| Лабораторная работа № 3 | Б09 | 2022 |
|---|---|---|
| ISA | Васильев Дмитрий Сергеевич | |

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

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

В моём случае работа выполнена на языке C++. Использовался компилятор Apple clang version 13.1.6 (clang-1316.0.21.2.5).

**Описание системы кодирования RISC-V -** это ISA, реализованная на принципах RISC (Reduced Instruction Set Architecture).

В отличие от большинства известных ISA - распространяется под open source лицензией. Это одно из главных её преимуществ.

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд и несколько стандартных опциональных расширений (в рамках этой работы мы рассмотрим расширение “M” - Standard Extension for Integer Multiplication and Division)

Регистры в этой архитектуре кодируется как показано в табличке. Из интересного: нулевой регистр захардкожен нулём (zero) попытки записи туда будут игнорироваться, а чтение равносильно чтению нуля.

Команды в этой архитектуре кодируются по - разному. Существуют 6 основных вариантов инструкций по типу кодирования:

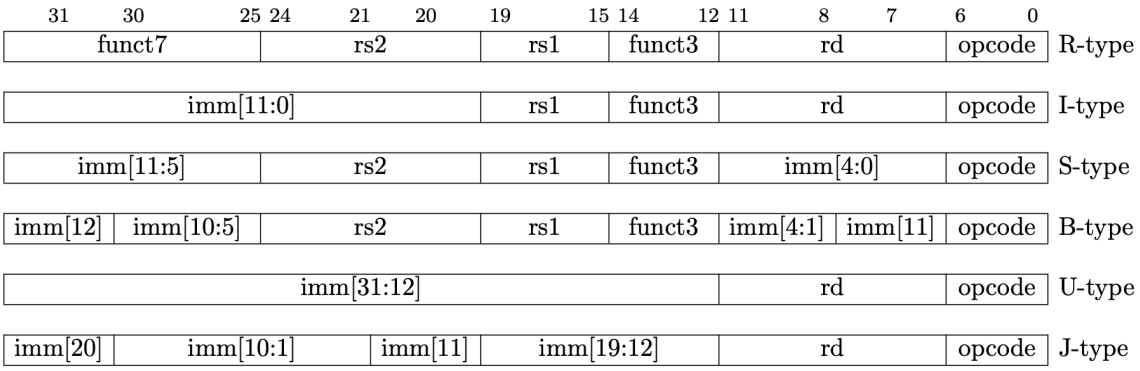


Рисунок 1 – Основные типы инструкций

Рассмотрим, например, S - type: `opcode` (биты с 6 по 0) и `funct3` (с 14 по 12) позволяют определить, какая инструкция нам попалась. `rs1` и `rs2` - по 5 битов - индексы регистров (как по индексу получить имя регистра показано в табличке выше). В случае `S type` - из `imm` собирается `offset:` с 31 по 25 биты инструкции находятся 7 старших битов оффсета, а с 11 по 7 - младшие биты оффсета.

Аналогично кодируются другие команды.

**Описание структуры ELF:** Elf файл устроен следующим образом: сначала идёт хедер всего файла - в нём собрана основная информация о файле. Тут можно найти: байт, обозначающий 32битный или 64битный у нас формат; байт, отвечающий за то, какой у файла эндиан; 2 байта, обозначающие целевую ISA; отступ от начала файла (в байтах), начиная с которого идут section хедеры; их количество; индекс section хедера, той секции, которая содержит имена остальных секций. И ещё много чего.

Дальше могут идти program хедеры, но они нам не нужны в рамках этой работы.

С помощью оффсета, найденного в основном хедере, мы можем определить место, где начинаются section хедеры. Это заголовки,

содержащие информацию о различных секциях файла. Например, здесь можно найти такую информацию о секции: её тип (таблица символом, таблица строк и другие); отступ до соответствующей секции от начала файла; её размер;  отступ от начала данных в `.shstrtab` до имени данной секции и другое.

В зависимости от секции в ней лежат разные данные.

Так в секции .text лежат инструкции, которые мы в дальнейшем будем парсить. (Лежат подряд: инструкци - 4 байта). Чтобы понять, когда остановится читать инструкции, нужно посмотреть на `sh_size` - размер этой секции (лежит в её хедере).

А в секции `.symtab` по порядку лежат `Elf32_Sym`  структуры. Их устройство подробно описано в документации.  Если вкратце, то там лежат все необходимые для парсинга символа поля. Например, поле `st_name` даёт offset от начала данных в string table, до имени данного символа, а байт `st_info` - это конкатенация 4х битов бинда и 4х битов типа.

Подводя итог: при работе с elf файлом мне пришлось:

1) Распарсить хедер
2) Найти нужные хедеры секций
3) По ним прыгнуть к соответствующим данным
4) В зависимости от того, какие это данные, распарсить их надлежащим образом.

