|  |  |  |
| --- | --- | --- |
| Лабораторная работа №3 | Группа 39 | 2022 |
| ISA | Яковлев Илья Игоревич | |
|

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

**Инструментарий и требования к работе:**работа должна быть выполнена на C, C++, Python или Java. В отчёте указываем язык и компилятор/интерпретатор, на котором вы работали.

**Описание**

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

Должен поддерживаться следующий набор команд: RISC-V RV32I, RV32M. Подробнее (volume 1): <https://riscv.org/technical/specifications/>

Кодирование: little endian.

Обрабатывать нужно только секции .text, .symtable.

Для каждой строки кода указывается её адрес в hex формате.

Обозначение меток нужно найти в Symbol Table (.symtab). Если же название метки там не найдено, то используется следующее обозначение: L%i, например, L2, L34. Нумерация начинается с 0. Для каждой метки перед названием указывается адрес (пример ниже).

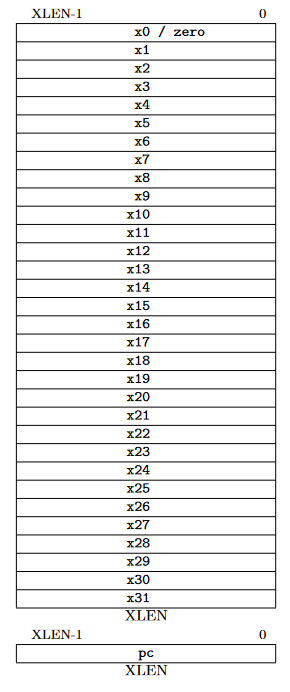
**Описание системы кодирования команд RISC-V**

RISC-V – это ISA, которая была разработана для поддержки исследований и обучения компьютерной архитектуре. То есть это специальная разработанная некоммерческая архитектура набора инструкций, предназначенная для изучения архитектуры компьютера. Из-за такой цели эта ISA максимально упрощена, также разработчики стараются не вносить радикальные изменения в новые версии, чтобы поддерживать обратную совместимость.

У данной ISA есть множество наборов инструкций, однако мы будем работать только с RV32I и RV32M.

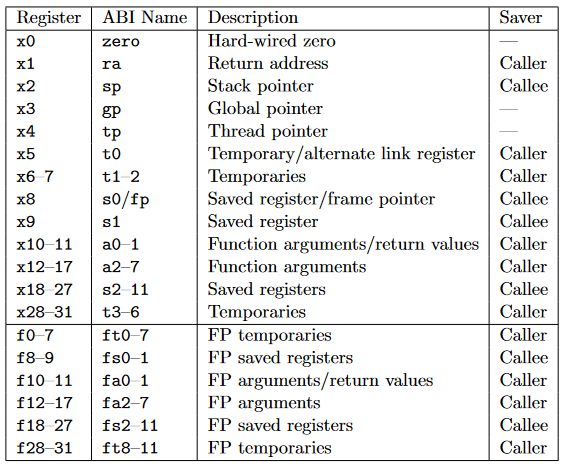
Устройство регистров:

RV32I – base integer instruction set, RV32M – расширение инструкций, включающее умножение и деление.



Всего есть 32 регистра, которые имеют размерность XLEN бит (в нашем случае XLEN = 32). x0 – особый регистр, связанный с со случаем, когда все биты 0. Регистр pc также особенный – он содержит адрес текущей инструкции. Остальные регистры x1-x31 – регистры общего назначения.

Также у регистров есть специальные названия:

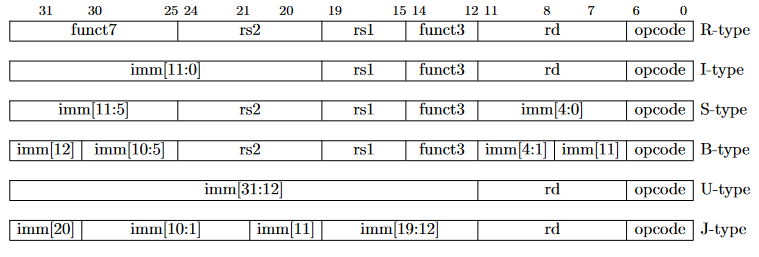


Инструкции:

Рассматриваемые нами инструкции имеют длину 32 бита, но в общем случае это неверно. Могут быть инструкции и другой длинны, есть инструменты для увеличения длины инструкций.

В инструкциях rs1 и rs2 – регистры источника, rd – регистр назначения, funct3, funct7, opcode нужны для опознавания инструкций. Также могут встречаться константы imm. Примечание: в некоторых инструкциях биты могут быть разбросаны, как показано на схеме.

Базовые форматы инструкций c несколькими вариантами кодирования:



Сразу замечу, что S-type и B-type отличаются только в распределении битов константы, потому что в некоторых случаях удобнее пользоваться нестандартным распределением битов, в частности, при работе с адресами. Аналогично и с U-type и J-type.

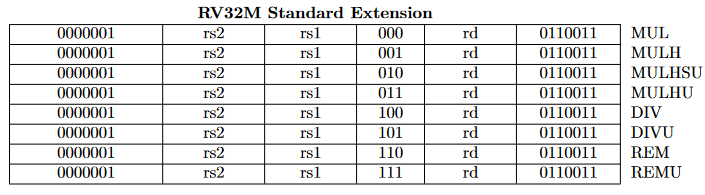
R-type это инструкции, в которых происходят какие-то манипуляции с регистрами rs1 и rs2, а результат записывается в регистр rd. Например, сложение, умножение. Конкретная инструкция определяется по funct7, funct3, opcode.

