Лабораторная работа №3	M3136	2022
ISA	Шибанов Игорь Александрович	

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

Инструментарий и требования к работе: работа выполнена на языке программирования Java.

Описание

Необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Вариант

Должен поддерживаться следующий набор команд: RISC-V RV32I, RV32M.

Кодирование: little endian.

RISC-V

RISC — архитектура процессора, в котором быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. Термин «сокращённый» в названии описывает тот факт, что сокращён объём (и время) работы, выполняемый каждой отдельной инструкцией — как максимум один цикл доступа к памяти.

Характерные особенности RISC-процессоров

- Элемент маркированного списка.
- Фиксированная длина машинных инструкций, простой формат команды.
- Специализированные команды для операций с памятью — чтения или записи. Операции вида Read-Modify-Write («прочитать-изменить-записать») отсутствуют. Любые операции «изменить» выполняются только над содержимым регистров (т.н. архитектура load-and-store).
- Большое количество регистров общего назначения (32 и более).
- Отсутствие поддержки операций вида «изменить» над укороченными типами данных — байт, 16-битное слово.
- Отсутствие микропрограмм внутри самого процессора.

RISC-V — это открытая спецификация архитектуры набора команд (ISA, Instruction Set Architecture), основанного на принципах сокращенного набора инструкций (RISC, Reduced Instruction Set Computer). Другими словами, RISC-V представляет собой описание ассемблерных инструкций, способа их кодирования, и семантику работы. Придерживание принципов RISC означает, что RISC-V не вводит лишних инструкций без сильной надобности. За счет этого существенно упрощается архитектура процессора.

Инструкции RISC-V имеют переменную длину. При этом длина инструкции определяется достаточно просто. Набор команд является расширяемым. В любой реализации обязательно присутствует базовый набор инструкций, который может быть 32-х, 64-х или 128-и битным. Базовые наборы команд называются соответственно RV32I, RV64I и RV128I. Помимо базовых команд конкретная реализация может предлагать команды, относящиеся к тем или иным расширениям. Например,

расширение A содержит в себе атомарные операции, а расширения F и D — операции над числами с плавающей точкой одинарной и двойной точности соответственно (то есть, float и double).

Формат машинных команд

Тип	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
Регистр/регистр	funct7							rs2					rs1					funct3			rd					код операции					1	1
С операндом	±	imm[10:0]											rs1					funct3			rd					код операции					1	1
С длинным операндом	±	imm[30:12]																			rd					код операции					1	1
Сохранение	±	imm[10:5]						rs2					rs1					funct3			imm[4:0]					код операции					1	1
Ветвление	±	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	код операции					1	1
Переход	±	imm[10:1]										[11]	imm[19:12]								rd					код операции					1	1

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I — Integer) и несколько стандартных опциональных расширений.

В базовый набор входят инструкции условной и безусловной передачи управления/ветвления, минимальный набор арифметических/битовых операций на регистрах, операций с памятью (load/store), а также небольшое число служебных инструкций.

Операции ветвления не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций операнды просто меняются местами.

Базовое подмножество команд использует следующий набор регистров: специальный регистр x0 (zero), 31 целочисленный регистр общего назначения (x1 — x31), регистр счётчика команд (PC, используется только косвенно), а также множество CSR (Control and Status Registers, может быть адресовано до 4096 CSR).

Операции не сохраняют где-либо биты переноса или переполнения, что приближено к модели операций в языке программирования Си. Также аппаратно не генерируются исключения по переполнению и даже по делению на 0. Все необходимые проверки операндов и результатов операций должны производиться программно.

Архитектура использует только модель little-endian — первый байт операнда в памяти соответствует младшим битам значений регистрового операнда.

Операции умножения, деления и вычисления остатка не входят в минимальный набор инструкций, а выделены в отдельное расширение (M — Multiply extension). Имеется ряд доводов в пользу разделения и данного набора на два отдельных (умножение и деление).

Регистры

RISC-V имеет 32 (или 16 для встраиваемых применений) целочисленных регистра. При реализации вещественных групп команд есть дополнительно 32 вещественных регистра.

Для операций над числами в бинарных форматах плавающей запятой используется набор дополнительных 32 регистров FPU (Floating Point Unit), которые совместно используются расширениями базового набора инструкций для трёх вариантов точности: одинарной — 32 бита (F extension), двойной — 64 бита (D — Double precision extension), а также четверной — 128 бит (Q — Quadruple precision extension).

Elf file

ELF — формат исполняемых двоичных файлов, используемый во многих современных UNIX-подобных операционных системах, таких как FreeBSD, Linux, Solaris и др.

Каждый ELF файл состоит из следующих частей:

Заголовок файла

Заголовок файла (ELF Header) имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры и смещения остальных частей файла. Заголовок имеет размер 52 байта для 32-битных файлов или 64 для 64-битных.

Обрабатываемые поля

EI_MAG0 – EI_MAG3 – сигнатура файла

EI_CLASS – класс объектного файла

EI_DATA – метод кодирования данных

EI_VERSION – версия elf заголовка (актуальное значение 1)

e_machine – архитектура аппаратной платформы

e_version – номер версии формата (актуальное значение 1)

e_shoff - смещение таблицы заголовков секций от начала файла

e_ehsize – размер заголовка файла

e_shentsize – размер одного заголовка секции

e_shnum - число заголовков секций

e_shstrndx - индекс записи в таблице заголовков секций.

