Лабораторная работа №3	M3136	2022
ISA	Ястребов Григорий Дмитриевич	

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

Инструментарий и требования к работе: работа выполнена на Java 18 (Oracle OpenJDK 18.0.2).

Описание

1. Изучить систему кодирования команд RISC-V.
2. Изучить структуру elf файла.
3. Написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Вариант: отсутствует

Описание архитектуры RISC-V

RISC-V ISA – современная, бесплатная и общедоступная архитектура набора инструкций без упора в конкретную микроархитектуру, подходящая под реальную hardware-реализацию в 32- или 64-битном адресном пространстве, в том числе и с использованием нескольких ядер и поддержкой стандарта IEEE-754. Архитектура старается ничего не предписывать и быть максимально обобщенной, позволяющей разные имплементации.

В архитектуру заложен принцип модульности – RISC-V ISA разделена на небольшую базовую архитектуру для работы с целыми числами (2 основных варианта: RV32I и RV64I для соответствующих адресных пространств) и опциональные стандартные расширения (например, RV32M для поддержки целочисленного умножения и деления или RV32F для поддержки чисел с плавающей запятой).

Базовая архитектура RV32I

Базовая архитектура имеет зафиксированную длину инструкции в 32 бита с правильным выравниванием. Вид – Reg-Reg 3. Упакованный формат разрешает реализацию даже в 16 бит, но, как и расширенное кодирование в >32 бита, далее это не рассматривается.

RV32I, использующая 32-битное адресное пространство, имеет 32 регистра x0-x31 длиной 32 бита. Также есть 1 дополнительный регистр pc - program counter, который содержит адрес действующей инструкции. Отсутствует stack pointer и регистр адреса возврата, в качестве них используется x2 и x1(доп. x5).

В архитектуре 4 главных формата инструкций (R/I/S/U) и еще 2 (B/J) для работы с константой. В 32-битном коде инструкции содержится вся

необходимая информация о ней: регистр назначения (rd), аргумента 1 (rs1), аргумента 2 (rs2), константа (imm), код операции (opcode), дополнительные коды типа операции (funct3, funct7). Расположение номеров регистров и кодов операций не зависят от формата (см. рис 1).

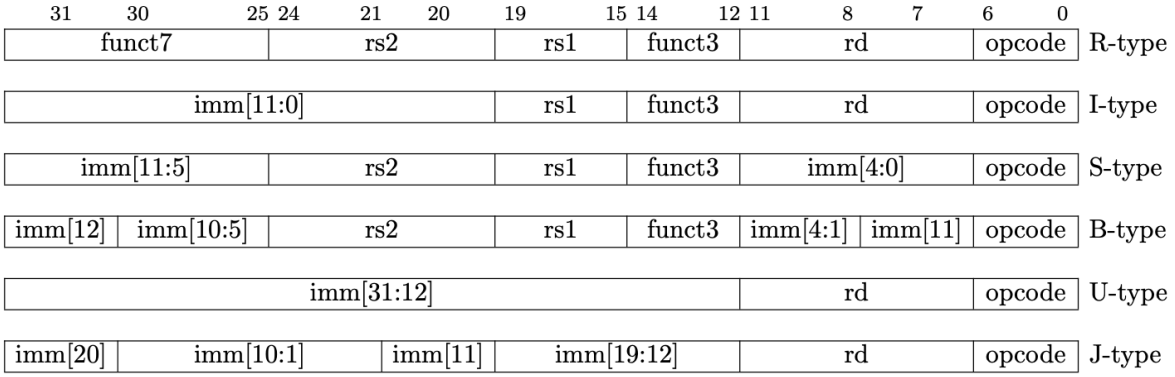

Рисунок 1 – 6 форматов инструкций базовой архитектуры.

Все константы (imm) знаковые, и бит знака есть самый левый бит из всех битов константы в коде инструкции (в частности, во всех форматах RV32I с константой это 31 бит). В форматах типа B и J константа есть частное при делении на два, поэтому наименьший ее бит в коде ставится на индекс 1 (чтобы лишний раз не сдвигать результат влево). Аналогичным образом в типе U константа сдвинута на 12 бит влево.

Краткое описание типов форматов и реализуемых инструкций

Формат типа R описывает операции регистр-регистр – чтение из rs1 и rs2, вычисление и запись в rd: арифметические ADD, SUB, логические AND, OR, XOR, знаковое/беззнаковое сравнение SLT/SLTU, сдвиги SLL, SRA, SRL (влево, вправо алгебраически и логически).

Тип I – регистр-константа, чтение константы imm и регистра rs1, вычисление, запись в rd: те же самые ADDI, SUBI, ANDI, ORI, XORI, SLTI/SLTIU, SLLI, SRAI, SRLI. Команда NOP, которая ничего не делает,

кодируется как ADDI x0, x0, 0. Также к I-формату относятся инструкции ECALL и EBREAK

Тип U отличается отсутствием rs1: загрузка верхней части константы LUI (для построения 32-битной константы), построение pc-зависимого адреса AUIPC (добавление к pc константы к старшим 20 битам).

J-формат реализуется только одной инструкцией JAL. В ней константа есть отступ, на который должен измениться адрес в pc, в rd записывает адрес следующей инструкции после прыжка. Еще есть JALR, который получает адрес пункта назначения прыжка как сумму константы и значения rs1 с округлением к четному.

Все инструкции ветвления реализованы форматом B-типа. Константа также хранит отступ для прыжка, регистры rs1 и rs2 держат сравниваемые значения. Прыжок происходит, если: в BEQ rs1 == rs2, в BNQ rs1 != rs2, в BLT/BLTU (знаковый/беззнаковый) rs1 < rs2, в BGT/BGTU rs1 > rs2, в BEQ rs1 == rs2. rd работает также, как и в J-типе.

Команды загрузки в память (STORE) реализует тип S-тип, из памяти (LOAD) – I-тип. Адрес памяти получается путем сложения константы отступа со значением rs1, в случае STORE в память загружаются нижние биты из rs2. SW, SH, SB хранят 32, 16, 8 бит соответственно, аналогично загружают LW, LH/LHU, LB/LBU (суффикс U указывает на то, что неиспользованные биты заполняются нулями, без суффикса – битом знака).

Инструкции ECALL и EBREAK реализуют I-формат, ничего не хранят в rs1 и rd, используются для отправки запроса в среду исполнения и остановки для дебага соответственно.

Все кодировки инструкций RV32I представлены на рисунке 2.

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Рисунок 2 – Кодировки инструкций базового набора RV32I.

RV32M

RV32M – стандартное расширение набора инструкций для выполнений целочисленного деления (DIV/DIVU), умножения (MUL, MULH/MULHU, MULSU) и взятия остатка (REM/REMU. Все инструкции в нем имеют R-формат (см. Рисунок 3).

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Рисунок 3 – Кодировки инструкций расширения RV32M.

Описание структуры ELF-файла

ELF (Executable and Linkable Format) определяет формат бинарного исполняемого файла (объектного файла) для некоторых операционных систем. В файле находятся непосредственно данные (в частности, инструкции процессора), необходимые для исполнения программы.

ELF-файл состоит из заголовка и последовательного массива секций, в конце файла находится табличка заголовков для каждой секции.

Данные самого ELF-файла представляются через типы Addr, Off, Word (размером 4 байта) и SWord, Half (размером 2 байта).

В заголовке ELF-файла содержится главная информация о разметке всего файла: отступ таблиц заголовков секций e_shoff в байтах, их размер и количество e_shentsize и e_shnum; виртуальный адрес e_entry, начиная с которого идет индексация инструкции; индекс таблицы строк в таблице секций e_shstrndx; дополнительная информация (например, архитектура процессора или его флаги).