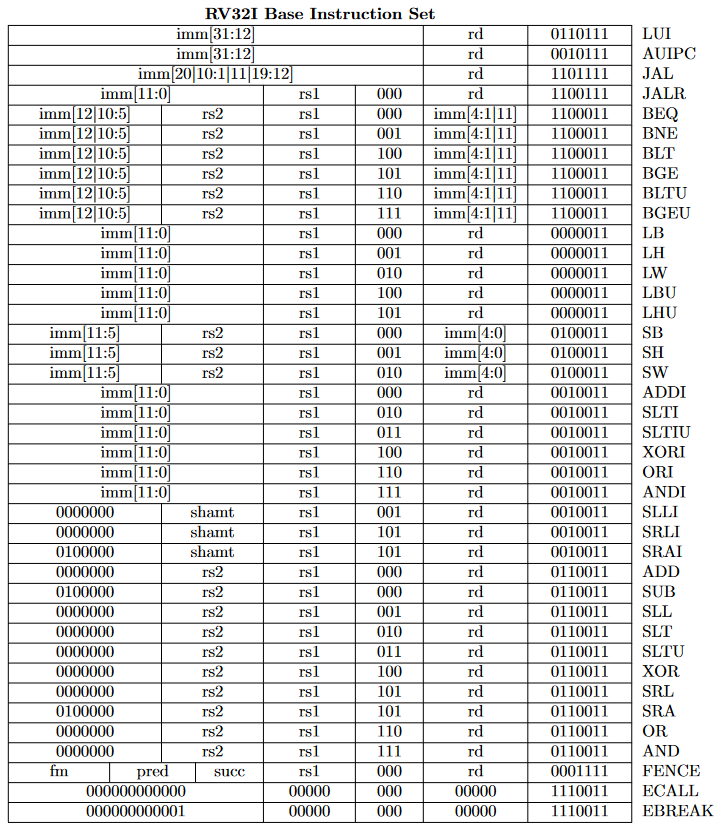
I-type это инструкции, в которых происходят какие-то манипуляции с регистром rs1 и imm, а результат записывается в регистр rd. Например сложение с константой, считывание с какой-то операцией. Конкретная инструкция определяется по funct3, opcode.

S-type в данных инструкциях отсутствует регистр назначения. Обычно это инструкции для записи данных в память. Конкретная инструкция определяется по funct3, opcode.

U-type имеет большую константу и регистр назначения. Обычно это инструкции для записи константы в регистр. Конкретная инструкция определяется по opcode.

Список инструкций:





Видно, что ECALL, EBREAK описываются полностью неизменяемым кодом. ECALL и EBREAK – специальные системные инструкции, которые могут потребовать привилегированных прав.

FENCE и вовсе вне всех типов, она используется для упорядочивания ввода и вывода.

**ELF файл**

В современных POSIX-системах основным форматом исполняемых файлов, объектных файлов, динамических библиотек является формат ELF. Этот формат используется и на 32-битных (Elf32), и на 64-битных (Elf64) системах и для машин с порядком байт Little-endian, и для машин с порядком байт Big-endian.

**Описание структуры файла ELF**

В начале файла идет заголовок ELF файла. Заголовок содержит в себе основные сведения об ELF файле. Он описывается такой структурой:

typedef struct

{

unsigned char e\_ident[16];

uint16\_t e\_type;

uint16\_t e\_machine;

uint32\_t e\_version;

uint32\_t e\_entry;

uint32\_t e\_phoff;

uint32\_t e\_shoff;

uint32\_t e\_flags;

uint16\_t e\_ehsize;

uint16\_t e\_phentsize;

uint16\_t e\_phnum;

uint16\_t e\_shentsize;

uint16\_t e\_shnum;

uint16\_t e\_shstrndx;

} Elf32\_Ehdr;

Опишу наиболее важные поля:

e\_ident[4] – характеризует размерность слова: 0 – неизвестно, 1- 32, 2 - 64 (в нашем случае 1)

e\_ident[5] – характеризует порядок байт: 0 – неизвестно, 1- little-endian, 2 - big-endian (в нашем случае 1)

e\_type – тип файла: 0 - неизвестно, 1 - объектный файл, 2 - исполняемый файл), 3 - разделяемая библиотека, 4 - core-файл (в нашем случае 2)

e\_entry - определяет виртуальный адрес точки входа в программу. После загрузки программы в память управление передается на этот адрес.

e\_phoff - задает смещение от начала файла до начала таблицы заголовков программы (program header table)

e\_shoff - задает смещение от начала файла до начала таблицы заголовков секций (program section table)

e\_phnum - хранит количество записей в таблице заголовков программы

e\_shnum - хранит количество записей в таблице заголовков секций

e\_shstrndx - хранит индекс заголовка секции, которая хранит имена всех секций

Информация в ELF файле организована в секции. Описывают эти секции их заголовки, которые лежат в таблице заголовков секций. Она представляет из себя массив структур (массив заголовков секций). Этот массив находится по смещению e\_shoff. Элемент массива 0 зарезервирован. Таким образом, описания секций - это элементы массива с индексами от 1 до e\_shnum – 1.

Структура заголовка секции:

typedef struct

{

uint32\_t sh\_name;

uint32\_t sh\_type;

uint32\_t sh\_flags;

uint32\_t sh\_addr;

uint32\_t sh\_offset;

uint32\_t sh\_size;

uint32\_t sh\_link;

uint32\_t sh\_info;

uint32\_t sh\_addralign;

uint32\_t sh\_entsize;

} Elf32\_Shdr;

Опишу наиболее важные поля:

sh\_name - хранит индекс имени секции, то есть хранит смещение в секции, хранящей имена (секция e\_shstrndx). Имя – это последовательность символов, заканчивающаяся нулевым символом. Можно сказать, что это своеобразное расширение данных, которое специфицирует их назначение.

sh\_offset - хранит смещение от начала файла, по которому размещаются данные секции

sh\_size - хранит размер секции в байтах

Для написания дизассемблера нужно обработать секции .text и .symtable.

С секцией .text все просто: она содержит закодированные команды, с которыми нам нужно работать.

