| *Лабораторная работа №3* | **M3136** | *2022* |
|---|---|---|
| ***ISA*** | **Артемьев Иван Вадимович** | |

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

**Инструментарий:** Java 19.

**Описание:** Написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код из elf файла в текст программы на языке ассемблера.

## *1. Описание системы кодирования команд RISC-V.*

RISC-V — это открытая стандартная архитектура набора инструкций (ISA), которая изначально создавалась в учебных целях. Каждая команда выполняется за 1 такт и имеет фиксированную длину. Процесс выполнения команды называется конвейером: считывание команды из памяти, раскодирование, выполнение операции, обращение к памяти(если надо), запись результата в регистр. Также архитектура RISC-V расширяема, а именно можно легко добавить новые функции и расширения.

Наборы инструкций RISC-V:

1. **RV32I** - 32-ух битный набор базовых инструкций взаимодействия с целочисленными данными. Состоит из инструкций следующих типов:
    a. Integer Register-Immediate Instructions(addi, ori, xori…)
    b. Integer Register-Register Instructions(and, sub, or…)
    c. Control Transfer Instructions(jal, jalr, beq, bne…)
    d. Load and store Instructions(lb, lh, sb, sh…)
    e. Memory Ordering Instructions(fence)
    f. Call and breakpoint Instructions(ebreak, ecall)

    Также в этом наборе инструкций содержится 32 регистра общего назначения. x0 - аппаратный ноль (zero) в него можно сохранять любые данные и он всё равно будет нулём.

2. **RV64I** - дополнение к 32-ух битному набору *RV32I*, для взаимодействия с 64-ех битными инструкциями.
3. **RV32/64M** - набор инструкций выполняющий базовые арифметические операции: сложение, умножение…
4. **RV32/64A** - Стандартное расширение атомарных инструкций и содержит инструкции, которые атомарно считывают, изменяют и записывают память для поддержки синхронизации между несколькими RISC-V HART, работающими в одном и том же пространстве памяти.
5. **RV32/64C** - Расширение совместимо со всеми другими стандартными расширениями инструкций. Расширение C позволяет свободно смешивать 16-битные инструкции с 32-битными, причем последние теперь могут начинаться на любой 16-битной границе. С добавлением расширения инструкции JAL и JALR больше не будут вызывать исключение неправильного выравнивания инструкции.
6. **RV32/64Zicsr** - Инструкции SYSTEM используются для доступа к системным функциям, которые могут потребовать привилегированного доступа, и кодируются с использованием формата инструкций I-типа.
7. **RV32/63F** - инструкции для работы с данными типа float (single-precision).
8. **RV32/64D** - инструкции для работы с данными типа double (double-precision)

Существует 6 форматов инструкций, каждый из которых необходим для определенного типа функций.

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

*рис 1. "32-bit RISC-V instruction formats"*

***rs1, rs2*** - номера регистров в которых находятся первый и второй операнды соответственно.

**rd** - номер регистра в который записывается результат

**opcode**, **funct3**, **funct7** - задают команду.

**imm[x:y]** - знаково-расширенное число.

## *2. Описание структуры файла ELF.*

ELF(executable and linkable format) файл - это исполняемый файл, который имеет одинаковый стандарт для всех ОС, поэтому широко используется.

Файл состоит из сегментов и секций. Сегменты содержат информацию необходимую для исполнения файла. Секции содержат информацию о связи и реаллокации, при этом любой байт из файла содержится не более чем в одной секции.

Первые 52 байта elf файла - это ELF-Header (заголовок), он содержит информацию о типе файла, о кодировке и др. Также в заголовке содержатся данные о количестве секций и секторов, о том где начинаются заголовки каждой из секций и секторов и их размеры. Также индекс секции “.shstrtab” - эта секция содержит информацию об именах всех остальных секций.

Заголовки секторов содержат атрибуты сектора либо другую информацию, необходимую ОС для подготовки программы к исполнению.

Заголовки секций содержат информацию о местонахождении данных секции, название и некоторые параметры необходимые для компоновщика, чтобы правильно разместить все данные.

Самих секций может содержаться любое количество внутри файл, но есть несколько основных:

“***.text***” - содержит исполняемый код, который загружается 1 раз. В нашем случае в этой секции содержатся все команды ассемблера

“*.data*” - Инициализированные данные, с правами на чтение и запись.

“*.rodata*” - Инициализированные данные, с правами только на чтение.

“*.bss*” - Неинициализированные данные, с правами на чтение/запись.

“***.symtab***” - symbol table - таблица символов - это таблица состоящая из строк фиксированной длины(16 байт), каждая из которых содержит информацию для локации и релокации символьных определений и ссылок. В ней есть переменные с рис 3.

```
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;
```

*рис 2. поля symbol table entry*

Из которых также можно получить необходимые st_bind и st_type.