Таблица заголовков программы

Таблица заголовков программы содержит заголовки, каждый из которых описывает отдельный сегмент программы и его атрибуты либо другую информацию, необходимую операционной системе для подготовки программы к исполнению. Данная таблица может располагаться в любом месте файла, её местоположение (смещение относительно начала файла) описывается в поле e_phoff заголовка ELF.

Таблица заголовков секций

Код и данные в логически разделены на секции, которые представляют собой непересекающиеся смежные блоки, расположенные в файле друг за другом без промежутков. У секций нет определенной общей структуры: в каждой секции организация размещения данных или кода зависит от ее назначения. Более того, некоторые секции вообще могут не иметь какой-либо структуры, а представлять собой неструктурированный блок кода или данных. Каждая секция описывается своим заголовком, который хранится в таблице заголовков секций. В заголовке перечислены свойства секции, а также местонахождение содержимого самой секции в файле.

Обрабатываемые поля секции

sh_name - смещение строки относительно начала таблицы названий секций

sh_addr - адрес, начиная с которого секция будет загружена

sh_offset - смещение секции от начала файла

sh_size – размер секции

sh_link – индекс ассоциированной секции

Обрабатываемые секции

.shstrtab – эта секция содержит имена всех остальных секций ELF-файла в кодировке ASCII с завершающим нулем.

.symtab — секция содержит так называемую таблицу статических символов (под символами в данном случае понимаются имена функций или переменных). Как правило, секция с таблицей статических символов носит имя .symtab, каждая запись в этой секции нужна для сопоставления того или иного символа с местонахождением функции или переменной, имя которой и определено данным символом. Все это в подавляющем большинстве случаев нужно, чтобы облегчить отладку программы, а непосредственно для выполнения эта секция не используется.

Обрабатываемые поля в строке symtab

name – индекс в таблице строк, которое содержит символьное представления имен

value – значение связанного символа

size – связанный размер

info – тип символа и атрибуты привязки

other – определяет видимость символа

shndx - содержит соответствующий индекс таблицы заголовков разделов

.text – секция, в которой находится весь исходный код.

Описание работы написанного кода

Первое, что я написал класс BinaryFile. Он сохраняет elf-file в массив и может выдавать нужное количество байт.

Далее я написал класс ELFHeader, он берет первые 52 байта из файла и создает все поля заголовка файла.

Далее я создал класс SectionHeader. Он берет смещение (e_shoff), размер секции (e_shentsize), количество секций (e_shnum), индекс StringTable (e_shstrndx) из ELFHeader и создает массив всех секций (Section). В свою очередь, класс Section принимает в себя массив байт и создает соответствующие поля.

После этого нужно достать StringTable. Из ELFHeader возьмем индекс StringTable и по нему возьмем из SectionHeader секцию StringTable. Для обработки StringTable я написал класс ParseString. В этот класс передается расположение байт в памяти, после чего он преобразовывает их в строку. С помощью SectionHeader можно найти имя для каждой секции.

Далее я создал класс Symtab и нашел по имени «.symtab» (getSectionByName) нужную секцию в SectionHeader. Из секции я получил расположение в файле нужных мне данных. В классе Symtab хранится массив с SymtabLine. В свою очередь SymtabLine хранит в себе нужные поля для одной строчки Symtab. Чтобы узнать имена для каждой SymtabLine, я взял секцию из SectionHeader (names), индекс которой находится в секции symtab (symtabSection.getSh_link()). Аналогично StringTable, данная секция (names) обрабатывается классом ParseString.

Осталось обработать секцию text. Этой задачей занимается класс ParseText. Аналогично symtab, я нашел по имени «.text» нужную секцию. Из секции я получил расположение нужных мне данных. В классе ParseText хранится массив с RiscV командами. Для их обработки создан класс RiscV, куда передается исходная команда в типе int. Адрес первой команды хранится в секции text (поле sh_addr).

Метки

Для того чтобы узнать название метки, в symtab написан специальный метод (getFunctionNameByValue). Он ищет название функции по полю value и возвращает его. Если же он не нашел функцию, он возвращает L метку в соответствии с порядком. Команды, которые расставляют метки, имеют offset. Чтобы получить значение, по которому можно найти название метки, нужно сложить offset и текущий адрес команды, получив индекс метки (название в коде markIndex). Также markIndex нужен, чтобы ставить метки при выводе. Перед каждой командой, адрес которой совпадает с markIndex, стоит нужная метка.

Обработка ошибок

1. Ошибки в ELF Header. В данном классе создан специальный method isValid, который проверят правильность полей. В случае некорректного ввода - бросает соответствующие исключение.
2. Ошибки в SectionHeader. В функциях getSectionByName и getSection, в случае ненахождения секции - бросается исключение об этом.
3. Ошибки RiscV. Если данная команда не поддерживается, то вместо нее выводится unknown_instruction.

