САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИТМО
Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №4

ISA. Ассемблер, дизассемблер

Выполнил: Рожков Денис Валерьевич
Номер ИСУ: 334899
студ. гр. M3134

Санкт-Петербург

2021

Теоретическая часть

ELF (executable and linkable format) – формат двоичных файлов часто используемый в unix системах. По своему дизайну ELF очень гибок, расширяем и кроссплатформенен. Например, он поддерживает возможность указывать порядок байтов или размеры адресов, чтобы не исключить возможность исполнения на некоторых и ISA.

Каждый ELF файл состоит из заголовка и данных. Заголовок состоит из такой информации как – класс (32 или 64 бита на адрес), ABI – описание интерфейса взаимодействия с операционной системой, целевая ISA, адрес entry – места откуда программа начнёт исполнение, адрес начала таблицы заголовков программ, адрес начала таблицы заголовков секций, индекс секции с именами секций в таблице заголовков секций и др.

Заголовок программы содержит всю необходимую информацию для размещения исполняемых данных в памяти компьютера.

Заголовок секции содержит в себе указатель на строку с названием секции, тип секции, фактическое расположение секции в данном файле. В секции могут храниться совершенно разные данные. Существует особая секция, которая содержит в себе строки с названиями секций. Её индекс хранится в заголовке elf файла.

Система кодирования команд RISC-V

RISC-V – открытая и свободная ISA основанная на концепции RISC. Основная ISA содержит в себе 53 команды, но может быть очень просто расширена. Существуют расширения для перемножения чисел (M), работы с плавающей точкой (F), сжатых команд (C), атомарных операций (A) и т.д.

RISC-V работает с 32-мя регистрами, соответственно для кодирования регистра нужно 5 бит. Регистры называют в формате x%d, где %d – число от 0 до 31. X0 всегда равен нулю.

По спецификации в X1 хранится указатель на возвращаемое значение.

В unix системах существуют соглашения по названию регистров (см. **Error! Reference source not found.**).

Таблица – использование регистров.

Регистр	**Название**	**Расшифровка**
x0	zero	Ноль
x1	ra	Возвращаемое значение
x2	sp	Указатель на стек
x3	gp	Глобальный указатель
x4	tp	Указатель потока
x5-x7	t0-t2	Временные регистры
x8-x9	s0-s1	Регистры используемые вызывающим
x10-x17	a0-a7	Регистры аргументов
x18-x27	s2-s11	Регистры используемые вызывающим
x28-x31	t3-t6	Временные регистры

– Базовая RV32i имеет длину инструкции 32 бита. Команды бывают нескольких типов – R, I, S, B, U и J.

– Каждая инструкция содержит opcode – располагается с 2 по 6 бит.

– Длину инструкции для модификаций где длина инструкции не равна 32 битам, определяют 1-ые 2 бита.

Распишем, для чего нужна каждая из интсрукций в RV32i :

1) **R** инструкция нужна для операций которые работают только на регистрах. Содержит 3 указателя на регистры: rs1, rs2, rd, два для чтения значений и один для записи, funct3, funct7 для определения операции.

2) **I** инструкция нужна для операций требующих временное значение imm (immediate) размером не боль ше 12 бит. Похожа на R, только место funct7 и rs2 занимает imm.

3) **S** инструкция нужна для записи значений в память. Похожа на R тип, но место rd и funct7 занимает imm – который в этих операциях играет роль дополнительного сдвиг для адреса памяти.

4) **B** работает с условными переходами. Почти как S тип, но поле imm записано по-другому.

5) **U** инструкция нужна для записи верхних бит 20 бит в какой либо регистр. Содержит только указатель на регистр и сохраняемое значение.

6) **J** инструкция нужна чтобы совершить прыжок в другое место.

Структура elf-файла

Изначально elf-файл представлен как набор битов, которые на 1-ый взгляд между собой никак не связаны, но если открыть википедию и внимательно почитать, станет понятно, что есть определенные последовательности байтов, которые помогают распарсить данные нам файл. Внимание стоит обратить на первые 52 (64) байта файла.

