САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

«ISA. Ассемблер, дизассемблер»

Выполнил(а): Елисеев Александр Сергеевич

Номер ИСУ: 334855

студ. гр. М3138

Санкт-Петербург

2021

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

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

Теоретическая часть

RISC-V - открытая ISA, представленная в 2010 году. Существуют две базовые версии: RV32I и RV64I для работы с 32- и 64- битными целыми числами. Также имеется довольно большое количество расширений, таких как M, предоставляющее команды умножения и деления, F, предоставляющее комады для работы с числами с плавающей точкой и др. Версия спецификации, используемая мной находится по адресу https://github.com/eliseevh/RISC-V-disassembler/blob/master/pdfs/riscv-spec-20191213.pdf.

Всего в RISC-V есть 32 регистра, которые называются x0, x1, …, x31. Помимо этих названий у них есть ABI-имена, отражающие их назначение, например x0 называется zero, т.к. он всегда равен нулю. Список имен регистров можно найти на 137 странице приложенной спецификации.

В RV32I всего есть 40 команд, из которых нам требуется обрабатывать 39 (FENCE не требуется). Также требуется обрабатывать (В задании написано, что эти команды являются частью RV32I, но в спецификации они выделены в расширение) RV32 Zicsr Standard Extension (команды для работы с Control and Status Register, CSR), содержащее 6 команд. Также требуется обрабатывать расширение M, содержащее 8 команд и расширение C (сжатые команды размером по 2 байта являющиеся сокращениями команд RV32I), содержащее 49 команд, но нам нужны только 27 из них, т.к. остальные являются сжатыми вариантами

команд из RV64I. RV128I и команд для работы с числами с плавающей точкой. Полный список команд можно найти в приложенной спецификации.

В базовой версии все команды имеют длину ровно 4 байта, младшие 2 бита всегда равны 11. Младшие 7 бит задают opcode, определяющий тип команды, остальные биты задают номера регистров, константы и конкретные команды. Всего есть 6 типов команд:

R - команды, совершающие операции над двумя регистрами, и записывающие результат в третий регистр.

I - команды, совершающие операции над регистром и константой, и записывающие результат в третий регистр. Также сюда относятся команды, загружающие в регистр значение из памяти по адресу константа + регистр.

S - команды, сохраняющее значение регистра в память по адресу константа + регистр.

B - команды условного перехода. Сравнивают значения двух регистров и в зависимости от результата прибавляют к pc указанную константу.

J - команда безусловного перехода, записывает в указанный регистр pc + 4 и прибавляет к pc указанную константу.

Также можно выделить седьмой тип - команды, обращающиеся к системе, это ecall, ebreak и все команды расширения Zicsr. У ecall и ebreak нет никаких аргументов, у команд из Zicsr есть три аргумента: csr, регистр и либо регистр, либо константа в зависимости от команды. csr - 12-битное число, задающее номер регистра CSR, все выделенные адреса регистров

можно найти в privileged спецификации. Версия, использованная мной: https://github.com/eliseevh/RISC-V-disassembler/blob/master/pdfs/riscv-privileged-20190608.pdf.

В сжатых командах младшие два бита не равны 11, именно так можно определить размер команды. Эти 2 бита задают opcode сжатой команды, однако их намного тяжелее разбить на типы, т.к. формат их записи достаточно сильно отличается друг от друга.

ELF - Executable and Linkable Format - формат двоичных файлов, широко используемый во многих Unix-подобных системах. Позволяет достаточно удобно хранить исполняемый код вместе с дополнительными данными. Также хранит достаточное количество дополнительной информации, позволяющей использовать формат для разных операционных систем и ISA.

Состоит из нескольких частей:

1) Заголовок файла. Хранит информацию о формате файла: в начале идут 4 байта, позволяющие определить, что это ELF-файл, дальше идут несколько однобайтных значений, задающих 32- или 64-битный формат, little endian или big endian формат многобайтных констант, целевую ISA и т.д. После этого хранятся адреса таблицы заголовков программы и таблицы разделов, а также кол-во вхождений в этих таблицах и размер одного вхождения.
2) Таблица заголовков программы. Хранит информацию о том, как система должна создать образ процесса. Не имеет ценности в рамках домашнего задания.

3) Таблица разделов. Хранит информацию о разделах файла. Каждая запись в таблице соответствует одному разделу. Есть особый раздел, содержащий информацию о именах других разделов, его индекс в таблице также хранится в заголовке файла. В каждой записи есть смещение адреса имени раздела, которое позволяет понять, где находится строка с именем раздела (все строки в ELF-файле заканчиваются нулевым байтом, поэтому достаточно хранить адрес начала строки). Также хранится байт, позволяющий определить тип раздела, и хранится некоторая информация, позволяющая корректно прочитать раздел, её интерпретация зависит от типа раздела.

Нам интересны только 2 раздела: .text и .symtab. В .text хранится код дизассемблируемой программы, в .symtab хранятся различные значения, нам интересны только метки.

Подробнее про .symtab:

1) Содержит список вхождений, размер каждого вхождения - 16 байт (в 32-битном ELF’e)
2) Каждое вхождение содержит значение, тип, имя и некоторые неважные нам поля. Нам нужны метки, они имеют тип FUNC, значение - адрес метки, имя - имя метки.

   В .text просто подряд хранятся команды RISC-V, нам известны адрес начала .text, совпадающий с адресом первой команды и размер раздела, т.е. суммарный размер всех команд.

Практическая часть

Язык программирования: Java.

При запуске программа проверяет кол-во переданных аргументов, если их меньше, чем 2, то выводится формат запуска. Если же их не меньше, то программа читает входной файл в массив байт, который далее преобразуется в объект класса ELFFile. Далее на его основе создаются объекты классов SymtabSection и TextSection, из которых создается объект класса DisassemblerToString, содержащий лишь один public метод: disasm(), который возвращает строку. Эта строка после этого записывается в выходной файл, после чего программа завершает свое выполнение.

DisassemblerToString достает из SymtabSection и TextSection метки (в SymtabSection это записи типа FUNC, в TextSection из всех условных и безусловных переходов на константное смещение мы можем получить метку для адреса, на который идет этот переход). Далее он дизассемблирует TextSection (для этого используется InstructionsParser, который преобразует TextSection в список Instruction'ов, каждый из которых умеет преобразовывать себя в строку) добавляя к строкам инструкций слева адрес и метку (если она есть), под которую выделяется столько символов, сколько в самой длинной метке, чтобы все команды имели одинаковый отступ от левого края, а также, если это переход на константное смещение, то в конец дописывает комментарий с меткой, на которую происходит переход. После этого он дизассемблирует SymtabSection (который, вообще-то сам преобразует себя в строку выводя заголовок таблицы, после которого выдает преобразованные в строки SymtabEntry).

Большая часть кода находится в static методах классов InstructionDecoding и CompressedInstructionDecoding. Эти методы позволяют по номеру

регистра получить его ABI имя, по номеру CSR регистра получить его имя (считается, что номера, не указанные в privileged спецификации, имеют имя равное записи этого номера в десятичной системе счисления), и по коду инструкции получить её строковое представление.

