[] args) {
        System.out.println("Input file must be in the package src!");
        String input = args[0];
        String output = args[1];
        try {
            byte[] array = Files.readAllBytes(Paths.get("src/" + input));
            //System.err.println("Array length: " + array.length + " bytes");
            try {
```

```
                new RiscVParser(array, output).parse();
            } catch (IllegalArgumentException e) {
                System.err.println("Something bad happened during parsing: ");
                e.printStackTrace();
            }
        } catch (IOException e) {
            System.err.println("File " + input + " does not exist or it is not in
src.");
        }

    }

    private static class RiscVParser {

        private SectionHeaderEntry symtab;
        private SectionHeaderEntry text;
        private List<Symbol> symbols;
        private static SectionHeaderEntry strtable;

        public static class Symbol {
            private final int start;
            public int st_name;
            public int st_value;
            public int st_size;
            public int st_info;
            public int st_other;
            public short st_shndx;
            public Symbol(int start) {
```

```
        this.start = start;
        parse();
    }

    public void parse() {
        st_name = getInteger(start, 4);
        st_value = getInteger(start + 4, 4);
        st_size = getInteger(start + 8, 4);
        st_info = getInteger(start + 12, 1);
        st_other = getInteger(start + 13, 1);
        st_shndx = (short) getInteger(start + 14, 2);
    }

    public String getType() {
        int type = st_info & 0xF;
        return switch (type) {
            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 -> "UNKNOWN";
        };
    }

    public String getBind() {
        int bind = st_info >>> 4;
        return switch (bind) {
            case 0 -> "LOCAL";
            case 1 -> "GLOBAL";
            case 2 -> "WEAK";
            case 10 -> "LOOS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> "UNKNOWN";
        };
    }

    public String getVis() {
        int vis = st_other & 0x3;
        return switch (vis) {
            case 0 -> "DEFAULT";
            case 1 -> "INTERNAL";
            case 2 -> "HIDDEN";
            case 3 -> "PROTECTED";
            default -> "UNKNOWN";
        };
```

```
}

public String getIndex() {
    if (st_shndx == (short) 0xfff1) {
        return "ABS";
    }
    if (st_shndx == (short) 0xfff2) {
        return "COMMON";
    }
    //0xff00 = LOPROC
    //0xff1f = HIPROC
    if (Short.compareUnsigned((short) 0xff00, st_shndx) <= 0 &&
    Short.compareUnsigned(st_shndx, (short) 0xff1f) <= 0) {
        return "PROC_RES";
    }
    //0xff20 = LOOC
    //0xff3f = HIOC
    if (Short.compareUnsigned((short)0xff20, st_shndx) <= 0 &&
    Short.compareUnsigned(st_shndx, (short) 0xff3f) <= 0) {
        return "OP_RES";
    }
    if (st_shndx == 0) {
        return "UNDEF";
    }
    if (st_shndx == (short) 0xffff) {
        return "XINDEX";
    }
```

```
            //0xff00 = LORESERVE
            //0xffff = HIRESERVE
            if ((short)0xff00 <= st_shndx && st_shndx <= (short)0xffff) {
                return "RESERVED";
            }
            return String.valueOf(st_shndx);
        }

        public String getName() {
            StringBuilder sb = new StringBuilder();
            int i = 0;
            while (array[strtable.sh_offset + st_name + i] != 0) {
                sb.append((char) array[strtable.sh_offset + st_name + i]);
                i++;
            }
            return sb.toString();
        }
    }
    public static class SectionHeaderEntry {
        private final int start;
        public int sh_name;
        public int sh_type;
        public int sh_flags;
        public int sh_addr;
        public int sh_offset;
        public int sh_size;
        public int sh_link;
```

```
public int sh_info;
public int sh_addralign;
public int sh_entsize;

public SectionHeaderEntry(int start) {
    this.start = start;
    parse();
}

public void parse() {
    sh_name = getInteger(start, 4);
    sh_type = getInteger(start + 0x04, 4);
    sh_flags = getInteger(start + 0x08, 4);
    sh_addr = getInteger(start + 0x0C, 4);
    sh_offset = getInteger(start + 0x10, 4);
    sh_size = getInteger(start + 0x14, 4);
    sh_link = getInteger(start + 0x18, 4);
    sh_info = getInteger(start + 0x1C, 4);
    sh_addralign = getInteger(start + 0x20, 4);
    sh_entsize = getInteger(start + 0x24, 4);
}

public void print() {
    System.err.printf("%16s%08x%n","sh_name: ", sh_name);
    System.err.printf("%16s%08x%n","sh_type: ", sh_type);
    System.err.printf("%16s%08x%n","sh_flags: ", sh_flags);
    System.err.printf("%16s%08x%n","sh_addr: ", sh_addr);
```