**Описание работы написанного кода:** Сначала рассматривается хедер ELF файла. Производятся необходимые проверки (совпадение магической константы, правильная битность, эндиан, актуальность версии,

целевая ISA). Считываются необходимые данные: `e_shoff, e_shnum, e_shstrndx.`

После этого парсятся все section хедеры (в таблице `shstr` были найдены их имена). Находятся оффсеты `sym table` и `string table` (нужно пройтись по всем section хедерам и найти те, которые называются `.symtab` и `.strtab`)

Читаем символы из таблицы символов. Запоминаем о них необходимую информацию. Если тип символа - функция, то заносим его имя в специальную мапу, которая пригодится при выводе (словарь принимает адрес, возвращает имя). После того, как все символы распаршены, переходим к парсингу инструкций.

Чтобы распарсить инструкции я завёл специальный массив, в который положил все инструкции, который могут нам встретиться в рамках этой работы (вот часть заполнения этого массива):

```
instructions[index++] = {"lui", 0b0110111, U};
instructions[index++] = {"auipc", 0b0010111, U};
instructions[index++] = {"jal", 0b1101111, J};
instructions[index++] = {"beq", 0b1100011, B, 0b000};
instructions[index++] = {"bne", 0b1100011, B, 0b001};
instructions[index++] = {"blt", 0b1100011, B, 0b100};
instructions[index++] = {"bge", 0b1100011, B, 0b101};
instructions[index++] = {"bltu", 0b1100011, B, 0b110};
instructions[index++] = {"sb", 0b0100011, S, 0b000};
instructions[index++] = {"sh", 0b0100011, S, 0b001};
instructions[index++] = {"sw", 0b0100011, S, 0b010};
instructions[index++] = {"addi", 0b0010011, I, 0b000};
instructions[index++] = {"slti", 0b0010011, I, 0b010};
instructions[index++] = {"sltiu", 0b0010011, I, 0b011};
instructions[index++] = {"xori", 0b0010011, I, 0b100};
instructions[index++] = {"ori", 0b0010011, I, 0b110};
instructions[index++] = {"andi", 0b0010011, I, 0b111};
```

```
instructions[index++] = {"slli", 0b0010011, I, 0b001, 0b0000000};
instructions[index++] = {"srli", 0b0010011, I, 0b101, 0b0000000};
instructions[index++] = {"srai", 0b0010011, I, 0b101, 0b0100000};
```

Структура инструкции такова (заполняются только те поля, которые есть). (Тип инструкции - enum (R - type, B - type и другие). bit20 - двадцатый бит, позволяющий отличить `ecall` от `ebreak`)

```
struct Instruction {
    std::string name;
    std::bitset<7> opcode;
    InstructionType instructionType = UNKNOWN;
    std::bitset<3> func3;
    std::bitset<7> func7;

    std::bitset<1> bit20;
};
```

Чтобы определить имя и тип инструкции, берется её opcode и в списке известных инструкций оставляются только те, у которых opcode такой же.

Если потенциально подходящих инструкций осталось больше одной, то из них берутся те, у которых такой же func3, потом func7, потом bit20.

Если на каком - то этапе вектор потенциально удовлетворяющих инструкций опустел, значит мы встретили unknown_instruction.

Когда мы уже знаем имя и тип инструкции, то нам не составит труда ее распарсить (для каждого типа инструкций - свой парсер) и вывести.

Вот пример парсера для инструкций U - type:

```
void processUType(Byte4 instruction, const std::string &name) {
    std::string rd = getRegName(getBits<11, 7, 32>(instruction));
    uint imm = getBits<31, 12, 32>(instruction).to_ulong();
```

```
        fprintf(out, "%7s\t%s, %d", name.c_str(), rd.c_str(), imm);
    }
```

(`getRegName` по пятибитовому числу возвращает имя регистра, функция `getBits<from, to, size>` принимает `size` битовое число и возвращает из него биты с `from` по `to`).

Также стоит отметить, что инструкции типа I пришлось разбить на два вида: обычные и I_LOAD так как у них разный формат вывода.