Результат работы написанной программы

```
.text
00010074   <main>:
10074:        ff010113       addi       sp, sp, -16
10078:        00112623         sw       ra, 12(sp)
1007c:        030000ef        jal       ra, 100ac <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, 10074 <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, 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, 100e4 <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, 100d8 <L1>
10114:        050f0f13       addi       t5, t5, 80
10118:        01478513       addi       a0, a5, 20
1011c:        fa5f16e3        bne       t5, t0, 100c8 <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
```

Источники

https://ru.bmstu.wiki/RISC_(Reduced_Instruction_Set_Computing)

https://eax.me/fpga-risc-v/

https://ru.wikipedia.org/wiki/RISC-V

https://habr.com/ru/company/samsung/blog/668810/

https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format

https://tech-geek.ru/elf-files-linux/

https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html

https://refspecs.linuxbase.org/elf/gabi4+/ch4.symtab.html

Листинг кода

BinaryFile.java

```java
import java.io.FileInputStream;
import java.io.IOException;
```

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

public class BinaryFile implements AutoCloseable {
    private final List<Integer> bytes;
    private final FileInputStream reader;

    public BinaryFile(final String source) throws IOException {
        bytes = new ArrayList<>();
        reader = new FileInputStream(source);
        int read;
        while ((read = reader.read()) != -1) {
            bytes.add(read);
        }
    }

    public int[] getBytes(int index, int size) {
        try {
            int[] outBytes = new int[size];
            for (int i = 0; i < size; i++) {
                outBytes[i] = bytes.get(index + i);
            }
            return outBytes;
        } catch (IndexOutOfBoundsException e) {
            throw new IllegalArgumentException("File not supported: invalid
file size");
        }
    }

    @Override
    public void close() throws IOException {
        reader.close();
    }


}
```

ELFHeader.java

```java
public class ELFHeader {
    private final int[] bytes;
    private final int EI_MAG0;
    private final int EI_MAG1;
    private final int EI_MAG2;
    private final int EI_MAG3;
    private final int EI_CLASS;
    private final int EI_DATA;
    private final int EI_VERSION;
    private final int EI_OSABI;
```

```
private final int EI_ABIVERSION;
private final int[] e_type = new int[2];
private final int[] e_machine = new int[2];
private final int[] e_version = new int[4];
private final int[] e_entry = new int[4];
private final int[] e_phoff = new int[4];
private final int[] e_shoff = new int[4];
private final int[] e_flags = new int[4];
private final int[] e_ehsize = new int[2];
private final int[] e_phentsize = new int[2];
private final int[] e_phum = new int[2];
private final int[] e_shentsize = new int[2];
private final int[] e_shnum = new int[2];
private final int[] e_shstrndx = new int[2];

private int byteIndex;

public ELFHeader(BinaryFile input) {
    this.bytes = input.getBytes(0, 52);
    EI_MAG0 = bytes[0];
    EI_MAG1 = bytes[1];
    EI_MAG2 = bytes[2];
    EI_MAG3 = bytes[3];
    EI_CLASS = bytes[4];
    EI_DATA = bytes[5];
    EI_VERSION = bytes[6];
    EI_OSABI = bytes[7];
    EI_ABIVERSION = bytes[8];
    byteIndex = 16;
    initArray(e_type);
    initArray(e_machine);
    initArray(e_version);
    initArray(e_entry);
    initArray(e_phoff);
    initArray(e_shoff);
    initArray(e_flags);
    initArray(e_ehsize);
    initArray(e_phentsize);
    initArray(e_phum);
    initArray(e_shentsize);
    initArray(e_shnum);
    initArray(e_shstrndx);
    isValid();
}

private void isValid() {
    if (getE_ehsize() != 52) {
```

```
            throw new IllegalArgumentException("File is not supported:
incorrect file header size");
        }
        if (getEI_MAG0() != 0x7f || getEI_MAG1() != 0x45 ||
                getEI_MAG2() != 0x4c || getEI_MAG3() != 0x46) {
            throw new IllegalArgumentException("File is not supported:
incorrect file signature");
        }
        if (getEI_CLASS() != 1) {
            throw new IllegalArgumentException("File is not supported:
incorrect file class");
        }
        if (getEI_DATA() != 1) {
            throw new IllegalArgumentException("File is not supported:
incorrect encoding method");
        }
        if (getEI_VERSION() != 1) {
            throw new IllegalArgumentException("File is not supported:
incorrect elf header version");
        }
        if (getE_machine() != 0xF3) {
            throw new IllegalArgumentException("File is not supported:
incorrect hardware platform architecture");
        }
        if (getE_version() != 1) {
            throw new IllegalArgumentException("File is not supported:
incorrect format version number");
        }
        if (getE_shoff() == 0) {
            throw new IllegalArgumentException("File is not supported: there
is no section header table");
        }
        if (getE_shentsize() != 40) {
            throw new IllegalArgumentException("File is not supported:
incorrect section header size");
        }
        if (getE_shnum() == 0) {
            throw new IllegalArgumentException("File is not supported: there
is no section header table");
        }
        if (getE_shstrndx() == 0) {
            throw new IllegalArgumentException("File is not supported: there
is no string table");
        }
    }

    private void initArray(int[] source) {
        for (int i = 0; i < source.length; i++) {
```