```
            System.err.printf("%16s%08x%n","sh_offset: ", sh_offset);
            System.err.printf("%16s%08x%n","sh_size: ", sh_size);
            System.err.printf("%16s%08x%n","sh_link: ", sh_link);
            System.err.printf("%16s%08x%n","sh_info: ",  sh_info);
            System.err.printf("%16s%08x%n","sh_addralign: ", sh_addralign);
            System.err.printf("%16s%08x%n","sh_entsize: ", sh_entsize);
        }

        public void print(String s) {
            System.err.println("----------");
            System.err.println(s);
            print();
            System.err.println("----------");
        }
    }
    private static byte[] array;
    private final String output;
    private int e_entry;
    private int e_phoff;
    private int e_shoff;
    private int e_flags;
    private int e_ehsize;
    private int e_shentsize;
    private int e_shnum;
    private int e_shstrndx;
    private int e_phnum;
    private int e_phentsize;
```

```
        public RiscVParser(byte[] array, String output) {
            this.array = array;
            this.output = output;
        }

        public void parse() {
            parse32FileHeader();
            SectionHeaderEntry shrtrtab = new SectionHeaderEntry(e_shoff +
e_shstrndx*e_shentsize);
            shrtrtab.print();
            Map<Integer, String> namesAndOffset =
getNamesAndOffsets(shrtrtab.sh_offset, shrtrtab.sh_size, 1);
            System.err.println(namesAndOffset);
            findSymtabAndText(namesAndOffset);

            text.print(".text");
            symtab.print(".symtab");
            strtable.print(".strtable");
            symbols = getAllSymb();
            try (PrintWriter writer = new PrintWriter(output,"UTF-8")) {
                dumpInstructions();
                dumpSymbols(writer);
            } catch (IOException e) {
                System.err.println("Something bad happened with output file!");
                e.printStackTrace();
            }
```

```
            //System.err.println(dump(0, array.length));
        }

        private void dumpInstructions() {
            int curOffset = 0;
            Set<Integer> unmarked = findUnmarked();
            for (Integer arg : unmarked) {
                System.err.println(Integer.toHexString(arg));
            }
            //And now we need to disassemble instructions from RISC_V RV32I,
RV32M, RVC
            do {
                curOffset += 4;
            } while (curOffset < text.sh_size);
        }

        private Set<Integer> findUnmarked() {
            int curOffset = 0;
            Set<Integer> answer = new TreeSet<>();
            do {
                int virtualAddress = curOffset + text.sh_addr;
//                System.err.println("_________________________");
//                System.err.println("virtual address: " +
Integer.toHexString(virtualAddress));
                int instruction = getInteger(text.sh_offset + curOffset, 4);
                //System.err.println("Instruction: " +
Integer.toBinaryString(instruction));
                int opcode = instruction & ((1 << 7) - 1);
```

```
                //System.err.println("opcode: " + Integer.toBinaryString(opcode));
                if (opcode == 0b1101111) { //JAL
                    int offset = getOffsetForJType(instruction);
                    //System.err.println("ofset(J): " + offset);
                    if (checkUnmarked(offset + virtualAddress)) {
                        //System.err.println("sum: " +
Integer.toHexString(virtualAddress + offset));
                        answer.add(virtualAddress + offset);
                    }
                } else if (opcode == 0b1100011) {
                    int offset = getOffsetForBType(instruction);
                    //System.err.println("offset(B): " + offset);
                    if (checkUnmarked(offset + virtualAddress)) {
                        //System.err.println("sum: " +
Integer.toHexString(virtualAddress + offset));
                        answer.add(virtualAddress + offset);
                    }
                }
                curOffset += 4;
            } while (curOffset < text.sh_size);
            return answer;
        }
        private int signExtend(int val, int nBits) {
            if ((val & (1 << nBits)) != 0) {
                val = -(-val & ((1 << nBits) - 1));
            }
            return val;
        }
```