Пример работы программы:

Часть .text:

```
0001010c                  sum: c.addi sp, -32
0001010e                       c.swsp s0, 28
00010110                       addi4spn s0, sp, 32
00010112                       sw a0, -20(s0)
00010116                       sw a1, -24(s0)
0001011a                       lw a4, -20(s0)
0001011e                       lw a5, -24(s0)
00010122                       c.add a5, a4
00010124                       c.mv a0, a5
00010126                       c.lwsp s0, 28
00010128                       c.addi16sp sp, 32
0001012a                       c.jr ra
0001012c                 main: c.addi sp, -16
0001012e                       c.swsp ra, 12
00010130                       c.swsp s0, 8
00010132                       addi4spn s0, sp, 16
00010134                       c.li a1, 3
00010136                       c.li a0, 3
00010138                       c.jal -44 # sum
0001013a                       c.li a5, 0
0001013c                       c.mv a0, a5
0001013e                       c.lwsp ra, 12
```

```
00010140                        c.lwsp s0, 8

00010142                        c.addi sp, 16

00010144                        c.jr ra
```

Часть .symtab:

```
Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x11404               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x11408               0 SECTION  LOCAL    DEFAULT       3
[   4] 0x11410               0 SECTION  LOCAL    DEFAULT       4
[   5] 0x11418               0 SECTION  LOCAL    DEFAULT       5
[   6] 0x11848               0 SECTION  LOCAL    DEFAULT       6
[   7] 0x1185C               0 SECTION  LOCAL    DEFAULT       7
[   8] 0x0                   0 SECTION  LOCAL    DEFAULT       8
[   9] 0x0                   0 SECTION  LOCAL    DEFAULT       9
[  10] 0x0                   0 FILE     LOCAL    DEFAULT     ABS __call_atexit.c
[  11] 0x10074              18 FUNC     LOCAL    DEFAULT       1 register_fini
[  12] 0x0                   0 FILE     LOCAL    DEFAULT     ABS crtstuff.c
```

Листинг

Java

openjdk 17.0.1 2021-10-19

OpenJDK Runtime Environment Temurin-17.0.1+12 (build 17.0.1+12)

OpenJDK 64-Bit Server VM Temurin-17.0.1+12 (build 17.0.1+12, mixed mode, sharing)

(https://adoptium.net/?variant=openjdk17&jvmVariant=hotspot)

disasm/Disassembler.java

```
package disasm;


import disasm.ELF.ELFFile;
import disasm.ELF.SymtabSection;
import disasm.ELF.TextSection;
import disasm.RISC_V.DisassemblerToString;
import disasm.util.BytesOperations;


import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileWriter;
import java.io.IOException;


public class Disassembler {
   public static void main(String[] args) {
       if (args.length < 2) {
           System.out.printf("USAGE: java %s <input_filename> <output_filename>",
Disassembler.class.getName());
           return;
       }
       String input = args[0];
       String output = args[1];
       try {
           byte[] bytes = BytesOperations.readFile(input);
           ELFFile file = new ELFFile(bytes);
           SymtabSection symtab = new SymtabSection(file);
           TextSection text = new TextSection(file);
           DisassemblerToString disasm = new DisassemblerToString(symtab, text);
           try (BufferedWriter writer = new BufferedWriter(new
FileWriter(output))){
               writer.write(disasm.disasm());
           } catch (IOException e) {
               System.out.println("Cannot write to output file: " +
e.getMessage());
           }
       } catch (FileNotFoundException e) {
```

```
            System.out.println("Input file not found: " + e.getMessage());
        } catch (IOException e) {
            System.out.println("Cannot read input file: " + e.getMessage());
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

disasm/ELF/ELFFile.java

```
package disasm.ELF;

import static disasm.util.BytesOperations.*;

public class ELFFile {
    private final ELFSectionHeader[] sectionHeaders;
    private final ELFSectionHeader stringTableHeader;
    private final String[] sectionNames;

    private final byte[] file;

    private final static int SECTION_HEADER_ENTRY_SIZE = 0x28;


    public ELFFile(byte[] file) {
        this.file = file;
        ELFHeader header = new ELFHeader(file);
        int shoff = header.getSectionHeaderOffset();
        int shnum = header.getSectionHeaderEntriesNum();
        sectionHeaders = new ELFSectionHeader[shnum];
        sectionNames = new String[shnum];
        ELFSectionHeader stringTableSectionHeader = new ELFSectionHeader(file,
                shoff + header.getSectionStringTableIndex() *
SECTION_HEADER_ENTRY_SIZE);
        int sectionHeaderNamesOffset = stringTableSectionHeader.getOffset();
        for (int i = 0; i < shnum; i++) {
```

```
            sectionHeaders[i] = new ELFSectionHeader(file, shoff + i *
SECTION_HEADER_ENTRY_SIZE);
            sectionNames[i] = getNullTerminatedString(file,
                    sectionHeaderNamesOffset + sectionHeaders[i].getNameOffset());
        }
        stringTableHeader = getSectionHeader(".strtab");
    }

    public ELFSectionHeader getSectionHeader(String name) {
        for (int i = 0; i < sectionNames.length; i++) {
            if (sectionNames[i].equals(name)) {
                return sectionHeaders[i];
            }
        }
        throw new AssertionError("There is no section with name \"" + name + "\"");
    }

    public byte[] getFile() {
        return file;
    }

    public String getString(int offset) {
        return getNullTerminatedString(file, offset +
stringTableHeader.getOffset());
    }
}
```

disasm/ELF/ELFHeader.java

```
package disasm.ELF;

import static disasm.util.BytesOperations.*;

public class ELFHeader {
    private final static int MAGIC_CONST_LITTLE_ENDIAN = 0x46_4c_45_7f;
    private final static byte CLASS_32_BIT = 1;
    private final static byte DATA_LITTLE_ENDIAN = 1;
```

```
private final static short MACHINE_RISC_V = 0xf3;

private final int e_shoff;
private final short e_shnum;
private final short e_shstrndx;

public ELFHeader(byte[] header) {
    int ei_mag = getIntLittleEndian(header, 0);
    if (ei_mag != MAGIC_CONST_LITTLE_ENDIAN) {
        throw new AssertionError("Not an elf header");
    }
    byte ei_class = header[4];
    if (ei_class != CLASS_32_BIT) {
        throw new AssertionError("Not a 32-bit ELF");
    }
    byte ei_data = header[5];
    if (ei_data != DATA_LITTLE_ENDIAN) {
        throw  new AssertionError("Not a little endian encoded ELF");
    }
    short e_machine = getShortLittleEndian(header, 0x12);
    if (e_machine != MACHINE_RISC_V) {
        throw new AssertionError("Not a RISC-V ELF");
    }
    e_shoff = getIntLittleEndian(header, 0x20);
    short e_shentsize = getShortLittleEndian(header, 0x2e);
    if (e_shentsize != 0x28) {
        throw new AssertionError("Wrong section header table entry size");
    }
    e_shnum = getShortLittleEndian(header, 0x30);
    e_shstrndx = getShortLittleEndian(header, 0x32);
}

public int getSectionHeaderOffset() {
    return e_shoff;
}
```

```
    public short getSectionHeaderEntriesNum() {
        return e_shnum;
    }

    public short getSectionStringTableIndex() {
        return e_shstrndx;
    }
}
```

disasm/ELF/ELFSection.java

