САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

«ISA»

Выполнил(а): Ступников Александр Сергеевич

студ. гр. M3135

Санкт-Петербург

2020

Цель работы: знакомство со системой набора команд RISC-V.

Инструментарий и требования к работе: работа может быть выполнена на любом из следующих языков: C, C++, Python, Java.

Теоретическая часть

Описание системы кодирования команд RISC-V

RISC-V ISA представляет из себя базовую ISA, реализующую операции для работы с целыми числами, и дополнительные расширения. База должна присутствовать в любой реализации RISC-V, расширения же опциональны. Вообще, если быть точным, RISC-V это не просто одна ISA, а несколько связанных, всего существует четыре базовых ISA RISC-V. Основные из них RV32I и RV64I, которые позволяют работать с 32-битными и 64-битными адресами соответственно. Дальше речь будет идти именно о RV32I.

Базовая ISA RV32I содержит инструкции по работе с целыми числами, запись в память, чтение из памяти, controlflow инструкции. Стандартное расширение, реализующее целочисленное деление и умножение называется “M”.

В RV32I есть 32 регистра, каждый из них имеет размер 32 бита. Регистр x0 всегда хранит значение 0. Также существует один дополнительный регистр – pc, хранящий адрес текущей инструкции.

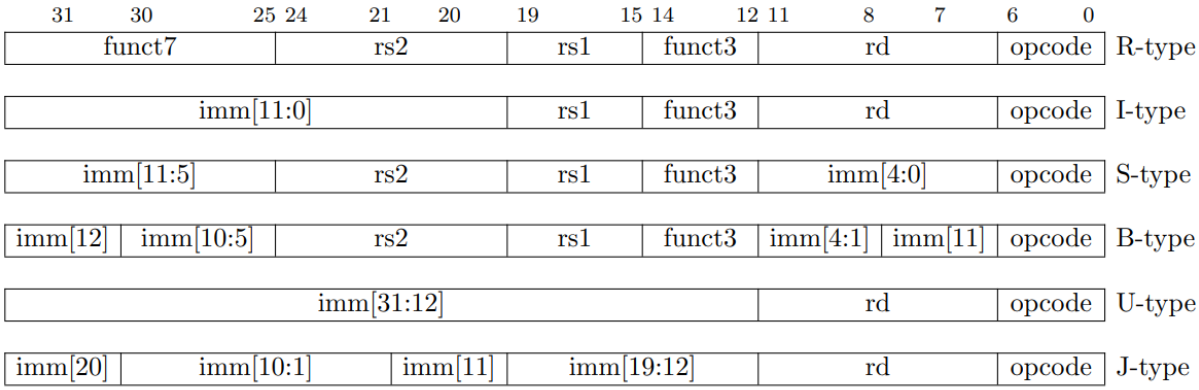

Рисунок №1 – базовые форматы инструкций RV32I

На рисунке 1 изображены базовые форматы инструкций. Для каждого поля immediate (константы) в квадратных скобках указаны позиции битов внутри immediate, а не позиция битов внутри инструкции, как это делается обычно.

Заметим, что RISC-V ISA хранит source (rs1 и rs2) и destination (rd) регистры в одних и тех же позиция для всех форматов инструкций. Immediates всегда sign-extended, бит знака всегда находится в 31 бите инструкции. В RV32I ISA все инструкции имеют фиксированную длину, равную 32 бита, и должны быть выравнены с четырёхбайтовым отступом в памяти.

В формате B immediate сдвинута на один бит влево, нулевой бит immediate в инструкциях B формата всегда равен 0. Аналогично для инструкций U и J формата immediate сдвинута влево на 12 и 1 бит соответственно. Эта информация обобщена на рисунке №2.

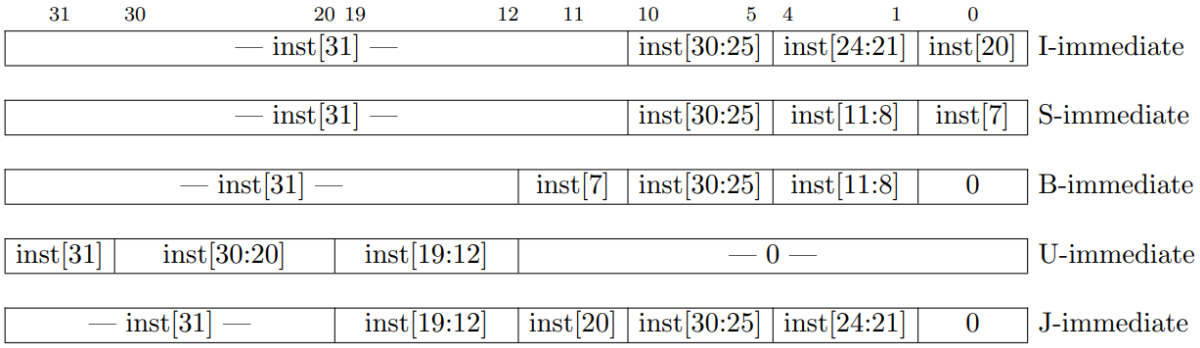

Рисунок №2 – соответствие битов immediate (подписаны сверху) битам инструкции (inst[x])

Почти все инструкции относятся к одному из 6 типов и декодируются понятным образом. Однако есть инструкции, выбивающиеся из общего числа, их структура приведена на рисунке №3.

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

31 20	19 15	14 12	11 7	6 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

Рисунок №3 – специализированные инструкции

Заметим, что все инструкции, из расширения RV32M, относятся к R-типу и декодируются тривиальным образом.