```
            source[i] = bytes[byteIndex];
            byteIndex++;
        }
    }

    public void out() {
        System.out.println("ELF Header:\t");
        System.out.println("Magic:\t" + getEI_MAG());
        System.out.println("Class:\t" + getEI_CLASS());
        System.out.println("Data:\t" + getEI_DATA());
        System.out.println("Version:\t" + getEI_VERSION());
        System.out.println("OS/ABI:\t" + getEI_OSABI());
        System.out.println("ABI Version:\t" + getEI_ABIVERSION());
        System.out.println("Type:\t" + getE_type());
        System.out.println(String.format("Machine: \t %x", getE_machine()));
        System.out.println("Version:\t" + getE_version());
        System.out.println(String.format("Entry point address: \t %x",
getE_entry()));
        System.out.println("Start of program headers:\t" + getE_phoff());
        System.out.println("Start of section headers:\t" + getE_shoff());
        System.out.println("Flags:\t" + getE_flags());
        System.out.println("Size of this header:\t" + getE_ehsize());
        System.out.println("Size of program headers:\t" + getE_phentsize());
        System.out.println("Number of program headers:\t" + getE_phum());
        System.out.println("Size of section headers:\t" + getE_shentsize());
        System.out.println("Number of section headers:\t" + getE_shnum());
        System.out.println("Section header string table index:\t" +
getE_shstrndx());
    }


    public int getEI_MAG0() {
        return EI_MAG0;
    }

    public int getEI_MAG1() {
        return EI_MAG1;
    }

    public int getEI_MAG2() {
        return EI_MAG2;
    }

    public int getEI_MAG3() {
        return EI_MAG3;
    }

    public String getEI_MAG() {
```

```
        return String.format("%x %x %x %x", getEI_MAG0(), getEI_MAG1(),
getEI_MAG2(), getEI_MAG3());
    }

    public int getEI_CLASS() {
        return EI_CLASS;
    }

    public int getEI_DATA() {
        return EI_DATA;
    }

    public int getEI_VERSION() {
        return EI_VERSION;
    }

    public int getEI_OSABI() {
        return EI_OSABI;
    }

    public int getEI_ABIVERSION() {
        return EI_ABIVERSION;
    }

    public int getE_type() {
        return Functions.bytesToInt(e_type);
    }

    public int getE_machine() {
        return Functions.bytesToInt(e_machine);
    }

    public int getE_version() {
        return Functions.bytesToInt(e_version);
    }

    public int getE_entry() {
        return Functions.bytesToInt(e_entry);
    }

    public int getE_phoff() {
        return Functions.bytesToInt(e_phoff);
    }

    public int getE_shoff() {
        return Functions.bytesToInt(e_shoff);
    }
```

```
    public int getE_flags() {
        return Functions.bytesToInt(e_flags);
    }

    public int getE_ehsize() {
        return Functions.bytesToInt(e_ehsize);
    }

    public int getE_phentsize() {
        return Functions.bytesToInt(e_phentsize);
    }

    public int getE_phum() {
        return Functions.bytesToInt(e_phum);
    }

    public int getE_shentsize() {
        return Functions.bytesToInt(e_shentsize);
    }

    public int getE_shnum() {
        return Functions.bytesToInt(e_shnum);
    }

    public int getE_shstrndx() {
        return Functions.bytesToInt(e_shstrndx);
    }

}
```

Functions.java

```
public class Functions {
    public static int bytesToInt(int[] source) { // перевод массива байт в int
(little endian)
        int ans = 0;
        int multiplier = 1;
        for (int i = 0; i < source.length; i++) {
            ans = ans + source[i] * multiplier;
            multiplier = multiplier * 256;
        }
        return ans;
    }

    public static int slice(int value, int finish, int start) {// срез числа
        if (finish == 31 && start == 0) {
            return value;
        }
```

```
        return (value >>> start) & ((1 << (finish - start + 1)) - 1);
    }

    public static int getValue(int value, int numberOfBytes) {// вывод числа с
учетом дополнения до двух
        if (((value >> numberOfBytes) & 1) == 0) {
            return value;
        }
        return -slice((~value) + 1, numberOfBytes, 0);
    }

    public static String getReg(int value) {// нужные регистры
        switch (value) {
            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" + (value - 5);
            }
            case 8, 9 -> {
                return "s" + (value - 8);
            }
            case 10, 11, 12, 13, 14, 15, 16, 17 -> {
                return "a" + (value - 10);
            }
            case 18, 19, 20, 21, 22, 23, 24, 25, 26, 27 -> {
                return "s" + (value - 16);
            }
            case 28, 29, 30, 31 -> {
                return "t" + (value - 25);
            }
        }
        return "";
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        Parser.parse(args[1], args[2]);
    }
}
```

Parser.java

```
import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class Parser {
    public static void parse(String inputFile, String outputFile) {
        try {
            BinaryFile elfFile = new BinaryFile(inputFile);
            try {
                BufferedWriter writer = new BufferedWriter(new
OutputStreamWriter(
                        new FileOutputStream(outputFile),
                        "UTF8"
                ));
                try {
                    ELFHeader elfHeader = new ELFHeader(elfFile);
                    SectionHeader sectionHeader = new SectionHeader(elfFile,
elfHeader);
                    Symtab symtab = new Symtab(elfFile, sectionHeader);
                    ParseText text = new ParseText(elfFile,
sectionHeader.getSectionByName(".text"), symtab);
                    writer.write(text.toString());
                    writer.newLine();
                    writer.write(symtab.toString());
                } finally {
                    writer.close();
                }
            } catch (IOException e) {
                System.out.println("Output file exception:" + e.getMessage());
            } finally {
                elfFile.close();
            }
        } catch (IOException e) {
            System.out.println("Input file exception:" + e.getMessage());
        }
    }
}
```

ParseString.java