```
package disasm.ELF;

public class ELFSection {
    protected final byte[] bytes;
    protected final ELFSectionHeader header;

    public ELFSection(ELFFile file, String name) {
        header = file.getSectionHeader(name);
        int size = header.getSize();
        int offset = header.getOffset();
        bytes = new byte[size];
        System.arraycopy(file.getFile(), offset, bytes, 0, size);
    }
}
```

disasm/ELF/ELFSectionHeader.java

```
package disasm.ELF;

import static disasm.util.BytesOperations.*;

public class ELFSectionHeader {
    private final int sh_name;
    private final int sh_addr;
    private final int sh_offset;
```

```
    private final int sh_size;

    public ELFSectionHeader(byte[] header, int shoff) {
        sh_name = getIntLittleEndian(header, shoff);
        sh_addr = getIntLittleEndian(header, shoff + 0xc);
        sh_offset = getIntLittleEndian(header, shoff + 0x10);
        sh_size = getIntLittleEndian(header, shoff + 0x14);
    }

    public int getOffset() {
        return sh_offset;
    }

    public int getNameOffset() {
        return sh_name;
    }

    public int getSize() {
        return sh_size;
    }

    public int getAddress() {
        return sh_addr;
    }
}
```

disasm/ELF/SymtabEntry.java

```
package disasm.ELF;

import java.util.Map;

import static disasm.util.BytesOperations.*;

public class SymtabEntry {
    private final int value;
```

```
private final int size;
private final byte type;
private final byte bind;
private final byte visibility;
private final short idx;

private final int entryIdx;

private final String name;

private final static Map<Byte, String> TYPE_DECODE = Map.ofEntries(
        Map.entry((byte) 0, "NOTYPE"),
        Map.entry((byte) 1, "OBJECT"),
        Map.entry((byte) 2, "FUNC"),
        Map.entry((byte) 3, "SECTION"),
        Map.entry((byte) 4, "FILE"),
        Map.entry((byte) 5, "COMMON"),
        Map.entry((byte) 6, "TLS"),
        Map.entry((byte) 10, "OS"),
        Map.entry((byte) 11, "OS"),
        Map.entry((byte) 12, "OS"),
        Map.entry((byte) 13, "PROC"),
        Map.entry((byte) 14, "PROC"),
        Map.entry((byte) 15, "PROC")
);

private final static Map<Byte, String> BIND_DECODE = Map.of(
        (byte) 0, "LOCAL",
        (byte) 1, "GLOBAL",
        (byte) 3, "WEAK",
        (byte) 10, "OS",
        (byte) 11, "OS",
        (byte) 12, "OS",
        (byte) 13, "PROC",
        (byte) 14, "PROC",
```

```
        (byte) 15, "PROC"
);

private final static Map<Byte, String> VISIBILITY_DECODE = Map.of(
        (byte) 0, "DEFAULT",
        (byte) 1, "INTERNAL",
        (byte) 2, "HIDDEN",
        (byte) 3, "PROTECTED",
        (byte) 4, "EXPORTED",
        (byte) 5, "SINGLETON",
        (byte) 6, "ELIMINATE"
);


public static String decodeIdx(short index) {
    int idx = Short.toUnsignedInt(index);
    if (idx == 0) {
        return "UNDEF";
    }
    if (idx == 0xff00) {
        return "BEFORE";
    }
    if (idx == 0xff01) {
        return "AFTER";
    }
    if (idx == 0xfff1) {
        return "ABS";
    }
    if (idx == 0xfff2) {
        return "IDX";
    }
    if (idx == 0xffff) {
        return "XINDEX";
    }
    if (0xff00 <= idx && idx <= 0xff1f) {
```

```
            return "PROC";
        }
        if (0xff20 <= idx && idx <= 0xff3f) {
            return "OS";
        }
        if (0xff00 <= idx) {
            return "RESERVE";
        }
        return Integer.toString(idx);
    }

    public String getType() {
        return TYPE_DECODE.get(type);
    }

    public String getName() {
        return name;
    }

    public int getValue() {
        return value;
    }

    public SymtabEntry(byte[] bytes, int offset, ELFFile file, int entryIdx) {
        this.entryIdx = entryIdx;
        int nameOffset = getIntLittleEndian(bytes, offset);
        value = getIntLittleEndian(bytes, offset + 4);
        size = getIntLittleEndian(bytes, offset + 8);
        byte info = bytes[offset + 0xc];
        type = (byte) (info & 0xf);
        bind = (byte) (info >>> 4);
        byte other = bytes[offset + 0xd];
        visibility = (byte) (other & 0x3);
        idx = getShortLittleEndian(bytes, offset + 0xe);
        if (nameOffset == 0) {
```

```
            name = "";
        } else {
            name = file.getString(nameOffset);
        }
    }

    @Override
    public String toString() {
        return String.format("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                entryIdx,
                value,
                size,
                TYPE_DECODE.get(type),
                BIND_DECODE.get(bind),
                VISIBILITY_DECODE.get(visibility),
                decodeIdx(idx),
                name);
    }
}
```

disasm/ELF/SymtabSection.java

```
package disasm.ELF;

public class SymtabSection extends ELFSection {
    private final static int ENTRY_SIZE = 0x10;
    private final SymtabEntry[] entries;
    private final static String TABLE_HEADER = "Symbol Value              Size Type
Bind     Vis       Index Name\n";

    public SymtabSection(ELFFile file) {
        super(file, ".symtab");
        if (header.getSize() % ENTRY_SIZE != 0) {
            throw new AssertionError("Incorrect symtab size (must be divisible by "
                    + ENTRY_SIZE + "(entry size))");
        }
        entries = new SymtabEntry[header.getSize() / ENTRY_SIZE];
```

```
        for (int i = 0; i < entries.length; i++) {
            entries[i] = new SymtabEntry(bytes, i * ENTRY_SIZE, file, i);
        }
    }

    public SymtabEntry[] getEntries() {
        return entries;
    }

    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder(TABLE_HEADER);
        for (SymtabEntry entry : entries) {
            builder.append(entry.toString());
        }
        return builder.toString();
    }
}
```

disasm/ELF/TextSection.java

```
package disasm.ELF;

public class TextSection extends ELFSection {
    private final int address;

    public TextSection(ELFFile file) {
        super(file, ".text");
        address = header.getAddress();
    }

    public byte getByte(int idx) {
        return bytes[idx];
    }

    public byte[] getBytes(int offset, int size) {
```

```
        byte[] result = new byte[size];
        System.arraycopy(bytes, offset, result, 0, size);
        return result;
    }

    public int getSize() {
        return bytes.length;
    }

    public int getAddress() {
        return address;
    }
}
```

disasm/RISC_V/DisassemblerToString.java