Обзор структуры ELF файла

The Executable and Linkable Format, или ELF, это типичный формат исполняемых файлов в Linux системах. ELF файлы составлены из трёх основных частей:

1. ELF Header (заголовок файла)
2. Sections (секции)
3. Segments (сегменты)

Elf Header

ELF header представляет из себя Elfxx_Ehdr структуру (см. рисунок №4). Header содержит основную информацию о бинарном файле. Рассмотрим некоторые поля ELF header’a.

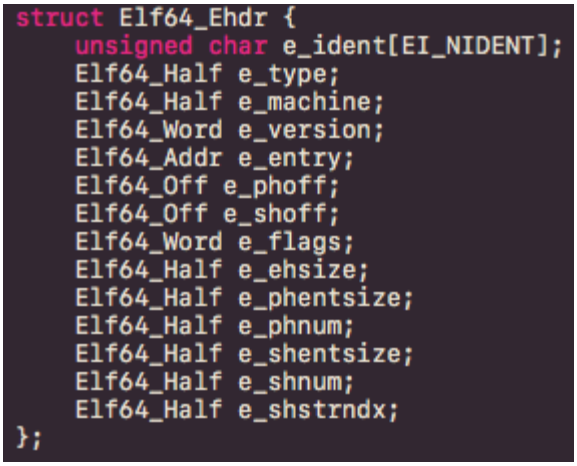

Рисунок №4 – ELF Header

- e_ident: 16 байтовый массив, содержащий идентификационную информацию о файле, необходим для кодирования и интерпретации содержимого файла. В e_ident помимо прочего входит поле EI_MAG0-3, содержащее ELF magic numbers. Эти числа показывают, что данные файл это ELF.
- e_type: тип файла, например, ET_REL, ET_EXEC и т.д.
- e_shoff: файловый отступ (далее offset) таблицы заголовков секций (Section Header Table).
- e_shentsize: размер каждого Section Header в Section Header Table.
- e_shnum: Число Section Header'ов.
- e_shstrndx: номер секции в Section Header Table, хранящей информацию о именах секций.

Sections

Секции хранят основную информацию ELF файла (и что самое главное код). Информацию о секциях хранит Section Header Table. Эта таблица представляет собой массив из Elfxx_Shdr структур. Каждой секции сопоставлен единственный заголовок. Рассмотрим некоторые поля Section Header'а (см. рисунок №5).

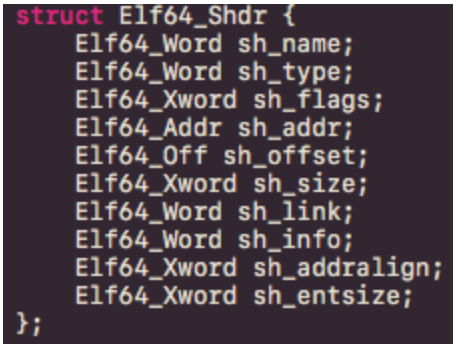

Рисунок №5 – Section Header

- sh_name: индекс имени секции в string table (offset от начала string table)
- sh_addr: виртуальный адрес секции.
- sh_offset: offset секции в файле.

Некоторые распространённые секции:

- .text: секция с кодом.
- .symtab: таблица символов.
- .strtab: таблица строк секции .symtab (может также являться таблицей строк и для Section Headers).

Segments

Сегменты, также известные, как Program Headers, разбивают ELF файл на части, удобные для загрузки в память. Но в рамках поставленной задачи они не слишком интересны (в файле test_elf их вообще нет).

ELF Symbol structure

В процессе написания программ, часто используются имена для обращения к определённым объектам в коде, таким как функции и переменные. Информация о таких именах называется program’s symbolic information. Для её хранения в ELF файле используется Symbol Table. В качестве конкретного примера информации, хранящейся в Symbol Table, можно привести метки ассемблера.

В ELF файле каждый символ представляет собой экземпляр Elfxx_Sym структуры, находящей внутри Symbol Table (см. рисунок №6).

```
struct Elf64_Sym{
    Elf64_Word      st_name;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf64_Half      st_shndx;
    Elf64_Addr      st_value;
    Elf64_Xword     st_size;
};
```

Рисунок №6 – символ

Рассмотрим некоторые поля этой структуры.

- st_name: индекс имени символа в string table. Если поле пусто, символ не имеет имени.
- st_info: содержит bind и тип символа.
- st_other: информация о видимости символа.
- st_shndx: каждый символ относится к некоторой секции. Значение st_shndx определяет индекс секции в Section Header Table, к которой относится символ. Например, метки ассемблера – это символы, относящиеся к секции кода .text.
- st_value: определяет значение символа. Интерпретация этого поля может быть разной в зависимости от конкретного символа. Например, если символ относится к секции .text, то это поле показывает виртуальный адрес инструкции, к которой относится метка. (Для файла test_elf это просто отступ от начала .text, так как тип файла ET_REL).
- st_size: размер символа.

Практическая часть

Начиная практическую часть, хочу заметить, что в теоретической части я постарался сделать акцент на том, что необходимо для решения поставленной задачи. Моменты, не слишком важные для решения, были описаны лишь в общих чертах.

Для начала я постараюсь подробно описать алгоритм решения задачи, не углубляясь в особенности реализации.

Стоит сказать, что далее, когда я использую слова «извлечь из полей», «посмотреть на значение в поле», это значит, что offset извлекаемых данных нам уже известен. Либо заранее, как в случае с elf header, либо на основании предыдущих действий.