Каждая секция содержит некоторый кластер данных об объектном файле. Например, .data – инициализированные данные с правами на чтение и запись, которые нужны для исполнения программы, .rodata – с правами только на чтение, .bss – неинициализированные, .strtab – массив строк-названий секций (массив последовательны нуль-терминированных строк), .text – инструкции, .symtab – таблица символов.

Каждая секция имеет заголовок (структура Elf32_Shdr), в полях которого лежит информация о нем. Например, sh_name – смещение от начала .strtab до начала строчки с соответствующим именем секции; sh_offset – смещение от начала файла до начала блока данных данной секции; sh_size – размер блока данных; sh_entsize – если блок данных

секции есть массив, то это – количество записей в нем. Все величины выражены в байтах.

Структура секций .text и .symtab

Секция .text – последовательные инструкций в виде массива байтов (в формате никак не описывается, как представлены инструкция, это уточняется спецификацией архитектуры). В нашем случае с RV32 все байты секции .text делятся на 4 байта на инструкцию.

В секции .symtab содержится информация о символьных определениях и ссылках в программе. Стандартно, это массив структур Elf32_Sym с полями:

- st_name – аналогично полю в заголовке секции, смещение от начала файла до начала блока данных данной секции;
- st_value – значение, ассоциированное с символом. В зависимости от контекста, может значить разные вещи (адрес, абсолютное значение, и т.п.)
- st_size – если символ ассоциирован с каким-то объектом, то это его размер в байтах, иначе 0.
- st_info – отсюда можно получить тип (st_info & 0xf) – общую классификацию ассоциированного с символом объекта, – и бинд-связку (st_info >> 4) – определитель поведения и видимости при линковке (см. таблицу 1).
- st_other – видимость (st_other & 0x3) – значение того, как можно получить доступ к этому символу после того, как он стал частью исполняемого или общего объекта (см. таблицу 1).
- st_shndx – символ определяется всегда в связи с какой-нибудь секцией, значение держит индекс заголовка соответствующей секции

в таблице заголовков секций. Причем, значения 0xfff1 – транслируются в “ABS”, 0xfff2 – в “COMMON”.

Например, в приложенном файле-примере есть символ с именем main, значением 0x10074 (адрес), размером 28 байт, типом FUNC, биндом GLOBAL, видимостью DEFAULT, и соответствующей секцией .text.

Ниже представлена таблица перевода значения бинда, типа и видимости в строчное имя.

Бинд		Тип		Видимость	
Значение	Имя	Значение	Имя	Значение	Имя
0	NOTYPE	0	LOCAL	0	DEFAULT
1	OBJECT	1	GLOBAL	1	INTERNAL
2	FUNC	2	WEAK	2	HIDDEN
3	SECTION	10	LOOS	3	PROTECTED
4	FILE	12	HIOS		
5	COMMON	13	LOPROC		
10	LOOS	15	HIPROC		
12	HIOS				
13	LOPROC				
15	HIPROC				

Таблица 1 – Имя/значение для бинда, типа и видимости

Описание работы программы-транслятора

Весь код находится в модуле src/main/java/rv3. В нем лежат классы Main, Labels, Elf32 и подмодуль riscv с классом InstructionTranslator, интерфейсом RiscvInstruction и перечислениями Type, RV32I, RV32M.

Main, Labels

Labels – класс, инкапсулирующий запись метки по адресу в словарь. Метод insert(addr, name) добавит в словарь название метки по адресу, а getLabel(addr) вернет его или создаст новую L-метку, если адреса в словаре нет.

Main - класс с запускаемым методом main. В нем обрабатываются ошибки при подаче аргументов, создается экземпляр elfFile класса Elf32, который затем читается и парсится из FileInputStream. Далее из elfFile мы достаем таблицу символов как массив объектов Elf32Sym, инструкции как массив int, виртуальный адрес. Затем создается экземлпяр Labels, в который мы загружаем метки из таблицы символов и инструкций (методы labelSymtab и labelInstructions). Наконец, через BufferedWriter в выходной файл записываются инструкции и таблица символов в необходимом формате через dumpInstructions и dumpSymtab соответственно.

В методе labelSymtab для каждого Elf32Sym находится имя и значение через инкапсулированные методы и записываются в labels. В labelInstructions с помощью методов InstructionTranslator из кода инструкции определяем ее тип (getInstruction, getType), и, если она типа J и B, достаем константы (get_typeImm).

Elf32

Класс парсит и хранит данные ELF-файла. В себе содержит классы заголовка секции Elf32Shdr, заголовка символа Elf32Sym и самого парсера ELF-файлов ElfParser.

В качестве данных содержит заголовок ELF-файла, таблицу заголовков секций sections, таблицу заголовков символов symtable, табличку строк-названий секций stringTable и табличку строк-названий символов symStringTable, int-массив инструкций instructions32.

В классах заголовка секции и заголовка символа есть метод getName, вызывающий getString от соответствующей таблички строк – метод по отступу читает строку до нулевого char.

Также в классе заголовка символа есть все методы, инкапсулирующие получение его данных для вывода. Для бинда, типа и видимости добавлены словари-переводчики в строчное имя.

Парсинг начинается вызовом метода from. Так как файл все равно необходимо читать до конца (в конце находится таблица заголовков секций), и почти все данные итак будут хранится в памяти класса Elf32, для удобства из файла сразу читаются все биты в массив data. Методы getInt и getShort извлекают из data по отступу один int или short соответственно.

Далее в методе parse читаются (parseHeader) данные заголовка, необходимые для парсинга далее (в парсере опущен парсинг почти всей ненужной информации): виртуальный адрес, индекс .strtab, отступ и размер shdr. По полученным данным теперь читается (parseSectionHeader) таблица заголовков секций: имя, отступ, размер, количество записей. В parseSections достаем таблицу строчек-имен по полученному ранее

индексу, и затем ищем нужные имена и парсим данные в секциях (parseInstructions, parseSymtable).

В parseInstructions просто поочередно записываем все инструкции в int и пишем в массив. В parseSymtable инициализируем массив размера размер / размер записи, циклом читаем символы – все 6 полей.

InstructionTranslator и пакет riscv

Интерфейс RiscvInstruction декларирует функции получения имени, типа (R/I/S/B/U/J) и маски кода инструкции (во всех полях кода кроме opcode, funct3 и funct7 лежат нули, для удобства восприятия блоки кодов разделены ‘_’, пример – `100_00000_0010011` для XORI). Перечисление Type как раз хранит 6 типов формата инструкции. Интерфейс реализуют перечисления RV32I и RV32M, хранящие записи об инструкциях.

InstructionTranslator – класс статических методов для трансляции инструкции в читаемый формат.

Метод getInstuction принимает код инструкции и возвращает запись о ней в перечислениях RV32I-RV32M (они статически загружены в словарь), перебирая маски кодов операций.

Метод subcode принимает код инструкции и границы полуинтервала [l, r), на котором он вернет через сдвиги вернет битовый подотрезок кода. Это пригодится для извлечения кусочков констант из кодов. Также есть signedSubcode, который сохранит знак у подотрезка (здесь важна только граница l).

Далее идут методы для извлечения rd, rs1, rs2, а также констант в зависимости от типа. registerToAbi возвращает ABI-формат представления регистров.