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

```
.text
00010074   <main>:
   10074:    ff010113        addi      sp, sp, -16
   10078:    00112623          sw      ra, 12(sp)
   1007c:    030000ef         jal      ra, 0x100ac <mmul>
   10080:    00c12083          lw      ra, 12(sp)
   10084:    00000513        addi      a0, zero, 0
   10088:    01010113        addi      sp, sp, 16
   1008c:    00008067        jalr      zero, 0(ra)
   10090:    00000013        addi      zero, zero, 0
   10094:    00100137         lui      sp, 256
   10098:    fddff0ef         jal      ra, 0x10074 <main>
   1009c:    00050593        addi      a1, a0, 0
   100a0:    00a00893        addi      a7, zero, 10
   100a4:    0ff0000f        unknown_instruction
   100a8:    00000073        ecall
000100ac   <mmul>:
   100ac:    00011f37         lui      t5, 17
   100b0:    124f0513        addi      a0, t5, 292
   100b4:    65450513        addi      a0, a0, 1620
   100b8:    124f0f13        addi      t5, t5, 292
   100bc:    e4018293        addi      t0, gp, -448
   100c0:    fd018f93        addi      t6, gp, -48
   100c4:    02800e93        addi      t4, zero, 40
   100c8:    fec50e13        addi      t3, a0, -20
   100cc:    000f0313        addi      t1, t5, 0
   100d0:    000f8893        addi      a7, t6, 0
   100d4:    00000813        addi      a6, zero, 0
   100d8:    00088693        addi      a3, a7, 0
   100dc:    000e0793        addi      a5, t3, 0
   100e0:    00000613        addi      a2, zero, 0
   100e4:    00078703          lb      a4, 0(a5)
   100e8:    00069583          lh      a1, 0(a3)
   100ec:    00178793        addi      a5, a5, 1
   100f0:    02868693        addi      a3, a3, 40
   100f4:    02b70733         mul      a4, a4, a1
```

```
   100f8:    00e60633          add      a2, a2, a4
   100fc:    fea794e3          bne      a5, a0, 0x100e4 <L0>
   10100:    00c32023           sw      a2, 0(t1)
   10104:    00280813        addi      a6, a6, 2
   10108:    00430313        addi      t1, t1, 4
   1010c:    00288893        addi      a7, a7, 2
   10110:    fdd814e3          bne      a6, t4, 0x100d8 <L1>
   10114:    050f0f13        addi      t5, t5, 80
   10118:    01478513        addi      a0, a5, 20
   1011c:    fa5f16e3          bne      t5, t0, 0x100c8 <L2>
   10120:    00008067        jalr      zero, 0(ra)

.symtab
Symbol Value              Size Type     Bind     Vis       Index Name
[   0] 0x0                   0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074               0 SECTION  LOCAL    DEFAULT       1
[   2] 0x11124               0 SECTION  LOCAL    DEFAULT       2
[   3] 0x0                   0 SECTION  LOCAL    DEFAULT       3
[   4] 0x0                   0 SECTION  LOCAL    DEFAULT       4
[   5] 0x0                   0 FILE     LOCAL    DEFAULT     ABS test.c
[   6] 0x11924               0 NOTYPE   GLOBAL   DEFAULT     ABS __global_pointer$
[   7] 0x118F4             800 OBJECT   GLOBAL   DEFAULT       2 b
[   8] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1 __SDATA_BEGIN__
[   9] 0x100AC             120 FUNC     GLOBAL   DEFAULT       1 mmul
[  10] 0x0                   0 NOTYPE   GLOBAL   DEFAULT   UNDEF _start
[  11] 0x11124            1600 OBJECT   GLOBAL   DEFAULT       2 c
[  12] 0x11C14               0 NOTYPE   GLOBAL   DEFAULT       2 __BSS_END__
[  13] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       2 __bss_start
[  14] 0x10074              28 FUNC     GLOBAL   DEFAULT       1 main
[  15] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1 __DATA_BEGIN__
[  16] 0x11124               0 NOTYPE   GLOBAL   DEFAULT       1 _edata
[  17] 0x11C14               0 NOTYPE   GLOBAL   DEFAULT       2 _end
[  18] 0x11764             400 OBJECT   GLOBAL   DEFAULT       2 a
```

**Список источников:**


1) основная информация об устройстве elf файла: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
2) информация об устройсиве symtab https://refspecs.linuxbase.org/elf/gabi4+/ch4.symtab.html
3) документация по RISC - V, в которой подробно описаны типы инструкции и то, в каком месте каждой инструкции находится нужная информация: https://riscv.org/technical/specifications/
4) информация про регистры в RISC - V


**Листинг кода:**

**main.cpp**

```cpp
#include "ElfReader.h"

int main(int argc, char **argv) {
   if(argc < 2) {
       std::cout << "Error! Wrong amount of arguments.";
       return 0;
   }

   std::ifstream in(argv[1], std::ios::binary);
   if(!in.is_open()) {
       std::cout << "Error! Impossible to open input file.";
       return 0;
   }

   FILE *out = fopen(argv[2], "w");
   if(out == nullptr) {
       std::cout << "Error! Impossible to open output file.";
       return 0;
   }

   ElfReader er(in, out);
   er.process();
   in.close();
   fclose(out);
   return 0;
}
```

