Лабораторная работа №3	M3139	2023
ISA	Мутаева Олеся Богдановна	

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

Инструментарий и требования к работе: работа выполнена на Java (OpenJDK 17.0.4.1).

Описание

Изучить систему кодирования команд RISC-V (набор RV32I, RV32M), изучить структуру elf файла, написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Описание системы кодирования команд RISC-V.

RISC-V – открытая и свободная ISA с возможностью расширения, основанная на концепции RISC. Набор команд RISC-V состоит из базовой спецификации и нескольких расширений. Некоторые наборы инструкций:

- RV32I (I – Integer) – базовая спецификация, содержит 40 команд. Включает в себя команды для ветвлений, работы с памятью, служебных операций, простых целочисленных арифметических и логических операций.
- RV64I – базовая спецификация для 64-разрядных систем. Содержит 15 дополнительных команд, “дополняет” RV32I.
- RV128I – базовая спецификация для 128-разрядных систем с тем же назначением, что и RV32I/RV64I. На данный момент набор ее инструкций официально не является конечным (not ratified).
- RV32E (E – Embedded) – базовая спецификация для встраиваемых систем с набором регистров общего назначения, сокращенным до первых 16. Содержит 40 команд, аналогичных командам RV32I.
- M (Multiply) – расширение для целочисленных операций умножения и деления. Содержит 8 инструкций для 32-битных систем, 5 дополнительных инструкций для 64-битных систем.
- A (Atomic) – расширение для атомарных операций (операция гарантированно либо выполняется целиком, либо не выполняется вовсе). Содержит 11 и 22 инструкции для 32-битных и 64-битных систем соответственно.
- C (Compressed) – расширение, содержащее инструкции стандартной спецификации, но они имеют сокращенную

длину 16 бит, в то время как в базовой спецификации и остальных расширениях инструкция кодируется 32 битами.

- F (float) – расширение для арифметических операций над числами с плавающей точкой одинарной точности (Single-Precision Floating-Point). Содержит 26 и 30 инструкции для 32-битных и 64-битных систем соответственно.
- D (double) – расширение с таким же смыслом как и F, но для чисел двойной точности (Double-Precision Floating-Point). Содержит 26 и 32 инструкции для 32-битных и 64-битных систем соответственно.
- Q (quad) – расширение с таким же смыслом как и F, но для чисел четверной точности (Quad-Precision Floating-Point). Содержит 28 и 32 инструкции для 32-битных и 64-битных систем соответственно.
- L – расширение для арифметических операций над десятичными числами с плавающей запятой (Decimal Floating-Point), на данный момент not ratified.
- Zicsr – расширение для работы с регистрами контроля и статуса (Control/Status Registers).
- Zifencei – расширение для синхронизации потоков команд и данных (Instruction-Fetch Fence).
- B – битовые операции.
- J – двоичная трансляция.
- T – транзакционная память.
- P – Короткие SIMD-операции.
- V – Векторные расширения (Vector Operations).

Полный список расширений есть в спецификации RISC-V.

Регистры общего назначения

В RISC-V используются 32 регистра общего назначения (16 в встраиваемых системах). Регистры пронумерованы от 0 до 31, кодируются 5 битами, названия регистров выглядят как x+номер регистра. Соглашение об использовании регистров ABI:

Регистр	Название	Описание
x0	zero	Всегда ноль
x1	ra	return address – адрес возврата
x2	sp	stack pointer – указатель стека
x3	gp	global pointer – глобальный указатель
x4	tp	thread pointer – указатель потока
x5-x7	t0-t2	temporary – временные регистры
x8-x9	s0-s1	saved registers – сохраняемые регистры
x10-x17	a0-a7	argument – регистры аргументов
x18-x27	s2-s11	saved registers – сохраняемые регистры
x28-x31	t3-t6	temporary – временные регистры

Таблица №1 – Соглашение об использовании регистров.

Типы инструкций и их структура

В RV32I и RV32M размер инструкции – 4 байта. Для определения инструкции рассматривается ее побитовое представление. Младшие 7 бит инструкции представляют собой opcode – код операции. Виды команд RV32I/RV32M:

- R (Register). Инструкции этого типа работают только с регистрами, содержат указатели на регистры для чтения значений из них (rs1, rs2), указатель на регистр для записи в него (rd), funct7 и funct3 для определения операции.
- I (Immediate). Инструкции этого типа похожи на R тип, но вместо второго регистра для чтения (rs2) и funct7 записана 12-битная константа со знаком imm – временное значение.

- S (Store). Инструкции этого типа предназначены для записи в память, используют два регистра для чтения из них (rs1, rs2), 12-битная константа показывает смещение (offset) необходимого адреса в памяти.
- B (Branch). Инструкции типа B предназначены для условных переходов. Выглядят как S-тип, но immediate (offset) записывается иначе (см. рис. №1).
- U (Upper immediate). Инструкции типа U предназначены для записи верхних 20 бит в регистр rd.
- J (Jump). Предназначены для прыжка в другое место программы, адрес прыжка определяется offset-ом (20-битная константа immediate).