```
public class ParseString {
    private final int[] bytes;

    public ParseString(int[] bytes) {
        this.bytes = bytes;
    }

    public String bytesToString(int index) {
        StringBuilder name = new StringBuilder();
        for (int i = index; i < bytes.length; i++) {
            if (bytes[i] == 0) {
                break;
            }
            name.append((char) bytes[i]);
        }
        return name.toString();
    }
}
```

ParseText.java

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

public class ParseText {
    private final List<RiscV> commands;
    private final Map<Integer, String> marks;

    public ParseText(BinaryFile input, Section text, Symtab symtab) {
        marks = new HashMap<>();
        commands = new ArrayList<>();
        int sh_addr = text.getSh_addr();
        int index = text.getSh_offset();
        for (int i = 0; i < text.getSh_size(); i += 4, index += 4, sh_addr +=
4) {
            RiscV r = new RiscV(Functions.bytesToInt(input.getBytes(index,
4)), sh_addr, symtab);
            commands.add(r);
            if (r.hasMark()) {
                marks.put(r.getMarkIndex(), r.getMark());
            }
        }
    }
```

```
    @Override
    public String toString() {
        StringBuilder out = new StringBuilder();
        out.append(".text");
        out.append(System.lineSeparator());
        for (int i = 0; i < commands.size(); i++) {
            RiscV command = commands.get(i);
            if (marks.containsKey(command.getAddr())) {
                out.append(String.format("%08x   %s:", command.getAddr(),
marks.get(command.getAddr())));
                out.append(System.lineSeparator());
            }
            out.append(command);
            out.append(System.lineSeparator());
        }
        return out.toString();
    }

}
```

