САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №_4__

«ISA. Ассемблер, дизассемблер»

Выполнил(а): Мавлютов Эрвин Акимович

Номер ИСУ: 334918

студ. гр. М3139

Санкт-Петербург

2021

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

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

Теоретическая часть

RISC-V – открытая и свободная система команд и процессорная архитектура на основе концепции RISC для микропроцессоров и микроконтроллеров. В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I — Integer) и несколько стандартных опциональных расширений. Архитектура использует только модель little-endian — первый байт операнда в памяти соответствует наименее значащим битам значений регистрового операнда. Операции умножения, деления и вычисления остатка не входят в минимальный набор инструкций, а выделены в отдельное расширение (M — Multiply extension). Имеется ряд доводов в пользу разделения и данного набора на два отдельных (умножение и деление). Для наиболее часто используемых инструкций стандартизовано применение их аналогов в более компактной 16-битной кодировке (C — Compressed extension).

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

Операции ветвления не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения

минимален, а для поддержки комплементарных операций операнды просто меняются местами.

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

Для встраиваемых применений может использоваться вариант архитектуры RV32E (Embedded) с сокращённым набором регистров общего назначения (первые 16). Уменьшение количества регистров позволяет не только экономить аппаратные ресурсы, но и сократить затраты памяти и времени на сохранение/восстановление регистров при переключениях контекста.

При одинаковой кодировке инструкций в RISC-V предусмотрены реализации архитектур с 32, 64 и 128-битными регистрами общего назначения и операциями (RV32I, RV64I и RV128I соответственно).

Elf-файл (Executable and Linkable Format) — формат двоичных файлов, используемый во многих современных UNIX-подобных операционных системах. В elf-файле есть заголовок фиксированного содержания и фиксированного размера. Здесь же содержится смещение на таблицу заголовков секций. Каждая запись о секции фиксированной длины (40 байт), имя секции берется из секции .shstrtab – таблицы имен секций. У каждой секции есть поле offset – смещение от начала файла – ссылка на таблицу данного заголовка.

Практическая часть

Далее представлен код программы-транслятора, с помощью которой можно преобразовать машинный код в код на языке ассемблера. В программе поддерживаются команды в формате RISC-V и сокращенные команды RVC. Программа принимает в качестве аргументов командной строки имена входного и выходного файла.

Входной файл представляет собой 32-битный elf-файл. Программа читает его в бинарном виде, далее парсит Header, читаются все секции заголовков. Парсится каждая из секций .text и .symtab.

В выходной файл программа выводит две секции: .text со всеми командами ассемблера и .symtab – таблицы символов.

В программе поддерживаются команды RISC-V RV32-I, RV32-M, RVC (сокращенные RV-32I, RV32-M).

Программа может бросить ParserException – пользовательское исключение, возникающее при разборе elf-файла – при неправильном формате ввода или при ошибках, связанных с работой с ресурсами.

Рассмотрим структуру файлов.

`package .parsers`

Класс `ParserELF` парсит elf-файл. Конструктор принимает две строки: имена входного и выходного файлов. Далее вызывается метод `parse()`. Все остальные классы – детали реализации, при попытке создать экземпляры других классов или вызвать методы, кроме `parse()`, программа не скомпилируется.

`package .myBase`

Реализованы `MyFileWriter`, `MyDataInputStream` – `FileWriter`, `DataInputStream` соответственно, поддерживающие имена файлов.

Класс `MyPair<F, S>` хранит пару объектов типа F и S.