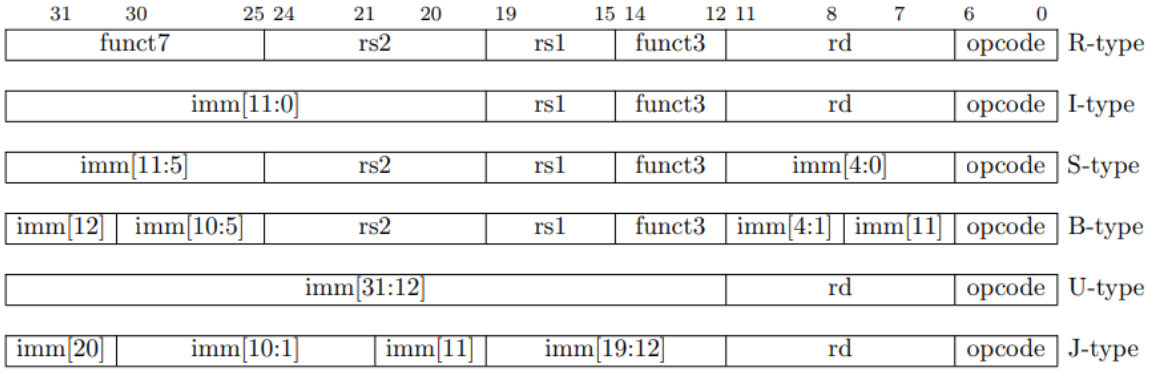

Рисунок №1 – виды команд в битовом представлении.

Подробное описание каждой команды есть в спецификации RISC-V.

Структура elf файла

ELF (Executable and Linking format) – расширяемый и кроссплатформенный формат исполняемых двоичных файлов, объектного кода, библиотек и дампов памяти. Распространен в Unix и Unix-подобных системах. Подробное описание структуры elf файла есть в спецификации.

Заголовок файла

Elf Header – заголовок файла – находится в начале файла и имеет фиксированный размер 52 байта в 32-разрядных системах и 64 байта в 64-разрядных системах (+12 байтов из-за наличия в заголовке 3 указателей размером на 4 байта больше, чем в 32-битных системах). Заголовок файла

содержит основную информацию о структуре файла и его основные характеристики. Некоторые поля заголовка elf файла:

- Сигнатура (magic number) – первые 4 байта, в ASCII их значение образует строку `.ELF` (. – непечатаемый символ)
- `EI_CLASS` – байт с индексом 4, определяет битность (1 – 32, 2 – 64). В работе используется 32-битный вариант.
- `EI_DATA` – байт с индексом 5, определяет порядок байтов (1 – little endian, 2 – big endian). RISC-V использует только little endian.
- `EI_VERSION` – байт с индексом 6, версия elf. На данный момент всегда равен 1.
- `e_machine` – байты с индексами 18, 19, определяют архитектуру, для которой предназначен этот файл. В случае RISC-V значение равно `0xF3`.
- `e_shstrndx` – байты с индексами 50, 51, определяют индекс секции имен в таблице заголовков секций.

Таблица заголовков программы

Program header table содержит заголовки, каждый из которых описывает отдельный сегмент программы и его атрибуты либо другую информацию, необходимую операционной системе для подготовки программы к исполнению. Смещение расположения таблицы относительно начала файла (offset) указано в поле `e_phoff` заголовка файла, сама таблица состоит из `e_phnum` заголовков, каждый размером `e_phentsize` (в 32-битном варианте размер равен 32 байтам).

Таблица заголовков секций

Section header table содержит атрибуты секций файла и необходима только компоновщику. Смещение расположения таблицы относительно начала файла указано в поле `e_shoff` заголовка файла, таблица описывает `e_shnum` секций, каждый заголовок секции имеет размер `e_shentsize` (в 32-битном варианте 40 байт). Некоторые поля заголовка секции:

- `sh_name` (4 байта) – offset названия данной секции относительно секции имен (`.shstrtab`)

- `sh_type` (4 байта) – тип заголовка. В данной работе важная секция, у которой это поле равно `SHT_SYMTAB` (2 – таблица символов). Такая секция должна быть ровно одна в файле.
- `sh_offset` (4 байта) – смещение данной секции относительно начала файла.
- `sh_size` (4 байта) – размер данной секции в файле.

Секции

В данной работе исследуются следующие секции:

`.text`

Содержит исполняемый код, который будет упакован в сегмент с правами на чтение и исполнение. Здесь находятся инструкции, которые необходимо дизассемблировать.

`.symtab`

Таблица символов, содержит информацию о символах (функциях и переменных), такую как расположение названия в таблице строк, размер, значение символа. Эта секция хранит адреса меток, используемых в коде.

`.strtab`

Таблица строк. В этой секции хранятся названия меток, используемых в коде.

`.shstrtab`

Таблица названий секций.

Дизассемблер

Дизассемблер – программа-транслятор, с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера. В данной работе программа предназначена для трансляции содержимого elf файла в ассемблер RISC-V с наборами команд RV32I и RV32M. Выходной файл содержит дизассемблированный код секции .text и содержимое таблицы символов .symtab в человекочитаемом виде.

Описание работы кода

1. Чтение входного файла, преобразование его содержимого в массив байтов.