У секции .symtable есть своя внутренняя структура, которую необходимо описать. В данных .symtable хранятся структуры, которые содержат информацию для поиска и перемещения кода программы (в частности, метки). Вид внутреней структуры .symtable:

typedef struct {  
Elf32\_Word st\_name;  
Elf32\_Addr st\_value;  
Elf32\_Word st\_size;  
unsigned char st\_info;  
unsigned char st\_other;  
Elf32\_Half st\_shndx;  
} Elf32\_Sym;

st\_name - хранит индекс имени, то есть хранит смещение в специальной секции (секция .strtab)

st\_value – хранит какое-то значение, причем что представляет из себя это значение контекстно зависимо

st\_info – хранит некоторую информацию о данном элементе

st\_shndx – хранит значение характеризующее данные элемента некоторым способом

С целью не рвать структуру описания ELF файла, опишу, что было посчитано при обработке symtable.

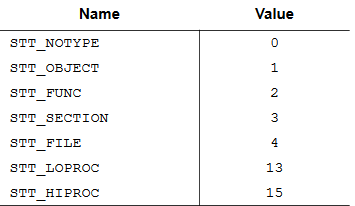
Обработка symtable осуществлена так, что для каждого значения секции symtable определены некоторые параметры:

Num – порядковый номер элемента

Value – его значение, равное st\_value

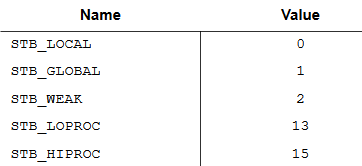
Size – его размер, равное st\_size

Type – тип элемента. Данный параметр получается из поля st\_info по формуле st\_info & 0xf. Таблица соответствия между полученным значением и названием типа:



Что значат типы, которые нам нужны, понятно из названия, так что подробной спецификации не будет.

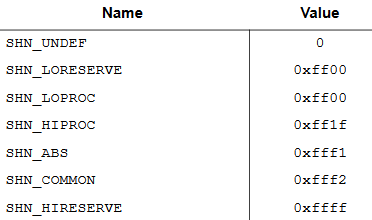
Bind – доступ элемента. Данный параметр получается из поля st\_info по формуле st\_info >> 4. Таблица соответствия между полученным значением и названием уровня доступа:



Что значат уровни доступа, которые нам нужны, понятно из названия, так что подробной спецификации не будет.

Vis – исключая ненужные детали, можно сказать, что в нормальной ситуации она равна default. Этот параметр зависит от st\_other.

Ndx (или index) – характеристика данных элемента, равная st\_shndx. Как конкретно этот параметр характеризует данные нам не важно, но важно, что есть специальные значения, которые имеют специальные названия. Таблица специальных значений:



Name – имя элемента по аналогии с st\_name.

Некоторые замечания по ELF файлам:

Для загрузки программы на выполнение необходима информация, находящаяся в заголовках программы. Однако для выполнения задания анализ этих данных не нужен, из-за этого они не были описаны.

Little-endian работает в ELF файлах немного необычно. Можно разбить ELF файл на блоки размера 1 байт, 2 байта, 4 байта. Эти блоки - это стандартные типы, например, 32 битное число или 8 битный char. Так вот, в ELF файлах при порядке байт little-endian меняется порядок байтов внутри описанных выше блоков. Порядок битов и блоков относительно друг друга не меняется.

**Описание работы кода**

Вначале нужно вытащить из ELF файла нужные нам данные. Для этого создан класс ElfParser. Замечу, что в него встроено средство отладки, которое выводит отладочную информацию в стандартный вывод, если при инициализации ElfParser log = true. При инициализации парсера сразу инициализируются поля заголовка ELF файла и проверяются на корректность. Далее в методе parseSectionHeader мы пробегаемся по всем SectionHeader и каждый раз инициализируем секцию с помощью метода makeSectionHeader. Далее если эта секция .text, .symtab или .strtab, то мы сохраняем необходимую информацию. В elfData – экземпляр TextSymtable - хранится нужная информация о .text и .symtab.

Также в ElfParser есть метод getSymtableString, который принимает OutputStream, куда записывает данные из .symtab в виде, который был описан выше, в описании ELF файлов. Наверное, стоит заметить, что в классе есть map’ы NDX, BIND, TYPE, которые хранят данные из соответственных табличек.

Еще в ElfParser есть метод getSymtableData, который возвращает map глобальных меток (первый аргумент – значение, второй – имя) из .symtab. Это нужно для дизассемблера. Теперь мы получили требуемые данные.

Эти данные принимает метод parseCommands (байты .text, адрес начала команд, полученную ранее map). Это метод класса DisassemblerBlock. Сразу отмечу, что из-за того, что нам нужно ставить в некоторые места метки, которые понадобятся (а значит и появятся) позже, придется дважды пробежаться по командам, но если пожертвовать наглядностью, то хватит и одного раза.

Разбор одной команды осуществляет метод parseComand. Она получает имя команды с помощью getName, которая ищет название инструкции по opcode, funct3, funct7 в заранее объявленных map’ах: OPCODE\_UJ, OPCODE\_ISB, OPCODE\_R. Затем тип инструкции определяется по другой map - TYPE, в которой осуществлено обратное сопоставление. Затем в зависимости от типа операции вызывается парсер операций конкретного типа. Описывать работу каждого парсера излишне, но отмечу, что некоторые инструкции вызывают переходы, из-за чего ставится новая метка, если ее нет (именно для этого нам и нужна была map из .symtab).