Результат работы программы-транслятора на примере приложенного файла

test/test_out

```
.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, 0x100
  10098:  fddff0ef      jal        ra, 10074 <main>
  1009c:  00050593      addi       a1, a0, 0
  100a0:  00a00893      addi       a7, zero, 10
  100a4:  0ff0000f      unknown_instruction
  100a8:  00000073      ecall
000100ac   <mmul>:
  100ac:  00011f37      lui        t5, 0x11
  100b0:  124f0513      addi       a0, t5, 292
  100b4:  65450513      addi       a0, a0, 1620
  100b8:  124f0f13      addi       t5, t5, 292
  100bc:  e4018293      addi       t0, gp, -448
  100c0:  fd018f93      addi       t6, gp, -48
  100c4:  02800e93      addi       t4, zero, 40
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      add        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
```

Список используемой литературы

1. https://riscv.org/technical/specifications/ – спецификация RISC-V
2. https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblj/index.html#chapter6-28341 – описание структуры ELF-файла

Листинг кода

src/main/java/rv3/Main.java

```
package rv3;

import java.io.*;
import rv3.riscv.RiscvInstruction;
import rv3.riscv.Type;

import static rv3.riscv.InstructionTranslator.*;

public class Main {
   public static void main(String[] args) {
       if (args.length == 0) {
           System.out.println("Can't find arguments");
           return;
       }
       if (!args[0].equals("rv3")) {
           System.out.println("Only rv3 is supported");
           return;
       }
       if (args.length != 3) {
           System.out.println("Usage: rv3 <input ELF file path> <disassembled
output file path>");
           return;
       }
       final Elf32 elfFile;
       try {
           try (final InputStream input = new FileInputStream(args[1])) {
               elfFile = Elf32.from(input);
           }
       } catch (FileNotFoundException e) {
           System.out.println("Unable to locate the input file: " +
e.getMessage());
           return;
       } catch (IOException e) {
           System.out.println("Unable to read the file: " + e.getMessage());
           return;
       }
       final Elf32.Elf32Sym[] symtab = elfFile.getSymtab();
       final int[] instructions = elfFile.getInstructions();
       final int virtualAddress = elfFile.getVirtualAddress();
       final Labels labels = new Labels();
       labelSymtab(symtab, labels);
       labelInstructions(instructions, labels, virtualAddress);
       try {
           try (final Writer writer = new BufferedWriter(
                   new OutputStreamWriter(new FileOutputStream(args[2])))) {
               dumpInstructions(writer, instructions, virtualAddress, labels);
               writer.write("\n");
               dumpSymtab(writer, elfFile.getSymtab());
           }
       } catch (FileNotFoundException e) {
           System.out.println("Unable to locate the output file: " +
e.getMessage());
       } catch (IOException e) {
```

```
            System.out.println("Unable to write to the file: " +
e.getMessage());
        }
    }

    private static void labelInstructions(int[] codes, Labels labels, int
virtualAddress) {
        for (int i = 0; i < codes.length; i++) {
            int curAddr = virtualAddress + i * 4;
            RiscvInstruction instr = getInstruction(codes[i]);
            if (instr == null) {
                continue;
            }
            if (instr.getType() == Type.J) {
                labels.getLabel(curAddr + getJtypeImm(codes[i]));
            } else if (instr.getType() == Type.B) {
                labels.getLabel(curAddr + getBtypeImm(codes[i]));
            }

        }
    }

    private static void labelSymtab(final Elf32.Elf32Sym[] symtab, final Labels
labels) {
        for (Elf32.Elf32Sym sym : symtab) {
            labels.insert(sym.getValue(), sym.getName());
        }
    }

    private static void dumpSymtab(final Writer writer, final Elf32.Elf32Sym[]
symtab)
            throws IOException {
        writer.write(".symtab\n");
        writer.write("Symbol Value              Size Type     Bind      Vis
Index   Name\n");
        for (int i = 0; i < symtab.length; i++) {
            writer.write(String.format("[%4d] 0x%-15x %5d %-8s %-8s %-8s %6s
%s\n",
                    i,
                    symtab[i].getValue(),
                    symtab[i].getSize(),
                    symtab[i].getType(),
                    symtab[i].getBind(),
                    symtab[i].getVisibility(),
                    symtab[i].getIndex(),
                    symtab[i].getName()
                    ));
        }
    }

    private static void dumpInstructions(final Writer writer, final int[]
codes,
                                         final int virtualAddress, final Labels
labels)
            throws IOException {
        writer.write(".text\n");
        for (int i = 0; i < codes.length; i++) {
            int code = codes[i];
            int curAddr = virtualAddress + i * 4;
```

```
            if (labels.containAddr(curAddr)) {
                writer.write(String.format("%08x   <%s>:\n",
                        curAddr,
                        labels.getLabel(curAddr)));
            }
            writer.write(String.format("   %05x:\t%08x\t\t", curAddr, code));
            final RiscvInstruction entry = getInstruction(code);
            if (entry == null) {
                writer.write("unknown_instruction\n");
                continue;
            }
            writer.write(String.format("%-7s\t",
entry.getName().toLowerCase()));
            writer.write(switch (entry.getType()) {
                case R -> String.format("%s, %s, %s\n",
                        registerToAbi(getRd(code)),
                        registerToAbi(getRs1(code)),
                        registerToAbi(getRs2(code))
                );
                case I -> {
                    final String name = entry.getName();
                    if (name.equals("ECALL") || name.equals("EBREAK")) {
                        yield "\n";
                    }
                    if (name.charAt(0) == 'L' || name.charAt(0) == 'J') {
                        yield String.format("%s, %s(%s)\n",
                                registerToAbi(getRd(code)),
                                getItypeImm(code),
                                registerToAbi(getRs1(code)));
                    }
                    yield String.format("%s, %s, %s\n",
                            registerToAbi(getRd(code)),
                            registerToAbi(getRs1(code)),
                            getItypeImm(code)
                    );
                }
                case S -> String.format("%s, %s(%s)\n",
                        registerToAbi(getRs2(code)),
                        getStypeImm(code),
                        registerToAbi(getRs1(code))
                );
                case B -> {
                    final int addr = getBtypeImm(code);
                    yield String.format("%s, %s, %x <%s>\n",
                            registerToAbi(getRs1(code)),
                            registerToAbi(getRs2(code)),
                            curAddr + addr,
                            labels.getLabel(curAddr + addr)
                    );
                }
                case U -> String.format("%s, 0x%x\n",
                        registerToAbi(getRd(code)),
                        getUtypeImm(code)
                );
                case J -> {
                    final int addr = getJtypeImm(code);
                    yield String.format("%s, %x <%s>\n",
                            registerToAbi(getRd(code)),
                            curAddr + addr,
```

```
                        labels.getLabel(curAddr + addr)
                    );
                }
            });
        }
    }
}
```