Листинг

Main.java

```
import parsers.ParserELF;

public class Main {
    public static void main(String[] args) {
        new ParserELF(args[0], args[1]).parse();
    }
}
```

myBase.MyPair.java

```
package myBase;

public class MyPair<F, S> {
    private final F first;
    private final S second;

    public MyPair(F first, S second) {
        this.first = first;
        this.second = second;
    }

    public F getFirst() {
        return first;
    }

    public S getSecond() {
        return second;
    }

    @Override
    public String toString() {
```

```
        return String.format("%s %s", first, second);
    }
}
```

myBase.MyReader.java

```
package myBase;

import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class MyReader {
    private final DataInputStream out;
    private final String name;

    public MyReader(String name) throws IOException {
        out = new DataInputStream(new FileInputStream(name));
        this.name = name;
    }

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

    public byte[] readAllBytes() throws IOException {
        return out.readAllBytes();
    }

    public void close() throws IOException {
        out.close();
    }
}
```

myBase.MyWriter.java

```
package myBase;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.nio.charset.StandardCharsets;

public class MyWriter {
    private final BufferedWriter in;
    private final String name;

    public MyWriter(String name) throws IOException {
        in = new BufferedWriter(new FileWriter(name,
StandardCharsets.UTF_8));
        this.name = name;
    }

    public void write(String str) throws IOException {
        in.write(str);
    }

    public void newLine() throws IOException {
        in.newLine();
    }

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

```
    public void close() throws IOException {
        in.close();
    }
}
```

parsers.ParserCommands.java

```
package parsers;

import myBase.MyPair;

import java.util.ArrayList;
import java.util.Map;

import static parsers.ParserRVC.error;

public class ParserCommands {
    protected static Map<Integer, Integer> symTabNodesDict = null;
    protected static ArrayList<SymTabNode> symTabNodes = null;
    protected static int text_addr;

    private ParserCommands() {}

    protected static MyPair<String[], Boolean> parseCommand(int[] bytes,
int left) {
        if (Integer.toBinaryString(bytes[left]).endsWith("11")) {
            return new MyPair<>(ParserCommands.parseRiscV(bytes, left),
true);
        }
        return new MyPair<>(ParserCommands.parseRvc(bytes, left),
false);
    }

    private static String[] parseRvc(int[] bytes, int left) {
```

```
        final String instruction = decToBin(bytes[left + 1]) +
decToBin(bytes[left]);
        final String fun03 = instruction.substring(0, 3);
        final String fun49 = instruction.substring(4, 9);
        final String fun69 = instruction.substring(6, 9);
        final String fun314 = instruction.substring(3, 14);
        final String fun911 = instruction.substring(9, 11);
        final String fun914 = instruction.substring(9, 14);
        final String reg49 = reg(fun49);
        final String fun1114 = instruction.substring(11, 14);
        final String imm1 = "" +
Integer.parseInt(Character.toString(instruction.charAt(3)), 2);
        final String imm2 = "" + Integer.parseInt(fun914, 2);
        return switch (Integer.parseInt(instruction.substring(14), 2)) {
            case 0 -> {
                if (instruction.startsWith("000")) {
                    yield new String[]{"c.addi4spn", reg(fun1114),
                            "sp", "" +
Integer.parseInt(instruction.substring(3, 11), 2)};
                } else if (instruction.startsWith("100")) {
                    throw error(String.format("%s (it is reserved)",
instruction));
                } else {
                    int imm11 =
Integer.parseInt(instruction.substring(3, 6), 2);
                    int imm21 = Integer.parseInt(fun911, 2);
                    yield new String[]{
                            "c." + ParserRVC.quadrant0(fun03),
                            reg(fun69), "" + imm11,
                            reg(fun1114), "" + imm21
                    };
                }
            }
            case 1 -> {
                String rs1 = reg(fun69);
```