## *3. Описание работы написанного кода.*

Для запуска программы необходимо передать ей аргументы через командую строку:

**rv3 <имя_входного_elf_файла> <имя_выходного_файла>**

Сперва распарсим elf header, мы знаем, что он состоит из 52 байтов, и знаем порядок следования данных в заголовке, при этом мы можем не запоминать некоторые данные (например первые 16 байт) потому что в нашей реализации они не нужны.

```
#define EI_NIDENT       16

typedef struct {
          unsigned char  e_ident[EI_NIDENT];
          Elf32_Half     e_type;
          Elf32_Half     e_machine;
          Elf32_Word     e_version;
          Elf32_Addr     e_entry;
          Elf32_Off      e_phoff;
          Elf32_Off      e_shoff;
          Elf32_Word     e_flags;
          Elf32_Half     e_ehsize;
          Elf32_Half     e_phentsize;
          Elf32_Half     e_phnum;
          Elf32_Half     e_shentsize;
          Elf32_Half     e_shnum;
          Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

*рис 3. ELF header*

После того, как распарсили заголовок всего elf файла, мы будем парсить заголовки всех секций. Мы знаем что они начинаются в e_shoff и знаем их количество(e_shnum), а также знаем длину одного заголовка (e_shentsize). Значит мы берем по e_shentsize байт и парсим заголовок.

Чтобы распарсить заголовок мы берем e_shentsize байтов и зная какие данные находятся в заголовке сохраняем их.

Теперь, мы знаем индекс “.shstrtab” и можем в заголовках распарсить имена, которые как раз хранятся в “.shstrtab”.

Получив имена, мы находим нужные нам “.text”, “.symtab”, “.strtab”. И теперь можем парсить symbol table.

Итак, мы знает offset, по которому хранятся symbol table entries. Также мы знаем длину одной строчки (16 байт) и можем распарсить её.

st_bind = st_info >> 4, теперь можно добавить вывод этого bind(см рис.  4)

| Name | Value |
|---|---|
| STB_LOCAL | 0 |
| STB_GLOBAL | 1 |
| STB_WEAK | 2 |
| STB_LOPROC | 13 |
| STB_HIPROC | 15 |

*рис 4. st_bind*

st_type = st_info & 0xf, теперь зная type, можем его вывести(см рис. 5)

| Name | Value |
|---|---|
| STT_NOTYPE | 0 |
| STT_OBJECT | 1 |
| STT_FUNC | 2 |
| STT_SECTION | 3 |
| STT_FILE | 4 |
| STT_LOPROC | 13 |
| STT_HIPROC | 15 |

*рис 5. st_type*

Чтобы получить имя метки в symbol table entry надо проделать те же операции, что и при получении имени секции, то теперь брать имена из “.strtab”, также я чтобы было похоже на то, что выводит readelf -a test_elf, решил сделать так: если st_name = 0 и st_type = “SECTION”, то тогда по индексу находим секцию и записываем её имя.

В Index мы выводим “UNDEF”, если st_shndx = 0, “ABS”, если st_shndx = -15, либо же сам st_shndx в других случаях.

Итак, теперь переходим к парсингу секции “.text”. Мы знаем, что каждая команда имеет длину в 32 бита, также знаем адрес по которому начинается первая команда(sh_addr). Теперь мы знаем инструкцию, и можем её разбить на нужные части(opcode, rd, imm…). По opcode, funct3, funct7 мы понимаем с какой командой мы работаем, и сохраняем её имя.

По opcode также мы можем понять с каким типом операции мы имеем дело и в зависимости от этого у нас будет различный вывод. В функциях J, B типа есть метки, которые нужно получать, для этого берём в symbol table метку которая находится по адресу, который мы получаем из imm. И записываем эту метку, а если такой метки нет внутри таблицы символов, то тогда пишем L0, L1, … в зависимости какая по счету это метка. При команде jal необходимо добавить в вывод по адресу который передаётся по этой функции, метку которая содержится по этому адресу в symtab-e.

В конце сохраняем в файл весь text и symtab согласно формату из тз.

## *4. Результат работы написанной программы на приложенном к заданию файле (дизассемблер и таблицу символов).*