RiscV.java

```
public class RiscV {
    private final int value;
    private final int addr;

    private final Symtab symtab;
    private String command = "unknown_instruction";
    private int markIndex;
    private String mark;

    public RiscV(int value, int addr, Symtab symtab) {
        this.value = value;
        this.addr = addr;
        this.symtab = symtab;
        parseOpcode();
    }

    public int getAddr() {
        return addr;
    }

    public boolean hasMark() {
        switch (command) {
            case "jal", "beq", "bne", "blt", "bge", "bltu", "bgeu" -> {
                return true;
            }
```

```
    }
    return false;
}

public int getMarkIndex() {
    return markIndex;
}

public String getMark() {
    if (mark == null) {
        mark = "<" + symtab.getFunctionNameByValue(getMarkIndex()) + ">";
    }
    return mark;
}

private int getOpcode() {
    return Functions.slice(value, 6, 0);
}

private String rType() {
    int funct7 = Functions.slice(value, 31, 25);
    String rs2 = Functions.getReg(Functions.slice(value, 24, 20));
    String rs1 = Functions.getReg(Functions.slice(value, 19, 15));
    int funct3 = Functions.slice(value, 14, 12);
    String rd = Functions.getReg(Functions.slice(value, 11, 7));
    int opcode = getOpcode();
    switch (opcode) {
        case 0b0010011 -> {
            switch (funct7) {
                case 0b0000000, 0b0000001 -> {
                    switch (funct3) {
                        case 0b001 -> command = "slli";
                        case 0b101 -> command = "srli";
                    }
                }
                case 0b0100000, 0b0100001 -> command = "srai";
            }
        }
        case 0b0110011 -> {
            switch (funct7) {
                case 0b0000000 -> {
                    switch (funct3) {
                        case 0b000 -> command = "add";
                        case 0b001 -> command = "sll";
                        case 0b010 -> command = "slt";
                        case 0b011 -> command = "sltu";
                        case 0b100 -> command = "xor";
                        case 0b101 -> command = "srl";
```

```
                        case 0b110 -> command = "or";
                        case 0b111 -> command = "and";
                    }
                }
                case 0b0100000 -> {
                    switch (funct3) {
                        case 0b000 -> command = "sub";
                        case 0b101 -> command = "sra";
                    }
                }
                case 0b0000001 -> {
                    switch (funct3) {
                        case 0b000 -> command = "mul";
                        case 0b001 -> command = "mulh";
                        case 0b010 -> command = "mulhsu";
                        case 0b011 -> command = "mulhu";
                        case 0b100 -> command = "div";
                        case 0b101 -> command = "divu";
                        case 0b110 -> command = "rem";
                        case 0b111 -> command = "remu";
                    }
                }
            }
        }
        return String.format("%7s \t %s, %s, %s", command, rd, rs1, rs2);
    }

    private String iType() {
        int imm = Functions.getValue(Functions.slice(value, 31, 20), 31 - 20);
        String rs1 = Functions.getReg(Functions.slice(value, 19, 15));
        int funct3 = Functions.slice(value, 14, 12);
        String rd = Functions.getReg(Functions.slice(value, 11, 7));
        int opcode = getOpcode();
        switch (opcode) {
            case 0b1100111 -> {
                command = "jalr";
                return String.format("%7s \t %s, %s(%s)", command, rd, imm,
rs1);
            }
            case 0b0000011 -> {
                switch (funct3) {
                    case 0b000 -> command = "lb";
                    case 0b001 -> command = "lh";
                    case 0b010 -> command = "lw";
                    case 0b100 -> command = "lbu";
                    case 0b101 -> command = "lhu";
                }
```

```
                return String.format("%7s \t %s, %s(%s)", command, rd, imm,
rs1);
            }
            case 0b0010011 -> {
                switch (funct3) {
                    case 0b000 -> command = "addi";
                    case 0b010 -> command = "slti";
                    case 0b011 -> command = "sltiu";
                    case 0b100 -> command = "xori";
                    case 0b110 -> command = "ori";
                    case 0b111 -> command = "andi";
                    case 0b001, 0b101 -> {
                        return rType();
                    }
                }
            }
        }
        return String.format("%7s \t %s, %s, %s", command, rd, rs1, imm);
    }

    private String sType() {
        int imm = (Functions.slice(value, 31, 25) << 5) +
Functions.slice(value, 11, 7);
        imm = Functions.getValue(imm, 11);
        String rs2 = Functions.getReg(Functions.slice(value, 24, 20));
        String rs1 = Functions.getReg(Functions.slice(value, 19, 15));
        int funct3 = Functions.slice(value, 14, 12);
        int opcode = getOpcode();
        switch (opcode) {
            case 0b0100011 -> {
                switch (funct3) {
                    case 0b000 -> command = "sb";
                    case 0b001 -> command = "sh";
                    case 0b010 -> command = "sw";
                }
            }
        }
        return String.format("%7s \t %s, %s(%s)", command, rs2, imm, rs1);
    }

    private String bType() {
        int offset = Functions.slice(value, 31, 31);
        offset = (offset << 1) + Functions.slice(value, 7, 7);
        offset = (offset << 6) + Functions.slice(value, 30, 25);
        offset = (offset << 4) + Functions.slice(value, 11, 8);
        offset = Functions.getValue(offset << 1, 12);
        String rs2 = Functions.getReg(Functions.slice(value, 24, 20));
        String rs1 = Functions.getReg(Functions.slice(value, 19, 15));
```

```
        int funct3 = Functions.slice(value, 14, 12);
        int opcode = getOpcode();
        markIndex = addr + offset;
        switch (Functions.slice(value, 14, 12)) {
            case 0b000 -> command = "beq";
            case 0b001 -> command = "bne";
            case 0b100 -> command = "blt";
            case 0b101 -> command = "bge";
            case 0b110 -> command = "bltu";
            case 0b111 -> command = "bgeu";
        }
        return String.format("%7s \t %s, %s, %x", command, rs1, rs2, addr +
offset);
    }

    private String uType() {
        int imm = Functions.getValue(Functions.slice(value, 31, 12), 31 - 12);
        String rd = Functions.getReg(Functions.slice(value, 11, 7));
        int opcode = getOpcode();
        switch (opcode) {
            case 0b0110111 -> command = "lui";
            case 0b0010111 -> command = "auipc";
        }
        return String.format("%7s \t %s, %d", command, rd, imm);
    }

    private String jType() {
        int offset = Functions.slice(value, 31, 31);
        offset = (offset << 8) + Functions.slice(value, 19, 12);
        offset = (offset << 1) + Functions.slice(value, 20, 20);
        offset = (offset << 10) + Functions.slice(value, 30, 21);
        offset = Functions.getValue((offset << 1), 20);
        int opcode = getOpcode();
        String rd = Functions.getReg(Functions.slice(value, 11, 7));
        command = "jal";
        markIndex = offset + addr;
        return String.format("%7s \t %s, %x", command, rd, (offset + addr));
    }

    private String csrType() {
        String rs1 = Functions.getReg(Functions.slice(value, 19, 15));
        String rd = Functions.getReg(Functions.slice(value, 11, 7));
        int csr = Functions.getValue(Functions.slice(value, 31, 27), 31 - 27);
        switch (Functions.slice(value, 14, 12)) {
            case 0b001 -> command = "csrrw";
            case 0b010 -> command = "csrrs";
            case 0b011 -> command = "csrrc";
            case 0b101 -> command = "csrrwi";
```

```
            case 0b110 -> command = "csrrsi";
            case 0b111 -> command = "csrrci";
        }
        return String.format("%7s \t %s, %s, %s", rd, csr, rs1);
    }

    private String argsFence(int value) {
        String ans = "";
        if (Functions.slice(value, 3, 3) == 1) {
            ans += 'i';
        }
        if (Functions.slice(value, 2, 2) == 1) {
            ans += 'o';
        }
        if (Functions.slice(value, 1, 1) == 1) {
            ans += 'r';
        }
        if (Functions.slice(value, 0, 0) == 1) {
            ans += 'w';
        }
        return ans;
    }

    private String fence() {
        switch (Functions.slice(value, 14, 12)) {
            case 0b000 -> {
                String pred = argsFence(Functions.slice(value, 27, 24));
                String succ = argsFence(Functions.slice(value, 23, 20));
                command = "fence";
                return String.format("%7s \t %s, %s", command, pred, succ);
            }
            case 0b001 -> {
                command = "fence.i";
                return String.format("%7s", command);
            }
        }
        return "";
    }

    private String other() {
        if (Functions.slice(value, 14, 12) != 0b000) {
            return csrType();
        }
        switch (Functions.slice(value, 31, 27)) {
            case 0b00000 -> {
                switch (Functions.slice(value, 24, 20)) {
                    case 0b00000 -> command = "ecall";
                    case 0b00001 -> command = "ebreak";
```

```
                    case 0b00010 -> command = "uret";
                }
            }
            case 0b00110 -> command = "mret";
            case 0b00010 -> {
                switch (Functions.slice(value, 26, 25)) {
                    case 0b00 -> {
                        switch (Functions.slice(value, 24, 20)) {
                            case 0b00010 -> command = "sret";
                            case 0b00101 -> command = "wfi";
                        }
                    }
                    case 0b01 -> {
                        String rs2 = Functions.getReg(Functions.slice(value,
24, 20));
                        String rs1 = Functions.getReg(Functions.slice(value,
19, 15));
                        command = "sfence.vma";
                        return String.format("%7s \t %s, %s", command, rs1,
rs2);
                    }
                }
            }
        }
        return String.format("%7s", command);
    }

    public String parseOpcode() {
        switch (getOpcode()) {
            case 0b0110111, 0b0010111 -> {
                return uType();
            }
            case 0b1100111, 0b0000011, 0b0010011 -> {
                return iType();
            }
            case 0b0100011 -> {
                return sType();
            }
            case 0b0110011 -> {
                return rType();
            }
            case 0b1101111 -> {
                return jType();
            }
            case 0b1100011 -> {
                return bType();
            }
            case 0b0001111 -> {
```

```
                return fence();
            }
            case 0b1110011 -> {
                return other();
            }
        }
        return "unknown_instruction";
    }

    @Override
    public String toString() {
        String out = String.format("   %05x: \t %08x \t", addr, value) +
parseOpcode();
        if (command.equals("unknown_instruction")) {
            out = "unknown_instruction";
        }
        if (hasMark()) {
            out += " " + getMark();
        }
        return out;
    }
}
```