Заголовок файла (ELF Header) имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки бита, размеры и смещения остальных частей файла. Заголовок имеет размер 52 байта для 32-битных файлов или 64 для 64-битных. Данное различие обуславливается тем, что в заголовке

файла содержится три поля, имеющих размер указателя, который составляет 4 и 8 байт для 32- и 64-битных процессоров соответственно. Такими полями являются: e_entry, e_phoff и e_shoff.

e_shnum – Число заголовков секций. Если у файла нет таблицы заголовков секций, это поле содержит 0.

e_shstrndx – Индекс записи в таблице заголовков секций, описывающей таблицу названий секций (обычно эта таблица называется .shstrtab и представляет собой отдельную секцию). Если файл не содержит таблицы названий секций, это поле содержит 0.

Так же есть и другие поля, которые тоже представляют собой какую либо другую полезную информацию о файле, но для выполнения данного задания из данной секции мне понадобились только описанные мной поля.

Таблица заголовков программы

Таблица заголовков программы содержит заголовки, каждый из которых

описывает отдельный сегмент программы и его атрибуты либо другую информацию, необходимую операционной системе для подготовки программы к исполнению. Данная таблица может располагаться в любом месте файла, её местоположение (смещение относительно начала файла) описывается в поле e-phoff заголовка ELF.

При анализе структуры заголовка программы можно обнаружить различное местоположение поля p-flags для 32- и 64-битных ELF файлов. Данное различие обуславливается выравниванием структуры для увеличения эффективности обработки.

При выполнении задания не требовалось парсить таблицу заголовков программы, поэтому более подробно я с ней не разбирался.

Таблица заголовков секций

Таблица заголовков секций содержит атрибуты секций файла. Данная таблица необходима только компоновщику, исполняемые файлы в наличии этой таблицы не нуждаются (ELF загрузчик её игнорирует). Предоставленную в таблице заголовков секций информацию компоновщик использует для оптимального размещения данных секций по сегментам при сборке файла с учётом их атрибутов.

Самая важная часть elf-файла, которая хранит в себе определенные секции.

Секции с которыми я буду работать в программе опишу более подробно

1) .symtab – Секция содержит таблицу символов. В настоящий момент в файле может быть только одна такая секция.

2) .text – Секция содержит информацию, определённую программой, т.е. операции которые собственно и требуется распарсить.

3).shstrtab – содержит названия каждого из заголовков, каждое имя екции ограничено справа нулевым байтом, до него и надо парсить.

Теперь рассмотрим структуру одной конкретно взятой секции, стоит отметить что инструкции для каждой секции идентичны.

Итак,

sh_name – Смещение строки, содержащей название данной секции, относительно начала таблицы названий секций.

sh_addr – с этого номера байта начинаются номера команд, которые содержит в себе секция.

sh_offset – Смещение секции от начала файла в байтах. Секции типа не занимают места в файле, для них данное поле содержит концептуальное местоположение в файле.

sh_size – размер секции.

Как парсится каждая из нужных секций я распишу в разделе ниже.

Описание работы кода.

1) Побайтовое считывание файла.

2) Поиск параметров необходимых для того, чтобы распарсить section header.

3) Парсинг каждой секции.

4) Парсинг секции, отвечающей за названия остальных секций и сразу опреленение каждой секции.

5) Парсинг отдельно взятой секции, .symtab парсится по 16 байт на одну команду.

6) Парсинг отдельно взятой секции, .text парсится по 4 или по 2 байта на одну команду. Читаем по 4 байта(2 байта) и дисассемблируем каждую инструкцию. Если это инструкция прыжка и, то куда она прыгает не указывает на начало символа, то запишем адрес в множество «неизвестных адресов»

7) Пройдёмся по файлу ещё раз и будем снова дисассемблировать каждую инструкцию. Добавим метку в начало если у нас есть символ, указывающий на этот адрес или этот адрес есть во множестве «неизвестных адресов». Будем выводить построчно.