```
.text
00010074   <main>:
   10074:   ff010113      addi    sp, sp, -16
   10078:   00112623        sw    ra, 12(sp)
   1007c:   030000ef       jal    ra, 0x100ac <mmul>
   10080:   00c12083        lw    ra, 12(sp)
   10084:   00000513      addi    a0, zero, 0
   10088:   01010113      addi    sp, sp, 16
   1008c:   00008067      jalr    zero, 0(ra)
   10090:   00000013      addi    zero, zero, 0
   10094:   00100137       lui    sp, 0x100
   10098:   fddff0ef       jal    ra, 0x10074 <main>
   1009c:   00050593      addi    a1, a0, 0
   100a0:   00a00893      addi    a7, zero, 10
   100a4:   0ff0000f     fence    iorw, iorw
   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
   100c8:   fec50e13      addi    t3, a0, -20
   100cc:   000f0313      addi    t1, t5, 0
   100d0:   000f8893      addi    a7, t6, 0
   100d4:   00000813      addi    a6, zero, 0
   100d8:   00088693      addi    a3, a7, 0
   100dc:   000e0793      addi    a5, t3, 0
   100e0:   00000613      addi    a2, zero, 0
   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   UNDEF
[    1] 0x10074              0 SECTION  LOCAL    DEFAULT       1 .text
[    2] 0x11124              0 SECTION  LOCAL    DEFAULT       2 .bss
[    3] 0x0                  0 SECTION  LOCAL    DEFAULT       3 .comment
[    4] 0x0                  0 SECTION  LOCAL    DEFAULT       4
            .riscv.attributes
[    5] 0x0                  0 FILE     LOCAL    DEFAULT     ABS test.c
[    6] 0x11924              0 NOTYPE   GLOBAL   DEFAULT     ABS
            __global_pointer$
[    7] 0x118F4            800 OBJECT   GLOBAL   DEFAULT       2 b
[    8] 0x11124              0 NOTYPE   GLOBAL   DEFAULT       1 __SDATA_BEGIN__
[    9] 0x100AC            120 FUNC     GLOBAL   DEFAULT       1 mmul
[   10] 0x0                  0 NOTYPE   GLOBAL   DEFAULT   UNDEF _start
[   11] 0x11124           1600 OBJECT   GLOBAL   DEFAULT       2 c
[   12] 0x11C14              0 NOTYPE   GLOBAL   DEFAULT       2 __BSS_END__
[   13] 0x11124              0 NOTYPE   GLOBAL   DEFAULT       2 __bss_start
[   14] 0x10074             28 FUNC     GLOBAL   DEFAULT       1 main
[   15] 0x11124              0 NOTYPE   GLOBAL   DEFAULT       1 __DATA_BEGIN__
[   16] 0x11124              0 NOTYPE   GLOBAL   DEFAULT       1 _edata
[   17] 0x11C14              0 NOTYPE   GLOBAL   DEFAULT       2 _end
[   18] 0x11764            400 OBJECT   GLOBAL   DEFAULT       2 a
```

## 5. *Список источников.*


*https://refspecs.linuxfoundation.org/elf/elf.pdf*

*https://riscv.org*

*https://en.wikipedia.org/wiki/RISC-V*

*https://docs.oracle.com/cd/E37838_01/html/E61063/elf-23207.html*

*In-depth: ELF - The Extensible & Linkable Format*

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


## 6. *Листинг кода.*

## *Main.java*

```
import disassembler.ParseElf;

import java.io.*;
import java.nio.charset.StandardCharsets;

public class Main {
   public static void main(String[] args) throws IOException {

       ParseElf parseElf = new ParseElf(args[1]);

                   try   (BufferedWriter   writer   =   new   BufferedWriter(new
OutputStreamWriter(
               new FileOutputStream(args[2]),
               StandardCharsets.UTF_8
       ))) {
           writer.write(parseElf.parseText().toString());
           writer.newLine();
           writer.write(parseElf.parseSymbolTable().toString());
       } catch (FileNotFoundException e) {
           System.out.println("File not found: " + e.getMessage());
       } catch (UnsupportedEncodingException e) {
           System.out.println("Unsupported encoding: " + e.getMessage());
       } catch (IOException e) {
            System.out.println("Input/output file reading/writing error: " +
e.getMessage());
       }
   }
}
```

## *disassembler.ParseElf.java*

```
package disassembler;

import java.io.IOException;
import java.util.ArrayList;
```

```
import java.util.Map;

public class ParseElf {
   private final ElfFile elfFile;
   protected final SymbolTable symbolTable;

   public ParseElf(String fileName) throws IOException {
       this.elfFile = new ElfFile(fileName);
       symbolTable = parseSymbolTable();
       elfFile.setSymbolTable(symbolTable);
       elfFile.setMapOfFunctions();
   }

   public SymbolTable parseSymbolTable() {
       ArrayList<SymbolTableEntry> symbolTableEntries = new ArrayList<>();
       SectionHeader symTab = elfFile.symTab;
       int offset = symTab.sh_offset.getIntData();
       int startOffset = offset;
       int index = 0;
       while(offset - startOffset < symTab.sh_size.getIntData()) {
           symbolTableEntries.add(parseSymbolTableEntry(offset, index));
           offset += 16;
           index++;
       }
       return new SymbolTable(symbolTableEntries);
   }

   private SymbolTableEntry parseSymbolTableEntry(int offset, int index) {
       Elf32_Word st_name = new Elf32_Word(elfFile.elfFileGetInt(offset));
       offset += 4;
       Elf32_Addr st_value = new Elf32_Addr(elfFile.elfFileGetInt(offset));
       offset += 4;
       Elf32_Word st_size = new Elf32_Word(elfFile.elfFileGetInt(offset));
       offset += 4;
       byte st_info = elfFile.elfFileGet(offset);
       offset += 1;
```