```
        private int getOffsetForBType(int instruction) {
            //12 | 10 : 5
            int imm1 = instruction >>> 25;
            //System.err.println("imm1: " + Integer.toBinaryString(imm1));
            int firstPart = getPerm(imm1, List.of(12, 10, 9, 8, 7, 6, 5));
            //System.err.println("firstPart: " +
Integer.toBinaryString(firstPart));
            int imm2 = (instruction >>> 7) & (0b11111);
            //System.err.println("imm2: " + Integer.toBinaryString(imm2));
            int secondPart = getPerm(imm2, List.of(4, 3, 2, 1, 11));
            //System.err.println("secondPart: " +
Integer.toBinaryString(secondPart));
            //System.err.println(Integer.toBinaryString(firstPart | secondPart));
            return signExtend((firstPart | secondPart)*2, 12);
        }

        private boolean checkUnmarked(int offset) {
            for (int i = 0; i < symbols.size(); i++) {
                if (symbols.get(i).st_value == offset) {
                    return false;
                }
            }
            return true;
        }

        private int getOffsetForJType(int instruction) {
            int imm = instruction >>> 12;
            //System.err.println("imm: " + imm);
```

```
            int offset = getPerm(imm, List.of(20, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
11, 19, 18, 17, 16, 15, 14, 13, 12));
            //20 | 10:1 | 11 | 19:12
            // 20 10 9 8 7 6 5 5 3 2 1 11 19 18 17 16 15 14 12
            return signExtend(offset*2, 20);
        }

        private int getPerm(int imm, List<Integer> integers) {
            String bin = Integer.toBinaryString(imm);
            StringBuilder binb = new StringBuilder();
            binb.append("0".repeat(Math.max(0, integers.size() - bin.length())));
            binb.append(bin);
            //System.err.println("binb: " + binb);
            int answer = 0;
            for (int i = 0; i < integers.size(); i++) {
                answer |= Integer.parseInt(binb.substring(i, i + 1)) <<
(integers.get(i) - 1);
            }
            return answer;
        }

        private long getLong(int start, int size) {
            long answer = 0;
            for (int i = 0; i < size; i++) {
                //System.err.println(String.format("%08x", ((array[start + i] &
0xFF) << (i*8))));
                answer |=  ((array[start + i] & 0xFF) << (i*8));
            }
```

```
            return answer;
        }

        private void dumpSymbols(PrintWriter writer) {
            writer.println(".symtab");
            writer.printf("%s %-15s %7s %-8s %-8s %-8s %6s %s\n", "Symbol",
"Value", "Size", "Type", "Bind", "Vis", "Index", "Name");
            for (int i = 0; i < symbols.size(); i++) {
                Symbol symbol = symbols.get(i);
                writer.printf("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                        i,
                        symbol.st_value,
                        ((long) symbol.st_size) & 0xFFFFl,
                        symbol.getType(),
                        symbol.getBind(),
                        symbol.getVis(),
                        symbol.getIndex(),
                        symbol.getName());
            }
        }

        private void dumpSymbols() {
            System.out.printf("%s %-15s %7s %-8s %-8s %-8s %6s %s\n", "Symbol",
"Value", "Size", "Type", "Bind", "Vis", "Index", "Name");
            for (int i = 0; i < symbols.size(); i++) {
                Symbol symbol = symbols.get(i);
                System.out.printf("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                        i,
```

```
                symbol.st_value,
                ((long) symbol.st_size) & 0xFFFFl,
                symbol.getType(),
                symbol.getBind(),
                symbol.getVis(),
                symbol.getIndex(),
                symbol.getName());
        }
    }

    private List<Symbol> getAllSymb() {
        List<Symbol> answer = new ArrayList<>();
        int start = symtab.sh_offset;
        for (int i = 0; i < symtab.sh_size / 16; i++) {
            answer.add(new Symbol(start + i*16));
        }
        return answer;
    }

    private void findSymtabAndText(Map<Integer, String> namesAndOffset) {
        symtab = null;
        text = null;

        for (int i = 0; i < e_shnum; i++) {
            if (i == e_shstrndx) {
                continue;
            }
```

```
                SectionHeaderEntry entry = new SectionHeaderEntry(e_shoff +
e_shentsize*i);
//                System.err.println("----------");
//                System.err.println(entry.sh_name + " " +
namesAndOffset.get(entry.sh_name));
                if (namesAndOffset.getOrDefault(entry.sh_name,
"").equals(".text")) {
                    text = entry;
                }
                if (namesAndOffset.getOrDefault(entry.sh_name,
"").equals(".symtab")) {
                    symtab = entry;
                }
                if (namesAndOffset.getOrDefault(entry.sh_name,
"").equals(".strtab")) {
                    strtable = entry;
                }
            }
            if (text == null) {
                throw new IllegalArgumentException(".text is not found but its
name was found!");
            }
            if (symtab == null) {
                throw new IllegalArgumentException(".symtab is not found but its
name was found!");
            }
            if (strtable == null) {
                throw new IllegalArgumentException(".strtable is not found but its
name was found!");
            }
```