**constants.h**

```cpp
#pragma once

#include <bitset>
#include <vector>

#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_VISIBILITY(o) ((o)&0x3)

using uchar = unsigned char;
using uint = unsigned int;
using Byte = std::bitset<8>;
using Byte4 = std::bitset<32>;
using Bytes = std::vector<Byte>;

using Reg = std::bitset<5>;

const uint sectionHeaderSize = 0x28;

const uint SHT_SYMTAB = 0x2;
const uint SHT_STRTAB = 0x3;

const uchar STB_LOCAL = 0;
const uchar STB_GLOBAL = 1;
const uchar STB_WEAK = 2;
const uchar STB_LOOS = 10;
const uchar STB_HIOS = 12;
const uchar STB_LOPROC = 13;
```

```
const uchar STB_HIPROC = 15;

const uchar STT_NOTYPE = 0;
const uchar STT_OBJECT = 1;
const uchar STT_FUNC = 2;
const uchar STT_SECTION = 3;
const uchar STT_FILE = 4;
const uchar STT_COMMON = 5;
const uchar STT_TLS = 6;
const uchar STT_LOOS = 10;
const uchar STT_HIOS = 12;
const uchar STT_LOPROC = 13;
const uchar STT_HIPROC = 15;

const uchar STV_DEFAULT = 0;
const uchar STV_INTERNAL = 1;
const uchar STV_HIDDEN = 2;
const uchar STV_PROTECTED = 3;

const uint16_t SHN_UNDEF = 0;
const uint16_t SHN_LORESERVE = 0xff00;
const uint16_t SHN_LOPROC = 0xff00;
const uint16_t SHN_HIPROC = 0xff1f;
const uint16_t SHN_LOOS = 0xff20;
const uint16_t SHN_HIOS = 0xff3f;
const uint16_t SHN_ABS = 0xfff1;
const uint16_t SHN_COMMON = 0xfff2;
const uint16_t SHN_XINDEX = 0xffff;
const uint16_t SHN_HIRESERVE = 0xffff;

const uint reversedELFMagicConstantOffset = 0x00;
const uint reversedELFMagicConstant = 0x464C457F;
const uint bitnessOffset = 0x04;
const uint bitnessIs32 = 1;
const uint endianOffset = 0x05;
const uint endianIsLittle = 1;
const uint currentVersionOffset = 0x06;
const uint isCurrentVersion = 1;
const uint targetISAOffset = 0x12;
const uint RISCVIsTargetISA = 0xf3;

const uint e_shoff_offset = 0x20;
const uint e_shnum_offset = 0x30;
const uint e_shstrndx_offset = 0x32;

const uint sh_name_offset = 0x00;
const uint sh_addr_offset = 0x0C;
const uint sh_offset_offset = 0x10;
const uint sh_size_offset = 0x14;

const uint symbolSizeBits = 16;
const uint instructionSize = 4;
```

**bitsReader.h**

```
#pragma once

#include <string>
```

```cpp
#include "constants.h"

template<uint size1, uint size2>
inline static std::bitset<size1 + size2> concat(const std::bitset<size1> &b1,
const std::bitset<size2> &b2) {
   std::string s1 = b1.to_string();
   std::string s2 = b2.to_string();
   return std::bitset<size1 + size2>(s1 + s2);
}

template<uint size1, uint size2, uint size3, uint size4>
static std::bitset<size1 + size2 + size3 + size4> inline
concat(const std::bitset<size1> &b1, const std::bitset<size2> &b2, const
std::bitset<size3> &b3,
      const std::bitset<size4> &b4) {
   return std::bitset<size1 + size2 + size3 + size4>(
           concat<size1 + size2, size3 + size4>(concat<size1, size2>(b1, b2),
concat<size3, size4>(b3, b4)));
}

template<uint from, uint to, uint srcSize = 32>
inline std::bitset<from - to + 1> getBits(std::bitset<srcSize> bits) {
   return std::bitset<from - to + 1>(
           bits.to_string().substr(srcSize - 1 - from, from - to + 1));
}

template<uint size>
inline int32_t convert2signed(std::bitset<size> num) {
   int32_t result = num.to_ulong();
   int pow = 1;
   for (int i = 0; i < size; i++) pow *= 2;
   if (num[size - 1]) result -= pow;
   return result;
}

class BitsReader {
public:
   BitsReader(Bytes &&bytes) : source(bytes) {}

   inline uint getByte(uint offset) const { return source[offset].to_ulong();
}

   inline uint getByte2(uint offset) const {
       return concat<8, 8>(source[offset + 1], source[offset]).to_ulong();
   }

   inline uint getByte4(uint offset) const {
       return concat<8, 8, 8, 8>(source[offset + 3], source[offset + 2],
source[offset + 1],
                                 source[offset]).to_ulong();
   }

   uint operator[](uint offset) const { return getByte(offset); }

private:
   Bytes source = {};
};
```

InstructionParser.h

```
#pragma once

#include "constants.h"
#include "bitsReader.h"
#include <unordered_map>
#include <array>
#include <string>
#include <fstream>

const uint amountOfInstructions = 47;

class InstructionParser {
public:
   InstructionParser(FILE *out);

   void parsInstruction(Byte4 instruction, uint address);

   std::unordered_map<uint, std::string> address2FunctionName;
private:
   enum InstructionType {
       R,
       I,
       I_LOAD,
       S,
       B,
       U,
       J,
       ECALL_EBREAK,
       UNKNOWN
   };

   struct Instruction {
       std::string name;
       std::bitset<7> opcode;
       InstructionType instructionType = UNKNOWN;
       std::bitset<3> func3;
       std::bitset<7> func7;

       std::bitset<1> bit20;
   };

   void fillInstructions();

   void getTypeAndProcess(Byte4 instruction, InstructionType type, const
std::string &name, uint address);

   void processRType(Byte4 instruction, const std::string &name);

   void processIType(Byte4 instruction, const std::string &name,
InstructionType type);

   void processSType(Byte4 instruction, const std::string &name);

   void processBType(Byte4 instruction, const std::string &name, uint
address);

   void processUType(Byte4 instruction, const std::string &name);
```

```
    void processJType(Byte4 instruction, const std::string &name, uint
address);

    void processECALL_EBREAKType(const std::string &name);

    void processUnknown();

    bool checkSuccess(Byte4 instruction, std::vector<Instruction>
&availableInstructions, uint address);

    void printRelevantObjectName(uint address);

    static std::string getRegName(Reg rg);

    FILE *out;
    std::string unknownInstructionName = "unknown_instruction";
    std::array<Instruction, amountOfInstructions> instructions;

    uint amountOfLocalObjects = 0;
};
```