2. Чтение заголовка elf файла с проверкой корректности файла: правильно указаны формат файла, архитектура, порядок байтов.
3. Нахождение расположения таблицы заголовков секций в заголовке файла.
4. Нахождение расположения секций .text и .symtab в таблице заголовков секций.
5. Чтение секции .symtab, сохранение адресов меток из таблицы символов.
6. Чтение секции .text: дизассемблирование инструкций и расстановка меток, отсутствующих в таблице символов, в множество неизвестных адресов. Сохранение дизассемблированных инструкций в исходном порядке.
7. Проход по дизассемблированным инструкциям, запись в выходной файл с расстановкой всех меток.

Код написан на языке Java (openjdk 17.0.4.1). Классы для дизассемблирования находятся в пакете elf, главный класс имеет название RVDisassembler, принимает два аргумента – название входного файла, название выходного файла.

Результат работы программы

```
.text
00010074   <main>:
   10074:    ff010113        addi      sp, sp, -16
   10078:    00112623          sw      ra, 12(sp)
   1007c:    030000ef         jal      ra, 0x100ac <mmul>
   10080:    00c12083          lw      ra, 12(sp)
   10084:    00000513        addi      a0, zero, 0
   10088:    01010113        addi      sp, sp, 16
   1008c:    00008067        jalr      zero, 0(ra)
   10090:    00000013        addi      zero, zero, 0
   10094:    00100137         lui      sp, 256
   10098:    fddff0ef         jal      ra, 0x10074 <main>
   1009c:    00050593        addi      a1, a0, 0
   100a0:    00a00893        addi      a7, zero, 10
   100a4:    0ff0000f       fence
   100a8:    00000073       ecall
000100ac   <mmul>:
```

```
   100ac:    00011f37         lui      t5, 17
   100b0:    124f0513        addi      a0, t5, 292
   100b4:    65450513        addi      a0, a0, 1620
   100b8:    124f0f13        addi      t5, t5, 292
   100bc:    e4018293        addi      t0, gp, -448
   100c0:    fd018f93        addi      t6, gp, -48
   100c4:    02800e93        addi      t4, zero, 40
000100c8   <L2>:
   100c8:    fec50e13        addi      t3, a0, -20
   100cc:    000f0313        addi      t1, t5, 0
   100d0:    000f8893        addi      a7, t6, 0
   100d4:    00000813        addi      a6, zero, 0
000100d8   <L1>:
   100d8:    00088693        addi      a3, a7, 0
   100dc:    000e0793        addi      a5, t3, 0
   100e0:    00000613        addi      a2, zero, 0
000100e4   <L0>:
   100e4:    00078703          lb      a4, 0(a5)
   100e8:    00069583          lh      a1, 0(a3)
   100ec:    00178793        addi      a5, a5, 1
   100f0:    02868693        addi      a3, a3, 40
   100f4:    02b70733         mul      a4, a4, a1
   100f8:    00e60633         add      a2, a2, a4
   100fc:    fea794e3         bne      a5, a0, 0x100e4 <L0>
   10100:    00c32023          sw      a2, 0(t1)
   10104:    00280813        addi      a6, a6, 2
   10108:    00430313        addi      t1, t1, 4
   1010c:    00288893        addi      a7, a7, 2
   10110:    fdd814e3         bne      a6, t4, 0x100d8 <L1>
   10114:    050f0f13        addi      t5, t5, 80
   10118:    01478513        addi      a0, a5, 20
   1011c:    fa5f16e3         bne      t5, t0, 0x100c8 <L2>
   10120:    00008067        jalr      zero, 0(ra)

.symtab
```

```
Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x11124               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x0                   0 SECTION  LOCAL    DEFAULT       3
[   4] 0x0                   0 SECTION  LOCAL    DEFAULT       4
[   5] 0x0                   0 FILE     LOCAL    DEFAULT     ABS test.c
[   6] 0x11924               0 NOTYPE   GLOBAL   DEFAULT     ABS __global_pointer$
[   7] 0x118f4             800 OBJECT   GLOBAL   DEFAULT       2 b
[   8] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1 __SDATA_BEGIN__
[   9] 0x100ac             120 FUNC     GLOBAL   DEFAULT       1 mmul
[  10] 0x0                   0 NOTYPE   GLOBAL   DEFAULT   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
```

Листинг кода

RVDisassembler.java

```java
import elf.ElfFile;


public class RVDisassembler {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Enter 2 arguments: input file name (elf) and output file name");
        }
```

```
        ElfFile elf = new ElfFile(args[0]);
        elf.parse();
        elf.write(args[1]);
    }
}
```

Labels.java

```
package elf;

import java.util.Map;
import java.util.TreeMap;

public class Labels {
    private Map<Integer, String> labels = new TreeMap<>();

    public void add(int adr, String name) {
        labels.put(adr, name);
    }

    public boolean checkLabel(int adr) {
        return labels.containsKey(adr);
    }

    public String getLabel(int adr) {
        return labels.get(adr);
    }
}
```