Итак, чтобы решить задачу нужно

1. Извлечь из header'а файла значения полей e_shoff, e_shentsize, e_shentnum, e_shtrndx.
2. Пойти в байт с отступом e_shoff + e_shtrndx * e_shentsize. Там храниться header секции .shstrtab (она может называться просто .strtab, но это неважно, так как у нас есть индекс заголовка этой секции e_shtrndx). Смотрим в поле s_offset и находим таким образом откуда начинается таблица имён секций (Section header string table). Назовём отступ этой таблицы SectionHeaderStringTableOffset.
3. Далее перемещаемся в байт с отступом e_shoff и идём по всем Section Header, пропуская e_shentsize байт. В каждом header смотрим на первые четыре байта, где хранится значение поля s_name, которое показывает offset имени секции, header которой мы рассматриваем, в section header string table (локальный отступ относительно начала section header string table). Проверяем, равны ли 6 подряд идущих байт, где у первого из них offset = s_name + SectionHeaderStringTableOffset в точности 0x2E 0x74 0x65 0x78 0x74 0x00, т.е. ASCII коду строки ".text" после которого записан байт 0x00.
4. Когда нашли header секции с именем .text, достаём из него значения полей s_offset и s_size, которые равны соответственно отступу и размеру секции исполняемого кода в байтах.
5. Теперь можно достать из elf файла часть, в которой записан этот код и отдать его парсеру.

После этого необходимо запарсить Symbol table. Там храняться метки ассемблера и не только, но нас интересуют только метки.

1. Находим заголовок секции с именем .symtab (это делается так же, как для секции .text).
2. Достаём из заголовка значения полей s_offset, s_size, s_entsize, которые показывают соответственно отступ секции symbol table, её размер и размер блока, хранящего информацию об одном символе внутри symbol table.
3. Имея эту информацию, пройдёмся по всем символам из symbol table. Первые четыре байта каждого символа хранят значение поля sym_name, имя символа, они показывают отступ имени символа в string table. Но чтобы получить имя символа нужно найти отступ string table (не путать с section header string table, эти таблицы могут отличаться). Для этого найдём отступ секции с именем .strtab, пусть он будет равен StringTableOffset. И вот теперь уже можно получить имя символа в виде кода ASCII, для этого смотрим на байт с offset = StringTableOffset + sym_name и читаем байты пока не встретим 0x00. Остальную информацию о символе получить легко, она просто храниться в symbol table.
4. Теперь мы запарсили symbol table. Осталось получить из неё метки ассемблера для кода. Для этого нужно найти все символы, относящиеся к секции .text. Чтобы сделать это, мы получаем индекс этой секции в section header table (это просто номер .text в таблице заголовков секций), пусть это будет SectionTextIndex. Теперь в каждом символе смотрим на значение поля sym_shndx и если оно равно SectionTextIndex, то мы нашли одну из меток. В поле sym_value метки будет храниться отступ команды, к которой она относится в секции .text. Проблема в том, что sym_value хранит виртуальный адрес символа в executable и shared object elf файлах (файл test_elf к ним не относится, однако подобный алгоритм всё даст верный результат, так как s_addr секции .text в relocatable файлах имеет значение 0). Итак, нужно перевести виртуальный адрес символа в файловый адрес. Для этого в заголовке секции .text

найдём поле s_addr. Его значение показывает виртуальный адрес, где начинает исполняться код, записанный в .text. Просто вычтем из виртуального адреса символа sym_value виртуальный адрес s_addr и получим адрес внутри секции команды, к которой относится метка. Вот и всё, теперь мы знаем, какие метки относятся к каким командам.

Отлично, теперь у нас есть вся необходимая информация из elf файла, и мы можем приступить к процессу дизассемблирования секции .text.

Информации, приведённой в теоретической части должно быть достаточно для понимания устройства типов инструкций (instructions) и механизма парсинга констант (immediate). Но для дизассемблирования конкретных команд нужно больше данных, получить их можно из таблицы инструкций для RV32M (см. рисунок №7) и таблицы инструкций для RV32I (см. рисунок №8).

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 №7 – список инструкций RV32M

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

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 №8 – список инструкций RV32I (сверху для удобства приведены типы инструкций)

Читая по 4 байта из секции .text и декодируя их из little endian в формат, где биты идут от старшего к младшему, будем сопоставлять опкоду тип инструкции, а потом:

1. Если инструкция относится к I/S/B типу, то посмотрим на func3 (12 – 14 биты) и определим с помощью полученной информации мнемонику.
2. Если инструкция относится к R типу, то посмотрим на func3 и func7 (12 – 14 и 25 – 31 биты соответственно) и определим с помощью полученной информации мнемонику.
3. Если инструкция относится к U или J типу, то мнемонику можно определить на основании только опкода.

Отлично, теперь зная мнемонику и, что даже важнее, тип, мы можем запарсить инструкцию. Т.е. получить значения rs1, rs2, rd, immediate и т.д., если соответствующие поля вообще присутствуют в инструкции.

Как сделать это для каждого типа инструкции описано в теоретической части. Также не стоит забывать о специальных инструкциях, для которых схема парсинга уникальна. К таким относятся, например, ecall, ebreak, slli, srli, srai.

Итак, я описал алгоритм решения задачи и теперь обращу внимание на то, как работает непосредственно мой код.

Весь код можно разделить на 2 относительно независимые части.

1. Класс ElfParser – отвечает за парсинг elf файла.
2. Класс InstructionSet – сопоставляет опкод и поля func3, func7 типу инструкции и её мнемонике. Создаёт экземпляры классов RTypeInstruction, ITypeInstruction, SpecialTypeInstruction (для ebreak, ecall) и т.д. Классы типов инструкций расширяют класс AbstractInstruction и отвечают за парсинг immediate, регистров и для конкретных типов инструкций. Класс InstructionSet использует

строку source, которая представляет из себя список инструкций в подготовленном виде.