InstructionParser.cpp

```
#include "InstructionParser.h"

void InstructionParser::fillInstructions() {
    uint index = 0;
    instructions[index++] = {"lui", 0b0110111, U};
    instructions[index++] = {"auipc", 0b0010111, U};
    instructions[index++] = {"jal", 0b1101111, J};
    instructions[index++] = {"jalr", 0b1100111, I_LOAD, 0b000};
    instructions[index++] = {"beq", 0b1100011, B, 0b000};
    instructions[index++] = {"bne", 0b1100011, B, 0b001};
    instructions[index++] = {"blt", 0b1100011, B, 0b100};
    instructions[index++] = {"bge", 0b1100011, B, 0b101};
    instructions[index++] = {"bltu", 0b1100011, B, 0b110};
    instructions[index++] = {"bgeu", 0b1100011, B, 0b111};
    instructions[index++] = {"lb", 0b0000011, I_LOAD, 0b000};
    instructions[index++] = {"lh", 0b0000011, I_LOAD, 0b001};
    instructions[index++] = {"lw", 0b0000011, I_LOAD, 0b010};
    instructions[index++] = {"lbu", 0b0000011, I_LOAD, 0b100};
    instructions[index++] = {"lhu", 0b0000011, I_LOAD, 0b101};
    instructions[index++] = {"sb", 0b0100011, S, 0b000};
    instructions[index++] = {"sh", 0b0100011, S, 0b001};
    instructions[index++] = {"sw", 0b0100011, S, 0b010};
    instructions[index++] = {"addi", 0b0010011, I, 0b000};
    instructions[index++] = {"slti", 0b0010011, I, 0b010};
    instructions[index++] = {"sltiu", 0b0010011, I, 0b011};
    instructions[index++] = {"xori", 0b0010011, I, 0b100};
    instructions[index++] = {"ori", 0b0010011, I, 0b110};
    instructions[index++] = {"andi", 0b0010011, I, 0b111};
    instructions[index++] = {"slli", 0b0010011, I, 0b001, 0b0000000};
    instructions[index++] = {"srli", 0b0010011, I, 0b101, 0b0000000};
    instructions[index++] = {"srai", 0b0010011, I, 0b101, 0b0100000};
    instructions[index++] = {"add", 0b0110011, R, 0b000, 0b0000000};
    instructions[index++] = {"sub", 0b0110011, R, 0b000, 0b0100000};
    instructions[index++] = {"sll", 0b0110011, R, 0b001, 0b0000000};
```

```
    instructions[index++] = {"slt", 0b0110011, R, 0b010, 0b0000000};
    instructions[index++] = {"sltu", 0b0110011, R, 0b011, 0b0000000};
    instructions[index++] = {"xor", 0b0110011, R, 0b100, 0b0000000};
    instructions[index++] = {"srl", 0b0110011, R, 0b101, 0b0000000};
    instructions[index++] = {"sra", 0b0110011, R, 0b101, 0b0100000};
    instructions[index++] = {"or", 0b0110011, R, 0b110, 0b0000000};
    instructions[index++] = {"and", 0b0110011, R, 0b111, 0b0000000};
    instructions[index++] = {"ecall", 0b1110011, ECALL_EBREAK, 0b000, 0b0000000,
0b0};
    instructions[index++] = {"ebreak", 0b1110011, ECALL_EBREAK, 0b000, 0b0000000,
0b1};
    instructions[index++] = {"mul", 0b0110011, R, 0b000, 0b0000001};
    instructions[index++] = {"mulh", 0b0110011, R, 0b001, 0b0000001};
    instructions[index++] = {"mulhsu", 0b0110011, R, 0b010, 0b0000001};
    instructions[index++] = {"mulhu", 0b0110011, R, 0b011, 0b0000001};
    instructions[index++] = {"div", 0b0110011, R, 0b100, 0b0000001};
    instructions[index++] = {"divu", 0b0110011, R, 0b101, 0b0000001};
    instructions[index++] = {"rem", 0b0110011, R, 0b110, 0b0000001};
    instructions[index++] = {"remu", 0b0110011, R, 0b111, 0b0000001};
}

InstructionParser::InstructionParser(FILE *out_) {
    out = out_;
    fillInstructions();
}

void
InstructionParser::getTypeAndProcess(Byte4 instruction, InstructionType type,
const std::string &name, uint address) {
    if (type == R) processRType(instruction, name);
    if (type == I or type == I_LOAD) processIType(instruction, name, type);
    if (type == S) processSType(instruction, name);
    if (type == B) processBType(instruction, name, address);
    if (type == U) processUType(instruction, name);
    if (type == J) processJType(instruction, name, address);
    if (type == ECALL_EBREAK) processECALL_EBREAKType(name);
    fprintf(out, "\n");
}

bool InstructionParser::checkSuccess(Byte4 instruction, std::vector<Instruction>
&availableInstructions, uint address) {
    if (availableInstructions.empty()) {
        processUnknown();
        return true;
    }

    if (availableInstructions.size() == 1) {
        getTypeAndProcess(instruction, availableInstructions[0].instructionType,
availableInstructions[0].name,
                          address);
        return true;
    }

    return false;
}

void InstructionParser::parsInstruction(Byte4 instruction, uint address) {
    if (address2FunctionName.count(address))
        fprintf(out, "%08x   <%s>:\n", address,
```

```
            address2FunctionName[address].c_str());
    fprintf(out, "   %05x:\t%08x\t", address,
            (uint)instruction.to_ulong());

    std::vector<Instruction> accessibleInstructions1;
    std::bitset<7> opcode = getBits<6, 0, 32>(instruction);
    for (auto &com: instructions)
        if (com.opcode == opcode) accessibleInstructions1.push_back(com);

    if (checkSuccess(instruction, accessibleInstructions1, address)) return;

    std::vector<Instruction> accessibleInstructions2;
    std::bitset<3> func3 = getBits<14, 12, 32>(instruction);
    for (auto &com: accessibleInstructions1)
        if (com.func3 == func3) accessibleInstructions2.push_back(com);