**Результат работы программы на приложенном к заданию файле**

Результат работы программы находится в одном текстовом файле, но разделен согласно ТЗ. Вот его содержимое:

.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 mul a4,a4,a1

100f8: 00e60633 add a2,a2,a4

100fc: fea794e3 bne a5,a0,100e4 <L0>

10100: 00c32023 sw a2,0(t1)

10104: 00280813 addi a6,a6,2

10108: 00430313 addi t1,t1,4

1010c: 00288893 addi a7,a7,2

10110: fdd814e3 bne a6,t4,100d8 <L1>

10114: 050f0f13 addi t5,t5,80

10118: 01478513 addi a0,a5,20

1011c: fa5f16e3 bne t5,t0,100c8 <L2>

10120: 00008067 jalr zero,0(ra)

.symtab

Num: Value Size Type Bind Vis Ndx Name

0: 00000000 0 NOTYPE LOCAL DEFAULT UND

1: 00010074 0 SECTION LOCAL DEFAULT 1

2: 00011124 0 SECTION LOCAL DEFAULT 2

3: 00000000 0 SECTION LOCAL DEFAULT 3

4: 00000000 0 SECTION LOCAL DEFAULT 4

5: 00000000 0 FILE LOCAL DEFAULT ABS test.c

6: 00011924 0 NOTYPE GLOBAL DEFAULT ABS \_\_global\_pointer$

7: 000118f4 800 OBJECT GLOBAL DEFAULT 2 b

8: 00011124 0 NOTYPE GLOBAL DEFAULT 1 \_\_SDATA\_BEGIN\_\_

9: 000100ac 120 FUNC GLOBAL DEFAULT 1 mmul

10: 00000000 0 NOTYPE GLOBAL DEFAULT UND \_start

11: 00011124 1600 OBJECT GLOBAL DEFAULT 2 c

12: 00011c14 0 NOTYPE GLOBAL DEFAULT 2 \_\_BSS\_END\_\_

13: 00011124 0 NOTYPE GLOBAL DEFAULT 2 \_\_bss\_start

14: 00010074 28 FUNC GLOBAL DEFAULT 1 main

15: 00011124 0 NOTYPE GLOBAL DEFAULT 1 \_\_DATA\_BEGIN\_\_

16: 00011124 0 NOTYPE GLOBAL DEFAULT 1 \_edata

17: 00011c14 0 NOTYPE GLOBAL DEFAULT 2 \_end

18: 00011764 400 OBJECT GLOBAL DEFAULT 2 a

Список источников:

<https://refspecs.linuxfoundation.org/elf/elf.pdf>

<https://ejudge.ru/study/3sem/elf.html>

<https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>

<https://habr.com/ru/post/480642/>

<https://riscv.org/technical/specifications/>

Листинг кода:

**import** java.io.\*;  
  
**public class** Main {  
 **public static void** main(String[] args) {  
 **try** {  
 InputStream input = **new** FileInputStream(args[1]);  
 ElfParser parser = **new** ElfParser(input, **false**);  
 DisassemblerBlock disassembler = **new** DisassemblerBlock();  
 OutputStream output = **new** FileOutputStream(args[2]);  
 output.write(**".text"**.getBytes());  
 output.write(disassembler.parseCommands(parser.getTextData(), parser.getStartAddr(), parser.getSymtableData()).getBytes());  
 output.write(**"\n.symtab\n"**.getBytes());  
 parser.getSymtableString(output);  
 output.close();  
 } **catch** (IOException e) {  
 System.***out***.println(**"Open file error "** + e.getMessage());  
 }  
 }  
}

**import** java.io.IOException;  
**import** java.io.InputStream;  
**import** java.io.OutputStream;  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**public class** ElfParser {  
 **private final** Map<Integer, String> **NDX** = Map.*of*(  
 0, **"UND"**,  
 65280, **"LORESERVE"**,  
 65311, **"HIPROC"**,  
 65521, **"ABS"**,  
 65522, **"COMMON"**,  
 65535, **"HIRESERVE"** );  
 **private final** Map<Integer, String> **BIND** = Map.*of*(  
 0, **"LOCAL"**,  
 1, **"GLOBAL"**,  
 2, **"WEAK"**,  
 13, **"LOPROC"**,  
 15, **"HIPROC"** );  
 **private final** Map<Integer, String> **TYPE** = Map.*of*(  
 0, **"NOTYPE"**,  
 1, **"OBJECT"**,  
 2, **"FUNC"**,  
 3, **"SECTION"**,  
 4, **"FILE"**,  
 13, **"LOPROC"**,  
 15, **"HIPROC"** );  
 **private** TextSymtable **elfData**;  
 **private int strtabOffset**;  
 **private int startAddr**;  
 **private int**[] **elf**;  
 **private char**[] **ident**;  
 **private int type**;  
 **private int machine**;  
 **private int version**;  
 **private int entry**;  
 **private int phoff**;  
 **private int shoff**;  
 **private int flags**;  
 **private int ehsize**;  
 **private int phentsize**;  
 **private int phnum**;  
 **private int shentsize**;  
 **private int shnum**;  
 **private int shstrndx**;  
 **private boolean log**;  
  
 ElfParser(InputStream elfFile, **boolean** log) {  
 **this**.**log** = log;  
 **ident** = **new char**[16];  
 **try** {  
 **byte**[] elf\_byte = elfFile.readAllBytes();  
 elfFile.close();  
 **elf** = **new int**[elf\_byte.**length**];  
 **for** (**int** i = 0; i != **elf**.**length**; i++) {  
 **elf**[i] = Byte.*toUnsignedInt*(elf\_byte[i]);  
 }  
 **for** (**int** i = 0; i != 16; i++) {  
 **ident**[i] = (**char**) elf\_byte[i];  
 }  
 **type** = decodeLSB16(16);  
 **machine** = decodeLSB16(18);  
 **version** = decodeLSB32(20);  
 **entry** = decodeLSB32(24);  
 **phoff** = decodeLSB32(28);  
 **shoff** = decodeLSB32(32);  
 **flags** = decodeLSB32(36);  
 **ehsize** = decodeLSB16(40);  
 **phentsize** = decodeLSB16(42);  
 **phnum** = decodeLSB16(44);  
 **shentsize** = decodeLSB16(46);  
 **shnum** = decodeLSB16(48);  
 **shstrndx** = decodeLSB16(50);  
 **if** (!isValidElf()) {  
 **throw new** IllegalArgumentException(**"Incorrect elf file"**);  
 }  
 writeElfByteCode();  
 **elfData** = parse();  
 } **catch** (IOException e) {  
 System.***out***.println(**"Reading error "** + e.getMessage());  
 }  
 }  
  