Класс Main использует функционал ElfParser и InstructionSet, открывает и закрывает файлы, помимо этого, Main дополняет assembler код метками. Метки из .symtable main получает от ElfParser, остальные генерирует сам на основании инструкций.

В коде присутствует базовая обработка ошибок, проверка на то, что файл является корректным elf файлом, что с входным и выходным файлами можно работать.

В конце я хотел бы обратить внимание на несколько важных моментов, которые, как мне кажется, помогут понять логику работы кода.

При чтении elf файла, я заношу его в память целиком, создавая массив байт. Это не слишком эффективно по памяти, однако значительно облегчает работу с содержимым файла, ведь нам нужно перемещаться по нему как вперёд, так и назад. Таким образом класс elfParser представляет собой набор функций, которые обращаются к определённым байтам elf файла или друг к другу и на основании этого выдают необходимую информацию. То есть я не пытаюсь сохранить информацию из elf файла в виде структур, вместо этого я использую функцию, которая читает данные из массива байт, представляющих собой elf файл.

Со всеми двоичными данными код работает, как с числами, используя битовые сдвиги, маски и другие побитовые операции.

Результат работы написанной программы на приложенном к заданию файле:

```
00000000: <main>    addi  x2, x2, -32
00000004:           sw    x2, x1, 28
00000008:           sw    x2, x8, 24
0000000c:           addi  x8, x2, 32
00000010:           addi  x10, zero, 0
00000014:           sw    x8, x10, -12
00000018:           addi  x11, zero, 64
0000001c:           sw    x8, x11, -16
00000020:           sw    x8, x10, -20
```

```
00000024:           addi  x10, zero, 1
00000028:           sw    x8, x10, -24
0000002c: <LOC_0000002c> jal   zero, 0
00000030: <.LBB0_1> lw    x10, x8, -24
00000034:           lw    x11, x8, -16
00000038: <LOC_00000038> bge   x10, x11, 0
0000003c: <LOC_0000003c> jal   zero, 0
00000040: <.LBB0_2> lw    x10, x8, -24
00000044:           mul   x10, x10, x10
00000048:           lw    x11, x8, -20
0000004c:           add   x10, x11, x10
00000050:           sw    x8, x10, -20
00000054: <LOC_00000054> jal   zero, 0
00000058: <.LBB0_3> lw    x10, x8, -24
0000005c:           addi  x10, x10, 1
00000060:           sw    x8, x10, -24
00000064: <LOC_00000064> jal   zero, 0
00000068: <.LBB0_4> lw    x10, x8, -20
0000006c:           lw    x8, x2, 24
00000070:           lw    x1, x2, 28
00000074:           addi  x2, x2, 32
00000078:           jalr  zero, x1, 0
```

Symbol Table (.symtab) для test_elf

Symbol	Value	Size	Type	Bind	Vis	Index	Name
[0]	0x0	0	NOTYPE	LOCAL	DEFAULT	UNDEF	
[1]	0x0	0	FILE	LOCAL	DEFAULT	ABS	test2.c
[2]	0x30	0	NOTYPE	LOCAL	DEFAULT	2	.LBB0_1
[3]	0x40	0	NOTYPE	LOCAL	DEFAULT	2	.LBB0_2
[4]	0x58	0	NOTYPE	LOCAL	DEFAULT	2	.LBB0_3
[5]	0x68	0	NOTYPE	LOCAL	DEFAULT	2	.LBB0_4
[6]	0x0	124	FUNC	GLOBAL	DEFAULT	2	main

Листинг

Компилятор javac 14.0.2.

Main.java

```java
package Disassembler;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;

public class Main {
    private static byte[][] SegregateCode(byte[] code) {
```

```
        byte[][] processed = new byte[code.length / 4][4];
        for (int i = 0; i < code.length / 4; i++) {
            System.arraycopy(code, i * 4, processed[i], 0, 4);
        }
        return processed;
    }

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Input file name expected");
            System.exit(0);
        }
        InstructionSet instructionSet = new InstructionSet();
        ElfParser elf = null;
        try {
            elf = new ElfParser(args[0]);
        } catch (ParseException e) {
            System.out.println(e.getMessage());
            System.exit(0);
        } catch (IOException e) {
            System.err.println("I/O error: " + e.getMessage());
            System.exit(0);
        }
        byte[] code = elf.getCode();
        byte[][] processedCode = SegregateCode(code);
        HashMap<Integer, String> labels = elf.getLabels();

        int sectionTextVirtualAddress = elf.getSectionTextVirtualAddress();

        //Getting labels from code
        int offset;
        for (int i = 0; i < processedCode.length; i++) {
            AbstractInstruction inst = instructionSet.get(processedCode[i]);
            switch (inst.getMnemonic()) {
                case ("JAL"):
                    offset = i * 4 + ((JTypeInstruction) inst).getImmediate();
                    labels.put(offset, String.format("LOC_%08x",
                            offset + sectionTextVirtualAddress));
                    break;
                case ("BEQ"):
                case ("BLT"):
                case ("BGE"):
                case ("BLTU"):
                case ("BGEU"):
                    offset = i * 4 + ((BTypeInstruction) inst).getImmediate();
                    labels.put(offset, String.format("LOC_%08x",
                            offset + sectionTextVirtualAddress));
                    break;
            }
        }

        try {
            BufferedWriter out;
            if (args.length > 1) {
                out = new BufferedWriter(new FileWriter(args[1]));
            } else {
                out = new BufferedWriter(new PrintWriter(System.out));
            }
            try {
```

```
                for (int i = 0; i < processedCode.length; i++) {
                    AbstractInstruction inst =
                            instructionSet.get(processedCode[i]);
                    int address = sectionTextVirtualAddress + i * 4;
                    if (labels.containsKey(i * 4)) {
                        String label = labels.get(i * 4);
                        if (label.equals("")) {
                            label = String.format("LOC_%08x", address);
                        }
                        out.write(String.format("%08x: %-9s %s%n",
                                address, '<' + label + '>', inst.toString()));
                    } else {
                        out.write(String.format("%08x:           %s%n",
                                address, inst.toString()));
                    }
                }
                if (args.length > 2 && args[2].equals("e")) {
                    out.write("\nSymbol Table (.symtab)\n");
                    elf.dumpSymbolTable(out);
                }
            } finally {
                out.close();
            }
        } catch (ParseException e) {
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.err.println("I/O error: " + e.getMessage());
        }
    }
}
```