```
                String rs2 = reg(fun1114);
                int regB = Integer.parseInt(instruction.charAt(3) +
fun911 + instruction.charAt(13)
                        + instruction.substring(4, 6) +
instruction.substring(11, 13), 2);
                yield switch (Integer.parseInt(fun03, 2)) {
                    case 0 -> {
                        if (instruction.equals("0".repeat(15) + "1")) {
                            yield new String[]{"c.nop"};
                        } else {
                            yield new String[]{"c.addi", reg49, imm2};
                        }
                    }
                    case 1 -> new String[]{"c.jal",
getLabel(Integer.parseInt(fun314, 2))};
                    case 2 -> new String[]{"c.li", reg49, imm2};
                    case 3 -> {
                        if (Integer.parseInt(fun49, 2) == 2) {
                            yield new String[]{"c.addi16sp", imm1, "sp",
imm2};
                        } else {
                            yield new String[]{"c.lui", reg49, "" +
(Integer.parseInt(imm1) + text_addr)};
                        }
                    }
                    case 4 -> switch
(Integer.parseInt(instruction.substring(4, 6), 2)) {
                        case 0 -> new String[]{"c.srli", rs1, imm2};
                        case 1 -> new String[]{"c.srai", rs1, imm2};
                        case 2 -> new String[]{"c.andi", rs1, imm2};
                        case 3 -> new String[]{"c." +
ParserRVC.quadrant1(instruction, imm1 + fun911), rs1, rs2};
                        default -> throw error(instruction);
                    };
                    case 5 -> new String[]{"c.j",
getLabel(Integer.parseInt(
```

```
                        "" + instruction.charAt(3) +
instruction.charAt(7) + instruction.substring(5, 7)
                                + instruction.charAt(9) +
instruction.charAt(8) + instruction.charAt(13)
                                + instruction.charAt(4) +
instruction.substring(10, 13), 2))};
                    case 6 -> new String[]{"c.beqz", rs1, getLabel(regB
+ 4)};
                    case 7 -> new String[]{"c.bnez", rs1, getLabel(regB
+ 4)};
                    default -> throw error(instruction);
                };
            }
            case 2 -> switch (Integer.parseInt(fun03, 2)) {
                case 0, 1, 2, 3 -> new String[]{String.format(
                        "c.%s %s, %s(sp)", ParserRVC.quadrant2(fun03),
reg49, imm2
                )};
                case 4 -> {
                    if (instruction.charAt(3) == '0' &&
!isZeroes(fun49)) {
                        if (isZeroes(fun914)) {
                            yield new String[]{"c.jr", reg49};
                        } else {
                            yield new String[]{"c.mv", reg49,
reg(fun914)};
                        }
                    } else if (instruction.charAt(3) == '1') {
                        if (isZeroes(fun49) && isZeroes(fun914)) {
                            yield new String[]{"c.ebreak"};
                        } else if (isZeroes(fun914)) {
                            yield new String[]{"c.jalr", reg49};
                        } else {
                            yield new String[]{"c.add", reg49,
reg(fun914)};
                        }
```

```
                    } else {
                        throw error(instruction);
                    }
                }
                case 5, 6, 7 -> {
                    int imm = Integer.parseInt(instruction.substring(3,
9), 2);
                    yield new String[]{"c." +
ParserRVC.quadrant2(fun03), reg(fun914), "" + imm};
                }
                default -> throw error(instruction);
            };
            default -> throw error(instruction);
        };
    }

    private static String[] parseRiscV(int[] bytes, int left) {
        StringBuilder sb = new StringBuilder()
                .append(decToBin(bytes[left +
3])).append(decToBin(bytes[left + 2]))
                .append(decToBin(bytes[left +
1])).append(decToBin(bytes[left]));
        String opcode = sb.substring(25, 32);
        String rd = sb.substring(20, 25);
        String func3 = sb.substring(17, 20);
        String rs1 = sb.substring(12, 17);
        String rs2 = sb.substring(7, 12);
        String func7 = sb.substring(0, 7);

        if (sb.toString().equals("00000000000000000000000001110011")) {
            return new String[] {"ecall"};
        } else if
(sb.toString().equals("00000000000100000000000001110011")) {
            return new String[] {"ebreak"};
        }
```