Section.java

```
public class Section {
    private final int[] bytes;
    private final int[] sh_name = new int[4];
    private final int[] sh_type = new int[4];
    private final int[] sh_flags = new int[4];
    private final int[] sh_addr = new int[4];
    private final int[] sh_offset = new int[4];
    private final int[] sh_size = new int[4];
    private final int[] sh_link = new int[4];
    private final int[] sh_info = new int[4];
    private final int[] sh_addralign = new int[4];
    private final int[] sh_entsize = new int[4];
    private int byteIndex;

    public Section(int[] bytes) {
        this.bytes = bytes;
        initArray(sh_name);
        initArray(sh_type);
        initArray(sh_flags);
        initArray(sh_addr);
        initArray(sh_offset);
        initArray(sh_size);
```

```
        initArray(sh_link);
        initArray(sh_info);
        initArray(sh_addralign);
        initArray(sh_entsize);
    }

    private void initArray(int[] source) {
        for (int i = 0; i < source.length; i++) {
            source[i] = bytes[byteIndex];
            byteIndex++;
        }
    }


    public int getSh_name() {
        return Functions.bytesToInt(sh_name);
    }

    public int getSh_type() {
        return Functions.bytesToInt(sh_type);
    }

    public int getSh_flags() {
        return Functions.bytesToInt(sh_flags);
    }

    public int getSh_addr() {
        return Functions.bytesToInt(sh_addr);
    }

    public int getSh_offset() {
        return Functions.bytesToInt(sh_offset);
    }

    public int getSh_size() {
        return Functions.bytesToInt(sh_size);
    }

    public int getSh_link() {
        return Functions.bytesToInt(sh_link);
    }

    public int getSh_info() {
        return Functions.bytesToInt(sh_info);
    }

    public int getSh_addralign() {
        return Functions.bytesToInt(sh_addralign);
    }
```

```
    public int getSh_entsize() {
        return Functions.bytesToInt(sh_entsize);
    }


    public void out() {
        System.out.println(getSh_name() + "\t" +
                getSh_type() + "\t" +
                getSh_flags() + "\t" +
                getSh_addr() + "\t" +
                getSh_offset() + "\t" +
                getSh_size() + "\t" +
                getSh_link() + "\t" +
                getSh_info() + "\t" +
                getSh_addralign() + "\t" +
                getSh_entsize() + "\t");
    }

    public void out(String name) {
        System.out.println(name + "\t" +
                getSh_type() + "\t" +
                getSh_flags() + "\t" +
                getSh_addr() + "\t" +
                getSh_offset() + "\t" +
                getSh_size() + "\t" +
                getSh_link() + "\t" +
                getSh_info() + "\t" +
                getSh_addralign() + "\t" +
                getSh_entsize() + "\t");
    }
}
```

SectionHeader.java

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

public class SectionHeader {
    BinaryFile input;
    private final int sectionSize;
    private final int number;
    private final int stingTableIndex;
    private List<Section> sections;
    private final ParseString stringTable;


    public SectionHeader(BinaryFile input, ELFHeader elfHeader) {
        this.input = input;
```

```
        int start = elfHeader.getE_shoff();
        this.sectionSize = elfHeader.getE_shentsize();
        this.number = elfHeader.getE_shnum();
        this.stingTableIndex = elfHeader.getE_shstrndx();
        sections = new ArrayList<>();
        for (int i = 0; i < number; i++) {
            int startSection = start + i * sectionSize;
            sections.add(new Section(input.getBytes(startSection,
sectionSize)));
        }
        stringTable = getParseStringByInd(stingTableIndex);
    }


    public Section getSectionByName(String name) {
        for (int i = 0; i < sections.size(); i++) {
            Section section = sections.get(i);
            if (stringTable.bytesToString(section.getSh_name()).equals(name))
{
                return section;
            }
        }
        throw new IllegalArgumentException("There is no " + name);
    }

    public String getSectionName(int index) {
        return stringTable.bytesToString(getSection(index).getSh_name());
    }

    public ParseString getStringTable() {
        return stringTable;
    }

    public ParseString getParseStringByInd(int index) {
        Section section = sections.get(index);
        return new ParseString(input.getBytes(section.getSh_offset(),
                section.getSh_size()));
    }

    public Section getSection(int index) {
        try {
            return sections.get(index);
        } catch (IndexOutOfBoundsException e){
            throw new IllegalArgumentException("Section " + index + "not
found");
        }
    }
```