ElfParser.java

```
package Disassembler;

import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.Writer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class ElfParser {
    byte[] bytes;
    public ElfParser(String fileName) throws IOException {
        try (BufferedInputStream file = new BufferedInputStream(new
FileInputStream(fileName))) {
            bytes = file.readAllBytes();
        }
        checkElf();
    }

    private byte read(int offset) {
        return read(offset, 1)[0];
    }

    private byte[] read(int offset, int size) {
```

```
        if (offset + size > bytes.length) {
            throw error("Unexpected end of file");
        }
        byte[] chunk = new byte[size];
        System.arraycopy(bytes, offset, chunk, 0,  size);
        return chunk;
    }

    public byte[] getCode() {
        int[] codeLocation = findSectionText();
        return read(codeLocation[0], codeLocation[1]);
    }

    public List<String[]> getSymbolTable() {
        List<String[]> symbols = new ArrayList<>();
        int[] pos = findSymbolTable();
        int symTableOffset = pos[0];
        int symTableSize = pos[1];
        int symTableEntrySize = pos[2];
        for (int i = symTableOffset; i < symTableOffset + symTableSize; i +=
symTableEntrySize) {
            symbols.add(getSymbol(i));
        }
        return symbols;
    }

    public void dumpSymbolTable() {
        List<String[]> symbols = getSymbolTable();
        String header = String.format("%s %-10s\t%-5s %-8s %-8s %-9s %-6s %s",
                "Symbol", "Value", "Size", "Type", "Bind", "Vis", "Index",
"Name");
        System.out.println(header);
        for (int i = 0; i < symbols.size(); i++) {
            String[] symbol = symbols.get(i);
            String out = String.format("[%4d] %-10s\t%-5s %-8s %-8s %-9s %-6s %s",
                    i, "0x" + symbol[0], symbol[1], symbol[2], symbol[3],
symbol[4], symbol[5], symbol[6]);
            System.out.println(out);
        }
    }

    public void dumpSymbolTable(Writer file) throws IOException {
        List<String[]> symbols = getSymbolTable();
        String header = String.format("%s %-10s\t%-5s %-8s %-8s %-9s %-6s %s",
                "Symbol", "Value", "Size", "Type", "Bind", "Vis", "Index",
"Name");
        file.write(header + '\n');
        for (int i = 0; i < symbols.size(); i++) {
            String[] symbol = symbols.get(i);
            String out = String.format("[%4d] %-10s\t%-5s %-8s %-8s %-9s %-6s %s",
                    i, "0x" + symbol[0], symbol[1], symbol[2], symbol[3],
symbol[4], symbol[5], symbol[6]);
            file.write(out + '\n');
        }
    }

    private String[] getSymbol(int offset) {
        int indexInStringTable = parseBytesToInt(read(offset, 4));
        String name = getSymbolName(indexInStringTable);
```

```
        String value = Integer.toHexString(parseBytesToInt(read(offset + 4, 4)));
        String size = String.valueOf(parseBytesToInt(read(offset + 8, 4)));
        String bind = getSymbolBind(read(offset + 12) >>> 4);
        String type = getSymbolType(read(offset + 12) & 0x0F);
        String vis = getSymbolVis(read(offset + 13));
        String index = getSymbolIndex(parseBytesToInt(read(offset + 14, 2)));
        return new String[] { value, size, type, bind, vis, index, name };
    }

    public HashMap<Integer, String> getLabels() {
        int index = getSectionTextIndex();
        List<String[]> symbols = getSymbolTable();
        HashMap<Integer, String> labels = new HashMap<>();
        for (String[] symbol : symbols) {
            if (symbol[5].equals(String.valueOf(index))) {

labels.put(translateVirtualAddressToOffsetInSectionText(Integer.parseInt(symbol[0]
, 16)), symbol[6]);
            }
        }
        return labels;
    }

    private String getSymbolName(int indexInStringTable) {
        int stringTableOffset = getStringTableOffset();
        StringBuilder name = new StringBuilder("");

        byte c = read(stringTableOffset + indexInStringTable);
        int i = 1;
        while (c != 0x00) {
            name.append((char) c);
            c = read(stringTableOffset + indexInStringTable + i);
            i++;
        }

        return name.toString();
    }

    private int[] findSymbolTable() {
        int offset = getSectionHeaderOffset(new int[] { 0x2E, 0X73, 0x79, 0x6D,
0x74, 0x61, 0x62, 0x00 });
        int symTableOffset = parseBytesToInt(read(offset + 16, 4));
        int symTableSize = parseBytesToInt(read(offset + 20, 4));
        int symTableEntrySize = parseBytesToInt(read(offset + 36, 4));
        return new int[] { symTableOffset, symTableSize, symTableEntrySize };
    }

    private int[] findSectionText() {
        int offset = getSectionHeaderOffset(new int[] { 0x2E, 0x74, 0x65, 0x78,
0x74, 0x00 });
        int sectionTextOffset = parseBytesToInt(read(offset + 16, 4));
        int sectionTextSize = parseBytesToInt(read(offset + 20, 4));
        return new int[] { sectionTextOffset, sectionTextSize };