src/main/java/rv3/Elf32.java

```
package rv3;

import java.io.IOException;
import java.io.InputStream;
import java.util.Map;

public class Elf32 {
    // elf32 file header
//    private static final int EI_NIDENT = 16;
//    private String e_ident;
//    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 int e_flags;
//    private short e_ehsize;
//    private short e_phentsize;
//    private short e_phnum;
    private short e_shentsize;
    private short e_shnum;
    private short e_shstrndx;


    public class ELf32Shdr {
        private int      sh_name;
//        private int      sh_type;
//        private int      sh_flags;
//        private int      sh_addr;
        private int      sh_offset;
        private int      sh_size;
//        private int      sh_link;
//        private int      sh_info;
//        private int      sh_addralign;
        private int      sh_entsize;

        public String getName() {
            return getString(stringTable, sh_name);
        }
    }
    private ELf32Shdr[] sections;


    public class Elf32Sym {
        private int    st_name;
        private int    st_value;
        private int    st_size;
```

```
private char   st_info;
private char   st_other;
private short  st_shndx;

private static final Map<Integer, String> ST_TYPE = Map.ofEntries(
        Map.entry(0, "NOTYPE"),
        Map.entry(1, "OBJECT"),
        Map.entry(2, "FUNC"),
        Map.entry(3, "SECTION"),
        Map.entry(4, "FILE"),
        Map.entry(5, "COMMON"),
        Map.entry(10, "LOOS"),
        Map.entry(12, "HIOS"),
        Map.entry(13, "LOPROC"),
        Map.entry(15, "HIPROC")
);

private static final Map<Integer, String> ST_BIND = Map.of(
        0, "LOCAL",
        1, "GLOBAL",
        2, "WEAK",
        10, "LOOS",
        12, "HIOS",
        13, "LOPROC",
        15, "HIPROC"
);

private static final Map<Integer, String> ST_VISIBILITY = Map.of(
        0, "DEFAULT",
        1, "INTERNAL",
        2, "HIDDEN",
        3, "PROTECTED"
);
public int getValue() {
    return st_value;
}
public int getSize() {
    return st_size;
}
public String getType() {
    return ST_TYPE.get(st_info & 0xf);
}
public String getBind() {
    return ST_BIND.get(st_info >> 4);
}
public String getVisibility() {
    return ST_VISIBILITY.get(st_other & 0x3);
}
public String getName() {
    return getString(symStringTable, st_name);
}
public String getIndex() {
    if (st_shndx == 0) {
        return "UNDEF";
    }
    if (st_shndx == (short)0xfff1) {
        return "ABS";
    }
    if (st_shndx == (short)0xfff2) {
```

```
                return "COMMON";
            }
            return Short.toString(st_shndx);
        }
    }
    private Elf32Sym[] symtable;

    private String stringTable;
    private String symStringTable;

    private int[] instructions32;

    private String getString(final String table, final int offset) {
        int i = offset;
        while (table.charAt(i) != 0) {
            i++;
        }
        return table.substring(offset, i);
    }
    public int[] getInstructions() {
        return instructions32;
    }

    public Elf32Sym[] getSymtab() {
        return symtable;
    }

    public int getVirtualAddress() {
        return e_entry;
    }

    public static Elf32 from(final InputStream input) throws IOException {
        return new ElfParser().parse(input);
    }

    private static class ElfParser {
        private byte[] data;
        public Elf32 parse(final InputStream input) throws IOException {
            data = input.readAllBytes();
            final Elf32 file = new Elf32();
            parseHeader(file);
            file.sections = new ELf32Shdr[file.e_shnum];
            for (int i = 0; i < file.sections.length; i++) {
                file.sections[i] = file.new ELf32Shdr();
                parseSectionHeader(file.sections[i], file.e_shoff + i *
file.e_shentsize);
            }
            parseSections(file);
            return file;
        }

        private void parseSections(Elf32 file) {
            final ELf32Shdr shstr = file.sections[file.e_shstrndx];
            file.stringTable = new String(data, shstr.sh_offset,
shstr.sh_size);
            for (ELf32Shdr shdr : file.sections) {
                switch (shdr.getName()) {
                    case ".text" -> parseInstructions(file, shdr);
                    case ".strtab" -> parseSymStrTable(file, shdr);
```

```
                    case ".symtab" -> parseSymtable(file, shdr);
                }
            }
        }

        private void parseSymStrTable(Elf32 file, ELf32Shdr shdr) {
            file.symStringTable = new String(data, shdr.sh_offset,
shdr.sh_size);
        }

        private void parseSymtable(Elf32 file, ELf32Shdr shdr) {
            file.symtable = new Elf32Sym[shdr.sh_size / shdr.sh_entsize];
            for (int i = 0; i < file.symtable.length; i++) {
                file.symtable[i] = file.new Elf32Sym();
                parseSym(file.symtable[i], shdr.sh_offset + i *
shdr.sh_entsize);
            }
        }

        private void parseSym(Elf32Sym sym, final int offset) {
            sym.st_name = getInt(offset);
            sym.st_value = getInt(offset + 4);
            sym.st_size = getInt(offset + 8);
            sym.st_info = (char)data[offset + 12];
            sym.st_other = (char)data[offset + 13];
            sym.st_shndx = getShort(offset + 14);
        }

        private void parseInstructions(Elf32 file, ELf32Shdr shdr) {
            file.instructions32 = new int[shdr.sh_size / 4];
            for (int i = 0; i < file.instructions32.length; i++) {
                file.instructions32[i] = getInt(shdr.sh_offset + 4 * i);
            }
        }

        private void parseSectionHeader(final ELf32Shdr section, final int
offset) {
            section.sh_name = getInt(offset);
//            section.sh_type = getInt(offset + 4);
//            section.sh_flags = getInt(offset + 8);
//            section.sh_addr = getInt(offset + 12);
            section.sh_offset = getInt(offset + 16);
            section.sh_size = getInt(offset + 20);
//            section.sh_link = getInt(offset + 24);
//            section.sh_info = getInt(offset + 28);
//            section.sh_addralign = getInt(offset + 32);
            section.sh_entsize = getInt(offset + 36);
        }

        private void parseHeader(Elf32 elf) {
//            String e_ident = new String(Arrays.copyOfRange(data, 0,
elf.EI_INDENT));
//            elf.e_type = getShort(16);
//            elf.e_machine = getShort(18);
//            elf.e_version = getInt(20);
            elf.e_entry = getInt(24);
//            elf.e_phoff = getInt(28);
            elf.e_shoff = getInt(32);
//            int      elf.e_flags = getInt(36);
```

```
//            short    elf.e_ehsize = getShort(40);
//            short    elf.e_phentsize = getShort(42);
//            short    elf.e_phnum = getShort(44);
            elf.e_shentsize = getShort(46);
            elf.e_shnum = getShort(48);
            elf.e_shstrndx = getShort(50);
        }

        private int getInt(final int offset) {
            int result = 0;
            for (int i = 0; i < 4; i++) {
                result |= (data[i + offset] & 0xFF) << (i * 8);
            }
            return result;
        }

        private short getShort(final int offset) {
            short result = 0;
            for (int i = 0; i < 2; i++) {
                result |= (data[i + offset] & 0xFF) << (i * 8);
            }
            return result;
        }
    }
}
```

src/main/java/rv3/Labels.java

```
package rv3;

import java.util.HashMap;
import java.util.Map;

public class Labels {
    private int counter;
    private final Map<Integer, String> values;

    public Labels() {
        values = new HashMap<>();
        counter = 0;
    }

    public void insert(final int addr, final String name) {
        values.put(addr, name);
    }

    public String getLabel(final int addr) {
        if (values.containsKey(addr)) {
            return values.get(addr);
        }
        values.put(addr, "L" + counter++);
        return values.get(addr);
    }

    public boolean containAddr(final int addr) {
        return values.containsKey(addr);
    }
```

```
}
```

src/main/java/rv3/riscv/Type.java

```
package rv3.riscv;

public enum Type {
   R, I, S, B, U, J,
}
```

src/main/java/rv3/riscv/RiscvInstruction.java

```
package rv3.riscv;

public interface RiscvInstruction {
   String getName();
   Type getType();
   int getCode();
}
```