```
        byte st_other = elfFile.elfFileGet(offset);
        offset += 1;
        Elf32_Half st_shndx = new Elf32_Half(elfFile.elfFileGetShort(offset));
             return new SymbolTableEntry(elfFile, index, st_name, st_value,
st_size, st_info, st_other, st_shndx);
    }

    public Text parseText() {
        Text text = elfFile.parseTextSection();
        int cnt = 0;
        for(Map.Entry<Integer, String> entry : elfFile.functions.entrySet()) {
            text.addCommand(
                    (entry.getKey() - elfFile.text.sh_addr.getIntData()) / 4 +
cnt,
                    new Command(entry.getKey(), entry.getValue())
            );
            cnt++;
        }
        return text;
    }
}
```

## *disassembler.ElfFIle.java*

```
package disassembler;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.*;

public class ElfFile {
    private SymbolTable symbolTable;
    private final ByteBuffer elfFile;
```

```
    protected ElfFileHeader elfFileHeader;

        protected final ArrayList<SectionHeader> sectionsHeaders = new
ArrayList<>();

    protected final SectionHeader shStrTab;

    protected final SectionHeader strTab;

    protected final SectionHeader symTab;

    protected final SectionHeader text;

    protected int textIndex;

    private int labelNumber = -1;

    protected final Map<Integer, String> functions = new TreeMap<>();

    public ElfFile(String fileName) throws IOException {

        elfFile = ByteBuffer.wrap(Files.readAllBytes(Path.of(fileName)));

        elfFile.order(ByteOrder.LITTLE_ENDIAN);

        setHeader();

        setSectionsHeaders();

                                                                shStrTab        =
sectionsHeaders.get((elfFileHeader.e_shstrndx).getIntData());

        symTab = findSectionWithName(".symtab");

        text = findSectionWithName(".text");

        strTab = findSectionWithName(".strtab");

    }

    protected void setMapOfFunctions() {

        for(SymbolTableEntry symbolTableEntry : symbolTable.symbolTable) {

                                if(symbolTableEntry.getType().equals("FUNC")   &&
symbolTableEntry.getSt_shndx().getIntData() == textIndex) {

                    functions.put(symbolTableEntry.getSt_value().getIntData(),
symbolTableEntry.getName());

            }

        }

    }

    protected String getFunctionName(int address) {

        if (functions.containsKey(address)) {

            return functions.get(address);

        } else {

            labelNumber++;

            functions.put(address, "L" + labelNumber);

            return "L" + labelNumber;

        }
```

```
    }
    public void setSymbolTable(SymbolTable symbolTable) {
        this.symbolTable = symbolTable;
    }
    private void setHeader() {
        ByteBuffer elfFileHeader = elfFile.duplicate();
        elfFileHeader.order(ByteOrder.LITTLE_ENDIAN);
        this.elfFileHeader = new ElfFileHeader(elfFileHeader);
        //System.out.println(this.elfFileHeader);
    }
    private void setSectionsHeaders() {
        int i = elfFileHeader.e_shoff.getIntData();
        //System.out.println(i);
        for(int j = 0; j < elfFileHeader.e_shnum.getShortData(); j++) {
            SectionHeader header = getNextSectionHeader(i);
            //System.out.println(header);
            sectionsHeaders.add(header);
            i += elfFileHeader.e_shentsize.getShortData();
        }

    }
    private SectionHeader getNextSectionHeader(int i) {
        SectionHeader header;
        ByteBuffer sectionHeader = elfFile.duplicate();
        sectionHeader.order(ByteOrder.LITTLE_ENDIAN);
        sectionHeader.position(i);
        header = new SectionHeader(sectionHeader);
        //System.out.println(header);
        return header;
    }
    private SectionHeader findSectionWithName(String name) {
        for(int i = 0; i < sectionsHeaders.size(); i++) {
            SectionHeader header = sectionsHeaders.get(i);
                        if(getNameByOffset(shStrTab.sh_offset.getIntData()  +
header.sh_name.getIntData()).equals(name)) {
                if (name.equals(".text")) {
```

```
                    textIndex = i;
                }
                return header;
            }
        }
        throw new RuntimeException("header with name '" + name + "' not found");
    }

    protected String getNameByOffset(int offset) {
        char c = (char)elfFile.get(offset);
        StringBuilder ans = new StringBuilder();
        while(c != '\0') {
            ans.append(c);
            offset++;
            c = (char)elfFile.get(offset);
        }
        return ans.toString();
    }

    protected Text parseTextSection() {
        Text parsedText = new Text();
        int offset = text.sh_offset.getIntData();
        int size = text.sh_size.getIntData();
        int end = offset + size;
        while(offset < end) {
            Command command = parseCommand(offset);
            offset += 4;
            parsedText.addCommand(command);
        }
        return parsedText;
    }

    private Command parseCommand(int offset) {
                int address = text.sh_addr.getIntData() + offset - text.sh_offset.getIntData();
        return new Command(this, address, elfFile.getInt(offset));
```