```
        return switch (opcode) {
            case "0110011" -> new String[]{ParserRiscV.parseR(func7,
func3), reg(rd), reg(rs1), reg(rs2)};
            case "1100011" -> {
                int imm_b = Integer.parseUnsignedInt(
                        (sb.charAt(0) + "").repeat(20) +
                                sb.charAt(24) + sb.substring(1, 7) +
sb.substring(20, 24) + "0",2
                );
                yield new String[]{ParserRiscV.parseB(func3), reg(rs1),
reg(rs2), getLabel(imm_b + 4)};
            }
            case "0100011" -> {
                int imm_s =
Integer.parseUnsignedInt(Character.toString(sb.charAt(0)).repeat(20)
                        + sb.substring(0, 7) + sb.substring(20, 25), 2);
                yield new String[]{
                        String.format("%s %s, %s(%s)",
ParserRiscV.parseS(func3), reg(rs2), "" + imm_s, reg(rs1))
                };
            }
            case "0110111", "0010111" -> {
                int imm_u = Integer.parseUnsignedInt(sb.substring(0, 20)
+ "0".repeat(12), 2);
                yield new String[]{ParserRiscV.parseU(opcode), reg(rd),
imm_u + ""};
            }
            case "1110011" -> new String[]{ParserRiscV.parseICsr(func3),
reg(rd), reg(sb.substring(0, 12)), reg(rs1)};
            case "0010011" -> {
                int imm_i = Integer.parseUnsignedInt(
                        Character.toString(sb.charAt(0)).repeat(20) +
sb.substring(0, 12), 2
                );
                yield new String[]{
```

```
                        ParserRiscV.parseISr(func3, func7), reg(rd),
reg(rs1),
                        "" + (func3.equals("101") || func3.equals("001")
?
                                Integer.parseUnsignedInt(sb.substring(7,
12), 2) : imm_i)
                };
            }
            case "0000011" -> {
                int imm_i = Integer.parseUnsignedInt(
                        Character.toString(sb.charAt(0)).repeat(20) +
sb.substring(0, 12), 2
                );
                yield new String[]{
                        String.format("%s %s, %s(%s)",
ParserRiscV.parseIL(func3), reg(rd), "" + imm_i, reg(rs1))
                };
            }
            case "1101111" -> {
                int imm_j =
Integer.parseUnsignedInt(Character.toString(sb.charAt(0)).repeat(12)
                        + sb.substring(12, 20) + sb.charAt(11) +
sb.substring(1, 11) + "0", 2);
                yield new String[]{"jal", reg(rd), getLabel(imm_j)};
            }
            case "1100111" -> {
                int imm_i = Integer.parseUnsignedInt(
                        Character.toString(sb.charAt(0)).repeat(20) +
sb.substring(0, 12), 2
                );
                yield new String[]{"jalr", reg(rd), reg(rs1), "" +
imm_i};
            }
            case "0001111" -> new String[]{"fence" +
(func3.equals("001") ? ".i" : "")};
            default -> throw new ParserException("Risc-V",
sb.toString());
```

```
        };
    }

    private static String reg(String a) {
        int reg = Integer.parseInt(a, 2);
        if (reg == 0) {
            return "zero";
        } else if (reg == 1) {
            return "ra";
        } else if (reg == 2) {
            return "sp";
        } else if (reg == 3) {
            return "gp";
        } else if (reg == 4) {
            return "tp";
        } else if (reg == 5) {
            return "t0";
        } else if (6 <= reg && reg <= 7) {
            String s = "t";
            s += (char) (reg - 5 + '0');
            return s;
        } else if (reg == 8) {
            return "s0";
        } else if (reg == 9) {
            return "s1";
        } else if (10 <= reg && reg <= 11) {
            String s = "a";
            s += (char) (reg - 10 + '0');
            return s;
        } else if (12 <= reg && reg <= 17) {
            String s = "a";
            s += (char) (reg - 10 + '0');
            return s;
```