src/main/java/rv3/riscv/RV32I.java

```
package rv3.riscv;

public enum RV32I implements RiscvInstruction {
   LUI     (Type.U, 0b0110111),
   AUIPC   (Type.U, 0b0010111),

   JAL     (Type.J, 0b1101111),

   JALR    (Type.I, 0b000_00000_1100111),

   BEQ     (Type.B, 0b000_00000_1100011),
   BNE     (Type.B, 0b001_00000_1100011),
   BLT     (Type.B, 0b100_00000_1100011),
   BGE     (Type.B, 0b101_00000_1100011),
   BLTU    (Type.B, 0b110_00000_1100011),
   BGEU    (Type.B, 0b111_00000_1100011),

   LB      (Type.I, 0b000_00000_0000011),
   LH      (Type.I, 0b001_00000_0000011),
   LW      (Type.I, 0b010_00000_0000011),
   LBU     (Type.I, 0b100_00000_0000011),
   LHU     (Type.I, 0b101_00000_0000011),

   SB      (Type.S, 0b000_00000_0100011),
   SH      (Type.S, 0b001_00000_0100011),
   SW      (Type.S, 0b010_00000_0100011),

   ADDI    (Type.I, 0b000_00000_0010011),
   SLTI    (Type.I, 0b010_00000_0010011),
   SLTIU   (Type.I, 0b011_00000_0010011),
   XORI    (Type.I, 0b100_00000_0010011),
   ORI     (Type.I, 0b110_00000_0010011),
   ANDI    (Type.I, 0b111_00000_0010011),

   SLLI    (Type.R, 0b0000000_0000000000_001_00000_0010011),
```

```
    SRLI    (Type.R, 0b0000000_0000000000_101_00000_0010011),
    SRAI    (Type.R, 0b0100000_0000000000_101_00000_0010011),
    ADD     (Type.R, 0b0000000_0000000000_000_00000_0110011),
    SUB     (Type.R, 0b0100000_0000000000_000_00000_0110011),
    SLL     (Type.R, 0b0000000_0000000000_001_00000_0110011),
    SLT     (Type.R, 0b0000000_0000000000_010_00000_0110011),
    SLTU    (Type.R, 0b0000000_0000000000_011_00000_0110011),
    XOR     (Type.R, 0b0000000_0000000000_100_00000_0110011),
    SRL     (Type.R, 0b0000000_0000000000_101_00000_0110011),
    SRA     (Type.R, 0b0100000_0000000000_101_00000_0110011),
    OR      (Type.R, 0b0000000_0000000000_110_00000_0110011),
    AND     (Type.R, 0b0000000_0000000000_111_00000_0110011),

    ECALL   (Type.I, 0b000000000000_00000_000_00000_1110011),
    EBREAK  (Type.I, 0b000000000001_00000_000_00000_1110011),
    ;
    private final Type type;
    private final int code;

    RV32I(final Type type, final int code) {
        this.type = type;
        this.code = code;
    }

    @Override
    public String getName() {
        return name();
    }

    @Override
    public Type getType() {
        return type;
    }

    @Override
    public int getCode() {
        return code;
    }
}
```

src/main/java/rv3/riscv/RV32M.java

```
package rv3.riscv;

public enum RV32M implements RiscvInstruction {

    MUL     (Type.R, 0b0000001_0000000000_000_00000_0110011),
    MULH    (Type.R, 0b0000001_0000000000_001_00000_0110011),
    MULSU   (Type.R, 0b0000001_0000000000_010_00000_0110011),
    MULHU   (Type.R, 0b0000001_0000000000_011_00000_0110011),
    DIV     (Type.R, 0b0000001_0000000000_100_00000_0110011),
    DIVU    (Type.R, 0b0000001_0000000000_101_00000_0110011),
    REM     (Type.R, 0b0000001_0000000000_110_00000_0110011),
    REMU    (Type.R, 0b0000001_0000000000_111_00000_0110011),
    ;
    private final Type type;
    private final int code;
```

```
    RV32M(final Type type, final int code) {
        this.type = type;
        this.code = code;
    }

    @Override
    public String getName() {
        return name();
    }

    @Override
    public Type getType() {
        return type;
    }

    @Override
    public int getCode() {
        return code;
    }
}
```

src/main/java/rv3/riscv/InstructionTranslator.java

```
package rv3.riscv;

import java.util.HashMap;
import java.util.Map;

public class InstructionTranslator {
    private static final int OPCODE_MASK    =
0b00000000000000000000000001111111;
    private static final int FUNCT3_MASK    =
0b00000000000000000111000000000000;
    private static final int FUNCT7_MASK    =
0b11111110000000000000000000000000;

    private static final Map<Integer, RiscvInstruction> CODE_TO_INSTRUCTION;

    static {
        CODE_TO_INSTRUCTION = new HashMap<>();
        for (RV32I v : RV32I.values()) {
            CODE_TO_INSTRUCTION.put(v.getCode(), v);
        }
        for (RV32M v : RV32M.values()) {
            CODE_TO_INSTRUCTION.put(v.getCode(), v);
        }
    }

    public static RiscvInstruction getInstruction(final int code) {
        int codeMask = 0;
        if (CODE_TO_INSTRUCTION.containsKey(code & (codeMask | OPCODE_MASK))) {
            codeMask |= OPCODE_MASK;
            if (CODE_TO_INSTRUCTION.containsKey(code & (codeMask |
FUNCT3_MASK))) {
                codeMask |= FUNCT3_MASK;
                if (CODE_TO_INSTRUCTION.containsKey(code & (codeMask |
FUNCT7_MASK))) {
                    return CODE_TO_INSTRUCTION.get(code & codeMask);
```

```
            }
            return CODE_TO_INSTRUCTION.get(code & codeMask);
        }
        return CODE_TO_INSTRUCTION.get(code & codeMask);
    }
    return null;
}

private static int subcode(final int code, final int l, final int r) {
    return code << (32 - r) >>> (32 - r + l);
}

private static int signedSubcode(final int code, final int l) {
    return code >> l;
}


public static int getRd(final int code) {
    return subcode(code, 7, 12);
}

public static int getRs1(final int code) {
    return subcode(code, 15, 20);
}

public static int getRs2(final int code) {
    return subcode(code, 20, 25);
}

public static int getItypeImm(final int code) {
    return signedSubcode(code, 20);
}

public static int getStypeImm(final int code) {
    return subcode(code, 7, 12) | signedSubcode(code, 25) << 5;
}

public static int getBtypeImm(final int code) {
    return subcode(code, 8, 12) << 1
            | subcode(code, 25, 31) << 5
            | subcode(code, 7, 8) << 11
            | signedSubcode(code, 31) << 12;
}

public static int getUtypeImm(final int code) {
    return signedSubcode(code, 12);
}

public static int getJtypeImm(final int code) {
    return subcode(code, 21, 31) << 1
            | subcode(code, 20, 21) << 11
            | subcode(code, 12, 20) << 12
            | signedSubcode(code, 31) << 20;
}

public static String registerToAbi(final int register) {
    if (register == 0) {
        return "zero";
    } else if (register == 1) {
```

```
            return "ra";
        } else if (register == 2) {
            return "sp";
        } else if (register == 3) {
            return "gp";
        } else if (register == 4) {
            return "tp";
        } else if (register >= 5 && register <= 7) {
            return "t" + (register - 5);
        } else if (register >= 8 && register <= 9) {
            return "s" + (register - 8);
        } else if (register >= 10 && register <= 17) {
            return "a" + (register - 10);
        } else if (register >= 18 && register <= 27) {
            return "s" + (register - 16);
        } else if (register >= 28 && register <= 31) {
            return "t" + (register - 25);
        } else {
            return "x" + register;
        }
    }
}
```

elf/type/Elf32_Sword.java

```java
package elf.type;

public class Elf32_Sword implements Elf32_Type {
   private static final int SIZE = 2;
   private static final int ALIGNMENT = 2;

   @Override
   public int getSize() {
       return SIZE;
   }
}
```

elf/type/Elf32_Sword.java

```java
package elf.type;

public class Elf32_Word implements Elf32_Type {
   private static final int SIZE = 4;
   private static final int ALIGNMENT = 4;
   @Override
   public int getSize() {
       return SIZE;
   }
}
```

elf/Elf32_Ehdr.java

```java
package elf;

import elf.type.*;

public class Elf32_Ehdr {

   public static final int EI_NIDENT = 16;

   public static char[]          e_ident = new char[EI_NIDENT];
   public static Elf32_Half      e_type;
   public static Elf32_Half      e_machine;
   public static Elf32_Word      e_version;
   public static Elf32_Addr      e_entry;
   public static Elf32_Off       e_phoff;
   public static Elf32_Off       e_shoff;
   public static Elf32_Word      e_flags;
   public static Elf32_Half      e_ehsize;
   public static Elf32_Half      e_phentsize;
   public static Elf32_Half      e_phnum;
   public static Elf32_Half      e_shentsize;
   public static Elf32_Half      e_shnum;
   public static Elf32_Half      e_shstrndx;
}
```