 **public int**[] getTextData() {  
 **return elfData**.getText();  
 }  
  
 **public int** getStartAddr() {  
 **return startAddr**;  
 }  
  
 **private** TextSymtable parse() {  
 **return** parseSectionHeader();  
 }  
  
 **private** TextSymtable parseSectionHeader() {  
 ElfSectionHeader nameSection = makeSectionHeader(**shoff** + **shstrndx** \* **shentsize**);  
 TextSymtable data = **new** TextSymtable();  
 **for** (**int** i = 1; i < **shnum**; i++) {  
 ElfSectionHeader programSection = makeSectionHeader(**shoff** + i \* **shentsize**);  
 write(**"ProgramSection "** + i + **"\n"**);  
 String name = getName(nameSection.**offset** + programSection.**name**);  
 write(name + **"\n"**);  
 programSection.write();  
 **if** (name.equals(**".text"**)) {  
 data.setText(readCode(programSection.**offset**, programSection.**size**));  
 **startAddr** = programSection.**addr**;  
 } **else if** (name.equals(**".symtab"**)) {  
 data.setSymtable(programSection);  
 } **else if** (name.equals(**".strtab"**)) {  
 **strtabOffset** = programSection.**offset**;  
 }  
 write(**"\n"**);  
 }  
 **return** data;  
 }  
  
 **public void** getSymtableString(OutputStream out) {  
 **int** offset = **elfData**.getSymtable().**offset**, size = **elfData**.getSymtable().**size**, nameOffset = **strtabOffset**;  
 **try** {  
 out.write(String.*format*(**"%3s:%9s%7s %-7s %-6s %-7s %9s %s\n"**, **"Num"**, **"Value"**, **"Size"**, **"Type"**, **"Bind"**, **"Vis"**, **"Ndx"**, **"Name"**).getBytes());  
 **for** (**int** i = 0; i + 15 < size; i += 16) {  
 String name = getName(nameOffset + decodeLSB32(offset + i));  
 **int** value = decodeLSB32(offset + i + 4),  
 symSize = decodeLSB32(offset + i + 8),  
 shndx = decodeLSB16(offset + i + 14),  
 info = decodeLSB16(offset + i + 12) % 256;  
 **int** bind = info >> 4, type = info & 0xf;  
 out.write(String.*format*(**"%3d: %08x %6d %-7s %-6s %-7s %9s %s\n"**, i / 16, value, symSize, **TYPE**.get(type), **BIND**.get(bind), **"DEFAULT"**, **NDX**.containsKey(shndx) ? **NDX**.get(shndx) : Integer.*toString*(shndx), name).getBytes());  
 }  
 } **catch** (IOException e) {  
 System.***out***.println(**"Writing error "** + e.getMessage());  
 }  
 }  
  
 **public** Map<Integer,String> getSymtableData() {  
 Map<Integer,String> data = **new** HashMap<>();  
 **int** offset = **elfData**.getSymtable().**offset**, size = **elfData**.getSymtable().**size**, nameOffset = **strtabOffset**;  
 **for** (**int** i = 0; i + 15 < size; i += 16) {  
 String name = getName(nameOffset + decodeLSB32(offset + i));  
 **int** value = decodeLSB32(offset + i + 4),  
 bind = (decodeLSB16(offset + i + 12) % 256) >> 4;  
 **if** (bind == 1) {  
 data.put(value, name);  
 }  
 }  
 **return** data;  
 }  
  
 **private** ElfSectionHeader makeSectionHeader(**int** index) {  
 **return new** ElfSectionHeader(decodeLSB32(index), decodeLSB32(index + 4),  
 decodeLSB32(index + 8), decodeLSB32(index + 12), decodeLSB32(index + 16),  
 decodeLSB32(index + 20), decodeLSB32(index + 24), decodeLSB32(index + 28),  
 decodeLSB32(index + 32), decodeLSB32(index + 36), **log**);  
 }  
  
 **private boolean** isValidElf() {  
 **return ident**[0] == 127 && **ident**[1] == **'E'** && **ident**[2] == **'L'** && **ident**[3] == **'F'** &&  
 **ident**[4] == 1 && **ident**[5] == 1 && **type** == 2 && **flags** == 0 && **ehsize** == 52;  
 }  
  
 **private int**[] readCode(**int** offset, **int** size) {  
 **int**[] data = **new int**[size];  
 **for** (**int** i = offset; i + 3 < offset + size; i += 4) {  
 writeBytes(data, i - offset, i, 4);  
 write(**"\n"**);  
 }  
 **return** data;  
 }  
  
 **private int** decodeLSB16(**int** index) {  
 **int** x1 = **elf**[index], x2 = **elf**[index + 1];  
 **return** x1 + x2 \* (1 << 8);  
 }  
  
 **private int** decodeLSB32(**int** index) {  
 **int** x1 = **elf**[index], x2 = **elf**[index + 1], x3 = **elf**[index + 2], x4 = **elf**[index + 3];  
 **return** x1 + x2 \* (1 << 8) + x3 \* (1 << 16) + x4 \* (1 << 24);  
 }  
  
 **private void** writeElfByteCode() {  
 **for** (**int** i = 0; i != 16; i++) {  
 write(+**ident**[i] + **" "**);  
 }  
 write(**"\n"**);  
 write(**"type: "** + **type** + **"\nmachine: "** + **machine** + **"\nversion: "** + **version** +  
 **"\nentry: "** + **entry** + **"\nphoff: "** + **phoff** + **"\nshoff: "** + **shoff** + **"\nflags: "** + **flags** +  
 **"\nehsize: "** + **ehsize** + **"\nphentsize: "** + **phentsize** + **"\nphnum: "** + **phnum** +  
 **"\nshentsize: "** + **shentsize** + **"\nshnum: "** + **shnum** + **"\nshstrndx: "** + **shstrndx** + **"\n"**);  
 **for** (**int** i = 0; i != **elf**.**length**; ++i) {  
 write(Integer.*toHexString*(**elf**[i]) + **" "**);  
 }  
 write(**"\n"**);  
 }  
  