Instruction.java

```
package elf;


public class Instruction {

    private int addr;

    private int instr;

    private String name;

    private String arg1;

    private String arg2;

    private String arg3;


    public Instruction(int addr, int instr, String name, String
arg1, String arg2, String arg3) {

        this.addr = addr;

        this.instr = instr;

        this.name = name;

        this.arg1 = arg1;

        this.arg2 = arg2;

        this.arg3 = arg3;

    }


    public int getAddr() {

        return addr;

    }


    public String toString() {


        String sAdr = Integer.toHexString(addr);

        if (sAdr.length() < 5) {

            sAdr = String.format("%0" + (5 - sAdr.length()) + "d%s",
0, sAdr);
```

```
        }
        String sInstr = Integer.toHexString(instr);
        if (sInstr.length() < 8) {
            sInstr = String.format("%0" + (8 - sInstr.length()) +
"d%s", 0, sInstr);
        }
        switch (this.name) {
            case "jalr", "lb", "lh", "lw", "lbu", "lhu", "sb", "sh",
"sw":
                return String.format("   %s:\t%s\t%7s\t%s,
%s(%s)\n", sAdr, sInstr, name, arg1, arg2, arg3);
        }
        if (this.arg3 != null) {
            return String.format("   %s:\t%s\t%7s\t%s, %s, %s\n",
sAdr, sInstr, name, arg1, arg2, arg3);
        }
        if (this.arg2 != null) {
            return String.format("   %s:\t%s\t%7s\t%s, %s\n", sAdr,
sInstr, name, arg1, arg2);
        }
        return String.format("   %s:\t%s\t%7s\n", sAdr, sInstr,
name);
    }
}
```

SymbolTable.java

```
package elf;

import java.io.BufferedWriter;
import java.io.IOException;
import java.nio.*;
import java.util.*;
```

```
public class SymbolTable {

    public static final int STB_LOCAL = 0;
    public static final int STB_GLOBAL = 1;
    public static final int STB_WEAK = 2;
    public static final int STB_LOOS = 10;
    public static final int STB_HIOS = 12;
    public static final int STB_LOPROC = 13;
    public static final int STB_HIPROC = 15;

    public static final int STV_DEFAULT = 0;
    public static final int STV_INTERNAL = 1;
    public static final int STV_HIDDEN = 2;
    public static final int STV_PROTECTED = 3;
    public static final int STV_EXPORTED = 4;
    public static final int STV_SINGLETON = 5;
    public static final int STV_ELIMINATE = 6;

    public static final short SHN_UNDEF = 0;
    public static final short SHN_LORESERVE = (short) 0xff00;
    public static final short SHN_LOPROC = (short) 0xff00;
    public static final short SHN_HIPROC = (short) 0xff1f;
    public static final short SHN_ABS = (short) 0xfff1;
    public static final short SHN_COMMON = (short) 0xfff2;
    public static final short SHN_XINDEX = (short) 0xffff;
    public static final short SHN_HIRESERVE = (short) 0xffff;
    public static final short SHN_LOOS = (short) 0xff20;
    public static final short SHN_HIOS = (short) 0xff3f;
```

```
    public static final int STT_NOTYPE = 0;
    public static final int STT_OBJECT = 1;
    public static final int STT_FUNC = 2;
    public static final int STT_SECTION = 3;
    public static final int STT_FILE = 4;
    public static final int STT_COMMON = 5;
    public static final int STT_TLS = 6;
    public static final int STT_LOOS = 10;
    public static final int STT_HIOS = 12;
    public static final int STT_LOPROC = 13;
    public static final int STT_HIPROC = 15;

    static class Symbol extends SymbolTable {
        private int symbol;
        private int value;
        private int size;
        private int type;
        private int bind;
        private int vis;
        private short index;
        private String name;

        public Symbol(int symbol, int value, int size, int type, int
bind, int vis, short index, String name) {
            this.symbol = symbol;
            this.value = value;
            this.size = size;
            this.type = type;
            this.bind = bind;
            this.vis = vis;
```

```
        this.index = index;
        this.name = name;
    }

    public String typeToString() {
        switch (this.type) {
            case (STT_NOTYPE):
                return "NOTYPE";
            case (STT_OBJECT):
                return "OBJECT";
            case (STT_FUNC):
                return "FUNC";
            case (STT_SECTION):
                return "SECTION";
            case (STT_FILE):
                return "FILE";
            case (STT_COMMON):
                return "COMMON";
            case (STT_TLS):
                return "TLS";
            case (STT_LOOS):
                return "LOOS";
            case (STT_HIOS):
                return "HIOS";
            case (STT_LOPROC):
                return "LOPROC";
            case (STT_HIPROC):
                return "HIPROC";
            default:
                return "UNKNOWN";
```

```
    }
}

public String bindToString() {
    switch (this.bind) {
        case (STB_LOCAL):
            return "LOCAL";
        case (STB_GLOBAL):
            return "GLOBAL";
        case (STB_WEAK):
            return "WEAK";
        case (STB_LOOS):
            return "LOOS";
        case (STB_HIOS):
            return "HIOS";
        case (STB_LOPROC):
            return "LOPROC";
        case (STB_HIPROC):
            return "HIPROC";
        default:
            return "UNKNOWN";
    }
}

public String visToString() {
    switch (this.vis) {
        case STV_HIDDEN:
            return "HIDDEN";
        case STV_DEFAULT:
            return "DEFAULT";
```

```
            case STV_INTERNAL:
                return "INTERNAL";
            case STV_PROTECTED:
                return "PROTECTED";
            case STV_EXPORTED:
                return "EXPORTED";
            case STV_SINGLETON:
                return "SINGLETON";
            case STV_ELIMINATE:
                return "ELIMINATE";
            default:
                return "UNKNOWN";
        }
    }

    public String indexToString() {
        switch (this.index) {
            case SHN_UNDEF:
                return "UNDEF";
            case SHN_LOPROC:
                return "LOPROC";
            case SHN_HIPROC:
                return "HIPROC";
            case SHN_ABS:
                return "ABS";
            case SHN_COMMON:
                return "COMMON";
            case SHN_XINDEX:
                return "XINDEX";
            case SHN_LOOS:
```

```
                    return "LOOS";
                case SHN_HIOS:
                    return "HIOS";
                default:
                    return Integer.toString((int) index & (0xffff));
            }
        }

        public String toString() {
             return String.format("[%4d] 0x%-15x %5d %-8s %-8s %-8s
%6s %s\n", symbol, value, size,
                      typeToString(), bindToString(), visToString(),
indexToString(), name);
        }

    }

    private List<Symbol> symtab = new ArrayList<Symbol>();

    public void add(Symbol e) {
        symtab.add(e);
    }

    public void write(BufferedWriter writer) throws IOException {
        writer.write("\n.symtab\n");
         writer.write(String.format("%s %-15s %7s %-8s %-8s %-8s %6s
%s\n",
                "Symbol", "Value", "Size", "Type",
                "Bind", "Vis", "Index", "Name"));
        for (Symbol i : symtab) {
            writer.write(i.toString());
```

```
        }
    }

    public Labels toLabels() {
        Labels labels = new Labels();
        for (Symbol symbol : symtab) {
            if (symbol.type == STT_FUNC) {
                labels.add(symbol.value, symbol.name);
            }
        }
        return labels;
    }
}
```