```
package disasm.RISC_V;

import disasm.ELF.SymtabSection;
import disasm.ELF.TextSection;

import java.util.List;
import java.util.Map;

public class DisassemblerToString {
    private final List<Instruction> instructions;
    private final Map<Integer, String> labels;
    private final SymtabSection symtab;
    private final int maxLabelLen;

    public DisassemblerToString(SymtabSection symtab, TextSection text) {
        instructions = new InstructionsParser(text).getInstructions();
        labels = new LabelsParser(symtab, instructions).getLabels();
        this.symtab = symtab;
        int maxv = 0;
        for (Map.Entry<Integer, String> label : labels.entrySet()) {
```

```
            maxv = Math.max(maxv, label.getValue().length());
        }
        maxLabelLen = maxv;
    }

    public String disasm() {
        return textSectionDisasm() +
                "\n" +
                symtabSectionDisasm();
    }

    private String symtabSectionDisasm() {
        return ".symtab\n" + symtab;
    }

    private String textSectionDisasm() {
        StringBuilder builder = new StringBuilder(".text\n");
        for (int i = 0; i < instructions.size(); i++) {
            builder.append(getInstruction(i));
        }
        return builder.toString();
    }
    private String getInstruction(int idx) {
        Instruction inst = instructions.get(idx);
        int addr = inst.getAddress();
        String label = "";
        if (labels.containsKey(addr)) {
            label = labels.get(addr) + ":";
        }
        StringBuilder builder = new StringBuilder(
                String.format("%08x %" + (maxLabelLen + 1) + "s %s", addr, label,
inst));



        // add comment with jump destination label
        if (inst.getJumpOffset() != null) {
```

```
            builder.append(" # ").append(labels.get(inst.getAddress() +
inst.getJumpOffset()));
        }

        builder.append("\n");
        return builder.toString();
    }
}
```

disasm/RISC_V/Instruction.java

```
package disasm.RISC_V;

import disasm.ELF.TextSection;
import disasm.util.BytesOperations;
import disasm.util.CompressedInstructionDecoding;
import disasm.util.InstructionDecoding;
import disasm.util.UnknownCommandError;

import static disasm.util.InstructionDecoding.*;

public class Instruction {
    private final byte[] instruction;
    private final InstructionFormat format;
    private final int address;

    public Instruction(TextSection text, int offset) {
        boolean compressed = isCompressedLowByte(text.getByte(offset));
        address = text.getAddress() + offset;
        byte opcode = getOpcode(text.getByte(offset));
        if (compressed) {
            format = InstructionFormat.COMPRESSED;
        } else {
            format = getStandardFormat(opcode);
        }
        int size = compressed ? 2 : 4;
        instruction = text.getBytes(offset, size);
```

```
    }

    public boolean isCompressed() {
        return format == InstructionFormat.COMPRESSED;
    }

    public int getAddress() {
        return address;
    }

    @Override
    public String toString() {
        if (isCompressed()) {
            short op = BytesOperations.getShortLittleEndian(instruction, 0);
            try {
                return
CompressedInstructionDecoding.getCompressedInstructionRepresentation(op);
            } catch (UnknownCommandError e) {
                return "unknown_command";
            }
        }
        int operation = BytesOperations.getIntLittleEndian(instruction, 0);
        try {
            return switch (format) {
                case B ->
InstructionDecoding.getBInstructionRepresentation(operation);
                case I ->
InstructionDecoding.getIInstructionRepresentation(operation);
                case J ->
InstructionDecoding.getJInstructionRepresentation(operation);
                case R ->
InstructionDecoding.getRInstructionRepresentation(operation);
                case S ->
InstructionDecoding.getSInstructionRepresentation(operation);
                case U ->
InstructionDecoding.getUInstructionRepresentation(operation);
                case SYSTEM ->
InstructionDecoding.getSystemInstructionRepresentation(operation);
                default -> "unknown_command";
```

```
            };
        } catch (UnknownCommandError e) {
            return "unknown_command";
        }
    }

    public Integer getJumpOffset() {
        if (format == InstructionFormat.COMPRESSED) {
            short command = BytesOperations.getShortLittleEndian(this.instruction,
0);
            return CompressedInstructionDecoding.getCompressedJumpOffset(command);
        }
        int instruction = BytesOperations.getIntLittleEndian(this.instruction, 0);
        return switch (format) {
            case B -> InstructionDecoding.getBFormatOffset(instruction);
            case J -> InstructionDecoding.getJFormatOffset(instruction);
            default -> null;
        };
    }
}
```

disasm/RISC_V/InstructionFormat.java

```
package disasm.RISC_V;

public enum InstructionFormat {
    R, I, S, B, U, J,
    SYSTEM,
    COMPRESSED
}
```

disasm/RISC_V/InstructionsParser.java

```
package disasm.RISC_V;

import disasm.ELF.TextSection;

import java.util.ArrayList;
```

```java
import java.util.List;

public class InstructionsParser {
   private final List<Instruction> instructions;
   public InstructionsParser(TextSection text) {
       instructions = new ArrayList<>();
       int pos = 0;
       int size = text.getSize();
       while (pos < size) {
           Instruction inst = new Instruction(text, pos);
           instructions.add(inst);
           pos += inst.isCompressed() ? 2 : 4;
       }
   }

   public List<Instruction> getInstructions() {
       return instructions;
   }
}
```

disasm/RISC_V/LabelsParser.java

```java
package disasm.RISC_V;

import disasm.ELF.SymtabEntry;
import disasm.ELF.SymtabSection;

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

public class LabelsParser {
   private final Map<Integer, String> labels;
   public LabelsParser(SymtabSection symtab, List<Instruction> instructions) {
       labels = new HashMap<>();
       for (SymtabEntry entry : symtab.getEntries()) {
```

```
            if (entry.getType().equals("FUNC")) {
                String name;
                if (entry.getName().isEmpty()) {
                    name = String.format("LOC_%05x", entry.getValue());
                } else {
                    name = entry.getName();
                }
                labels.put(entry.getValue(), name);
            }
        }
        for (Instruction instruction : instructions) {
            if (instruction.getJumpOffset() != null) {
                int address = instruction.getAddress() +
instruction.getJumpOffset();
                if (!labels.containsKey(address)) {
                    labels.put(address, String.format("LOC_%05x", address));
                }
            }
        }
    }

    public Map<Integer, String> getLabels() {
        return labels;
    }
}
```

disasm/util/BytesOperations.java

```
package disasm.util;