 **private void** writeBytes(**int**[] data, **int** offset, **int** index, **int** size) {  
 **for** (**int** i = index + size - 1; i >= index; i--) {  
 write(getBits(i));  
 data[offset + index + size - 1 - i] = **elf**[i];  
 **if** (i + 1 != index + size) {  
 write(**"'"**);  
 }  
 }  
 }  
  
 **private** String getBits(**int** index) {  
 String bits = Integer.*toBinaryString*(**elf**[index]);  
 **for** (; bits.length() != 8; bits = **"0"** + bits);  
 **return** bits;  
 }  
  
 **private** String getName(**int** index) {  
 String name = **""**;  
 **for** (**int** i = index; **elf**[i] != 0; i++) {  
 name = name + ((**char**) **elf**[i]);  
 }  
 **return** name;  
 }  
  
 **private void** write(String message) {  
 **if** (**log**) {  
 System.***out***.print(message);  
 }  
 }  
}

**public class** ElfSectionHeader {  
 **int name**;  
 **private int type**;  
 **private int flags**;  
 **int addr**;  
 **int offset**;  
 **int size**;  
 **private int link**;  
 **private int info**;  
 **private int addralign**;  
 **private int entsize**;  
 **private boolean log**;  
  
 **public** ElfSectionHeader(**int** name, **int** type, **int** flags, **int** addr, **int** offset, **int** size, **int** link, **int** info, **int** addralign, **int** entsize, **boolean** log) {  
 **this**.**name** = name;  
 **this**.**type** = type;  
 **this**.**flags** = flags;  
 **this**.**addr** = addr;  
 **this**.**offset** = offset;  
 **this**.**size** = size;  
 **this**.**link** = link;  
 **this**.**info** = info;  
 **this**.**addralign** = addralign;  
 **this**.**entsize** = entsize;  
 **this**.**log** = log;  
 }  
  
 **public void** write() {  
 **if** (**log**) {  
 System.***out***.println(**"name: "** + **name** + **"\ntype: "** + **type** + **"\nflags: "** + **flags** +  
 **"\naddr: "** + **addr** + **"\noffset: "** + **offset** + **"\nsize: "** +  
 **size** + **"\nlink: "** + **link** + **"\ninfo: "** + **info** +  
 **"\naddralign: "** + **addralign** + **"\nentsize: "** + **entsize**);  
 }  
 }  
}

**import** java.util.HashMap;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.Set;  
  
**public class** DisassemblerBlock {  
 **private final** Map<String, String> **OPCODE\_UJ** = Map.*of*(  
 **"0110111"**, **"LUI"**,  
 **"0010111"**, **"AUIPC"**,  
 **"1101111"**, **"JAL"** );  
  
 **private final** Map<String, Map <Integer, String>> **OPCODE\_ISB** = Map.*of*(  
 **"1100111"**, Map.*of*(  
 0,**"JALR"**),  
 **"1100011"**, Map.*of*(  
 0,**"BEQ"**,  
 1,**"BNE"**,  
 4,**"BLT"**,  
 5,**"BGE"**,  
 6,**"BLTU"**,  
 7,**"BGEU"** ),  
 **"0000011"**, Map.*of*(  
 0, **"LB"**,  
 1, **"LH"**,  
 2, **"LW"**,  
 4, **"LBU"**,  
 5, **"LHU"** ),  
 **"0100011"**, Map.*of*(  
 0, **"SB"**,  
 1, **"SH"**,  
 2, **"SW"** ),  
 **"0010011"**, Map.*of*(  
 0, **"ADDI"**,  
 2, **"SLTI"**,  
 3, **"SLTIU"**,  
 4, **"XORI"**,  
 6, **"ORI"**,  
 7, **"ANDI"** )  
 );  
 **private final** Map<String, Map <Integer, Map <String, String>>> **OPCODE\_R** = Map.*of*(  
 **"0010011"**, Map.*of*(  
 1, Map.*of*(  
 **"0000000"**, **"SLLI"** ),  
 5, Map.*of*(  
 **"0000000"**, **"SRLI"**,  
 **"0100000"**, **"SRAI"** )  
 ),  
 **"0110011"**, Map.*of*(  
 0, Map.*of*(  
 **"0000000"**, **"ADD"**,  
 **"0100000"**, **"SUB"**,  
 **"0000001"**, **"MUL"** ),  
 1, Map.*of*(  
 **"0000000"**, **"SLL"**,  
 **"0000001"**, **"MULH"** ),  
 2, Map.*of*(  
 **"0000000"**, **"SLT"**,  
 **"0000001"**, **"MULHSU"** ),  
 3, Map.*of*(  
 **"0000000"**, **"SLTU"**,  
 **"0000001"**, **"MULHU"** ),  
 4, Map.*of*(  
 **"0000000"**, **"XOR"**,  
 **"0000001"**, **"DIV"** ),  
 5, Map.*of*(  
 **"0000000"**, **"SRL"**,  
 **"0100000"**, **"SRA"**,  
 **"0000001"**, **"DIVU"** ),  
 6, Map.*of*(  
 **"0000000"**, **"OR"**,  
 **"0000001"**, **"REM"** ),  
 7, Map.*of*(  
 **"0000000"**, **"AND"**,  
 **"0000001"**, **"REMU"** )  
 )  
 );  
 **private final** List<String> **regName** = List.*of*(  
 **"zero"**,  
 **"ra"**,  
 **"sp"**,  
 **"gp"**,  
 **"tp"**,  
 **"t0"**,  
 **"t1"**,  
 **"t2"**,  
 **"s0"**,  
 **"s1"**,  
 **"a0"**,  
 **"a1"**,  
 **"a2"**,  
 **"a3"**,  
 **"a4"**,  
 **"a5"**,  
 **"a6"**,  
 **"a7"**,  
 **"s2"**,  
 **"s3"**,  
 **"s4"**,  
 **"s5"**,  
 **"s6"**,  
 **"s7"**,  
 **"s8"**,  
 **"s9"**,  
 **"s10"**,  
 **"s11"**,  
 **"t3"**,  
 **"t4"**,  
 **"t5"**,  
 **"t6"** );  
 **private final** Map<String, Character> **TYPE**;  
 **private int indexL**;  
 **public** DisassemblerBlock() {  
 **TYPE** = **new** HashMap<>();  
 **indexL** = 0;  
 **TYPE**.putAll(Map.*of*(  
 **"LUI"**, **'U'**,  
 **"AUIPC"**, **'U'**,  
 **"JAL"**, **'J'**,  
 **"JALR"**, **'I'**,  
 **"BEQ"**, **'B'**,  
 **"BNE"**, **'B'**,  
 **"BLT"**, **'B'**,  
 **"BGE"**, **'B'**,  
 **"BLTU"**, **'B'**,  
 **"BGEU"**, **'B'** ));  
 **TYPE**.putAll(Map.*of*(  
 **"LB"**, **'I'**,  
 **"LH"**, **'I'**,  
 **"LW"**, **'I'**,  
 **"LBU"**, **'I'**,  
 **"LHU"**, **'I'**,  
 **"SB"**, **'S'**,  
 **"SH"**, **'S'**,  
 **"SW"**, **'S'**,  
 **"ADDI"**, **'I'**,  
 **"SLTI"**, **'I'** ));  
 **TYPE**.putAll(Map.*of*(  
 **"SLTIU"**, **'I'**,  
 **"XORI"**, **'I'**,  
 **"ORI"**, **'I'**,  
 **"ANDI"**, **'I'**,  
 **"SLLI"**, **'R'**,  
 **"SRLI"**, **'R'**,  
 **"SRAI"**, **'R'**,  
 **"ADD"**, **'R'**,  
 **"SUB"**, **'R'**,  
 **"SLL"**, **'R'** ));  
 **TYPE**.putAll(Map.*of*(  
 **"SLTU"**, **'R'**,  
 **"XOR"**, **'R'**,  
 **"SRL"**, **'R'**,  
 **"SRA"**, **'R'**,  
 **"OR"**, **'R'**,  
 **"AND"**, **'R'**,  
 **"MUL"**, **'R'**,  
 **"MULH"**, **'R'**,  
 **"MULHSU"**, **'R'**,  
 **"MULHU"**, **'R'** ));  
 **TYPE**.putAll(Map.*of*(  
 **"DIV"**, **'R'**,  
 **"DIVU"**, **'R'**,  
 **"REM"**, **'R'**,  
 **"REMU"**, **'R'** ));  
 }  
  