```
    }

    protected int elfFileGetInt(int offset) {
        return elfFile.getInt(offset);
    }

    protected short elfFileGetShort(int offset) {
        return elfFile.getShort(offset);
    }

    protected byte elfFileGet(int offset) {
        return elfFile.get(offset);
    }


}
```

## *disassembler.ElfFileHeader.java*

```
package disassembler;

import java.nio.ByteBuffer;

public class ElfFileHeader {
    private final ByteBuffer elfHeader;
    protected Elf32_Half e_type; // type of file
    protected Elf32_Half e_machine; // required architecture
    protected Elf32_Word e_version; // version is valid or not
    protected Elf32_Addr e_entry; // entry point address
    protected Elf32_Off e_phoff; // start of program headers
    protected Elf32_Off e_shoff; // start of section headers
    protected Elf32_Word e_flags; // flag
    protected Elf32_Half e_ehsize; // size of this header
    protected Elf32_Half e_phentsize; // size of program header
    protected Elf32_Half e_phnum; // number of program headers
    protected Elf32_Half e_shentsize; // size of section header
```

```
protected Elf32_Half e_shnum; // number of section headers
protected Elf32_Half e_shstrndx; // section header string table index

public ElfFileHeader(ByteBuffer elfFile) {
    elfHeader = elfFile;
    setElfHeaderData();
}

public void setElfHeaderData() {
    //skip first 16 bytes for e_ident, it's not interesting in this task
    byte[] bytes = new byte[16];
    elfHeader.get(bytes, 0, 16);
    e_type = new Elf32_Half(elfHeader.getShort());
    e_machine = new Elf32_Half(elfHeader.getShort());
    e_version = new Elf32_Word(elfHeader.getInt());
    e_entry = new Elf32_Addr(elfHeader.getInt());
    e_phoff = new Elf32_Off(elfHeader.getInt());
    e_shoff = new Elf32_Off(elfHeader.getInt());
    e_flags = new Elf32_Word(elfHeader.getInt());
    e_ehsize = new Elf32_Half(elfHeader.getShort());
    e_phentsize = new Elf32_Half(elfHeader.getShort());
    e_phnum = new Elf32_Half(elfHeader.getShort());
    e_shentsize = new Elf32_Half(elfHeader.getShort());
    e_shnum = new Elf32_Half(elfHeader.getShort());
    e_shstrndx = new Elf32_Half(elfHeader.getShort());
}


//toString for debugging out
@Override
public String toString() {
    return "Type: " + e_type + '\n' +
            "Architecture: " + e_machine + '\n' +
            "Version: " + e_version + '\n' +
            "Entry point address: " + e_entry + '\n' +
            "Start of program headers: " + e_phoff + '\n' +
```

```
                "Start of section header: " + e_shoff + '\n' +
                "Flag: " + e_flags + '\n' +
                "Size of this header: " + e_ehsize + '\n' +
                "Size of program header: " + e_phentsize + '\n' +
                "Number of program headers: " + e_phnum + '\n' +
                "Size of section header: " + e_shentsize + '\n' +
                "Number of section headers: " + e_shnum + '\n' +
                "Section header string table index: " + e_shstrndx;
    }
}
```

## *disassembler.AbstractElf32DataTypes.java*

```
package disassembler;

public abstract class AbstractElf32DataTypes {
    protected final Number data;

    public AbstractElf32DataTypes(Number data) {
        this.data = data;
    }

    public int getIntData() {
        return data.intValue();
    }

    public int getShortData() {
        return data.shortValue();
    }

    @Override
    public String toString() {
        return String.valueOf(this.getIntData());
    }

}
```

## *disassembler.Elf32_Addr.java*

```
package disassembler;

public class Elf32_Addr extends AbstractElf32DataTypes {
    public Elf32_Addr(Number data) {
        super(data);
    }
}
```

## *disassembler.Elf32_Half.java*

```
package disassembler;

public class Elf32_Half extends AbstractElf32DataTypes {
    public Elf32_Half(Number data) {
        super(data);
    }
    @Override
    public String toString() {
        return String.valueOf(super.getShortData());
    }
}
```

## *disassembler.Elf32_Off.java*

```
package disassembler;

public class Elf32_Off extends AbstractElf32DataTypes{

    public Elf32_Off(Number data) {
        super(data);
    }
}
```

## *disassembler.Elf32_Word.java*

```
package disassembler;

public class Elf32_Word extends AbstractElf32DataTypes {
   public Elf32_Word(Number data) {
       super(data);
   }