elf/Elf32_Shdr.java

```
package elf;

import elf.type.*;

public class Elf32_Shdr {
   public static Elf32_Word      sh_name;
   public static Elf32_Word      sh_type;
   public static Elf32_Word      sh_flags;
   public static Elf32_Addr      sh_addr;
   public static Elf32_Off       sh_offset;
   public static Elf32_Word      sh_size;
   public static Elf32_Word      sh_link;
   public static Elf32_Word      sh_info;
   public static Elf32_Word      sh_addralign;
   public static Elf32_Word      sh_entsize;
}
```

elf/Elf32_Sym.java

```
package elf;

import elf.type.*;

public class Elf32_Sym {
   public static Elf32_Word    st_name;
   public static Elf32_Addr    st_value;
   public static Elf32_Word    st_size;
   public static char          st_info;
   public static char          st_other;
   public static Elf32_Half    st_shndx;
}
```

riscv/instruction/RiscvInstruction.java

```
package riscv.instruction;

public interface RiscvInstruction {
   String getName();
   Type getType();
   int getCode();
}
```

riscv/instruction/RV32I.java

```
package riscv.instruction;

public enum RV32I implements RiscvInstruction {
   LUI     (Type.U, 0b0110111),
   AUIPC   (Type.U, 0b0010111),

   JAL     (Type.J, 0b1101111),

   JALR    (Type.I, 0b000_00000_1100111),

   BEQ     (Type.B, 0b000_00000_1100011),
```

```
    BNE     (Type.B, 0b001_00000_1100011),
    BLT     (Type.B, 0b100_00000_1100011),
    BGE     (Type.B, 0b101_00000_1100011),
    BLTU    (Type.B, 0b110_00000_1100011),
    BGEU    (Type.B, 0b111_00000_1100011),

    LB      (Type.I, 0b000_00000_0000011),
    LH      (Type.I, 0b001_00000_0000011),
    LW      (Type.I, 0b010_00000_0000011),
    LBU     (Type.I, 0b100_00000_0000011),
    LHU     (Type.I, 0b101_00000_0000011),

    SB      (Type.S, 0b000_00000_0100111),
    SH      (Type.S, 0b001_00000_0100111),
    SW      (Type.S, 0b010_00000_0100111),

    ADDI    (Type.I, 0b000_00000_1100111),
    SLTI    (Type.I, 0b010_00000_1100111),
    SLTIU   (Type.I, 0b011_00000_1100111),
    XORI    (Type.I, 0b100_00000_1100111),
    ORI     (Type.I, 0b110_00000_1100111),
    ANDI    (Type.I, 0b111_00000_1100111),

    SLLI    (Type.R, 0b0000000_0000000000_001_00000_0010011),
    SRLI    (Type.R, 0b0000000_0000000000_101_00000_0010011),
    SRAI    (Type.R, 0b0100000_0000000000_101_00000_0010011),
    ADD     (Type.R, 0b0000000_0000000000_000_00000_0110011),
    SUB     (Type.R, 0b0100000_0000000000_000_00000_0110011),
    SLL     (Type.R, 0b0000000_0000000000_001_00000_0110011),
    SLT     (Type.R, 0b0000000_0000000000_010_00000_0110011),
    SLTU    (Type.R, 0b0000000_0000000000_011_00000_0110011),
    XOR     (Type.R, 0b0000000_0000000000_100_00000_0110011),
    SRL     (Type.R, 0b0000000_0000000000_101_00000_0110011),
    SRA     (Type.R, 0b0100000_0000000000_101_00000_0110011),
    OR      (Type.R, 0b0000000_0000000000_110_00000_0110011),
    AND     (Type.R, 0b0000000_0000000000_111_00000_0110011),

    ECALL   (Type.I, 0b000000000000_00000_000_00000_0110011),
    EBREAK  (Type.I, 0b000000000001_00000_000_00000_0110011),
    ;
    private final Type type;
    private final int code;

    RV32I(Type type, int code) {
        this.type = type;
        this.code = code;
    }

    @Override
    public String getName() {
        return name();
    }

    @Override
    public Type getType() {
        return type;
    }

    @Override
```

```
    public int getCode() {
        return code;
    }
}
```

riscv/instruction/RV32M.java

```
package riscv.instruction;

public enum RV32M implements RiscvInstruction {

    MUL(Type.R, 0b0000001_0000000000_000_00000_0110011),
    MULH(Type.R, 0b0000001_0000000000_001_00000_0110011),
    MULSU(Type.R, 0b0000001_0000000000_010_00000_0110011),
    MULHU(Type.R, 0b0000001_0000000000_011_00000_0110011),
    DIV(Type.R, 0b0000001_0000000000_100_00000_0110011),
    DIVU(Type.R, 0b0000001_0000000000_101_00000_0110011),
    REM(Type.R, 0b0000001_0000000000_110_00000_0110011),
    REMU(Type.R, 0b0000001_0000000000_111_00000_0110011),
    ;
    private final Type type;
    private final int code;

    RV32M(Type type, int code) {
        this.type = type;
        this.code = code;
    }

    @Override
    public String getName() {
        return name();
    }

    @Override
    public Type getType() {
        return type;
    }

    @Override
    public int getCode() {
        return code;
    }
}
```

riscv/instruction/Type.java

```
package riscv.instruction;

public enum Type {
    R, I, S, B, U, J,
}
```

rv3/BinaryParser.java

```java
package rv3;

import java.io.IOException;
import java.io.InputStream;

public class BinaryParser {
   private final InputStream input;

   public BinaryParser(InputStream input) {
       this.input = input;
   }

   public byte read() throws IOException {
       return (byte)input.read();
   }

   public byte[] read(int len) throws IOException {
       return input.readNBytes(len);
   }
}
```

rv3/BinaryParser.java

```java
package rv3;

import java.io.IOException;
import java.io.InputStream;

public class BinaryParser {
   private final InputStream input;

   public BinaryParser(InputStream input) {
       this.input = input;
   }

   public byte read() throws IOException {
       return (byte)input.read();
   }

   public byte[] read(int len) throws IOException {
       return input.readNBytes(len);
   }
}
```

rv3/BinaryTranslator.java

```java
package rv3;

public interface BinaryTranslator {
   static int bytesToInt(final byte[] bytes, int begin, int len) {
       if (len > 4) {
           throw new IllegalArgumentException("4 byte limit exceeded");
       }
       int result = 0;
       for (int i = 0; i < len; i++) {
           result |= (int) bytes[begin + i] << (i * 8);
       }
       return result;
   }
}
```

rv3/Labels.java

```java
package rv3;

import java.util.HashMap;
import java.util.Map;

public class Labels {
   private int counter;
   private final Map<Integer, String> values;

   public Labels() {
       values = new HashMap<>();
       counter = 0;
   }

   public void insert(final int addr, final String name) {
       values.put(addr, name);
   }

   public String getLabel(final int addr) {
       if (containAddr(addr)) {
           return values.get(addr);
       }
       return values.put(addr, "L" + counter++);
   }

   public boolean containAddr(final int addr) {
       return values.containsKey(addr);
   }
}
```