 **public** String parseCommands(**final int**[] commands, **int** addrStart, **final** Map<Integer,String> symtabData) {  
 **int** addr = addrStart;  
 **for** (**int** i = 0; i + 3 < commands.**length**; i += 4) {  
 parseCommand(get32Bits(commands[i], commands[i + 1], commands[i + 2], commands[i + 3]), addr, symtabData);  
 addr += 4;  
 }  
 StringBuilder ansWithLabel = **new** StringBuilder();  
 addr = addrStart;  
 **for** (**int** i = 0; i + 3 < commands.**length**; i += 4) {  
 **if** (symtabData.containsKey(addr)) {  
 ansWithLabel.append(String.*format*( **"\n%08x <%s>:\n"**, addr, symtabData.get(addr)));  
 }  
 ansWithLabel.append(String.*format*(**"%8x:\t"**, addr));  
 ansWithLabel.append(parseCommand(get32Bits(commands[i], commands[i + 1], commands[i + 2], commands[i + 3]), addr, symtabData));  
 ansWithLabel.append(**"\n"**);  
 addr += 4;  
 }  
 **return** ansWithLabel.toString();  
 }  
  
 **private** String parseCommand(**final** String bits, **int** addr, Map<Integer,String> symtabData) {  
 String name = getName(bits);  
 name = (name == **null** ? **"unknown\_instruction"** : name.toLowerCase());  
 String outString = String.*format*(**"%s\t\t%-20s"**, codeToHex(bits), name);  
 **if** (name.equals(**"unknown\_instruction"**) || name.equals(**"ecall"**) || name.equals(**"ebreak"**)) {  
 **return** outString;  
 }  
 **char** type = getType(name.toUpperCase());  
 **if** (type == **'R'**) {  
 **return** outString + parseTypeR(bits);  
 } **else if** (type == **'I'**) {  
 **return** outString + (Set.*of*(**"lb"**, **"lh"**, **"lw"**, **"lbu"**, **"lhu"**, **"jalr"**).contains(name) ? parseTypeI\_LJ(bits) : parseTypeI\_Other(bits));  
 } **else if** (type == **'S'**) {  
 **return** outString + parseTypeS(bits);  
 } **else if** (type == **'B'**) {  
 **return** outString + parseTypeB(bits, addr, symtabData);  
 } **else if** (type == **'U'**) {  
 **return** outString + parseTypeU(bits);  
 } **else if** (type == **'J'**) {  
 **return** outString + parseTypeJ(bits, addr, symtabData);  
 } **else** {  
 **return** codeToHex(bits) + **"\tunknown\_instruction"**;  
 }  
 }  
  
 **private** String parseTypeR(String bits) {  
 **int** rs2 = getRegisterNumber(bits, 7, 5), rs1 = getRegisterNumber(bits, 12, 5), rd = getRegisterNumber(bits, 20, 5);  
 **return regName**.get(rd) + **","** + **regName**.get(rs1) + **","** + **regName**.get(rs2);  
 }  
  
 **private** String parseTypeI\_LJ(String bits) {  
 **int** imm = getNumber(bits, 0, 12), rs1 = getRegisterNumber(bits, 12, 5), rd = getRegisterNumber(bits, 20, 5);  
 **return regName**.get(rd) + **","** + imm + **"("** + **regName**.get(rs1) + **")"**;  
 }  
  