}
```

## *disassembler.SectionHeader.java*

```
package disassembler;

import java.nio.ByteBuffer;

public class SectionHeader {
   protected final Elf32_Word sh_name; // Section name
   protected final Elf32_Word sh_type; // Section type
   protected final Elf32_Word sh_flags; // Section flags
   protected final Elf32_Addr sh_addr; // Address of first byte
   protected final Elf32_Off sh_offset; // Offset
   protected final Elf32_Word sh_size; // Size
   protected final Elf32_Word sh_link; // Link
   protected final Elf32_Word sh_info; // Info
   protected final Elf32_Word sh_addralign; // Addralign
   protected final Elf32_Word sh_entsize; // entry size

   public SectionHeader(ByteBuffer elfSectionHeader) {
       sh_name = new Elf32_Word(elfSectionHeader.getInt());
       sh_type = new Elf32_Word(elfSectionHeader.getInt());
       sh_flags = new Elf32_Word(elfSectionHeader.getInt());
       sh_addr = new Elf32_Addr(elfSectionHeader.getInt());
       sh_offset = new Elf32_Off(elfSectionHeader.getInt());
       sh_size = new Elf32_Word(elfSectionHeader.getInt());
```

```
        sh_link = new Elf32_Word(elfSectionHeader.getInt());
        sh_info = new Elf32_Word(elfSectionHeader.getInt());
        sh_addralign = new Elf32_Word(elfSectionHeader.getInt());
        sh_entsize = new Elf32_Word(elfSectionHeader.getInt());
    }

    @Override
    public String toString() {
        return "Name: " + sh_name + '\n' +
                "Type: " + sh_type + '\n' +
                "Flags: " + sh_flags + '\n' +
                "Entry point address: " + sh_addr + '\n' +
                "Offset: " + sh_offset + '\n' +
                "Size: " + sh_size + '\n' +
                "Link: " + sh_link + '\n' +
                "Info: " + sh_info + '\n' +
                "Address align: " + sh_addralign + '\n' +
                "Entry size: " + sh_entsize + '\n' + '\n';
    }

}
```

## *disassembler.SymbolTable.java*

```
package disassembler;

import java.util.ArrayList;

public class SymbolTable {
    protected final ArrayList<SymbolTableEntry> symbolTable;
    public SymbolTable(ArrayList<SymbolTableEntry> symbolTable) {
        this.symbolTable = symbolTable;
    }

    @Override
    public String toString() {
```

```
        StringBuilder sb = new StringBuilder();
        sb.append(".symtab\nSymbol Value              Size Type    Bind    Vis
Index Name\n");
        for (SymbolTableEntry symbolTableEntry : symbolTable) {
            sb.append(symbolTableEntry.toString());
        }
        return sb.toString();
    }
}
```

## *disassembler.SymbolTableEntry.java*

```
package disassembler;


public class SymbolTableEntry {
    private final ElfFile elfFile;
    private final int st_symndx; //index of symbol
    private final Elf32_Word st_name;
    private final Elf32_Addr st_value;
    private final Elf32_Word st_size;
    private final int st_info;
    private final int st_other;
    private final Elf32_Half st_shndx;

    protected Elf32_Addr getSt_value() {
        return st_value;
    }


    protected Elf32_Half getSt_shndx() {
        return st_shndx;
    }


      public SymbolTableEntry(ElfFile elfFile, int st_symndx, Elf32_Word
st_name, Elf32_Addr st_value, Elf32_Word st_size, byte st_info, byte
st_other, Elf32_Half st_shndx) {
        this.elfFile = elfFile;
        this.st_symndx = st_symndx;
```

```
        this.st_name = st_name;
        this.st_value = st_value;
        this.st_size = st_size;
        this.st_info = st_info;
        this.st_other = st_other;
        this.st_shndx = st_shndx;
    }

    @Override
    public String toString() {
        return String.format("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                st_symndx, st_value.getIntData(), st_size.getIntData(),
                getType(), getBind(), getVis(), getIndex(), getName()
        );
    }

    private String getIndex() {
        return switch (st_shndx.getIntData()) {
            case 0 -> "UNDEF";
            case -15 -> "ABS";
            default -> st_shndx.toString();
        };
    }
    private String getBind() {
        return switch (st_info >> 4) {
            case 0 -> "LOCAL";
            case 1 -> "GLOBAL";
            case 2 -> "WEAK";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
             default -> throw new IllegalStateException("Unexpected value: " +
st_info);
        };
    }

    private String getVis() {
```

```
        return switch (st_other) {
            case 0 -> "DEFAULT";
            case 1 -> "INTERNAL";
            case 2 -> "HIDDEN";
            case 3 -> "PROTECTED";
             default -> throw new IllegalStateException("Unexpected value: " +
st_other);
        };
    }


    protected String getName() {
        if (getType().equals("SECTION")) {
                                                                        return
elfFile.getNameByOffset(elfFile.sectionsHeaders.get(st_shndx.getIntData()).sh
_name.getIntData() + elfFile.shStrTab.sh_offset.getIntData());
        } else {
                                                                        return
elfFile.getNameByOffset(elfFile.strTab.sh_offset.getIntData()                +
st_name.getIntData());
        }
    }


    protected String getType() {
        return switch (st_info & 0xf) {
            case 0 -> "NOTYPE";
            case 1 -> "OBJECT";
            case 2 -> "FUNC";
            case 3 -> "SECTION";
            case 4 -> "FILE";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
             default -> throw new IllegalStateException("Unexpected value: " +
st_info);
        };
    }
}
```

## *disassembler.Text.java*