    if (checkSuccess(instruction, accessibleInstructions2, address)) return;

    std::vector<Instruction> accessibleInstructions3;
    std::bitset<7> func7 = getBits<31, 25, 32>(instruction);
    for (auto &com: accessibleInstructions2)
        if (com.func7 == func7) accessibleInstructions3.push_back(com);

    if (checkSuccess(instruction, accessibleInstructions3, address)) return;

    std::vector<Instruction> accessibleInstructions4;
    std::bitset<1> bit20 = getBits<20, 20, 32>(instruction);
    for (auto &com: accessibleInstructions3)
        if (com.bit20 == bit20) accessibleInstructions4.push_back(com);

    if (checkSuccess(instruction, accessibleInstructions4, address)) return;
}

void InstructionParser::processRType(Byte4 instruction, const std::string &name) {
    std::string rs1 = getRegName(getBits<19, 15, 32>(instruction));
    std::string rs2 = getRegName(getBits<24, 20, 32>(instruction));
    std::string rd = getRegName(getBits<11, 7, 32>(instruction));

    fprintf(out, "%7s\t%s, %s, %s", name.c_str(), rd.c_str(), rs1.c_str(),
rs2.c_str());
}

void InstructionParser::processIType(Byte4 instruction, const std::string &name,
InstructionType type) {
    std::string rs1 = getRegName(getBits<19, 15, 32>(instruction));
    uint imm;
    if (name == "slli" or name == "srli" or name == "srai") imm = getBits<24, 20,
32>(instruction).to_ulong();
    else imm = convert2signed<12>(getBits<31, 20, 32>(instruction));
    std::string rd = getRegName(getBits<11, 7, 32>(instruction));
    fprintf(out, "%7s\t%s, ", name.c_str(), rd.c_str());
    if (type == I) fprintf(out, "%s, %d", rs1.c_str(), imm);
    if (type == I_LOAD) fprintf(out, "%d(%s)", imm, rs1.c_str());
}

void InstructionParser::processSType(Byte4 instruction, const std::string &name) {
    std::string rs1 = getRegName(getBits<19, 15, 32>(instruction));
    std::string rs2 = getRegName(getBits<24, 20, 32>(instruction));
```

```
    std::bitset<7> offset11_5 = getBits<31, 25, 32>(instruction);
    std::bitset<5> offset4_0 = getBits<11, 7, 32>(instruction);
    int offset = convert2signed<12>(concat<7, 5>(offset11_5, offset4_0));

    fprintf(out, "%7s\t%s, %d(%s)", name.c_str(), rs2.c_str(), offset,
rs1.c_str());
}

void InstructionParser::processBType(Byte4 instruction, const std::string &name,
uint address) {
    std::string rs1 = getRegName(getBits<19, 15, 32>(instruction));
    std::string rs2 = getRegName(getBits<24, 20, 32>(instruction));

    std::bitset<1> offset12 = getBits<31, 31, 32>(instruction);
    std::bitset<6> offset10_5 = getBits<30, 25, 32>(instruction);
    std::bitset<4> offset4_1 = getBits<11, 8, 32>(instruction);
    std::bitset<1> offset11 = getBits<7, 7, 32>(instruction);
    int32_t offset = convert2signed<13>(
            concat<12, 1>(concat<1, 1, 6, 4>(offset12, offset11, offset10_5,
offset4_1), 0));

    fprintf(out, "%7s\t%s, %s, 0x%0x", name.c_str(), rs1.c_str(), rs2.c_str(),
offset + address);
    printRelevantObjectName(offset + address);
}

void InstructionParser::processUType(Byte4 instruction, const std::string &name) {
    std::string rd = getRegName(getBits<11, 7, 32>(instruction));
    uint imm = getBits<31, 12, 32>(instruction).to_ulong();

    fprintf(out, "%7s\t%s, %d", name.c_str(), rd.c_str(), imm);
}

void InstructionParser::processJType(Byte4 instruction, const std::string &name,
uint address) {
    std::string rd = getRegName(getBits<11, 7, 32>(instruction));
    std::bitset<20> imm = getBits<31, 32 - 20, 32>(instruction);
    std::bitset<1> offset20 = getBits<19, 19, 20>(imm);
    std::bitset<8> offset19_12 = getBits<7, 0, 20>(imm);
    std::bitset<1> offset11 = getBits<8, 8, 20>(imm);
    std::bitset<10> offset10_1 = getBits<18, 9, 20>(imm);

    int32_t offset = convert2signed<21>(
            concat<20, 1>(concat<1, 8, 1, 10>(offset20, offset19_12, offset11,
offset10_1), 0));

    fprintf(out, "%7s\t%s, 0x%0x", name.c_str(), rd.c_str(), offset + address);
    printRelevantObjectName(offset + address);
}

void InstructionParser::processECALL_EBREAKType(const std::string &name) {
    fprintf(out, " %7s", name.c_str());
}

void InstructionParser::processUnknown() {
    fprintf(out, "   %7s\n", unknownInstructionName.c_str());
}

std::string InstructionParser::getRegName(Reg rg) {
```

```
    uchar reg = rg.to_ulong();
    if (reg == 0) return "zero";
    if (reg == 1) return "ra";
    if (reg == 2) return "sp";
    if (reg == 3) return "gp";
    if (reg == 4) return "tp";
    if (reg >= 5 && reg <= 7) return "t" + std::to_string(reg - 5);
    if (reg >= 8 && reg <= 9) return "s" + std::to_string(reg - 8);
    if (reg >= 10 && reg <= 17) return "a" + std::to_string(reg - 10);
    if (reg >= 18 && reg <= 27) return "s" + std::to_string(reg - 16);
    if (reg >= 28 && reg <= 31) return "t" + std::to_string(reg - 25);
    return "";
}

void InstructionParser::printRelevantObjectName(uint address) {
    if (address2FunctionName.count(address)) fprintf(out, " <%s>",
address2FunctionName[address].c_str());
    else fprintf(out, " <%s>", (address2FunctionName[address] = "L" +
std::to_string(amountOfLocalObjects++)).c_str());
}
```