 **private** String parseTypeI\_Other(String bits) {  
 **int** imm = getNumber(bits, 0, 12), rs1 = getRegisterNumber(bits, 12, 5), rd = getRegisterNumber(bits, 20, 5);  
 **return regName**.get(rd) + **","** + **regName**.get(rs1) + **","** + imm;  
 }  
  
 **private** String parseTypeS(String bits) {  
 **int** rs2 = getRegisterNumber(bits, 7, 5), rs1 = getRegisterNumber(bits, 12, 5),  
 imm = getNumber(bits.substring(0,7) + bits.substring(20,25), 0, 12);  
 **return regName**.get(rs2) + **","** + imm + **"("** + **regName**.get(rs1) + **")"**;  
 }  
  
 **private** String parseTypeB(String bits, **int** addr, Map<Integer,String> symtabData) {  
 **int** rs2 = getRegisterNumber(bits, 7, 5), rs1 = getRegisterNumber(bits, 12, 5),  
 imm = 2 \* getNumber(bits.substring(0,1) + bits.substring(24,25) + bits.substring(1,7) + bits.substring(20,24), 0, 12);  
 **return regName**.get(rs1) + **","** + **regName**.get(rs2) + **","** + Integer.*toHexString*(imm + addr) + **" <"** + getLabel(imm + addr, symtabData) + **">"**;  
 }  
  
 **private** String parseTypeU(String bits) {  
 **int** imm = getNumber(bits, 0, 20), rd = getRegisterNumber(bits, 20, 5);  
 **return regName**.get(rd) + **",0x"** + Integer.*toHexString*(imm);  
 }  
  
 **private** String parseTypeJ(String bits, **int** addr, Map<Integer,String> symtabData) {  
 **int** rd = getRegisterNumber(bits, 20, 5),  
 imm = 2 \* getNumber(bits.substring(0,1) + bits.substring(12,20) + bits.substring(11,12) + bits.substring(1,11), 0, 20);  
 **return regName**.get(rd) + **","** + Integer.*toHexString*(imm + addr) + **" <"** + getLabel(imm + addr, symtabData) + **">"**;  
 }  
  
 **private** String getLabel(**int** addr, Map<Integer,String> symtabData) {  
 **if** (!symtabData.containsKey(addr)) {  
 symtabData.put(addr, **"L"** + **indexL**);  
 **indexL**++;  
 }  
 **return** symtabData.get(addr);  
 }  
  
 **private int** getRegisterNumber(String bits, **int** first, **int** size) {  
 **int** value = 0;  
 **for** (**int** i = first; i != first + size; i++) {  
 value = value<<1;  
 **if** (bits.charAt(i) == **'1'**) {  
 value += 1;  
 }  
 }  
 **return** value;  
 }  
  
 **private int** getNumber(String bits, **int** first, **int** size) {  
 **int** value = 0;  
 **for** (**int** i = first; i != first + size; i++) {  
 value = value<<1;  
 **if** (bits.charAt(i) == **'1'**) {  
 **if** (i == first) {  
 value -= 1;  
 } **else** {  
 value += 1;  
 }  
 }  
 }  
 **return** value;  
 }  
  
 **private char** getType(**final** String name) {  
 **return TYPE**.get(name);  
 }  
  
 **private** String codeToHex(String bits) {  
 **return** Integer.*toHexString*(getRegisterNumber(bits, 0 , 4)) + Integer.*toHexString*(getRegisterNumber(bits, 4 , 4)) +  
 Integer.*toHexString*(getRegisterNumber(bits, 8 , 4)) + Integer.*toHexString*(getRegisterNumber(bits, 12 , 4)) +  
 Integer.*toHexString*(getRegisterNumber(bits, 16 , 4)) + Integer.*toHexString*(getRegisterNumber(bits, 20 , 4)) +  
 Integer.*toHexString*(getRegisterNumber(bits, 24 , 4)) + Integer.*toHexString*(getRegisterNumber(bits, 28 , 4));  
 }  
  
 **private** String getName(**final** String bits) {  
 **if** (bits.equals(**"00000000000000000000000001110011"**)) {  
 **return "ECALL"**;  
 }  
 **if** (bits.equals(**"00000000000100000000000001110011"**)) {  
 **return "EBREAK"**;  
 }  
 String opcode = bits.substring(25, 32),  
 funct7 = bits.substring(0, 7);  
 **int** funct3 = Integer.*parseInt*(bits.substring(17, 20), 2);  
 **if** (**OPCODE\_UJ**.containsKey(opcode)) {  
 **return OPCODE\_UJ**.get(opcode);  
 } **else if** (**OPCODE\_ISB**.containsKey(opcode) && **OPCODE\_ISB**.get(opcode).containsKey(funct3)) {  
 **return OPCODE\_ISB**.get(opcode).get(funct3);  
 } **else if** (**OPCODE\_R**.containsKey(opcode) && **OPCODE\_R**.get(opcode).containsKey(funct3) && **OPCODE\_R**.get(opcode).get(funct3).containsKey(funct7)) {  
 **return OPCODE\_R**.get(opcode).get(funct3).get(funct7);  
 } **else** {  
 **return null**;  
 }  
 }  
  
 **private** String get32Bits(**int** b1, **int** b2, **int** b3, **int** b4) {  
 **return** get8Bits(b1) + get8Bits(b2) + get8Bits(b3) + get8Bits(b4);  
 }  
  
 **private** String get8Bits(**int** x) {  
 String bits = Integer.*toBinaryString*(x);  
 **for** (; bits.length() != 8; bits = **"0"** + bits);  
 **return** bits;  
 }  
}

**public class** TextSymtable {  
 **private int**[] **text**;  
 **private** ElfSectionHeader **symtable**;  
  
 **public** TextSymtable() {}  
  
 **public int**[] getText() {  
 **return text**;  
 }  
  
 **public** ElfSectionHeader getSymtable() {  
 **return symtable**;  
 }  
  
 **public void** setText(**int**[] text) {  
 **this**.**text** = text;  
 }  
  
 **public void** setSymtable(ElfSectionHeader symtable) {  
 **this**.**symtable** = symtable;  
 }  
}