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

public final class BytesOperations {
    private BytesOperations() {}
```

```java
public static byte[] readFile(String filename) throws IOException {
    InputStream input = new FileInputStream(filename);
    byte[] bytes = new byte[1024];
    int read;
    int size = 0;
    while ((read = input.read(bytes, size, bytes.length - size)) != -1) {
        size += read;
        if (size == bytes.length) {
            bytes = Arrays.copyOf(bytes, 2 * size);
        }
    }
    input.close();
    return Arrays.copyOf(bytes, size);
}

private static int getNumLittleEndian(byte[] bytes, int offset, int numBytes) {
    int num = 0;
    for (int i = offset; i < offset + numBytes; i++) {
        int byteVal = bytes[i] + (bytes[i] >= 0 ? 0 : 256);
        num |= byteVal << 8 * (i - offset);
    }
    return num;
}

public static int getSignExtension(int val, int size) {
    // >> is arithmetic shift, so it works fine
    return (val << (32 - size)) >> (32 - size);
}

public static int getIntLittleEndian(byte[] bytes, int offset) {
    return getNumLittleEndian(bytes, offset, 4);
}
public static short getShortLittleEndian(byte[] bytes, int offset) {
    return (short) getNumLittleEndian(bytes, offset, 2);
```

```
    }

    public static String getNullTerminatedString(byte[] bytes, int offset) {
        StringBuilder builder = new StringBuilder();
        char ch;
        int i = 0;
        while ((ch = (char)getNumLittleEndian(bytes, offset + i, 1)) != 0) {
            builder.append(ch);
            i++;
        }
        return builder.toString();
    }
}
```

disasm/util/CompressedInstructionDecoding.java

```
package disasm.util;

import static disasm.util.BytesOperations.getSignExtension;
import static disasm.util.InstructionDecoding.getRegisterName;

public final class CompressedInstructionDecoding {
    private CompressedInstructionDecoding() {}

    public static String getCompressedRegisterName(int register) {
        if (register < 2) {
            return "s" + register;
        }
        return "a" + (register - 2);
    }

    public static String getCompressedInstructionRepresentation(short command)
throws UnknownCommandError {
        byte b2to4 =   (byte) ((command & 0b0000000000011100) >>> 2);
        byte b5to6 =   (byte) ((command & 0b0000000001100000) >>> 5);
        byte b7to9 =   (byte) ((command & 0b0000001110000000) >>> 7);
        byte b10to12 = (byte) ((command & 0b0001110000000000) >>> 10);
```

```
        byte b13to15 = (byte) ((command & 0b1110000000000000) >>> 13);
        byte b5to12 = (byte) (b5to6 | (b7to9 << 2) | (b10to12 << 5));
        byte b2to6 = (byte) (b2to4 | (b5to6 << 3));
        byte b7to11 = (byte) (b7to9 | ((b10to12 & 0b011) << 3));
        byte b12 = (byte) ((b10to12 & 0b100) >>> 2);
        byte b7to12 = (byte) (b7to11 | (b12 << 5));
        short b2to12 = (short) (b2to6 | (b7to11 << 5) | (b12 << 10));
        byte b10to11 = (byte) (b10to12 & 0b11);
        byte opcode = (byte) (command & 0b11);

        int jimm = getSignExtension(
                ((b2to12 & 0b10000000000) << 1) +//11
                ((b2to12 & 0b01000000000) >>> 5) +   //4
                ((b2to12 & 0b00110000000) << 1) +    //9:8
                ((b2to12 & 0b00001000000) << 4) +    //10
                ((b2to12 & 0b00000100000) << 1) +    //6
                ((b2to12 & 0b00000010000) << 3) +    //7
                (b2to12 & 0b00000001110) +           //3:1
                ((b2to12 & 0b00000000001) << 5),     //5
        12);
        switch (opcode) {
            case 0b00:
                switch (b13to15) {
                    case 0b000 -> {
                        int nzuimm = ((b5to12 & 0b11000000) >>> 2) +
                                     ((b5to12 & 0b00111100) << 4) +
                                     ((b5to12 & 0b00000010) << 1) +
                                     ((b5to12 & 0b00000001) << 3);
                        // rd is 2-4
                        return String.format("addi4spn %s, sp, %d",
getCompressedRegisterName(b2to4), nzuimm);
                    }
                    case 0b010, 0b110 -> {
                        int uimm = ((b5to12 & 0b11100000) >>> 2) +
                                   ((b5to12 & 0b00000010) << 1) +
                                   ((b5to12 & 0b00000001) << 6);
```

```
                        final String inst;
                        if (b13to15 == 0b010) {
                            inst = "c.lw";
                        } else {
                            inst = "c.sw";
                        }
                        // rd is 2-4, rs is 7-9
                        return String.format("%s %s, %d(%s)",
                                inst, getCompressedRegisterName(b2to4), uimm,
getCompressedRegisterName(b7to9));
                    }
                    default -> throw new UnknownCommandError();
                }
            case 0b01:
                switch (b13to15) {
                    case 0b000 -> {
                        if (b7to11 == 0) {
                            return "c.nop";
                        } else {
                            int nzimm = (b2to6) + (b12 << 5);
                            // rd/rs is 7-11
                            return String.format("c.addi %s, %d",
                                    getRegisterName(b7to11),
getSignExtension(nzimm, 6));
                        }
                    }
                    case 0b001 -> {
                        return String.format("c.jal %d", jimm);
                    }
                    case 0b010 -> {
                        int imm = (b2to6) + (b12 << 5);
                        // rd is 7-11
                        return String.format("c.li %s, %d",
                                getRegisterName(b7to11), getSignExtension(imm, 6));
                    }
                    case 0b011 -> {
```

```
                    if (b7to11 == 2) {
                        int nzimm =  (b12 << 9) +           //9
                                ((b2to6 & 0b10000)) +       //4
                                ((b2to6 & 0b01000) << 3) +  //6
                                ((b2to6 & 0b00110) << 6) +  //8:7
                                ((b2to6 & 0b00001) << 5);   //5
                        return String.format("c.addi16sp sp, %d",
getSignExtension(nzimm, 10));
                    } else {
                        int nzimm = (b12 << 17) +           //17
                                ((b2to6) << 10);            //16:12
                        // rd is 7-11
                        return String.format("c.lui %s, %d",
                                getRegisterName(b7to11),
getSignExtension(nzimm, 18));
                    }
                }
                case 0b100 -> {
                    switch (b10to11) {
                        case 0b00 -> {
                            int shamt = (b12 << 5) + b2to6;
                            // rd/rs is 7-9
                            return String.format("c.srli %s, %d",
                                    getCompressedRegisterName(b7to9), shamt);
                        }
                        case 0b01 -> {
                            int shamt = (b12 << 5) + b2to6;
                            // rd/rs is 7-9
                            return String.format("c.srai %s, %d",
                                    getCompressedRegisterName(b7to9), shamt);
                        }
                        case 0b10 -> {
                            int imm = (b12 << 5) + b2to6;
                            // rd/rs is 7-9
                            return String.format("c.andi %s, %d",
                                    getCompressedRegisterName(b7to9),
getSignExtension(imm, 6));
```