ElfReader.h

```
#pragma once

#include <fstream>
#include <string>
#include <iostream>
#include "bitsReader.h"
#include "InstructionParser.h"

class ElfReader {
public:
    ElfReader(std::ifstream &in_, FILE *out);

    void process();

private:
    std::string isValid();

    void parseSectionHeader(uint index);

    void parseSectionHeaders();

    uint findTabIndex(const std::string &sectionName);

    std::string getSectionName(uint offset, uint sectionIndex);

    void parseSymbol(uint index);

    void parseSymbols();

    void processText();

    void printSymbol(uint index);

    void printSymbols();
```

```
struct SectionHeader {
    uint sh_name;
    uint sh_offset;
    uint sh_size;
    uint sh_addr;

    std::string name;
};

struct Symbol {
    uint st_name;
    uint st_value;
    uint st_size;
    uchar st_info;
    uchar st_other;
    uint16_t st_shndx;

    std::string name;

    inline std::string getTypeName() {
        auto type = ELF32_ST_TYPE(st_info);
        if (type == STT_NOTYPE) return "NOTYPE";
        if (type == STT_OBJECT) return "OBJECT";
        if (type == STT_FUNC) return "FUNC";
        if (type == STT_SECTION) return "SECTION";
        if (type == STT_FILE) return "FILE";
        if (type == STT_COMMON) return "COMMON";
        if (type == STT_TLS) return "TLS";
        if (type == STT_LOOS) return "LOOS";
        if (type == STT_HIOS) return "HIOS";
        if (type == STT_LOPROC) return "LOPROC";
        if (type == STT_HIPROC) return "HIPROC";

        return "UNKNOWN";
    }

    inline std::string getBindName() {
        auto type = ELF32_ST_BIND(st_info);
        if (type == STB_LOCAL) return "LOCAL";
        if (type == STB_GLOBAL) return "GLOBAL";
        if (type == STB_WEAK) return "WEAK";
        if (type == STB_LOOS) return "LOOS";
        if (type == STB_HIOS) return "HIOS";
        if (type == STB_LOPROC) return "LOPROC";
        if (type == STB_HIPROC) return "HIPROC";

        return "UNKNOWN";
    }

    inline std::string getVisName() {
        auto type = ELF32_ST_VISIBILITY(st_other);
        if (type == STV_DEFAULT) return "DEFAULT";
        if (type == STV_INTERNAL) return "INTERNAL";
        if (type == STV_HIDDEN) return "HIDDEN";
        if (type == STV_PROTECTED) return "PROTECTED";

        return "UNKNOWN";
    }
```

```
        inline std::string getIndexName() {
            auto type = st_shndx;
            if (type == SHN_UNDEF) return "UNDEF";
            if (type == SHN_LORESERVE) return "LORESERVE";
            if (type == SHN_LOPROC) return "LOPROC";
            if (type == SHN_HIPROC) return "HIPROC";
            if (type == SHN_LOOS) return "LOOS";
            if (type == SHN_HIOS) return "HIOS";
            if (type == SHN_ABS) return "ABS";
            if (type == SHN_COMMON) return "COMMON";
            if (type == SHN_XINDEX) return "XINDEX";
            if (type == SHN_HIRESERVE) return "HIRESERVE";

            return std::to_string(type);
        }
    };

    FILE *out;
    BitsReader byteReader = BitsReader({});
    InstructionParser instructionParser;
    std::vector<SectionHeader> sectionHeaders;
    std::vector<Symbol> symbols;

    uint e_shoff;
    uint e_shnum;
    uint e_shstrndx;

    uint e_symtabndx;
    uint e_strtabndx;

    SectionHeader textSectionHeader;
};
```