```java
package disassembler;

import java.util.LinkedList;
import java.util.List;

public class Text {
   private final List<Command> commands;

   protected Text() {
       this.commands = new LinkedList<>();
   }

   protected void addCommand(Command command) {
       commands.add(command);
   }

   protected void addCommand(int pos, Command command) {
       commands.add(pos, command);
   }

   @Override
   public String toString() {
       StringBuilder sb = new StringBuilder(".text\n");
       for(Command command : commands) {
           sb.append(command.toString());
       }
       return sb.toString();
   }
}
```

## *disassembler.Command.java*

```java
package disassembler;

public class Command {
   protected final int address;
```

```
private int instruction;
private Registr regSource1, regSource2;
private Registr regDest;
private int imm31_12, imm11_0, imm11_5, imm4_0;
private int opcode;
private Function function;
private String labelName;
protected ElfFile elfFile;
private final boolean isLabel;
public Command(ElfFile elfFile, int address, int instruction) {
    this.elfFile = elfFile;
    this.address = address;
    this.instruction = instruction;
    this.isLabel = false;
    String line = Integer.toBinaryString(instruction);
    line = "0".repeat(32 - line.length()) + line;
    String opcodeStr = line.substring(32 - 7, 32);
    String rd = line.substring(32 - 12, 32 - 7);
    String funct3 = line.substring(32 - 15, 32 - 12);
    String rs1 = line.substring(32 - 20, 32 - 15);
    String rs2 = line.substring(32 - 25, 32 - 20);
    String funct7 = line.substring(0, 32 - 25);

    opcode = Integer.parseInt(opcodeStr, 2);
    regDest = new Registr(Integer.parseInt(rd, 2));
    regSource1 = new Registr(Integer.parseInt(rs1, 2));
    regSource2 = new Registr(Integer.parseInt(rs2, 2));

    instruction >>= 7;
    imm4_0 = instruction & ((1 << 5) - 1);
    instruction >>= 5;
    imm31_12 = instruction;
    instruction >>= 8;
    imm11_0 = instruction;
    instruction >>= 5;
    imm11_5 = instruction;
```

```
            function = new Function(opcode, Integer.parseInt(funct3, 2),
Integer.parseInt(funct7, 2));
    }
    public Command(int address, String labelName) {
        this.isLabel = true;
        this.address = address;
        this.labelName = labelName;
    }

    @Override
    public String toString() {
        if (isLabel) {
            return String.format("%08x   <%s>:\n", address, labelName);
        }
        return switch (opcode) {
            case 19 -> String.format("   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                    address, instruction, function, regDest, regSource1,
imm11_0);
            case 51 -> String.format("   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                    address, instruction, function, regDest, regSource1,
regSource2);
            case 55, 23 -> String.format("   %05x:\t%08x\t%7s\t%s, 0x%s\n",
                                address, instruction, function, regDest,
Integer.toHexString(imm31_12));
            case 3, 103 -> String.format("   %05x:\t%08x\t%7s\t%s, %s(%s)\n",
                    address, instruction, function, regDest, imm11_0,
regSource1);
            case 35 -> String.format("   %05x:\t%08x\t%7s\t%s, %s(%s)\n",
                    address, instruction, function, regSource2, (imm11_5 << 5)
+ imm4_0, regSource1);
            case 111 -> String.format("   %05x:\t%08x\t%7s\t%s, %s\n",
                    address, instruction, function, regDest, getLabelJal());
            case 99 -> String.format("   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                    address, instruction, function, regSource1, regSource2,
getLabelBranch());
            case 115 -> switch (regSource2.getReg()) {
                case 0 -> String.format("   %05x:\t%08x\t%7s\n",
                        address, instruction, "ecall");
                case 1 -> String.format("   %05x:\t%08x\t%7s\n",
```

```
                        address, instruction, "ebreak");
                default -> throw new IllegalStateException("Unexpected value:
" + regSource2.getReg());
            };
            case 15 -> String.format("   %05x:\t%08x\t%7s\t%s, %s\n",
                    address, instruction, function, "iorw", "iorw");
            default -> String.format("   %05x:\t%08x\t%7s\n",
                    address, instruction, function);
        };
    }
    private String getLabelBranch() {
        int line = imm11_5;
        int imm10_5 = line & ((1 << 6) - 1);
        line >>= 6;
        int imm12 = line & 1;
        line = imm4_0;
        int imm11 = line & 1;
        line >>= 1;
        int imm4_1 = line & ((1 << 4) - 1);
        int offset = imm4_1 + (imm10_5 << 4) + (imm11 << 10) + (imm12 << 11);
        offset <<= 1;
        if(imm12 == 1) {
            offset |= -(1 << 12);
        }
        int address = this.address + offset;

                        return   String.format("0x%05x   <%s>",   address,
elfFile.getFunctionName(address));
    }
    private String getLabelJal() {
        int line = imm31_12;
        int imm19_12 = line & ((1 << 8) - 1);
        line >>= 8;
        int imm11 = line & 1;
        line >>= 1;
        int imm10_1 = line & ((1 << 10) - 1);
        line >>= 10;
```