```
        } else if (18 <= reg && reg <= 27) {
            String s = "s";
            s += (char) (reg - 16 + '0');
            return s;
        } else if (28 <= reg && reg <= 31) {
            String s = "t";
            s += (char) (reg - 25 + '0');
            return s;
        }
        return null;
    }

    protected static String decToBin(int b) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 8; i++) {
            sb.append(b % 2);
            b /= 2;
        }
        return sb.reverse().toString();
    }

    protected static boolean isZeroes(final String x) {
        return x.equals("0".repeat(x.length()));
    }

    private static String getLabel(int i) {
        final int addr_command = text_addr + i;
        if (symTabNodesDict.containsKey(addr_command)) {
            return
symTabNodes.get(symTabNodesDict.get(addr_command)).getName();
        }
        symTabNodesDict.put(addr_command, symTabNodes.size());
```

```
        symTabNodes.add(new SymTabNode(String.format("LOC_%05x",
addr_command)));
        return getLabel(i);
    }
}
```

parsers.ParserELF.java

```
package parsers;

import myBase.MyReader;
import myBase.MyWriter;
import myBase.MyPair;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;

import static parsers.ParserCommands.decToBin;

public class ParserELF {
    private int[] bytes;
    private final MyReader in;
    private final MyWriter out;
    private final Map<String, Section> sections = new LinkedHashMap<>();
    private final ArrayList<SymTabNode> symTabNodes = new ArrayList<>();
    private final Map<Integer, Integer> symTabNodesDict = new
HashMap<>();

    public ParserELF(String fileIn, String fileOut) {
        try {
            in = new MyReader(fileIn);
```

```
            enterData();
        } catch (IOException e) {
            throw new ParserException(String.format("Can't open file
\"%s\"", fileIn));
        }
        try {
            out = new MyWriter(fileOut);
        } catch (IOException e) {
            throw new ParserException(String.format("Can't open file
\"%s\"", fileOut));
        }
    }


    public void parse() {
        parseHeader();
        parseSymtab();
        try {
            parseAndDumpText();
        } catch (IOException e) {
            throw new ParserException(".text", "Can't write to file",
e.getMessage());
        }
        try {
            dumpSymtab();
        } catch (IOException e) {
            throw new ParserException(".symtab", "Can't write to file",
e.getMessage());
        }
        try {
            out.close();
        } catch (IOException e) {
            throw new ParserException(String.format("Error while closing
file \"%s\"", out));
        }
        System.out.println("Successfully parsed!");
```

```
    }

    private void enterData() {
        final byte[] bytes1;
        try {
            bytes1 = in.readAllBytes();
            in.close();
        } catch (IOException e) {
            throw new ParserException(
                    String.format("file \"%s\" format not supported:
%s", in, e.getMessage())
            );
        }
        bytes = new int[bytes1.length];
        for (int i = 0; i < bytes1.length; i++) {
            bytes[i] = ((bytes1[i] < 0) ? 256 : 0) + bytes1[i];
        }
    }

    private void parseHeader() {
        final int e_shoff = cnt(32, 4);
        final int e_shentsiz = cnt(46, 2);
        final int e_shnum = cnt(48, 2);
        final int e_shstrndx = cnt(50, 2);
        final int go = e_shoff + e_shentsiz * e_shstrndx;
        final int sh_offset12 = cnt(go + 16, 4);
        for (int j = 0, tmp = e_shoff; j < e_shnum; j++, tmp += 40) {
            String name = getName(sh_offset12 + cnt(tmp, 4));
            int[] param = new int[10];
            for (int r = 0; r < 10; r++) {
                param[r] = cnt(tmp + 4 * r, 4);
            }
            sections.put(name, new Section(param));
```