7) Вывод .symtab

8) Вывод .text

Более подробные детали на мой взгляд не нужны, ибо в них уже будет расписываться каким способом я реализовывал парсинг операций.

Листинг кода.

Main.java

```
package babakapa;

import java.io.*;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;


public class Main {

   public static void main(String[] args) {

       //1

       if (args.length < 1) {
```

```
            System.err.println("Usage: <input file> [<output file>]");
            return;
        }
        String inputFile = args[0];
        String outputFile = args[1];
        int[] mas = new int[10];
        int pos = 0;
        try (
            InputStream inputStream = new
FileInputStream(inputFile)
        ) {
            int byteRead;
            while ((byteRead = inputStream.read()) != -1) {
                if (pos == mas.length) {
                    mas = Arrays.copyOf(mas, mas.length * 2);
                }
                mas[pos++] = byteRead;
            }
        } catch (IOException ex) {
            ex.printStackTrace();
```

```
        }
        //2
        Parse gg = new Parse(mas);
        int e_shoff = gg.parseIntBytes(32, 36); // ок
        int e_shnum = gg.parseIntBytes(48, 50); // ок
        int e_shstrndx = gg.parseIntBytes(50, 52); // ок
        int[] shstrtab = new int[10];


        List<Section> baseSection = new ArrayList<>();
        //3
        for (int i = 0; i < e_shnum; i++, e_shoff += 40) {
            baseSection.add(new Section(
                    gg.parseIntBytes(e_shoff, e_shoff + 4),
                    gg.parseIntBytes(e_shoff + 4, e_shoff + 8),
                    gg.parseIntBytes(e_shoff + 12, e_shoff + 16),
                    gg.parseIntBytes(e_shoff + 16, e_shoff + 20),
                    gg.parseIntBytes(e_shoff + 20, e_shoff + 24)
            ));
        }
        //4
```

```
        pos = 0;
        Section sec = baseSection.get(e_shstrndx);
        for (int j = sec.sh_offset; j < sec.sh_offset + sec.sh_size; j++) {
            if (pos == shstrtab.length) {
                shstrtab = Arrays.copyOf(shstrtab, shstrtab.length * 2);
            }
            shstrtab[pos++] = mas[j];
        }
        shstrtab = Arrays.copyOf(shstrtab, pos);
        for (Section value : baseSection) {
            int j = value.sh_name;
            StringBuilder sb = new StringBuilder();
            while (shstrtab[j] != 0) {
                sb.append((char) shstrtab[j++]);
            }
            value.setName(sb.toString());
        }
        List<Character> baseStrtab = new ArrayList<>();
        List<Symtab> baseSymtab = new ArrayList<>();
        List<Operation> text = new ArrayList<>();
```

```
        for (Section section : baseSection) {
            if (section.name.equals(".strtab")) {
                for (int j = section.sh_offset; j < section.sh_offset + section.sh_size; j++) {
                    baseStrtab.add((char) mas[j]);
                }
            }
        //5
            if (section.name.equals(".symtab")) {
                for (int j = section.sh_offset; j < section.sh_offset + section.sh_size; j += 16) {
                    Symtab sym = new Symtab(
                            gg.parseIntBytes(j, j + 4),