```
                        }
                        case 0b11 -> {
                            String inst = switch (b5to6) {
                                case 0b00 -> "c.sub";
                                case 0b01 -> "c.xor";
                                case 0b10 -> "c.or";
                                case 0b11 -> "c.and";
                                default -> throw new AssertionError("Wrong
b5to6");
                            };
                            // rd/rs1 is 7-9, rs2 is 2-4
                            return String.format("%s %s, %s",
                                    inst, getCompressedRegisterName(b7to9),
getCompressedRegisterName(b2to4));
                        }
                        default -> throw new AssertionError("Wrong b10to11");
                    }
                }
                case 0b101 -> {
                    return String.format("c.j %d", jimm);
                }
                case 0b110, 0b111 -> {
                    int imm = ((b10to12 & 0b100) << 6) +   //8
                            ((b10to12 & 0b011) << 3) +     //4:3
                            ((b2to6 & 0b11000) << 3) +     //7:6
                            ((b2to6 & 0b00110)) +          //2:1
                            ((b2to6 & 0b00001) << 5);      //5
                    String inst;
                    if (b13to15 == 0b110) {
                        inst = "c.beqz";
                    } else {
                        inst = "c.bnez";
                    }
                    // rs is 7-9
                    return String.format("%s %s, %d",
                            inst, getCompressedRegisterName(b7to9),
getSignExtension(imm, 9));
```

```
                    }
                    default -> throw new UnknownCommandError();
                }
            case 0b10:
                switch (b13to15) {
                    case 0b000 -> {
                        int shamt = (b12 << 5) + b2to6;
                        // rd/rs is 7-11
                        return String.format("c.slli %s, %d",
                                getRegisterName(b7to11), shamt);
                    }
                    case 0b010 -> {
                        int uimm = (b12 << 5) +           //5
                                ((b2to6 & 0b11100)) +     //4:2
                                ((b2to6 & 0b00011) << 6); //7:6
                        // rd is 7-11
                        return String.format("c.lwsp %s, %d",
getRegisterName(b7to11), uimm);
                    }
                    case 0b100 -> {
                        if (b12 == 0) {
                            if (b2to6 == 0) {
                                // rs is 7-11
                                return String.format("c.jr %s",
getRegisterName(b7to11));
                            } else {
                                // rd is 7-11, rs is 2-6
                                return String.format("c.mv %s, %s",
getRegisterName(b7to11), getRegisterName(b2to6));
                            }
                        } else {
                            if (b7to11 == 0) {
                                if (b2to6 == 0) {
                                    return "c.ebreak";
                                } else {
                                    // rs2 is 2-6
```

```
                                return String.format("c.add zero, %s",
getRegisterName(b2to6));
                            }
                        } else {
                            if (b2to6 == 0) {
                                // rs is 7-11
                                return String.format("c.jalr %s",
getRegisterName(b7to11));
                            } else {
                                // rd/rs1 is 7-11, rs2 is 2-6
                                return String.format("c.add %s, %s",
                                        getRegisterName(b7to11),
getRegisterName(b2to6));
                            }
                        }
                    }
                    case 0b110 -> {
                        int uimm = (b7to12 & 0b111100) +     //5:2
                                ((b7to12 & 0b000011) << 6);  //7:6
                        // rs is 2-6
                        return String.format("c.swsp %s, %d",
getRegisterName(b2to6), uimm);
                    }
                    default -> throw new UnknownCommandError();
                }
            default:
                throw new AssertionError("Wrong opcode");
        }
    }

    public static Integer getCompressedJumpOffset(short command) {
        byte b2to4 =   (byte) ((command & 0b0000000000011100) >>> 2);
        byte b5to6 =   (byte) ((command & 0b0000000001100000) >>> 5);
        byte b7to9 =   (byte) ((command & 0b0000001110000000) >>> 7);
        byte b10to12 = (byte) ((command & 0b0001110000000000) >>> 10);
        byte b13to15 = (byte) ((command & 0b1110000000000000) >>> 13);
```

```
byte b2to6 = (byte) (b2to4 | (b5to6 << 3));
byte b7to11 = (byte) (b7to9 | ((b10to12 & 0b011) << 3));
byte b12 = (byte) ((b10to12 & 0b100) >>> 2);
short b2to12 = (short) (b2to6 | (b7to11 << 5) | (b12 << 10));
byte opcode = (byte) (command & 0b11);

int jimm = getSignExtension(
        ((b2to12 & 0b10000000000) << 1) +//11
                ((b2to12 & 0b01000000000) >>> 5) +   //4
                ((b2to12 & 0b00110000000) << 1) +    //9:8
                ((b2to12 & 0b00001000000) << 4) +    //10
                ((b2to12 & 0b00000100000) << 1) +    //6
                ((b2to12 & 0b00000010000) << 3) +    //7
                (b2to12 & 0b00000001110) +           //3:1
                ((b2to12 & 0b00000000001) << 5),     //5
        12);

if (opcode == 0b01) {
    switch (b13to15) {
        case 0b001, 0b101 -> {
            return jimm;
        }
        case 0b110, 0b111 -> {
            int imm = ((b10to12 & 0b100) << 6) +   //8
                    ((b10to12 & 0b011) << 3) +     //4:3
                    ((b2to6 & 0b11000) << 3) +     //7:6
                    ((b2to6 & 0b00110)) +          //2:1
                    ((b2to6 & 0b00001) << 5);      //5
            return getSignExtension(imm, 9);
        }
        default -> {
            return null;
        }
    }
} else {
```

```
            return null;
        }
    }
}
```

disasm/util/InstructionDecoding.java

```
package disasm.util;

import disasm.RISC_V.InstructionFormat;

import static disasm.util.BytesOperations.getSignExtension;

public final class InstructionDecoding {
    public static final int OPCODE_MASK = 0x00_00_00_7f;
    public static final int RD_MASK = 0x00_00_0f_80;
    public static final int RD_SHIFT = 7;
    public static final int FUNCT_3_MASK = 0x00_00_70_00;
    public static final int FUNCT_3_SHIFT = 12;
    public static final int RS_1_MASK = 0x00_0f_80_00;
    public static final int RS_1_SHIFT = 15;
    public static final int RS_2_MASK = 0x01_f0_00_00;
    public static final int RS_2_SHIFT = 20;
    public static final int FUNCT_7_MASK = 0xfe_00_00_00;
    public static final int FUNCT_7_SHIFT = 25;

    private InstructionDecoding() {}

    public static byte getOpcode(byte lowByte) {
        // in compressed instruction opcode is 2 lowest bits, in standard
instruction opcode is 7 lowest bits
        if (isCompressedLowByte(lowByte)) {
            return (byte) (lowByte & 0b11);
        } else {
            return (byte) (lowByte & 0b01_11_11_11);
        }
    }
```

```
    public static boolean isCompressedLowByte(byte lowByte) {
        byte low2 = (byte) (lowByte & 0b11);
        return low2 != 0b11;
    }

    public static InstructionFormat getStandardFormat(byte opcode) {
        return switch (opcode) {
            case 0b0110011 -> InstructionFormat.R;
            case 0b0010011, 0b0000011, 0b1100111 -> InstructionFormat.I;
            case 0b1110011 -> InstructionFormat.SYSTEM;
            case 0b0100011 -> InstructionFormat.S;
            case 0b1100011 -> InstructionFormat.B;
            case 0b1101111 -> InstructionFormat.J;
            case 0b0110111, 0b0010111 -> InstructionFormat.U;
            default -> throw new AssertionError("Not a standard format opcode");
        };
    }

    public static String getRegisterName(int register) {
        if (register < 0 || register > 31) {
            throw new IllegalArgumentException("In RISC-V register number must be
between 0 and 31, not " + register);
        }
        switch (register) {
            case 0:
                return "zero";
            case 1:
                return "ra";
            case 2:
                return "sp";
            case 3:
                return "gp";
            case 4:
                return "tp";
        }
```