```
        int imm20 = line & 1;
         int offset = imm10_1 + (imm11 << 10) + (imm19_12 << 11) + (imm20 <<
19);
        offset <<= 1;
        if(imm20 == 1) {
            offset |= -(1 << 21);
        }
        int address = this.address + offset;

                            return    String.format("0x%05x    <%s>",    address,
elfFile.getFunctionName(address));
    }
}
```

## *disassembler.Registr.java*

```
package disassembler;

public class Registr {
    private final int reg;
    public Registr(int reg) {
        this.reg = reg;
    }

    protected int getReg(){
        return reg;
    }
    private String parseReg() {
        return switch (reg) {
            case 0 -> "zero";
            case 1 -> "ra";
            case 2 -> "sp";
            case 3 -> "gp";
            case 4 -> "tp";
            case 5 -> "t0";
            case 6 -> "t1";
            case 7 -> "t2";
```

```
            case 8 -> "s0";
            case 9 -> "s1";
            case 10 -> "a0";
            case 11 -> "a1";
            case 12 -> "a2";
            case 13 -> "a3";
            case 14 -> "a4";
            case 15 -> "a5";
            case 16 -> "a6";
            case 17 -> "a7";
            case 18 -> "s2";
            case 19 -> "s3";
            case 20 -> "s4";
            case 21 -> "s5";
            case 22 -> "s6";
            case 23 -> "s7";
            case 24 -> "s8";
            case 25 -> "s9";
            case 26 -> "s10";
            case 27 -> "s11";
            case 28 -> "t3";
            case 29 -> "t4";
            case 30 -> "t5";
            case 31 -> "t6";
             default -> throw new IllegalStateException("Unexpected value: " +
reg);
        };
    }

    @Override
    public String toString() {
        return parseReg();
    }
}
```

## *disassembler.Function.java*

```
package disassembler;

public class Function {
   private final int opcode;
   private final int funct3;
   private final int funct7;
   public Function(int opcode, int funct3, int funct7) {
       this.opcode = opcode;
       this.funct3 = funct3;
       this.funct7 = funct7;
   }

   private String getFunctionName() {
       return switch (opcode) {
           case 19 -> switch (funct3) {
               case 0 -> "addi";
               case 2 -> "slti";
               case 3 -> "sltiu";
               case 4 -> "xori";
               case 6 -> "ori";
               case 7 -> "andi";
               case 1 -> "slli";
               case 5 -> switch (funct7) {
                   case 0 -> "srli";
                   case 32 -> "srai";
                   default -> "unknown_instruction";
               };
               default -> "unknown_instruction";
           };
           case 51 -> switch (funct3) {
               case 0 -> switch (funct7) {
                   case 0 -> "add";
                   case 1 -> "mul";
                   case 32 -> "sub";
                   default -> "unknown_instruction";
               };
```

```
case 1 -> switch (funct7) {
    case 0 -> "sll";
    case 1 -> "mulh";
    default -> "unknown_instruction";
};
case 2 -> switch (funct7) {
    case 0 -> "slt";
    case 1 -> "mulhsu";
    default -> "unknown_instruction";
};
case 3 -> switch (funct7) {
    case 0 -> "sltu";
    case 1 -> "mulhu";
    default -> "unknown_instruction";
};
case 4 -> switch (funct7) {
    case 0 -> "xor";
    case 1 -> "div";
    default -> "unknown_instruction";
};
case 5 -> switch (funct7) {
    case 0 -> "srl";
    case 1 -> "divu";
    case 32 -> "sra";
    default -> "unknown_instruction";
};
case 6 -> switch (funct7) {
    case 0 -> "or";
    case 1 -> "rem";
    default -> "unknown_instruction";
};
case 7 -> switch (funct7) {
    case 0 -> "and";
    case 1 -> "remu";
    default -> "unknown_instruction";
};
```

```
            default -> "unknown_instruction";
        };
        case 55 -> "lui";
        case 23 -> "auipc";
        case 3 -> switch (funct3) {
            case 0 -> "lb";
            case 1 -> "lh";
            case 2 -> "lw";
            case 4 -> "lbu";
            case 5 -> "lhu";
            default -> "unknown_instruction";
        };
        case 103 -> "jalr";
        case 35 -> switch (funct3) {
            case 0 -> "sb";
            case 1 -> "sh";
            case 2 -> "sw";
            default -> "unknown_instruction";
        };
        case 111 -> "jal";
        case 99 -> switch (funct3) {
            case 0 -> "beq";
            case 1 -> "bne";
            case 4 -> "blt";
            case 5 -> "bge";
            case 6 -> "bltu";
            case 7 -> "bgeu";
            default -> "unknown_instruction";
        };
        case 15 -> "fence";
        default -> "unknown_instruction";
    };
}

@Override
public String toString() {
```

```
        return getFunctionName();
    }
}
```