                            gg.parseIntBytes(j + 4, j + 8),
                            gg.parseIntBytes(j + 8, j + 12),
                            gg.parseIntBytes(j + 12, j + 13),
                            gg.parseIntBytes(j + 13, j + 14),
                            gg.parseIntBytes(j + 14, j + 16)
                    );
                    baseSymtab.add(sym);
```

```
            }
        }
    //6
        if (section.name.equals(".text")) {
            for (int j = section.sh_offset; j < section.sh_offset + section.sh_size; ) {
                long p = gg.parseLongBytes(j, j + 4);
                Operation operation = new Operation();
                operation.setAddr(section.sh_addr);
                if (p % 4 == 3) {
                    ParseOperation32bit now = new ParseOperation32bit(p);
                    operation.op = now.write();
                    operation.offset = now.findOffset();
                    j += 4;
                    section.sh_addr += 4;
                } else {
                    int v = gg.parseIntBytes(j, j + 2);
                    operation.op = (new ParseOperation16bit(v).write());
```

```
                j += 2;
                section.sh_addr += 2;
            }
            text.add(operation);
        }
    }
    //7
    try {
        OutputStreamWriter output = new OutputStreamWriter(new FileOutputStream(args[1]));
        output.write(".symtab" + '\n');
        output.write(String.format("%4s %-17s %5s %-8s %-8s %-8s %6s %s%n",
            "Symbol", "Value", "Size", "Type", "Bind", "Vis", "Index", "Name") + '\n') ;
        for (int i = 0; i < baseSymtab.size(); i++) {
            Symtab sym = baseSymtab.get(i);
            StringBuilder sb = new StringBuilder();
            int j = sym.st_name;
```

```
            while (baseStrtab.get(j) != 0) {
                sb.append(baseStrtab.get(j++));
            }
            baseSymtab.get(i).setName(sb.toString());
            sym.setName(sb.toString());
            output.write(String.format("[%4d] 0x%-15X %5d %-8s
%-8s %-8s %6s %s%n",
                    i, sym.st_value, sym.st_size, sym.st_type,
                    sym.st_bind,        sym.st_other,        sym.st_shndx,
sym.name));
            if (sym.st_type.equals("FUNC")) {
                for (j = 0; j < text.size(); j++) {
                    if (text.get(j).addr.equals(sym.st_value)) {
                        text.get(j).firstMark = sym.name;
                        break;
                    }
                }
            }
        }
        output.write(".text\n");
```

```
            for (Operation operation : text) {
                output.write("0x00" + operation.index);
                if (operation.firstMark != null) {
                    output.write(" " + operation.firstMark + ":");
                }
                output.write(" " + operation.op);
                if (operation.secondMark != null) {
                    output.write(" " + operation.secondMark);
                }
                output.write('\n');
            }
        } catch (FileNotFoundException e) {
            System.err.println("output file not found" + e);
        } catch (IOException e) {
            System.err.println("Unnkown exception" + e);
        }
    }
}
```