```
    if (register <= 7) {
        return "t" + (register - 5);
    }
    if (register <= 9) {
        return "s" + (register - 8);
    }
    if (register <= 17) {
        return "a" + (register - 10);
    }
    if (register <= 27) {
        return "s" + (register - 16);
    }
    return "t" + (register - 25);
}


public static String getCSRName(int csr) {
    switch (csr) {
        case 0x000:
            return "ustatus";
        case 0x004:
            return "uie";
        case 0x005:
            return "utvec";
        case 0x040:
            return "uscratch";
        case 0x041:
            return "uepc";
        case 0x042:
            return "ucause";
        case 0x043:
            return "utval";
        case 0x044:
            return "uip";
        case 0x001:
```

```
        return "fflags";
    case 0x002:
        return "frm";
    case 0x003:
        return "fcsr";
    case 0xc00:
        return "cycle";
    case 0xc01:
        return "time";
    case 0xc02:
        return "instret";
    case 0xc80:
        return "cycleh";
    case 0xc81:
        return "timeh";
    case 0xc82:
        return "instreth";
    case 0x100:
        return "sstatus";
    case 0x102:
        return "sedeleg";
    case 0x103:
        return "sideleg";
    case 0x104:
        return "sie";
    case 0x105:
        return "stvec";
    case 0x106:
        return "scounteren";
    case 0x140:
        return "sscratch";
    case 0x141:
        return "sepc";
    case 0x142:
        return "scause";
```

```
case 0x143:
    return "stval";
case 0x144:
    return "sip";
case 0x180:
    return "satp";
case 0xf11:
    return "mvendorid";
case 0xf12:
    return "marchid";
case 0xf13:
    return "mimpid";
case 0xf14:
    return "mhartid";
case 0x300:
    return "mstatus";
case 0x301:
    return "misa";
case 0x302:
    return "medeleg";
case 0x303:
    return "mideleg";
case 0x304:
    return "mie";
case 0x305:
    return "mtvec";
case 0x306:
    return "mcounteren";
case 0x340:
    return "mscratch";
case 0x341:
    return "mepc";
case 0x342:
    return "mcause";
case 0x343:
```

```
        return "mtval";
    case 0x344:
        return "mip";
    case 0xb00:
        return "mcycle";
    case 0xb02:
        return "minstret";
    case 0xb80:
        return "mcycleh";
    case 0xb82:
        return "minstreth";
    case 0x320:
        return "mcountinhibit";
    case 0x7a0:
        return "tselect";
    case 0x7b0:
        return "dcsr";
    case 0x7b1:
        return "dpc";
    case 0x7b2:
        return "dscratch0";
    case 0x7b3:
        return "dscratch1";

}
if (0xc03 <=  csr && csr <= 0xc1f) {
    return "hpmcounter" + (csr - 0xc00);
}
if (0xc83 <= csr && csr <= 0xc9f) {
    return "hpmcounter" + (csr - 0xc80) + "h";
}
if (0x3a0 <= csr && csr <= 0x3a3) {
    return "pmpcfg" + (csr - 0x3a0);
}
if (0x3b0 <= csr && csr <= 0x3bf) {
```

```
            return "pmpaddr" + (csr - 0x3b0);
        }
        if (0xb03 <= csr && csr <= 0xb1f) {
            return "mhpmcounter" + (csr - 0xb00);
        }
        if (0xb83 <= csr && csr <= 0xb9f) {
            return "mhpmcounter" + (csr - 0xb80) + "h";
        }
        if (0x323 <= csr && csr <= 0x33f) {
            return "mhpmevent" + (csr - 0x320);
        }
        if (0x7a1 <= csr && csr <= 0x7a3) {
            return "tdata" + (csr - 0x7a0);
        }
        return Integer.toString(csr);
    }



    public static String getRInstructionRepresentation(int instruction) throws
UnknownCommandError {
        byte rd = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        byte funct3 = (byte) ((instruction & FUNCT_3_MASK) >>> FUNCT_3_SHIFT);
        byte rs1 = (byte) ((instruction & RS_1_MASK) >>> RS_1_SHIFT);
        byte rs2 = (byte) ((instruction & RS_2_MASK) >>> RS_2_SHIFT);
        byte funct7 = (byte) ((instruction & FUNCT_7_MASK) >>> FUNCT_7_SHIFT);
        final String inst = switch (funct7) {
            case 0x00 -> switch (funct3) {
                case 0x0 -> "add";
                case 0x4 -> "xor";
                case 0x6 -> "or";
                case 0x7 -> "and";
                case 0x1 -> "sll";
                case 0x5 -> "srl";
                case 0x2 -> "slt";
                case 0x3 -> "sltu";
                default -> throw new UnknownCommandError();
```

```
            };
            case 0x20 -> switch (funct3) {
                case 0x0 -> "sub";
                case 0x5 -> "sra";
                default -> throw new UnknownCommandError();
            };
            case 0x01 -> switch (funct3) {
                case 0x0 -> "mul";
                case 0x1 -> "mulh";
                case 0x2 -> "mulsu";
                case 0x3 -> "mulu";
                case 0x4 -> "div";
                case 0x5 -> "divu";
                case 0x6 -> "rem";
                case 0x7 -> "remu";
                default -> throw new UnknownCommandError();
            };
            default -> throw new UnknownCommandError();
        };
        final String dest = getRegisterName(rd);
        final String first = getRegisterName(rs1);
        final String second = getRegisterName(rs2);
        return String.format("%s %s, %s, %s", inst, dest, first, second);
    }


    public static String getSystemInstructionRepresentation(int instruction) throws
UnknownCommandError {
        byte rd = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        byte funct3 = (byte) ((instruction & FUNCT_3_MASK) >>> FUNCT_3_SHIFT);
        byte rs1 = (byte) ((instruction & RS_1_MASK) >>> RS_1_SHIFT);
        short imm12 = (short) ((instruction & (RS_2_MASK | FUNCT_7_MASK)) >>>
RS_2_SHIFT);
        final String inst;
        switch (funct3) {
            case 0b000 -> {
                if (imm12 == 0x0) {
```