    }

    private int getSectionTextIndex() {
        return getSectionIndex(new int[] { 0x2E, 0x74, 0x65, 0x78, 0x74, 0x00 });
    }
```

```
    private int getSectionIndex(int[] name) {
        int shoff = getShoff(); //Section header offset
        int shnum = getShnum(); //Section headers number
        int shentsize = getShentsize(); //Section header entry size
        int stringTableOffset = getSectionHeaderStringTableOffset();
        for (int i = 0; i < shnum; i++) {
            if (isSection(shoff + i * shentsize, stringTableOffset, name)) {
                return i;
            }
        }
        throw error("Section header not found");
    }

    private int getSectionHeaderOffset(int[] name) {
        int shoff = getShoff(); //Section header offset
        int shnum = getShnum(); //Section headers number
        int shentsize = getShentsize(); //Section header entry size
        int sectionHeaderStringTableOffset = getSectionHeaderStringTableOffset();
        for (int i = 0; i < shnum; i++) {
            if (isSection(shoff + i * shentsize, sectionHeaderStringTableOffset,
name)) {
                return shoff + i * shentsize;
            }
        }
        throw error("Section header not found");
    }

    private int getStringTableOffset() {
        int sectionHeaderOffset = getSectionHeaderOffset(new int[] { 0x2E, 0x73,
0x74, 0x72, 0x74, 0x61, 0x62, 0x00 });
        int stringTableOffset = parseBytesToInt(read(sectionHeaderOffset + 16,
4));
        return stringTableOffset;
    }

    private int getSectionHeaderStringTableOffset() {
        int shtrndx = getShtrndx();
        int shoff = getShoff();
        int shentsize = getShentsize();
        return parseBytesToInt(read(shoff + shtrndx * shentsize + 16, 4));
    }

    private boolean isSection(int offset, int stringTableOffset, int[] name) {
        int offsetInStringTable = parseBytesToInt(read(offset, 4));
        return check(stringTableOffset + offsetInStringTable, name);
    }

    private String getSymbolIndex(int index) {
        switch(index) {
            case (0):
                return "UNDEF";
            case (0xff00):
                return "LOPROC";
            case (0xff1f):
                return "HIPROC";
            case (0xff20):
                return "LOOS";
            case (0xff3f):
                return "HIOS";
```

```
            case (0xfff1):
                return "ABS";
            case (0xfff2):
                return "COMMON";
            case (0xffff):
                return "XINDEX";
            default:
                return String.valueOf(index);
        }
    }

    private String getSymbolVis(int type) {
        switch(type) {
            case (0):
                return "DEFAULT";
            case (1):
                return "INTERNAL";
            case (2):
                return "HIDDEN";
            case (3):
                return "PROTECTED";
            default:
                return "UNKNOWN";
        }
    }

    private String getSymbolBind(int type) {
        switch(type) {
            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";
            default:
                return "UNKNOWN";
        }
    }

    private String getSymbolType(int type) {
        switch(type) {
            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";
            default:
                return "UNKNOWN";
        }
    }

    private int getShtrndx() {
        return parseBytesToInt(read(48, 4)) >>> 16;
    }

    private int getShnum() {
        return parseBytesToInt(read(48, 4)) & 0x0000FFFF;
    }

    private int getShoff() {
        return parseBytesToInt(read(32, 4));
    }

    private int getShentsize() {
        return parseBytesToInt(read(44, 4)) >>> 16;
    }

    public int getSectionTextVirtualAddress() {
        int offset = getSectionHeaderOffset(new int[] { 0x2E, 0x74, 0x65, 0x78,
0x74, 0x00 });
        int sectionTextVirtualAddress = parseBytesToInt(read(offset + 12, 4));
        return sectionTextVirtualAddress;
    }

    private int translateVirtualAddressToOffsetInSectionText(int address) {
        return address - getSectionTextVirtualAddress();
    }

    private void checkElf() {
        if (!check(0, new int[] { 0x7F, 0x45, 0x4C, 0x46 })) {
            throw error("Invalid file type: elf expected");
        }
    }

    private boolean check(int offset, int[] expected) {
        for (int i = 0; i < expected.length; i++) {
            if (expected[i] != read(offset + i)) {
                return false;
            }
        }
        return true;
    }
```

```
    public static int parseBytesToInt(byte[] bytes) {
        int parsed = bytes[bytes.length - 1] & 0b011111111;
        for (int i = bytes.length - 2; i >= 0; i--) {
            parsed <<= 8;
            parsed = parsed | (bytes[i] & 0b011111111);
        }
        return parsed;
    }

    public ParseException error(final String message) {
        return new ParseException(message);
    }
}
```