rv3/ElfParser.java

```java
package rv3;

import elf.Elf32_Ehdr;
import elf.Elf32_Shdr;
```

```
import elf.Elf32_Sym;
import elf.type.*;

import java.io.IOException;
import java.io.InputStream;
import java.util.Arrays;
import java.util.Map;

public class ElfParser extends BinaryParser {

   public ElfParser(InputStream input) {
       super(input);
   }

   private int headerSize;
   private int virtualAddress;
   private int sectionHeaderTableOffset;
   private int sectionHeaderTableEntryCount;
   private int sectionHeaderTableEntrySize;
   private int sectionHeaderStringTableIndex;
   private int sectionHeaderStringTableOffset;
   private int sectionHeaderStringTableSize;
   private byte[] headerToSectionHeaderGap;
   private byte[] sectionHeaderTable;
   private byte[] dotTextBytes;
   private int symtabOffset;
   private int symtabEntryCount;


   private void parse() throws IOException {
       parseHeader();
       headerToSectionHeaderGap = read(sectionHeaderTableOffset - headerSize);
       sectionHeaderTable =
         read(sectionHeaderTableEntryCount * sectionHeaderTableEntrySize);
       sectionHeaderStringTableOffset =
         getOffsetFromSectionHeader(sectionHeaderStringTableIndex);
       sectionHeaderStringTableOffset -= headerSize;
       for (int i = 0; i < sectionHeaderTableEntryCount; i++) {
           String name = getNameFromSectionHeader(i);
           if (name.equals(".text")) {
               int offset = getOffsetFromSectionHeader(i) - headerSize;
               dotTextBytes = Arrays.copyOfRange(
                       headerToSectionHeaderGap,
                       offset,
                       offset + getSizeFromSectionHeader(i)
               );
           } else if (name.equals(".symtab")) {
               symtabOffset = getOffsetFromSectionHeader(i) - headerSize;
               symtabEntryCount =
                getSizeFromSectionHeader(i) /
             getEntrySizeFromSectionHeader(i);
           }
       }
   }


   private int read(Elf32_Type type) throws IOException {
       return BinaryTranslator.bytesToInt(
```

```
                read(Elf32_Ehdr.e_shnum.getSize()), 0,
Elf32_Ehdr.e_shnum.getSize());
    }

    private void parseHeader() throws IOException {
        read(Elf32_Ehdr.e_type);
        read(Elf32_Ehdr.e_machine);
        read(Elf32_Ehdr.e_version);
        virtualAddress = read(Elf32_Ehdr.e_entry);
        read(Elf32_Ehdr.e_phoff);
        sectionHeaderTableOffset = read(Elf32_Ehdr.e_shoff);
        read(Elf32_Ehdr.e_flags);
        headerSize = read(Elf32_Ehdr.e_ehsize);
        read(Elf32_Ehdr.e_phentsize);
        read(Elf32_Ehdr.e_phnum);
        sectionHeaderTableEntrySize = read(Elf32_Ehdr.e_shentsize);
        sectionHeaderTableEntryCount = read(Elf32_Ehdr.e_shnum);
        sectionHeaderStringTableIndex = read(Elf32_Ehdr.e_shstrndx);
    }

    private String getNameFromSectionHeader(int entryId) {
        int nameOffset = BinaryTranslator.bytesToInt(
                sectionHeaderTable,
                entryId * sectionHeaderTableEntrySize,
                Elf32_Shdr.sh_name.getSize()
        );
        return getStringFromShstr(nameOffset);
    }

    private String getStringFromShstr(final int offset) {
        var sb = new StringBuilder();
        int curCharId = sectionHeaderStringTableOffset - headerSize + offset;
        while (headerToSectionHeaderGap[curCharId] != 0) {
            sb.append(headerToSectionHeaderGap[curCharId]);
        }
        return sb.toString();
    }

    private int getOffsetFromSectionHeader(int entryId) {
        int j = Elf32_Shdr.sh_name.getSize()
                + Elf32_Shdr.sh_type.getSize()
                + Elf32_Shdr.sh_flags.getSize()
                + Elf32_Shdr.sh_addr.getSize();
        return BinaryTranslator.bytesToInt(
                sectionHeaderTable,
                entryId * sectionHeaderTableEntrySize + j,
                Elf32_Shdr.sh_offset.getSize()
        );
    }

    private int getSizeFromSectionHeader(int entryId) {
        int j = Elf32_Shdr.sh_name.getSize()
                + Elf32_Shdr.sh_type.getSize()
                + Elf32_Shdr.sh_flags.getSize()
                + Elf32_Shdr.sh_addr.getSize()
                + Elf32_Shdr.sh_offset.getSize();
        return BinaryTranslator.bytesToInt(
                sectionHeaderTable,
                entryId * sectionHeaderTableEntrySize + j,
```

```
            Elf32_Shdr.sh_size.getSize()
    );
}

private int getEntrySizeFromSectionHeader(int entryId) {
    int j = Elf32_Shdr.sh_name.getSize()
            + Elf32_Shdr.sh_type.getSize()
            + Elf32_Shdr.sh_flags.getSize()
            + Elf32_Shdr.sh_addr.getSize()
            + Elf32_Shdr.sh_offset.getSize()
            + Elf32_Shdr.sh_link.getSize()
            + Elf32_Shdr.sh_info.getSize()
            + Elf32_Shdr.sh_addralign.getSize();
    return BinaryTranslator.bytesToInt(
            sectionHeaderTable,
            entryId * sectionHeaderTableEntrySize + j,
            Elf32_Shdr.sh_entsize.getSize()
    );
}

public byte[] getInstructionsBytes() throws IOException {
    return dotTextBytes;
}

private static final Map<Integer, String> ST_TYPE = Map.ofEntries(
        Map.entry(0, "NOTYPE"),
        Map.entry(1, "OBJECT"),
        Map.entry(2, "FUNC"),
        Map.entry(3, "SECTION"),
        Map.entry(4, "FILE"),
        Map.entry(5, "COMMON"),
        Map.entry(10, "LOOS"),
        Map.entry(12, "HIOS"),
        Map.entry(13, "LOPROC"),
        Map.entry(15, "HIPROC")
);

private static final Map<Integer, String> ST_BIND = Map.of(
        0, "LOCAL",
        1, "GLOBAL",
        2, "WEAK",
        10, "LOOS",
        12, "HIOS",
        13, "LOPROC",
        15, "HIPROC"
);

private static final Map<Integer, String> ST_VISIBILITY = Map.of(
        0, "DEFAULT",
        1, "INTERNAL",
        2, "HIDDEN",
        3, "PROTECTED"
);

public String[] parseSymTableWithLabels(Labels labels) throws IOException {
    String[] result = new String[symtabEntryCount + 1];
    result[0] =
       "Symbol Value              Size Type  Bind   Vis    Index Name\n";
    for (int i = 0; i < symtabEntryCount; i++) {
```

```
            int byteId = symtabOffset;
            String name = getStringFromShstr(BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId,
Elf32_Sym.st_name.getSize()
            ));
            byteId += Elf32_Sym.st_name.getSize();
            int value = BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId,
                        Elf32_Sym.st_value.getSize()
            );
            byteId += Elf32_Sym.st_value.getSize();
            int size = BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId,
Elf32_Sym.st_size.getSize()
            );
            byteId += Elf32_Sym.st_size.getSize();
            int info = BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId, 1
            );
            byteId++;
            String type = ST_TYPE.get(info & 0xf);
            String bind = ST_BIND.get(info >> 4);
            int other = BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId, 1
            );
            String vis = ST_VISIBILITY.get(other & 0x3);
            byteId++;
            int index = BinaryTranslator.bytesToInt(
                    headerToSectionHeaderGap, byteId,
                 Elf32_Sym.st_shndx.getSize()
            );
            result[i] =
                 String.format("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                    i,
                    value,
                    size,
                    type,
                    bind,
                    vis,
                    index,
                    name
            );
            labels.insert(value, name);
        }
        return result;
    }

    public int getVirtualAdress() {
        return virtualAddress;
    }
}
```