```
                    inst = "ecall";
                } else if (imm12 == 0x1) {
                    inst = "ebreak";
                } else {
                    throw new UnknownCommandError();
                }
                return inst;
            }
            case 0b001 -> inst = "csrrw";
            case 0b010 -> inst = "csrrs";
            case 0b011 -> inst = "csrrc";
            case 0b100 -> inst = "csrrwi";
            case 0b101 -> inst = "csrrsi";
            case 0b111 -> inst = "csrrci";
            default -> throw new AssertionError("Wrong funct3");
        }
        if (funct3 < 0b100) {
            return String.format("%s %s, %s, %s",
                    inst, getRegisterName(rd), getCSRName(imm12),
getRegisterName(rs1));
        } else {
            // imm is in rs1
            return String.format("%s %s, %s, %d",
                    inst, getRegisterName(rd), getCSRName(imm12), rs1);
        }
    }


    public static String getIInstructionRepresentation(int instruction) throws
UnknownCommandError {
        byte opcode = (byte) (instruction & OPCODE_MASK);
        byte rd = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        byte funct3 = (byte) ((instruction & FUNCT_3_MASK) >>> FUNCT_3_SHIFT);
        byte rs1 = (byte) ((instruction & RS_1_MASK) >>> RS_1_SHIFT);
        short imm12 = (short) ((instruction & (RS_2_MASK | FUNCT_7_MASK)) >>>
RS_2_SHIFT);
        final String inst;
        switch (opcode) {
```

```
case 0b0010011:
    switch (funct3) {
        case 0x0:
            inst = "addi";
            break;
        case 0x1:
            if ((imm12 & 0b111111100000) == 0) {
                inst = "slli";
            } else {
                throw new UnknownCommandError();
            }
            break;
        case 0x2:
            inst = "slti";
            break;
        case 0x3:
            inst = "sltiu";
            break;
        case 0x4:
            inst = "xori";
            break;
        case 0x5:
            if ((imm12 & 0b111111100000) == 0) {
                inst = "srli";
            } else if ((imm12 & 0b111111100000) == (0x20 << 5)) {
                inst = "srai";
            } else {
                throw new UnknownCommandError();
            }
            break;
        case 0x6:
            inst = "ori";
            break;
        case 0x7:
            inst = "andi";
```

```
                    break;
                default:
                    throw new UnknownCommandError();
            }
            break;
        case 0b0000011:
            inst = switch (funct3) {
                case 0x0 -> "lb";
                case 0x1 -> "lh";
                case 0x2 -> "lw";
                case 0x4 -> "lbu";
                case 0x5 -> "lhu";
                default -> throw new UnknownCommandError();
            };
            break;
        case 0b1100111:
            if (funct3 == 0x0) {
                inst = "jalr";
            } else {
                throw new UnknownCommandError();
            }
            break;
        default:
            throw new UnknownCommandError();
    }
    int imm = getSignExtension(imm12, 12);
    return switch (inst) {
        case "jalr" -> String.format("jalr %s, %d(%s)",
                getRegisterName(rd), imm, getRegisterName(rs1));

        case "lb", "lh", "lw", "lbu", "lhu" -> String.format("%s %s, %d(%s)",
                inst, getRegisterName(rd), imm, getRegisterName(rs1));

        case "slli", "srli", "srai" -> String.format("%s %s, %s, %d",
                inst, getRegisterName(rd), getRegisterName(rs1), imm12 &
0b11111);
```

```
            default -> String.format("%s %s, %s, %d",
                    inst, getRegisterName(rd), getRegisterName(rs1), imm);
        };
    }


    public static String getSInstructionRepresentation(int instruction) throws
UnknownCommandError {
        // imm5 is in the rd place
        byte imm5 = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        byte funct3 = (byte) ((instruction & FUNCT_3_MASK) >>> FUNCT_3_SHIFT);
        byte rs1 = (byte) ((instruction & RS_1_MASK) >>> RS_1_SHIFT);
        byte rs2 = (byte) ((instruction & RS_2_MASK) >>> RS_2_SHIFT);
        // imm7 is in the funct7 place
        byte imm7 = (byte) ((instruction & FUNCT_7_MASK) >>> FUNCT_7_SHIFT);
        final String inst = switch (funct3) {
            case 0x0 -> "sb";
            case 0x1 -> "sh";
            case 0x2 -> "sw";
            default -> throw new UnknownCommandError();
        };
        final String source = getRegisterName(rs2);
        final String base = getRegisterName(rs1);
        final String offset = Integer.toString(getSignExtension(imm5 + (imm7 << 5),
12));
        return String.format("%s %s, %s(%s)", inst, source, offset, base);
    }


    public static String getBInstructionRepresentation(int instruction) throws
UnknownCommandError {
        byte funct3 = (byte) ((instruction & FUNCT_3_MASK) >>> FUNCT_3_SHIFT);
        byte rs1 = (byte) ((instruction & RS_1_MASK) >>> RS_1_SHIFT);
        byte rs2 = (byte) ((instruction & RS_2_MASK) >>> RS_2_SHIFT);
        final String inst = switch (funct3) {
            case 0x0 -> "beq";
            case 0x1 -> "bne";
            case 0x4 -> "blt";
```

```
            case 0x5 -> "bge";
            case 0x6 -> "bltu";
            case 0x7 -> "bgeu";
            default -> throw new UnknownCommandError();
        };
        final String source1 = getRegisterName(rs1);
        final String source2 = getRegisterName(rs2);
        int label = getBFormatOffset(instruction);
        return String.format("%s %s, %s, %d", inst, source1, source2, label);
    }

    public static int getBFormatOffset(int instruction) {
        byte imm5 = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        byte imm7 = (byte) ((instruction & FUNCT_7_MASK) >>> FUNCT_7_SHIFT);
        int label =
                ((imm5 & 0b11110))
                        + ((imm7 & 0b0111111) << 5)
                        + ((imm5 & 0b00001) << 11)
                        + ((imm7 & 0b1000000) << 6);
        return  getSignExtension(label, 13);
    }

    public static String getJInstructionRepresentation(int instruction) {
        byte rd = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        final String inst = "jal";
        final String dest = getRegisterName(rd);
        int offset = getJFormatOffset(instruction);
        return String.format("%s %s, %d", inst, dest, offset);
    }

    public static int getJFormatOffset(int instruction) {
        int imm20 = instruction & (FUNCT_3_MASK | RS_1_MASK | RS_2_MASK |
FUNCT_7_MASK) >>> FUNCT_3_SHIFT;
        int offset = ((imm20 & 0b01111111111000000000) >>> 8)
                + ((imm20 & 0b00000000000100000000) << 3)
                + ((imm20 & 0b00000000000011111111) << 12)
```

```
                + ((imm20 & 0b10000000000000000000) << 1);
        return BytesOperations.getSignExtension(offset, 21);
    }

    public static String getUInstructionRepresentation(int instruction) throws
UnknownCommandError {
        byte opcode = (byte) (instruction & OPCODE_MASK);
        byte rd = (byte) ((instruction & RD_MASK) >>> RD_SHIFT);
        int imm20 = (instruction & (FUNCT_3_MASK | RS_1_MASK | RS_2_MASK |
FUNCT_7_MASK)) >>> FUNCT_3_SHIFT;
        final String inst = switch (opcode) {
            case 0b0110111 -> "lui";
            case 0b0010111 -> "auipc";
            default -> throw new UnknownCommandError();
        };
        final String dest = getRegisterName(rd);
        return String.format("%s %s, %d", inst, dest, getSignExtension(imm20, 20));
    }
}
```

disasm/util/UnknownCommandError.java

```
package disasm.util;

public class UnknownCommandError extends Exception {
}
```