```
        }
    }


    private void parseAndDumpText() throws IOException {
        Section Text = sections.get(".text");
        out.write(".text\n");
        int ind = Text.offset;
        final int size = Text.size;
        final int addr = Text.addr;
        for (int i = 0; i < size; i += 2) {
            MyPair<String[], Boolean> ans =
ParserCommands.parseCommand(bytes, ind);
            out.write(String.format("%08x", addr + i));
            String label = (symTabNodesDict.containsKey(addr + i) &&
                    symTabNodes.get(symTabNodesDict.get(addr +
i)).getType().equals("FUNC")) ?
                    symTabNodes.get(symTabNodesDict.get(addr +
i)).getName() : "";
            out.write(String.format(" %10s%s", label, label.isEmpty() ?
" " : ":"));
            int len = ans.getFirst().length;
            for (int j = 0; j < len; j++) {
                out.write(String.format(" %s", ans.getFirst()[j]));
                if (j != 0 && j != len - 1) {
                    out.write(",");
                }
            }
            out.newLine();
            ind += 2;
            if (ans.getSecond()) {
                i += 2;
                ind += 2;
            }
        }
```

```
        out.newLine();
    }

    private void parseSymtab() {
        final Section Symtab = sections.get(".symtab");
        final int ind = Symtab.offset;
        final int num = Symtab.size / 16;
        for (int i = 0; i < num; i++) {
            StringBuilder sb = new StringBuilder();
            for (int j = 0; j < 4; j++) {
                sb.append(new StringBuilder(decToBin(bytes[ind + i * 16
+ 12 + j])).reverse());
            }
            final String name = getName(cnt(ind + i * 16, 4) +
sections.get(".strtab").offset);
            final int value = cnt(ind + i * 16 + 4, 4);
            final int size = cnt(ind + i * 16 + 8, 4);
            final int info = Integer.parseInt(new
StringBuilder(sb.substring(0, 8)).reverse().toString(), 2);
            final int other = Integer.parseInt(new
StringBuilder(sb.substring(8)).reverse().toString(), 2);
            symTabNodesDict.put(value, symTabNodes.size());
            symTabNodes.add(new SymTabNode(i, name, value, size, info,
other));
        }
        updateDataInParserCommands();
    }

    private void updateDataInParserCommands() {
        ParserCommands.symTabNodes = symTabNodes;
        ParserCommands.symTabNodesDict = symTabNodesDict;
        ParserCommands.text_addr = sections.get(".text").addr;
    }

    private void dumpSymtab() throws IOException {
```

```
        out.write(".symtab\n");
        out.write(String.format("%s %-15s %7s %-8s %-8s %-8s %6s %s\n",
                "Symbol", "Value", "Size", "Type", "Bind", "Vis",
"Index", "Name"));
        for (SymTabNode node : symTabNodes) {
            out.write(node.toString());
        }
    }

    protected int cnt(final int left, final int num) {
        int ans = 0;
        for (int i = num - 1; i >= 0; i--) {
            ans = ans * 256 + bytes[left + i];
        }
        return ans;
    }

    protected String getName(int left) {
        final StringBuilder sb = new StringBuilder();
        while (bytes[left] != 0) {
            sb.append((char) bytes[left++]);
        }
        return sb.toString();
    }
}
```

parsers.ParserException.java

```
package parsers;

public class ParserException extends RuntimeException {
    protected ParserException(final String sectionName, final String
message, final String cause) {
        super(String.format(
```

```
                "%s while parsing \"%s\" section: %s",
                message, sectionName, cause
        ));
    }

    protected ParserException(final String type, final String message) {
        super(String.format("Unknown \"%s\"-type instruction: %s", type,
message));
    }

    protected ParserException(final String message) {
        super(String.format("Can't start parse ELF-file: %s", message));
    }
}
```

parsers.ParserRiscV.java