```
        }

        private Map<Integer, String> getNamesAndOffsets(int start, int size, int
offset) {
            Map<Integer, String> namesAndOffset = new HashMap<>();
            StringBuilder sb = new StringBuilder();
            for (int i = 1; i < size; i++) {
                if (array[start + i] != 0) {
                    sb.append((char) array[start + i]);
                } else {
                    namesAndOffset.put(offset, sb.toString());
                    sb = new StringBuilder();
                    offset = i + 1;
                }
            }
            if (!namesAndOffset.containsValue(".text")) {
                throw new IllegalArgumentException(".text section is not found!");
            }
            if (!namesAndOffset.containsValue(".symtab")) {
                throw new IllegalArgumentException(".symtab is not found!");
            }
            if (!namesAndOffset.containsValue(".strtab")) {
                throw new IllegalArgumentException(".strtab is not found!");
            }
            return namesAndOffset;
        }

        private void parse32FileHeader() {
```

```
            if (!(array[0x00] == (byte) 0x7f && array[0x01] == (byte) 0x45 &&
array[0x02] == (byte) 0x4c && array[0x03] == (byte) 0x46)) {
                throw new IllegalArgumentException("Magic numbers don't correspond
to magic numbers of Elf file! actual: " + dump(0, 4));
            }
            if (array[0x04] != 1) {
                throw new IllegalArgumentException("That elf doesn't have 32-bit
format! actual: " + dump(4, 1));
            }
            if (array[0x05] != 1) {
                throw new IllegalArgumentException("That elf doesn't have little
endian format! actual: " + dump(5, 1));
            }
            if (array[0x06] != 1) {
                throw new IllegalArgumentException("Elf version of that file is
not original or not current! actual: " + dump(6, 1));
            }
            // We consider that ABI version is fine ???
            if (!(checkZeros(0x09, 7))) {
                throw new IllegalArgumentException("EI_PAD currently unused and
should be filled with zeros! actual: " + dump(9, 7));
            }
//            if (!(array[0x10] == (byte) 0x02 && array[0x11] == (byte) 0x00)) {
//                throw new IllegalArgumentException("E_TYPE should ET.EXEC which
corresponds to 0x0200! actual: " + dump(16, 2));
//            }
            //Not sure about that one
            if (!(array[0x12] == (byte) 0xf3 && array[0x13] == (byte) 0x00)) {
                throw new IllegalArgumentException("E_MACHINE should be RISC-V!
which corresponds to 0xf300! actual: " + dump(18, 2));
```

```
            }
            if (!(array[0x14] == (byte) 1 && checkZeros(21, 3))) {
                throw new IllegalArgumentException("E_VERSION should be set to 1
for the original version of ELF! actual: " + dump(20, 2));
            }
            e_entry = getInteger(0x18, 4);
            e_phoff = getInteger(0x1C, 4);
            e_shoff = getInteger(0x20, 4);
            e_flags = getInteger(0x24, 4);
            e_ehsize = getInteger(0x28, 2);
            e_phentsize = getInteger(0x2A, 2);
            e_phnum = getInteger(0x2C, 2);
            e_shentsize = getInteger(0x2E, 2);
            e_shnum = getInteger(0x30, 2);
            e_shstrndx = getInteger(0x32, 2);
//            System.err.println("e_phoff: " + String.format("%02x", e_phoff));
//            System.err.println("e_phnum: " + String.format("%02x", e_phnum));
//            System.err.println("e_phentsize: " + String.format("%02x",
e_phentsize));
//            System.err.println("e_shentsize: " + String.format("%02x",
e_shentsize));
//            System.err.println("e_shstrndx: " + String.format("%02x",
e_shstrndx));
//            System.err.println("e_shoff: " + String.format("%04x", e_shoff));
        }

        private static int getInteger(int start, int size) {
            int answer = 0;
            for (int i = 0; i < size; i++) {
```

```
                //System.err.println(String.format("%08x", ((array[start + i] &
0xFF) << (i*8))));
                answer |= ((array[start + i] & 0xFF) << (i*8));
            }
            return answer;
        }

        private String dump(int start, int size) {
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < size; i++) {
                sb.append(String.format("%02x", array[start + i])).append(' ');
            }
            return sb.toString();
        }

        private boolean checkZeros(int start, int size) {
            for (int i = 0; i < size; i++) {
                if (array[start + i] != 0) {
                    return false;
                }
            }
            return true;
        }
    }
}
```