rv3/InstructionTranslator.java

```
package rv3;

import riscv.instruction.RV32I;
import riscv.instruction.RV32M;
```

```java
import riscv.instruction.RiscvInstruction;

import java.util.HashMap;
import java.util.Map;

public class InstructionTranslator {
   public static String[] translateInstructionSet32(final byte[] bytes,
                                                    final int virtualAddress,
                                                       Labels labels) {
       final int instructionSetSize = bytes.length / 4;
       String[] instructionSet = new String[instructionSetSize];
       for (int byteIndex = 0; byteIndex < bytes.length; byteIndex += 4) {
           instructionSet[byteIndex / 4] = translateInstruction(
                   BinaryTranslator.bytesToInt(bytes, byteIndex, 4),
                   virtualAddress + byteIndex, labels
           );
       }
       return instructionSet;
   }

   private static final int OPCODE_MASK    =
0b00000000000000000000000001111111;
   private static final int RD_MASK        =
0b00000000000000000000111110000000;
   private static final int IMM5_MASK      =
0b00000000000000000000111110000000;
   private static final int FUNCT3_MASK    =
0b00000000000000000111000000000000;
   private static final int RS1_MASK       =
0b00000000000011111000000000000000;
   private static final int RS2_MASK       =
0b00000001111100000000000000000000;
   private static final int FUNCT7_MASK    =
0b11111110000000000000000000000000;
   private static final int IMM7_MASK      =
0b11111110000000000000000000000000;
   private static final int IMM12_MASK     =
0b11111111111100000000000000000000;
   private static final int IMM20_MASK     =
0b11111111111111111111000000000000;

   private static final Map<Integer, RiscvInstruction> CODE_TO_INSTRUCTION;
   static {
       CODE_TO_INSTRUCTION = new HashMap<>();
       for (RV32I v : RV32I.values()) {
           CODE_TO_INSTRUCTION.put(v.getCode(), v);
       }
       for (RV32M v : RV32M.values()) {
           CODE_TO_INSTRUCTION.put(v.getCode(), v);
       }
   }

   private static RiscvInstruction getEntry(final int instruction) {
       int codeMask = OPCODE_MASK;
       if (CODE_TO_INSTRUCTION.containsKey(instruction & codeMask)) {
           return CODE_TO_INSTRUCTION.get(instruction & codeMask);
       }
       codeMask |= FUNCT3_MASK;
       if (CODE_TO_INSTRUCTION.containsKey(instruction & codeMask)) {
```

```
            return CODE_TO_INSTRUCTION.get(instruction & codeMask);
        }
        codeMask |= FUNCT7_MASK;
        if (CODE_TO_INSTRUCTION.containsKey(instruction & codeMask)) {
            return CODE_TO_INSTRUCTION.get(instruction & codeMask);
        }
        return null;
    }

    private static String translateInstruction(final int instruction,
                                               final int address,
                                              final Labels labels) {
        RiscvInstruction entry = getEntry(instruction);
        String result = String.format("   %05x:\t%08x\t", address,
instruction);
        if (entry == null) {
            return result + "unknown_instruction";
        }
        return result
                + String.format("%7s\t", entry.getName())
                + switch (entry.getType()) {
            case R -> String.format("%s, %s, %s\n",
                    registerToAbi(getRd(instruction)),
                    registerToAbi(getRs1(instruction)),
                    getRs2(instruction)
            );
            case I -> String.format("%s, %s, %s\n",
                    registerToAbi(getRd(instruction)),
                    registerToAbi(getRs1(instruction)),
                    getItypeImm(instruction)
            );
            case S -> String.format("%s, %s(%s)\n",
                    registerToAbi(getRs1(instruction)),
                    getStypeImm(instruction),
                    registerToAbi(getRs2(instruction))
            );
            case B -> {
                int addr = getBtypeImm(instruction);
                yield String.format("%s, %s, %s <%s>\n",
                        registerToAbi(getRs1(instruction)),
                        registerToAbi(getRs2(instruction)),
                        Integer.toHexString(addr),
                        labels.getLabel(addr)
                );
            }
            case U -> String.format("%s, %s\n",
                    registerToAbi(getRd(instruction)),
                    getUtypeImm(instruction)
            );
            case J -> {
                int addr = getJtypeImm(instruction);
                yield String.format("%s, %s <%s>\n",
                        registerToAbi(getRd(instruction)),
                        Integer.toHexString(addr),
                        labels.getLabel(addr)
                );
            }
        };
    }
```

```
private static int getRd(int instruction) {
    return instruction & RD_MASK;
}

private static int getRs1(int instruction) {
    return instruction & RS1_MASK;
}

private static int getRs2(int instruction) {
    return instruction & RS2_MASK;
}

private static int getItypeImm(int instruction) {
    return (instruction & IMM12_MASK) >>> 19;
}

private static int getStypeImm(int instruction) {
    return (instruction & IMM5_MASK) >>> 7
            | (instruction & IMM7_MASK) >>> (32 - 1 - 5);
}

private static int getBtypeImm(int instruction) {
    return instruction << 20              >>> (32 - 4)
                   | instruction << 1         >>> (32 - (6 + 4))
                   | instruction << (20 + 4) >>> (32 - (1 + 6 + 4))
                   | instruction             >>> (32 - (1 + 1 + 6 + 4));
}

private static int getUtypeImm(int instruction) {
    return (instruction & IMM20_MASK) >>> (32 - 20);
}

private static int getJtypeImm(int instruction) {
    return instruction << 1            >>> (32 - 10)
            | instruction << (1 + 10)      >>> (32 - (1 + 10))
            | instruction << (1 + 10 + 11) >>> (32 - (8 + 10 + 1))
            | instruction                  >>> (32 - (1 + 10 + 1 + 8));
}

private static String registerToAbi(int register) {
    if (register == 0) {
        return "zero";
    } else if (register == 1) {
        return "ra";
    } else if (register == 2) {
        return "sp";
    } else if (register == 3) {
        return "gp";
    } else if (register == 4) {
        return "tp";
    } else if (register >= 5 && register <= 7) {
        return "t" + (register - 5);
    } else if (register >= 8 && register <= 9) {
        return "s" + (register - 8);
    } else if (register >= 10 && register <= 17) {
        return "a" + (register - 10);
    } else if (register >= 18 && register <= 27) {
        return "s" + (register - 16);
```

```
        } else if (register >= 28 && register <= 31) {
            return "t" + (register - 25);
        } else {
            return "x" + register;
        }
    }
}
```

rv3/Main.java

```
package rv3;

import java.io.*;

public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Can't find arguments");
            return;
        }
        if (!args[0].equals("rv3")) {
            System.out.println("Only rv3 is supported");
            return;
        }
        if (args.length != 3) {
             System.out.println("Usage: rv3 <input ELF file path> <disassembled
output file path>");
            return;
        }
        byte[] instructions;
        String[] symTable;
        var labels = new Labels();
        int virtualAddress;
        try {
            try (InputStream stream = new FileInputStream(args[1])) {
                ElfParser parser = new ElfParser(stream);
                instructions = parser.getInstructionsBytes();
                symTable = parser.parseSymTableWithLabels(labels);
                virtualAddress = parser.getVirtualAdress();
            }
        } catch (FileNotFoundException e) {
                System.out.println("Unable to locate the input file: " +
e.getMessage());
            return;
        } catch (IOException e) {
            System.out.println("Unable to read the file: " + e.getMessage());
            return;
        }
        String[] disassembledInstructions =
                InstructionTranslator.translateInstructionSet32(instructions,
virtualAddress, labels);
        try {
            try (BufferedWriter writer = new BufferedWriter(
                    new OutputStreamWriter(new FileOutputStream(args[2])))) {
                writer.write(".text");
                writer.newLine();
                for (int i = 0; i < disassembledInstructions.length; i++) {
                    int addr = virtualAddress + i * 4;
```

```
                    if (labels.containAddr(addr)) {
                        writer.write(String.format("%08x   <%s>:\n",
                                addr,
                                labels.getLabel(addr)));
                        writer.newLine();
                    }
                    writer.write(disassembledInstructions[i]);
                    writer.newLine();
                }
                writer.write(".symtab");
                writer.newLine();
                for (String s : symTable) {
                    writer.write(s);
                    writer.newLine();
                }
            }
        } catch (FileNotFoundException e) {
                  System.out.println("Unable to locate the output file: " +
e.getMessage());
        } catch (IOException e) {
                    System.out.println("Unable to write to the file: " +
e.getMessage());
        }
    }
}
```