```
package parsers;

public class ParserRiscV {
    private ParserRiscV() {}

    protected static String parseIL(final String func3) {
        return switch (func3) {
            case "000" -> "lb";
            case "001" -> "lh";
            case "010" -> "lw";
            case "100" -> "lbu";
            case "101" -> "lhu";
            default -> throw new ParserException("I", func3);
        };
    }

    protected static String parseICsr(final String func3) {
```

```
        return switch (func3) {
            case "001" -> "csrrw";
            case "010" -> "csrrs";
            case "011" -> "csrrc";
            case "101" -> "csrrwi";
            case "110" -> "csrrsi";
            case "111" -> "csrrci";
            default -> throw new ParserException("I", func3);
        };
    }


    protected static String parseISr(final String func3, final String
func7) {
        return switch (func3) {
            case "000" -> "addi";
            case "001" -> {
                if (ParserCommands.isZeroes(func7)) {
                    yield "slli";
                } else {
                    throw new ParserException("I", func7);
                }
            }
            case "010" -> "slti";
            case "011" -> "sltiu";
            case "100" -> "xori";
            case "101" -> switch (func7) {
                case "0000000" -> "srli";
                case "0100000" -> "srai";
                default -> throw new ParserException("I", func7);
            };
            case "110" -> "ori";
            case "111" -> "andi";
            default -> throw new ParserException("I", func3);
```

```
        };
    }

    protected static String parseS(final String func3) {
        return switch (func3) {
            case "000" -> "sb";
            case "001" -> "sh";
            case "010" -> "sw";
            default -> throw new ParserException("S", func3);
        };
    }

    protected static String parseU(final String opcode) {
        return switch (opcode) {
            case "0110111" -> "lui";
            case "0010111" -> "auipc";
            default -> throw new ParserException("U", opcode);
        };
    }

    protected static String parseB(final String func3) {
        return switch (func3) {
            case "000" -> "beq";
            case "001" -> "bne";
            case "100" -> "blt";
            case "101" -> "bge";
            case "110" -> "bltu";
            case "111" -> "bgeu";
            default -> throw new ParserException("B", func3);
        };
    }
```

```
   protected static String parseR(final String func7, final String
func3) {
        return switch (func7) {
            case "0000000" -> switch (func3) {
                case "000" -> "add";
                case "001" -> "sll";
                case "010" -> "slt";
                case "011" -> "sltu";
                case "100" -> "xor";
                case "101" -> "srl";
                case "110" -> "or";
                case "111" -> "and";
                default -> throw new ParserException("R", func3);
            };
            case "0100000" -> switch (func3) {
                case "000" -> "sub";
                case "101" -> "sra";
                default -> throw new ParserException("R", func3);
            };
            case "0000001" -> switch (func3) {
                case "000" -> "mul";
                case "001" -> "mulh";
                case "010" -> "mulhsu";
                case "011" -> "mulhu";
                case "100" -> "div";
                case "101" -> "divu";
                case "110" -> "rem";
                case "111" -> "remu";
                default -> throw new ParserException("R", func3);
            };
            default -> throw new ParserException("R", func7);
        };
    }
```

```
}
```

parsers.ParserRVC.java

```
package parsers;

public class ParserRVC {
    private ParserRVC() {}

    protected static String quadrant0(String instruction) {
        return switch (instruction.substring(0, 3)) {
            case "001" -> "fld";
            case "010" -> "lw";
            case "011" -> "flw";
            case "101" -> "fsd";
            case "110" -> "sw";
            case "111" -> "fsw";
            default -> throw error(instruction);
        };
    }

    protected static String quadrant1(String instruction, String str3) {
        return switch (Integer.parseInt(str3, 2)) {
            case 0 -> "sub";
            case 1 -> "xor";
            case 2 -> "or";
            case 3 -> "and";
            case 4 -> "subw";
            case 5 -> "addw";
            default -> throw error(instruction);
        };
    }

    protected static String quadrant2(String instruction) {
```

```
        return switch (Integer.parseInt(instruction.substring(0, 3), 2))
{
            case 0 -> "slli";
            case 1 -> "fldsp";
            case 2 -> "lwsp";
            case 3 -> "flwsp";
            case 5 -> "fsdsp";
            case 6 -> "swsp";
            case 7 -> "fswsp";
            default -> throw error(instruction);
        };
    }

    protected static ParserException error(String instruction) {
        return new ParserException("RVC", instruction);
    }
}
```

parsers.ParserSymtab.java