InstructionSet.java

```
package Disassembler;

import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Scanner;

public class InstructionSet {
    Scanner sc;
    char[] typeTable = new char[128];
    HashMap<Integer, HashMap<Integer, String>> mnemonicTable = new HashMap<>();
    String source =
        "0110111 U LUI\n" +
        "0010111 U AUIPC\n" +
        "1101111 J JAL\n" +
        "1100111 I JALR 000\n" +
        "1100011 B BEQ 000\n" +
        "1100011 B BNE 001\n" +
        "1100011 B BLT 100\n" +
        "1100011 B BGE 101\n" +
        "1100011 B BLTU 110\n" +
        "1100011 B BGEU 111\n" +
        "0000011 I LB 000\n" +
        "0000011 I LH 001\n" +
        "0000011 I LW 010\n" +
        "0000011 I LBU 100\n" +
        "0000011 I LHU 101\n" +
        "0100011 S SB 000\n" +
        "0100011 S SH 001\n" +
        "0100011 S SW 010\n" +
        "0010011 I ADDI 000\n" +
        "0010011 I SLTI 010\n" +
        "0010011 I SLTIU 011\n" +
        "0010011 I XORI 100\n" +
        "0010011 I ORI 110\n" +
        "0010011 I ANDI 111\n" +
        "0010011 I SLLI 001\n" +
        "0010011 I SRLI 101\n" +
        "0010011 I SRAI 101\n" +
        "0110011 R ADD 000 0000000\n" +
```

```
        "0110011 R SUB 000 0100000\n" +
        "0110011 R SLL 001 0000000\n" +
        "0110011 R SLT 010 0000000\n" +
        "0110011 R SLTU 011 0000000\n" +
        "0110011 R XOR 100 0000000\n" +
        "0110011 R SRL 101 0000000\n" +
        "0110011 R SRA 101 0100000\n" +
        "0110011 R OR 110 0000000\n" +
        "0110011 R AND 111 0000000\n" +
        "0110011 R MUL 000 0000001\n" +
        "0110011 R MULH 001 0000001\n" +
        "0110011 R MULHSU 010 0000001\n" +
        "0110011 R MULHU 011 0000001\n" +
        "0110011 R DIV 100 0000001\n" +
        "0110011 R DIVU 101 0000001\n" +
        "0110011 R REM 110 0000001\n" +
        "0110011 R REMU 111 0000001";

    public InstructionSet() {
        try {
            try (Reader rd = new StringReader(source)) {
                sc = new Scanner(rd);
                parse();
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }

    private void parse() {
        while (sc.hasNext()) {
            int opcode = Integer.parseInt(sc.next(), 2);
            char type = sc.next().charAt(0);
            typeTable[opcode] = type;
            String mnemonic = sc.next();

            int func3;
            int func7;

            switch (type) {
                case ('R'):
                    func3 = Integer.parseInt(sc.next(), 2);
                    func7 = Integer.parseInt(sc.next(), 2);
                    int key = func7 * 8 + func3;
                    update(opcode, key, mnemonic);
                    break;
                case ('I'):
                case ('S'):
                case ('B'):
                    func3 = Integer.parseInt(sc.next(), 2);
                    update(opcode, func3, mnemonic);
                    break;
                case ('U'):
                case ('J'):
                    update(opcode, 0, mnemonic);
                    break;
                default:
                    throw new ParseException("Invalid instruction set (Unknown
instruction type)");
```

```
            }
        }
    }

    private void update(int opcode, int key, String mnemonic) {
        HashMap<Integer, String> tmp = mnemonicTable.getOrDefault(opcode, new
HashMap<>());
        tmp.put(key, mnemonic);
        mnemonicTable.put(opcode, tmp);
    }

    public static int parseBytes(byte[] bytes) {
        int parsed = bytes[3];
        for (int i = 2; i >= 0; i--) {
            parsed <<= 8;
            parsed = parsed | (bytes[i] & 0b011111111);

        }
        return parsed;
    }

    public AbstractInstruction get(byte[] bytes) {
        return get(parseBytes(bytes));
    }

    private boolean ifSpecialInstruction(int opcode) {
        return opcode == 0b1110011 || opcode == 0b0001111;
    }

    public int getBits(int hex, int first, int last) {
        return (hex << (31 - last)) >>> (31 - last + first);
    }

    private AbstractInstruction specialInstruction(int opcode, int hex) {
        if (opcode == 0b0001111) {
            return new SpecialTypeInstruction("FENCE", hex);
        } else if (opcode == 0b1110011) {
            if (getBits(hex, 20, 20) == 0) {
                return new SpecialTypeInstruction("ECALL", hex);
            } else if (getBits(hex, 20, 20) == 1) {
                return new SpecialTypeInstruction("EBREAK", hex);
            }
        }
        throw new ParseException("Unknown special case: " + hex);
    }

    public AbstractInstruction get(int hex) {
        int opcode = getOpcode(hex);

        if (ifSpecialInstruction(opcode)) {
            return specialInstruction(opcode, hex);
        }

        char type = getType(opcode, hex);
        int key;
        String mnemonic;
        switch (type) {
            case ('R'):
                key = getFunc3(hex) + getFunc7(hex) * 8;
```

```
                mnemonic = mnemonicTable.get(opcode).get(key);
                return new RTypeInstruction(mnemonic, type, hex);
            case ('I'):
                if (opcode == 0b0010011) { //Specialized shift instruction handle
                    int func3 = getFunc3(hex);
                    if (func3 == 0b101) {
                        int func7 = getFunc7(hex);
                        if (func7 == 0b0000000) {
                            return new SpecializedITypeShiftInstruction("SRLI",
type, hex);
                        } else if (func7 == 0b0100000) {
                            return new SpecializedITypeShiftInstruction("SRAI",
type, hex);
                        }
                    } else if (func3 == 0b001) {
                        return new SpecializedITypeShiftInstruction("SLLI", type,
hex);
                    }
                }
                key = getFunc3(hex);
                mnemonic = mnemonicTable.get(opcode).get(key);
                return new ITypeInstruction(mnemonic, type, hex);
            case ('S'):
                key = getFunc3(hex);
                mnemonic = mnemonicTable.get(opcode).get(key);
                return new STypeInstruction(mnemonic, type, hex);
            case ('B'):
                key = getFunc3(hex);
                mnemonic = mnemonicTable.get(opcode).get(key);
                return new BTypeInstruction(mnemonic, type, hex);
            case ('U'):
                mnemonic = mnemonicTable.get(opcode).get(0);
                return new UTypeInstruction(mnemonic, type, hex);
            case ('J'):
                mnemonic = mnemonicTable.get(opcode).get(0);
                return new JTypeInstruction(mnemonic, type, hex);
            default:
                return new SpecialTypeInstruction("unknown command", 0x00000000);
        }
    }

    private char getType(int opcode, int hex) {
        return typeTable[opcode];
    }

    public int getOpcode(int hex) {
        return hex & 0x0000007F;
    }

    public int getFunc3(int hex) {
        return (hex & 0x00007000) >>> 12;
    }

    public int getFunc7(int hex) {
        return (hex & 0xFE000000) >>> 25;
    }

    private void dumpTable() {
        for (int opcode : mnemonicTable.keySet())
```

```
        {
            System.out.println(opcode + ": ");
            for (int func : mnemonicTable.get(opcode).keySet()) {
                System.out.println("    " + func + ": " +
mnemonicTable.get(opcode).get(func));
            }
        }
    }
}
```