ParseOperation32bit.java

```
package babakapa;


public class ParseOperation32bit {
    private final long x;
    private final Format ans = new Format();
    private final int[] bits = new int[32];

    ParseOperation32bit(long x) {
        this.x = x;
        getBitsMas();
    }

    private void getBitsMas() {
        long x = this.x;
        int pos = 0;
        while (x > 0) {
            bits[pos++] = (int) x % 2;
```

```
            x /= 2;
        }
    }

    private int parseBites(int l, int r) {
        int p = 1, res = 0;
        for (int i = l; i < r; i++, p *= 2) {
            res += bits[i] * p;
        }
        return res;
    }

    public String write() {
        String op = getOperation(parseBites(2, 7), parseBites(12, 15),
parseBites(25, 32));
        return ans.codeOp + " " + op;
    }

    public Integer findOffset() {
        return ans.offset;
```

```
    }

    private String getOperation(int type, int numOp, int typeBit) {
        if (type == 0x18) {
            return getBtype(numOp);
        }
        if (type == 0x00) {
            return getLtype(numOp);
        }
        if (type == 0x0C) {
            return getBitType(numOp, typeBit);
        }
        if (type == 0x08) {
            return getStype(numOp);
        }
        if (type == 0x04) {
            return getItype(numOp);
        }
        if (type == 0x1C) {
            return getSystype(numOp, typeBit);
```

```
    }
    if (type == 0x03) {
        return getFtype(numOp);
    }
    if (type == 0x19) {
        ans.codeOp = "jalr";
        ans.rd = getReg(parseBites(7, 12));
        ans.rs1 = getReg(parseBites(15, 20));
        ans.offset = parseBites(20, 32) * 2;
        return ans.rd + " " + ans.rs1;
    }
    if (type == 0x1b) {
        ans.codeOp = "jal";
        ans.rd = getReg(parseBites(7, 12));
        ans.offset = getJalOffset();
        return ans.rd;
    }
    if (type == 0x0d) {
        ans.codeOp = "lui";
        ans.rd = getReg(parseBites(7, 12));
```

```
            ans.imm = parseBites(12, 32);
            return ans.rd + " " + ans.imm;
        }
        if (type == 0x05) {
            ans.codeOp = "auipc";
            ans.rd = getReg(parseBites(7, 12));
            ans.imm = parseBites(12, 32);
            return ans.rd + " " + ans.imm;
        }
        return "";
    }

    private int getJalOffset() {
        int[] val = new int[21];
        val[20] = bits[31];
        val[11] = bits[20];
        System.arraycopy(bits, 21, val, 1, 10);
        System.arraycopy(bits, 12, val, 12, 8);
        int p = 1;
        int res = 0;
```

```
        for (int i = 1; i < 21; i++, p *= 2) {
            res += val[i] * p;
        }
        return res * 2;
    }

    private String getSystype(int numOp, int typeBit) {
        if (typeBit == 0x09) {
            ans.codeOp = "sfence.vma";
            ans.rs1 = getReg(parseBites(15, 20));
            ans.rs2 = getReg(parseBites(20, 25));
            return ans.rs1 + " " + ans.rs2;
        }
        typeBit = parseBites(20, 32);
        ans.codeOp = switch (typeBit) {
            case 0x000 -> "ecall";
            case 0x001 -> "ebreak";
            case 0x102 -> "sret";
            case 0x302 -> "mret";
            case 0x7b2 -> "dret";
```

```
            case 0x105 -> "wfi";
            default -> null;
        };
        if (ans.codeOp != null) {
            return ans.codeOp;
        }
        ans.codeOp = switch (numOp) {
            case 1 -> "csrrw";
            case 2 -> "csrrs";
            case 3 -> "csrrc";
            case 5 -> "csrrwi";
            case 6 -> "csrrsi";
            case 7 -> "csrrci";
            default -> null;
        };
        ans.rd = getReg(parseBites(7, 12));
        ans.offset = parseBites(20, 32);
        ans.rs1 = getReg(parseBites(15, 20));
        ans.imm = parseBites(15, 20);
        if (numOp < 4)
```

```
            return ans.rd + " " + ans.offset + " " + ans.rs1;
        return ans.rd + " " + ans.offset + " " + ans.imm;
    }

    private String getFtype(int numOp) { // ok
        ans.codeOp = switch (numOp) {
            case 0 -> "fence";
            case 1 -> "fence.i";
            default -> null;
        };
        if (numOp == 0) {
            ans.pred = parseBites(24, 28);
            ans.succ = parseBites(20, 24);
            return ans.pred + " " + ans.succ;
        }
        return "";
    }

    private String getBitType(int numOp, int typeBit) { //ok
        if (typeBit == 1) {
```

```