ElfFile.java

```
package elf;

import java.io.*;
import java.nio.*;
import java.nio.file.*;
import java.util.*;

public class ElfFile {

    public static final int EI_MAG_ELF = 0x464c457f;
    public static final byte EI_CLASS_32 = 1;
    public static final byte EI_DATA_LE = 1;
    public static final short E_MACHINE_RISCV = 0xf3;
```

```java
public static final int SHT_SYMTAB = 0x02;
public static final int SHT_STRTAB = 0x03;

List<Instruction> text = new ArrayList<>();
SymbolTable symtab = new SymbolTable();
Labels labels;

private int textOffset = -1;
private int symtabOffset = -1;
private int namesOffset = -1;
private int strtabOffset = -1;

private int textSize = -1;
private int symtabSize = -1;

private int textAddr = -1;

private ByteBuffer bytes;
private BufferedWriter out;
private int bytesRead;

private int EI_MAG;
private byte EI_CLASS;
private byte EI_DATA;
private short EI_VERSION;
private short e_type;
private short e_machine;
private int e_version;
private int e_entry;
private int e_phoff;
```

```
private int e_shoff;
private short e_ehsize;
private short e_phentsize;
private short e_phnum;
private short e_shentsize;
private short e_shnum;
private short e_shstrndx;

private int unknownAddr = 0;

private void parseHeader() {
    if (bytesRead < 54) {
        ElfError("Only " + bytesRead + " bytes in file");
    }

    this.EI_MAG = bytes.getInt(0);
    this.EI_CLASS = bytes.get(0x04);
    this.EI_DATA = bytes.get(0x05);
    this.EI_VERSION = bytes.getShort(0x06);
    this.e_type = bytes.getShort(0x10);
    this.e_machine = bytes.getShort(0x12);
    this.e_version = bytes.getInt(0x14);
    this.e_entry = bytes.getInt(0x18);
    this.e_phoff = bytes.getInt(0x1c);
    this.e_shoff = bytes.getInt(0x20);
    this.e_ehsize = bytes.getShort(0x28);
    this.e_phentsize = bytes.getShort(0x2a);
    this.e_phnum = bytes.getShort(0x2c);
    this.e_shentsize = bytes.getShort(0x2e);
    this.e_shnum = bytes.getShort(0x30);
```

```
        this.e_shstrndx = bytes.getShort(0x32);

        if (EI_MAG != EI_MAG_ELF) {
            ElfError("Not elf file");
        }

        if (EI_CLASS != EI_CLASS_32) {
            ElfError("Not 32 bit elf");
        }

        if (EI_DATA != EI_DATA_LE) {
            ElfError("Not little-endian elf");
        }

        if (e_machine != E_MACHINE_RISCV) {
            ElfError("Not RISC-V elf file");
        }

    }

    public ElfFile(String inputName) {
        try {
                                                byte[]    arrayByte    =
Files.readAllBytes(Paths.get(inputName));
            bytesRead = arrayByte.length;
            bytes = ByteBuffer.wrap(arrayByte);
            bytes.order(ByteOrder.LITTLE_ENDIAN);
        } catch (FileNotFoundException e) {
                throw new IllegalArgumentException("Input file not
found: " + e.getMessage());
        } catch (IOException e) {
```

```
                throw new IllegalArgumentException("Could not read from
input file: " + e.getMessage());
            }
        }

        public void write(String outputName) {
            try {
                BufferedWriter writer = new BufferedWriter(
                                                new  OutputStreamWriter(new
FileOutputStream(outputName), "utf8"));
                writeText(writer);
                symtab.write(writer);
                writer.close();

            } catch (IOException e) {
                    throw new IllegalArgumentException("Could not open
output file: " + e.getMessage());
            }
        }

        public void parse() {
            parseHeader();
            parseSectionHeader();
            parseSymtab();
            labels = symtab.toLabels();
            parseText();
        }

        private String getBytesString(int start, int length) {
            StringBuilder str = new StringBuilder();
```

```
            for (int i = start; i < bytesRead && i < start + length;
i++) {
                str.append((char) bytes.get(i));
            }
            return str.toString();
        }

        private int getInstruction(int index) {
            return bytes.getInt(index);
        }

        public static String rToString(int d) {
            switch (d) {
                case 0:
                    return "zero";
                case 1:
                    return "ra";
                case 2:
                    return "sp";
                case 3:
                    return "gp";
                case 4:
                    return "tp";
                case 5, 6, 7:
                    return "t" + Integer.toString(d - 5);
                case 8, 9:
                    return "s" + Integer.toString(d - 8);
                case 10, 11, 12, 13, 14, 15, 16, 17:
                    return "a" + Integer.toString(d - 10);
                case 18, 19, 20, 21, 22, 23, 24, 25, 26, 27:
```

```
                return "s" + Integer.toString(d - 16);
            case 28, 29, 30, 31:
                return "t" + Integer.toString(d - 25);
        }
        ElfError("Unknown register x" + d);
        return null;
    }

    private String offsetToString(int addr, int offset) {
        addr += offset;
        if (labels.checkLabel(addr)) {
                                return   String.format("0x%s   <%s>",
Integer.toHexString(addr), labels.getLabel(addr));
        }
        String name = String.format("L%d", unknownAddr++);
        labels.add(addr, name);
        return String.format("0x%s <%s>", Integer.toHexString(addr),
name);
    }

    public static int getOpcode(int instr) {
        return instr & 0b1111111;
    }

    public int getBits(int instr, int r, int l) {
        return (instr >> l) & ((1 << (r - l + 1)) - 1);
    }

    public static int to12Bits(int x) {
        x = x & (0xfff);
        if ((x & (0x800)) != 0) {
```