AbstractInstruction.java

```
package Disassembler;

public abstract class AbstractInstruction {
    String mnemonic;
    char type;
    int hex;

    public AbstractInstruction(String mnemonic, char type, int hex) {
        this.mnemonic = mnemonic;
        this.type = type;
        this.hex = hex;
    }

    protected int getBits(int first, int last) {
        return (hex << (31 - last)) >>> (31 - last + first);
    }

    protected boolean isNegative() {
        return getBits(31, 31) == 1;
    }

    protected String getRegName(int R) {
        if (R == 0) {
            return "zero";
        } else {
            return "x" + R;
        }
    }

    protected int getRd() {
        return getBits(7, 11);
    }

    protected int getRs1() {
        return getBits(15, 19);
    }

    protected int getRs2() {
        return getBits(20, 24);
    }

    public String getMnemonic() {
        return mnemonic;
    }
```

```
    public int getHex() {
        return hex;
    }

    @Override
    public String toString() {
        return String.format("%-5s ", getMnemonic().toLowerCase());
    }
}
```

BTypeInstruction.java

```
package Disassembler;

public class BTypeInstruction extends AbstractInstruction {
    public BTypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    public int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF; //Immediates are sign-extended
        }
        initial <<= 12;
        initial |= (getBits(7, 7) << 11);
        initial |= (getBits(25, 30) << 5);
        initial |= (getBits(8, 11) << 1);
        return initial;
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %s, %d",
getRegName(getRs1()), getRegName(getRs2()), getImmediate());
    }
}
```

ITypeInstruction.java

```
package Disassembler;

public class ITypeInstruction extends AbstractInstruction {
    public ITypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    protected int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF; //Immediates are sign-extended
        }
        initial <<= 11;
        initial |= getBits(20, 30);
        return initial;
    }
```

```
    @Override
    public String toString() {
        return super.toString() + String.format("%s, %s, %d", getRegName(getRd()),
getRegName(getRs1()), getImmediate());
    }
}
```

JTypeInstruction.java

```
package Disassembler;

public class JTypeInstruction extends AbstractInstruction {
    public JTypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    public int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF; //Immediates are sign-extended
        }
        initial <<= 20;
        initial |= (getBits(12, 19) << 12);
        initial |= (getBits(20, 20) << 11);
        initial |= (getBits(21, 30) << 1);
        return initial;
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %d", getRegName(getRd()),
getImmediate());
    }
}
```

RTypeInstruction.java

```
package Disassembler;

public class RTypeInstruction extends AbstractInstruction {
    public RTypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %s, %s", getRegName(getRd()),
getRegName(getRs1()), getRegName(getRs2()));
    }
}
```

SpecializedITypeShiftInstruction.java

```
package Disassembler;

public class SpecializedITypeShiftInstruction extends AbstractInstruction {
```

```
    public SpecializedITypeShiftInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    protected int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF;
        }
        initial <<= 11;
        initial |= getBits(20, 30);
        return initial;
    }

    private int getShamt() {
        return getImmediate() & 0x0000_001F;
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %s, %d", getRegName(getRd()),
getRegName(getRs1()), getShamt());
    }
}
```

SpecialTypeInstruction.java

```
package Disassembler;

public class SpecialTypeInstruction extends AbstractInstruction {
    public SpecialTypeInstruction(String mnemonic, int hex) {
        super(mnemonic, 'X', hex);
    }

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

STypeInstruction.java

```
package Disassembler;

public class STypeInstruction extends AbstractInstruction {
    public STypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    private int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF; //Immediates are sign-extended
        }
        initial <<= 11;
        initial |= (getBits(25, 30) << 5);
```

```
        initial |= (getBits(8, 11) << 1);
        initial |= getBits(7, 7);
        return initial;
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %s, %d",
getRegName(getRs1()), getRegName(getRs2()), getImmediate());
    }
}
```

UTypeInstruction.java

```
package Disassembler;

public class UTypeInstruction extends AbstractInstruction {
    public UTypeInstruction(String mnemonic, char type, int hex) {
        super(mnemonic, type, hex);
    }

    private int getImmediate() {
        int initial = 0;
        if (isNegative()) {
            initial = 0xFFFFFFFF; //Immediates are sign-extended
        }
        initial <<= 31;
        initial |= (getBits(12, 30) << 12);
        return initial;
    }

    @Override
    public String toString() {
        return super.toString() + String.format("%s, %d", getRegName(getRd()),
getImmediate());
    }
}
```

ParseException.java

```
package Disassembler;

public class ParseException extends RuntimeException {
    public ParseException(final String message) {
        super(message);
    }
}
```