```
}
```

Symtab.java

```
import java.util.*;

public class Symtab {
    private final int[] bytes;
    private final Section symtabSection;
    private final int lineSize = 16;
    private final List<SymtabLine> lines;
    private final Map<Integer, String> convertFunctionName;
    private int functionNameIndex = 0;

    private final ParseString names;

    public Symtab(BinaryFile input, SectionHeader sectionHeader) {
        symtabSection = sectionHeader.getSectionByName(".symtab");
        bytes = input.getBytes(symtabSection.getSh_offset(),
symtabSection.getSh_size());
        lines = new ArrayList<>();
        convertFunctionName = new HashMap<>();
        names = sectionHeader.getParseStringByInd(symtabSection.getSh_link());
        for (int i = 0; i < bytes.length; i += lineSize) {
            lines.add(new SymtabLine(Arrays.copyOfRange(bytes, i, i +
lineSize), sectionHeader, names, i / lineSize));
        }
    }

    public String getFunctionNameByValue(int value) {
        for (int i = 0; i < lines.size(); i++) {
            SymtabLine line = lines.get(i);
            if (line.getValue() == value && line.getType().equals("FUNC")) {
                return line.getName();
            }
        }
        if (!convertFunctionName.containsKey(value)) {
            convertFunctionName.put(value, "L" + functionNameIndex);
            functionNameIndex++;
        }
        return convertFunctionName.get(value);
    }


    @Override
    public String toString() {
        StringBuilder out = new StringBuilder();
        out.append(".symtab");
        out.append(System.lineSeparator());
```

```
        out.append("Symbol Value              Size
Type     Bind     Vis       Index Name\n");
        for (int i = 0; i < lines.size(); i++) {
            out.append(lines.get(i));
            out.append(System.lineSeparator());
        }
        return out.toString();
    }
}
```

SymtabLine.java

```
public class SymtabLine {
    private final int[] bytes;
    private final int indexOfLine;
    private final int[] name = new int[4];
    private final int[] value = new int[4];
    private final int[] size = new int[4];
    private final int info;
    private final int other;
    private final int[] shndx = new int[2];
    private int byteIndex;
    private final SectionHeader sectionHeader;
    private final ParseString names;

    public SymtabLine(int[] bytes, SectionHeader sectionHeader, ParseString
names, int indexOfLine) {
        this.bytes = bytes;
        this.sectionHeader = sectionHeader;
        this.names = names;
        this.indexOfLine = indexOfLine;
        initArray(name);
        initArray(value);
        initArray(size);
        info = bytes[byteIndex++];
        other = bytes[byteIndex++];
        initArray(shndx);
    }

    @Override
    public String toString() {
        String out = String.format("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s",
getIndexOfLine(), getValue(), getSize(),
                getType(), getBind(), getVis(), getShndx(), getName()
        );
        return out;
    }
```

```
private void initArray(int[] source) {
    for (int i = 0; i < source.length; i++) {
        source[i] = bytes[byteIndex];
        byteIndex++;
    }
}

public String getName() {
    return names.bytesToString(Functions.bytesToInt(name));
}

public int getValue() {
    return Functions.bytesToInt(value);
}

public int getSize() {
    return Functions.bytesToInt(size);
}

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

public String getType() {
```

```
        int out = (((info) & 0xf));
        switch (out) {
            case 0 -> {
                return "NOTYPE";
            }
            case 1 -> {
                return "OBJECT";
            }

            case 2 -> {
                return "FUNC";
            }
            case 3 -> {
                return "SECTION";
            }
            case 4 -> {
                return "FILE";
            }
            case 5 -> {
                return "COMMON";
            }
            case 6 -> {
                return "TLS";
            }
            case 10 -> {
                return "LOOS";
            }
            case 12 -> {
                return "HIOS";
            }
            case 13 -> {
                return "LOPROC";
            }
            case 15 -> {
                return "HIPROC";
            }
        }
        return "";
    }

    public String getVis() {
        int out = ((other) & 0x3);
        switch (out) {
            case 0 -> {
                return "DEFAULT";
            }
            case 1 -> {
                return  "INTERNAL";
```

```
            }
            case 2 -> {
                return "HIDDEN";
            }
            case 3 -> {
                return "PROTECTED";
            }
        }
        return "";
    }

    public int getInfo() {
        return info;
    }

    public int getOther() {
        return other;
    }

    public String getShndx() {
        int index = Functions.bytesToInt(shndx);
        switch (index) {
            case 0 -> {
                return "UNDEF";
            }
            case 0xFF00 -> {
                return "LOPROC";
            }
            case 0xFF1F -> {
                return "HIPROC";
            }
            case 0xFFF1 -> {
                return "ABS";
            }
            case 0xFFF2 -> {
                return "COMMON";
            }
            case 0xFFFF -> {
                return "HIREVERVE";
            }
            default -> {
                return Integer.toString(index);
            }
        }

    }

    public int getIndexOfLine() {
```

```
            return indexOfLine;
        }
    }
```