```
            x = -(x ^ 0xfff) - 1;
        }
        return x;
    }

    public static int to7Bits(int x) {
        x = x & (0b1111111);
        if ((x & (0b1000000)) != 0) {
            x = -(x ^ 0b1111111) - 1;
        }
        return x;
    }

    public static int to20Bits(int x) {
        x = x & (0xfffff);
        if ((x & (0x80000)) != 0) {
            x = -(x ^ 0xfffff) - 1;
        }
        return x;
    }

    private void parseSectionHeader() {
        for (int i = e_shoff; i + 0x24 < bytesRead; i += 0x28) {
            int index = (i - e_shoff) / 0x28;
               if (index == e_shstrndx && bytes.getInt(i + 0x04) ==
SHT_STRTAB) {
                namesOffset = bytes.getInt(i + 0x10);
                break;
            }
        }
```

```
        if (namesOffset == -1) {
            ElfError("Section names not found");
        }
        for (int i = e_shoff; i + 0x24 < bytesRead; i += 0x28) {
            // .text -- 5 bytes
            // .symtab -- 7 bytes
            int sh_name = bytes.getInt(i);
             String textName = getBytesString(namesOffset + sh_name,
5);
            if (textName.equals(".text")) {
                textAddr = bytes.getInt(i + 0x0c);
                textOffset = bytes.getInt(i + 0x10);
                textSize = bytes.getInt(i + 0x14);
            }
                   String symtabName = getBytesString(namesOffset +
sh_name, 7);
            if (symtabName.equals(".symtab") &&
                    bytes.getInt(i + 0x04) == SHT_SYMTAB) {
                symtabOffset = bytes.getInt(i + 0x10);
                symtabSize = bytes.getInt(i + 0x14);
            }


                   String strtabName = getBytesString(namesOffset +
sh_name, 7);
            if (symtabName.equals(".strtab") &&
                    bytes.getInt(i + 0x04) == SHT_STRTAB) {
                strtabOffset = bytes.getInt(i + 0x10);
            }


        }
        if (symtabOffset == -1) {
```

```
            ElfError("Section .symtab not found");
        }
        if (symtabOffset == -1) {
            ElfError("Section .text not found");
        }
        if (strtabOffset == -1) {
            ElfError("Section .strtab not found");
        }
    }

    private void parseText() {
         for (int i = 0; i < textSize && textOffset + i < bytesRead;
i += 4) {
            int x = getInstruction(textOffset + i);
            int addr = textAddr + i;
            int opcode = getOpcode(x);
            int func3 = getBits(x, 14, 12);
            int func7 = getBits(x, 31, 25);
            int offset;
            String name = null;
            String arg1 = null;
            String arg2 = null;
            String arg3 = null;
            switch (opcode) {
                case (0b0110111):
                    name = "lui";
                    arg1 = rToString(getBits(x, 11, 7));
                    arg2 = Integer.toString(getBits(x, 31, 12));
                    break;
                case (0b0010111):
```

```
                    name = "auipc";
                    arg1 = rToString(getBits(x, 11, 7));
                    arg2 = Integer.toString(getBits(x, 31, 12));
                    break;
                case (0b0010011):
                    arg1 = rToString(getBits(x, 11, 7));
                    arg2 = rToString(getBits(x, 19, 15));
                    switch (func3) {
                        case (0b000):
                                                                arg3  =
Integer.toString(to12Bits(getBits(x, 31, 20)));
                            name = "addi";
                            break;
                        case (0b010):
                                                                arg3  =
Integer.toString(to12Bits(getBits(x, 31, 20)));
                            name = "slti";
                            break;
                        case (0b011):
                              arg3 = Integer.toString(getBits(x, 31,
20));
                            name = "sltiu";
                            break;
                        case (0b100):
                                                                arg3  =
Integer.toString(to12Bits(getBits(x, 31, 20)));
                            name = "xori";
                            break;
                        case (0b110):
                                                                arg3  =
Integer.toString(to12Bits(getBits(x, 31, 20)));
                            name = "ori";
```

```
                    break;
                case (0b111):
                                                    arg3  =
Integer.toString(to12Bits(getBits(x, 31, 20)));
                    name = "andi";
                    break;
                case (0b001):
                      arg3 = Integer.toString(getBits(x, 24,
20));
                    name = "slli";
                    break;
                case (0b101):
                    switch (func7 | 1) {
                        case (0b0000001):
                                                    arg3  =
Integer.toString(getBits(x, 24, 20));
                            name = "srli";
                            break;
                        case (0b0100001):
                                                    arg3  =
Integer.toString(getBits(x, 24, 20));
                            name = "srai";
                            break;
                        default:
                            name = "unknown_instruction";
                            arg1 = null;
                            arg2 = null;
                            arg3 = null;
                    }
                    break;
                default:
```

```
                name = "unknown_instruction";
                arg1 = null;
                arg2 = null;
                arg3 = null;

        }
        break;