            ans.codeOp = switch (numOp) { // rv32m
                case 0 -> "mul";
                case 1 -> "mulh";
                case 2 -> "mulhsu";
                case 3 -> "mulhu";
                case 4 -> "div";
                case 5 -> "divu";
                case 6 -> "rem";
                case 7 -> "remu";
                default -> null;
            };
        } else if (typeBit == 32) {
            if (numOp == 0)
                ans.codeOp = "sub";
            else {
                ans.codeOp = "sra";
            }
        } else {
            ans.codeOp = switch (numOp) {
                case 0 -> "add";
```

```
            case 1 -> "sll";
            case 2 -> "slt";
            case 3 -> "sltu";
            case 4 -> "xor";
            case 5 -> "srl";
            case 6 -> "or";
            case 7 -> "and";
            default -> null;
        };
    }
    ans.rd = getReg(parseBites(7, 12));
    ans.rs1 = getReg(parseBites(15, 20));
    ans.rs2 = getReg(parseBites(20, 25));
    return ans.rd + " " + ans.rs1 + " " + ans.rs2;
  }

  private String getItype(int numOp) { // ok
    ans.codeOp = switch (numOp) {
        case 0 -> "addi";
        case 1 -> "slli";
```

```
            case 2 -> "slti";
            case 3 -> "sltiu";
            case 4 -> "xori";
            case 6 -> "ori";
            case 7 -> "andi";
            default -> "";
        };
        ans.rd = getReg(parseBites(7, 12));
        ans.rs1 = getReg(parseBites(15, 20));
        if (numOp == 5 || numOp == 1) {
            ans.shamt = parseBites(20, 25);
            if (parseBites(27, 32) == 8)
                ans.codeOp = "srai";
            else
                ans.codeOp = "srli";
            return ans.rd + " " + ans.rs1 + " " + ans.shamt;
        }
        ans.imm = parseBites(20, 32);
        return ans.rd + " " + ans.rs1 + " " + ans.imm;
    }
```

```
    private String getStype(int numOp) { // ok
        ans.codeOp = switch (numOp) {
            case 0 -> "sb";
            case 1 -> "sh";
            case 2 -> "sw";
            default -> "";
        };
        ans.rs1 = getReg(parseBites(15, 20));
        ans.rs2 = getReg(parseBites(20, 25));
        return ans.rs2 + " " + getSoffset() + "(" + ans.rs1 + ")";
    }

    private String getLtype(int numOp) { // ok
        ans.codeOp = switch (numOp) {
            case 0 -> "lb";
            case 1 -> "lh";
            case 2 -> "lw";
            case 4 -> "lbu";
            case 5 -> "lhu";
```

```
            default -> null;
        };
        ans.rd = getReg(parseBites(7, 12));
        ans.rs1 = getReg(parseBites(15, 20));
        return ans.rd + " " + parseBites(20, 32) + "(" + ans.rs1 +
")";
    }

    private String getBtype(int numOp) { // ok
        ans.codeOp = switch (numOp) {
            case 0 -> "beq";
            case 1 -> "bne";
            case 4 -> "blt";
            case 5 -> "bge";
            case 6 -> "bltu";
            case 7 -> "bgeu";
            default -> "";
        };
        ans.rs1 = getReg(parseBites(15, 20));
        ans.rs2 = getReg(parseBites(20, 25));
```

```
        ans.offset = getBoffset();
        return ans.rs1 + " " + ans.rs2;
    }

    private int getSoffset() {
        int[] val = new int[12];
        System.arraycopy(bits, 25, val, 5, 7);
        System.arraycopy(bits, 7, val, 0, 5);
        int p = 1;
        int res = 0;
        for (int i = 0; i < 12; i++, p *= 2) {
            res += val[i] * p;
        }
        return res * 2;
    }

    private int getBoffset() {
        int[] val = new int[13];
        val[12] = bits[31];
        val[11] = bits[7];
```

```
        System.arraycopy(bits, 25, val, 5, 6);
        System.arraycopy(bits, 8, val, 1, 4);
        int p = 1;
        int res = 0;
        for (int i = 1; i < 13; i++, p *= 2) {
            res += val[i] * p;
        }
        return res * 2;
    }

    private String getReg(int reg) {
        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 (5 <= reg && reg <= 7)
            return "t" + (reg - 5);
        else if (reg == 8)
            return "s0";
        else if (reg == 9)
            return "s1";
        else if (10 <= reg && reg <= 17)
            return "a" + (reg - 10);
        else if (18 <= reg && reg <= 27)
            return "s" + (reg - 18 + 2);
        else if (28 <= reg && reg <= 31)
            return "t" + (reg - 28 + 3);
        else
            throw new AssertionError("RISC-V doesn't have register
-" + reg);
    }
}
```