```
package parsers;

public class ParserSymtab {
    private ParserSymtab() {}

    protected static String getIndex(final int other) {
        return switch (other) {
            case 0 -> "UNDEF";
            case 0xff00 -> "LOPROC";
            case 0xff01 -> "AFTER";
            case 0xff02 -> "AMD64_LCOMMON";
            case 0xff1f -> "HIPROC";
            case 0xff20 -> "LOOS";
            case 0xff3f -> "HIOS";
```

```
            case 0xfff1 -> "ABS";
            case 0xfff2 -> "COMMON";
            case 0xffff -> "XINDEX";
            default -> other + "";
        };
    }

    protected static String getVis(final int other) {
        return switch (other) {
            case 0 -> "DEFAULT";
            case 1 -> "INTERNAL";
            case 2 -> "HIDDEN";
            case 3 -> "PROTECTED";
            case 4 -> "EXPORTED";
            case 5 -> "SINGLETON";
            case 6 -> "ELIMINATE";
            default -> "UNKNOWN";
        };
    }

    protected static String getBind(final int info) {
        return switch (info) {
            case 0 -> "LOCAL";
            case 1 -> "GLOBAL";
            case 2 -> "WEAK";
            case 10 -> "LOOS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> "UNKNOWN";
        };
    }
```

```
    protected static String getType(final int info) {
        return switch (info) {
            case 0 -> "NOTYPE";
            case 1 -> "OBJECT";
            case 2 -> "FUNC";
            case 3 -> "SECTION";
            case 4 -> "FILE";
            case 5 -> "COMMON";
            case 6 -> "TLS";
            case 10 -> "LOOS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 14 -> "SPARC_REGISTER";
            case 15 -> "HIPROC";
            default -> "UNKNOWN";
        };
    }
}
```

parsers.Section.java

```
package parsers;


public class Section {
    protected final int name, type, flags, addr, offset, size, link,
info, addralign, entsize;


    protected Section(int[] param) {
        name = param[0];
        type = param[1];
        flags = param[2];
        addr = param[3];
        offset = param[4];
        size = param[5];
```

```
        link = param[6];
        info = param[7];
        addralign = param[8];
        entsize = param[9];
    }

    @Override
    public String toString() {
        return String.format("name = %2d, addr = %5d, offset = %4d, size
= %4d", name, addr, offset, size);
    }
}
```

parsers.SymTabNode.java

```
package parsers;

public class SymTabNode {
    private final int symbol, value, size;
    private final String type, bind, vis, index, name;

    protected SymTabNode(String name) {
        this.symbol = 0;
        this.value = 0;
        this.size = 0;
        this.type = "FUNC";
        this.bind = null;
        this.vis = null;
        this.index = null;
        this.name = name;
    }

    protected SymTabNode(
```

```
            final int i, final String name, final int value, final int
size, final int info, final int other
    ) {
        this.symbol = i;
        this.value = value;
        this.size = size;
        this.type = ParserSymtab.getType(info & 0xf);
        this.bind = ParserSymtab.getBind(info >> 4);
        this.vis = ParserSymtab.getVis(other & 0x3);
        this.index = ParserSymtab.getIndex(other >> 8);
        this.name = name;
    }

    public String getType() {
        return type;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        if (name.startsWith("LOC")) {
            return "";
        }
        return String.format(
                "[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n",
                symbol, value, size, type, bind, vis, index, name
        );
    }
}
```

Литература

1. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2", Editors Andrew Waterman and Krste Asanovic, RISC-V Foundation, May 2017.
2. John Winans. RISC-V. Assembly Language Programming. – June 29, 2021.
3. Peijie Li. Electrical Engineering and Computer Sciences University of California at Berkeley. – May 16, 2019

Электронные ресурсы:

1. https://ru.wikipedia.org/wiki/Executable_and_Linkable_Format
2. https://github.com/riscv/riscv-opcodes
3. https://habr.com/ru/post/480642/
4. https://www.felixcloutier.com/x86/