    case (0b0110011):
        arg1 = rToString(getBits(x, 11, 7));
        arg2 = rToString(getBits(x, 19, 15));
        arg3 = rToString(getBits(x, 24, 20));
        switch (func3) {
            case (0b000):
                switch (func7) {
                    case (0):
                        name = "add";
                        break;
                    case (0b0100000):
                        name = "sub";
                        break;
                    case (0b0000001):
                        name = "mul";
                        break;
                    default:
                        name = "unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                }
```

```
            break;
        case (0b001):
            switch (func7) {
                case (0):
                    name = "sll";
                    break;
                case (0b0000001):
                    name = "mulh";
                    break;
                default:
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
            }
            break;
        case (0b010):
            switch (func7) {
                case (0):
                    name = "slt";
                    break;
                case (0b0000001):
                    name = "mulhsu";
                    break;
                default:
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
            }
```

```
            break;
        case (0b011):
            switch (func7) {
                case (0):
                    name = "sltu";
                    break;
                case (0b0000001):
                    name = "mulhu";
                    break;
                default:
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
            }
            break;
        case (0b100):
            switch (func7) {
                case (0):
                    name = "xor";
                    break;
                case (0b0000001):
                    name = "div";
                    break;
                default:
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
            }
```

```
                break;
            case (0b101):
                switch (func7) {
                    case (0):
                        name = "srl";
                        break;
                    case (0b0100000):
                        name = "sra";
                        break;
                    case (0b0000001):
                        name = "divu";
                        break;
                    default:
                        name = "unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                }
                break;
            case (0b110):
                switch (func7) {
                    case (0):
                        name = "or";
                        break;
                    case (0b0000001):
                        name = "rem";
                        break;
                    default:
                        name = "unknown_instruction";
                        arg1 = null;
```

```
                    arg2 = null;
                    arg3 = null;
                }
                break;
            case (0b111):
                switch (func7) {
                    case (0):
                        name = "and";
                        break;
                    case (0b0000001):
                        name = "remu";
                        break;
                    default:
                        name = "unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                }
                break;
            default:
                name = "unknown_instruction";
                arg1 = null;
                arg2 = null;
                arg3 = null;
        }
        break;
    case (0b0001111):
        name = "fence";
        arg1 = null;
        arg2 = null;
```

```
            arg3 = null;
            break;
        case (0b1110011):
            arg1 = rToString(getBits(x, 11, 7));
            switch (func3) {
                case (0b000):
                    switch (getBits(x, 31, 7)) {
                        case (0):
                            arg1 = null;
                            arg2 = null;
                            arg3 = null;
                            name = "ecall";
                            break;
                        case (0b10000000000000):
                            arg1 = null;
                            arg2 = null;
                            arg3 = null;
                            name = "ebreak";
                            break;
                        case (0b100000000000000):
                            arg1 = null;
                            arg2 = null;
                            arg3 = null;
                            name = "uret";
                            break;
                        case (0b00010000001000000000000000):
                            arg1 = null;
                            arg2 = null;
                            arg3 = null;
                            name = "sret";
```

```
                    break;
                case (0b00110000001000000000000):
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
                    name = "mret";
                    break;
                case (0b00010000010100000000000):
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
                    name = "wfi";
                    break;
                default:
                    if (func7 == 0b0001001) {
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                        name = "sfence.vma";
                    } else {
                                                name =
"unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                    }
            }
            break;
        default:
            name = "unknown_instruction";
```

```
                arg1 = null;
                arg2 = null;
                arg3 = null;
            }
            break;

        case (0b0000011):
            arg1 = rToString(getBits(x, 11, 7));
            arg2 = Integer.toString(getBits(x, 31, 20));
            arg3 = rToString(getBits(x, 19, 15));
            switch (func3) {
                case 0b000:
                    name = "lb";
                    break;
                case 0b001:
                    name = "lh";
                    break;
                case 0b010:
                    name = "lw";
                    break;
                case 0b100:
                    name = "lbu";
                    break;
                case 0b101:
                    name = "lhu";
                    break;

                default:
                    name = "unknown_instruction";
                    arg1 = null;
```

```
                    arg2 = null;
                    arg3 = null;
                }
                break;

            case (0b0100011):
                offset = getBits(x, 11, 7) | (getBits(x, 31, 25)
<< 5);
                arg1 = rToString(getBits(x, 24, 20));
                arg2 = Integer.toString(offset);
                arg3 = rToString(getBits(x, 19, 15));
                switch (func3) {
                    case 0b000:
                        name = "sb";
                        break;
                    case 0b001:
                        name = "sh";
                        break;
                    case 0b010:
                        name = "sw";
                        break;

                    default:
                        name = "unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                }
                break;
```

```
                case (0b1101111):
                                offset = (getBits(x, 31, 31) << 20) |
(getBits(x, 30, 21) << 1)
                                        | (getBits(x, 20, 20) << 11) |
(getBits(x, 19, 12) << 12);
                    arg1 = rToString(getBits(x, 11, 7));
                    arg2 = offsetToString(addr, to20Bits(offset));
                    name = "jal";
                    break;
                case (0b1100111):
                    if (func3 != 0) {
                        name = "unknown_instruction";
                        arg1 = null;
                        arg2 = null;
                        arg3 = null;
                        break;
                    }
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
                    arg1 = rToString(getBits(x, 11, 7));
                    arg2 = Integer.toString(getBits(x, 31, 20));
                    arg3 = rToString(getBits(x, 19, 15));
                    name = "jalr";
                    break;
                case (0b1100011):
                                offset = (getBits(x, 31, 31) << 12) |
(getBits(x, 30, 25) << 5) |
                                  (getBits(x, 11, 8) << 1) | (getBits(x,
7, 7) << 11);
                    arg1 = rToString(getBits(x, 19, 15));
```

```
            arg2 = rToString(getBits(x, 24, 20));
            arg3 = offsetToString(addr, to12Bits(offset));
            switch (func3) {
                case 0b000:
                    name = "beq";
                    break;
                case 0b001:
                    name = "bne";
                    break;
                case 0b100:
                    name = "blt";
                    break;
                case 0b101:
                    name = "bge";
                    break;
                case 0b110:
                    name = "bltu";
                    break;
                case 0b111:
                    name = "bgeu";
                    break;
                default:
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
            }
            break;

        default:
```

```
                    name = "unknown_instruction";
                    arg1 = null;
                    arg2 = null;
                    arg3 = null;
                }
                text.add(new Instruction(addr, x, name, arg1, arg2,
arg3));
            }
        }

    private void parseSymtab() {
            for (int i = symtabOffset, symbol = 0; i < symtabOffset +
symtabSize; i += 0x10, symbol++) {
                int value = bytes.getInt(i + 4);
                int size = bytes.getInt(i + 8);
                int type = bytes.get(i + 12) % 0x10;
                int bind = bytes.get(i + 12) / 0x10;
                int vis = bytes.get(i + 13);
                short index = bytes.getShort(i + 14);
                String name = parseSymbolName(strtabOffset +
bytes.getInt(i));
                symtab.add(new SymbolTable.Symbol(symbol, value, size,
type, bind, vis, index, name));
            }
        }

    private String parseSymbolName(int index) {
        StringBuilder str = new StringBuilder();
        for (int i = index; i < bytesRead && bytes.get(i) != 0; i++)
{
            str.append((char) bytes.get(i));
        }
```

```
        return str.toString();
    }

    private void writeText(BufferedWriter writer) throws IOException
{
        writer.write(".text\n");
        for (Instruction i : text) {
            if (labels.checkLabel(i.getAddr())) {
                String addr = Integer.toHexString(i.getAddr());
                if (addr.length() < 8) {
                      addr = String.format("%0" + (8 - addr.length())
+ "d%s", 0, addr);
                }
                String label = labels.getLabel(i.getAddr());
                    writer.write(String.format("%s    <%s>:\n", addr,
label));
            }
            writer.write(i.toString());
        }
    }

    public static void ElfError(String msg) {
        throw new IllegalStateException(msg);
    }
}
```