ParseOperation16bit.java

```
package babakapa;


public class ParseOperation16bit {
    private final int x;
    Format ans = new Format();
    private final int[] bits = new int[16];


    private void getBitsMas() {
        int x = this.x;
        int pos = 0;
        while (x > 0) {
            bits[pos++] = x % 2;
            x /= 2;
        }
    }


    ParseOperation16bit(int x) {
```

```
        this.x = x;
        getBitsMas();
    }

    private int parseBites(int l, int r) {
        int p = 1, res = 0;
        for (int i = l; i < r; i++, p *= 2) {
            res += bits[i] * p;
        }
        return res;
    }

    public String write() {
        int type = parseBites(0, 2);
        switch (type) {
            case 0:
                getType0();
            case 1:
                getType1();
            case 2:
```

```
                getType2();
            default:
                checkSpecial();
        }
        if(ans.codeOp == null) {
            return x + "";
        }
        return ans.codeOp;
    }
    private int getImmAddi16sp() {
        int[] val = new int[12];
        System.arraycopy(bits, 25, val, 5, 7);
        System.arraycopy(bits, 7, val, 0, 5);
        int p = 1;
        int res = 0;
        for (int i = 0; i < 12; i++, p *= 2) {
            res += val[i] * p;
        }
        return res * 2;
    }
```

```
  private void checkSpecial() {
      int t1 = parseBites(0, 2);
      int t2 = parseBites(13, 16);
      int t3 = parseBites(12, 13);
      int t4 = parseBites(7, 12);
      int t5 = parseBites(2, 6);
      if (t1 == 1) {
          if (t2 == 0 && t3 == 0 && t4 == 0 && t5 == 0) {
              ans.codeOp = "c.nop";
          }
          if (t2 == 3 && t4 == 2) {
              ans.codeOp = "c.addi16sp";
          }
      }
      if (t1 == 2) {
          if (t2 == 4) {
              if (t3 == 0 && t5 == 0) {
                  ans.codeOp = "c.jr";
                  return;
              }
```

```
            if (t3 == 1) {
                if (t4 == 0 && t5 == 0) {
                    ans.codeOp = "c.ebreak";
                    return;
                }
                if (t5 == 0) {
                    ans.codeOp = "c.jalr";
                }
            }
        }
    }
}

private void getType2() {
    int numOp = parseBites(13, 16);
    ans.codeOp = switch (numOp) {
        case 0 -> "c.slli";
        case 1 -> "c.fldsp";
        case 2 -> "c.lwsp";
        case 3 -> "c.flwsp";
```

```
            case 5 -> "c.fsdsp";
            case 6 -> "c.swsp";
            case 7 -> "c.fswsp";
            default -> null;
        };
        if (numOp == 4) {
            numOp = parseBites(12, 13);
            if (numOp == 0) {
                ans.codeOp = "c.mv";
            } else {
                ans.codeOp = "c.add";
            }
        }
    }

    private void getType1() {
        int numOp = parseBites(13, 16);
        ans.codeOp = switch (numOp) {
            case 0 -> "c.addi";
            case 1 -> "c.jal";
```

```
            case 2 -> "c.li";
            case 3 -> "c.lui";
            case 5 -> "c.j";
            case 6 -> "c.beqz";
            case 7 -> "c.bnez";
            default -> null;
        };
        if (numOp == 4) {
            numOp = parseBites(10, 12);
            ans.codeOp = switch (numOp) {
                case 0 -> "c.srli";
                case 1 -> "c.srai";
                case 2 -> "c.andi";
                default -> null;
            };
            if (numOp == 3) {
                numOp = parseBites(5, 7);
                ans.codeOp = switch (numOp) {
                    case 0 -> "c.sub";
                    case 1 -> "c.xor";
```

```
                case 2 -> "c.or";
                case 3 -> "c.and";
                default -> null;
            };
        }
    }
}

private void getType0() {
    ans.codeOp = switch (parseBites(13, 16)) {
        case 0 -> "c.addi4spn";
        case 1 -> "c.fld";
        case 2 -> "c.lw";
        case 3 -> "c.flw";
        case 5 -> "c.fsd";
        case 6 -> "c.sw";
        case 7 -> "c.fsw";
        default -> null;
    };
}
```

```
}
```

Section.java

```
package babakapa;


public class Section {
   int sh_offset;
   int sh_size;
   int sh_addr;
   int sh_type;
   int sh_name;
   String name;
   Section(int sh_name, int sh_type, int sh_addr, int sh_offset, int
sh_size) {
       this.sh_offset = sh_offset;
       this.sh_size = sh_size;
       this.sh_addr = sh_addr;
       this.sh_type = sh_type;
       this.sh_name = sh_name;
   }
   public void setName(String name) {
```

```
        this.name = name;
    }
}
```

Symtab.java

```
package babakapa;


public class Symtab {
    int st_name;
    Integer st_value;
    int st_size;
    int st_info;
    String st_other;
    String st_shndx;
    String st_bind;
    String st_type;
    String name;


    Symtab(int st_name, Integer st_value, int st_size, int st_info, int st_other, int st_shndx) {
        this.st_name = st_name;
```

```
        this.st_value = st_value;
        this.st_size = st_size;
        this.st_info = st_info;
        this.st_other = checkStOther(st_other);
        this.st_shndx = checkStShndx(st_shndx);
        this.st_type = checkStType(this.st_info & 0xf);
        this.st_bind = checkStBind(this.st_info >> 4);
    }
    public void setName(String name) {
        this.name = name;
    }
    private String checkStShndx(int x) {
        return switch (x) {
            case 0 -> "UNDEF";
            case 0xfff1 -> "ABS";
            case 0xfff2 -> "COMMON";
            default -> x + "";
        };
    }
```

```
    private String checkStOther(int x) { // готово
        return switch (x) {
            case 0 -> "DEFAULT";
            case 1 -> "INTERNAL";
            case 2 -> "HIDDEN";
            case 3 -> "PROTECTED";
            default -> null;
        };
    }

    private String checkStBind(int x) {// готово
        return switch (x) {
            case 0 -> "LOCAL";
            case 1 -> "GLOBAL";
            case 2 -> "WEAK";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> null;
        };
    }
```

```
    private String checkStType(int x) {// готово
        return switch (x) {
            case 1 -> "OBJECT";
            case 2 -> "FUNC";
            case 3 -> "SECTION";
            case 4 -> "FILE";
            case 5 -> "COMMON";
            case 6 -> "TLS";
            case 12 -> "HIOS";
            case 13 -> "LOPROC";
            case 15 -> "HIPROC";
            default -> "NOTYPE";
        };
    }
}
```

Parse.java

```
package babakapa;
```

```
public class Parse {
    public int[] mas;
    Parse(int[] mas) {
        this.mas = mas;
    }
    Parse() {
    }

    public String getMark(int cnt) {
        StringBuilder res = new StringBuilder();
        while(cnt > 0) {
            res.append(cnt % 10);
            cnt /= 10;
        }
        while(res.length() < 6) {
            res.append("0");
        }
        res.reverse();
        return "LOC_" + res;
    }
```

```
    public int parseIntBytes(int l, int r) {
        int res = 0;
        for(int i = r - 1; i >= l; i--) {
            res <<= 8;
            res += mas[i];
        }
        return res;
    }

    public long parseLongBytes(int l, int r) {
        long res = 0L;
        for(int i = r - 1; i >= l; i--) {
            res <<= 8;
            res += mas[i];
        }
        return res;
    }
}
```

Operation.java

```
package babakapa;


public class Operation {
    Integer addr;
    String index;
    String op;
    Integer offset;
    String firstMark;
    String secondMark;
    Operation() {
        index = null;
        op = null;
        offset = null;
        firstMark = null;
        secondMark = null;
    }
    public void setAddr(int x) {
        addr = x;
        this.index = Integer.toHexString(x);
```

```
    }

}
```