ElfReader.cpp

```
#include "ElfReader.h"

ElfReader::ElfReader(std::ifstream &in, FILE *out_) : out(out_),
instructionParser(out_) {
    uchar tmpChar;
    Bytes source;
    while (in.read((char *) &tmpChar, 1)) {
        source.emplace_back(tmpChar);
    }
    instructionParser = InstructionParser(out);
    byteReader = BitsReader(move(source));
}

std::string ElfReader::isValid() {
    if (byteReader.getByte4(reversedELFMagicConstantOffset) !=
reversedELFMagicConstant)
        return "File is incorrect (magic number does not match).";

    if (byteReader[bitnessOffset] != bitnessIs32) return "File is not in 32 bit
format.";
    if (byteReader[endianOffset] != endianIsLittle) return "File is not in little
endian.";
```

```
    if (byteReader[currentVersionOffset] != isCurrentVersion)
        return "File is not in the original and current version of ELF.";
    if (byteReader.getByte2(targetISAOffset) != RISCVIsTargetISA) return "File's
target ISA is not RISC-V.";

    return "";
}

void ElfReader::parseSectionHeaders() {
    e_shoff = byteReader.getByte4(e_shoff_offset);
    e_shnum = byteReader.getByte2(e_shnum_offset);
    e_shstrndx = byteReader.getByte2(e_shstrndx_offset);

    sectionHeaders.resize(e_shnum);

    parseSectionHeader(e_shstrndx);

    for (uint i = 0; i < e_shnum; i++) parseSectionHeader(i);
}

void ElfReader::parseSectionHeader(uint index) {
    uint currentOffset = e_shoff + index * sectionHeaderSize;
    SectionHeader &sectionHeader = sectionHeaders[index];

    sectionHeader.sh_name = byteReader.getByte4(currentOffset + sh_name_offset);
    sectionHeader.sh_addr = byteReader.getByte4(currentOffset + sh_addr_offset);
    sectionHeader.sh_offset = byteReader.getByte4(currentOffset +
sh_offset_offset);
    sectionHeader.sh_size = byteReader.getByte4(currentOffset + sh_size_offset);

    sectionHeader.name = getSectionName(sectionHeader.sh_name, e_shstrndx);
}

std::string ElfReader::getSectionName(uint offset, uint sectionIndex) {
    offset += sectionHeaders[sectionIndex].sh_offset;
    std::string result;
    auto tmp = (char) byteReader.getByte(offset++);
    while (tmp) {
        result += tmp;
        tmp = (char) byteReader.getByte(offset++);
    }
    return result;
}

uint ElfReader::findTabIndex(const std::string &sectionName) {
    for (int i = 0; i < e_shnum; i++) if (sectionHeaders[i].name == sectionName)
return i;
    return -1;
}

void ElfReader::parseSymbol(uint index) {
    Symbol &symbol = symbols[index];
    uint currentOffset = index * symbolSizeBits +
sectionHeaders[e_symtabndx].sh_offset;
    symbol.st_name = byteReader.getByte4(currentOffset);
    currentOffset += 4;
    symbol.st_value = byteReader.getByte4(currentOffset);
    currentOffset += 4;
    symbol.st_size = byteReader.getByte4(currentOffset);
```

```
    currentOffset += 4;
    symbol.st_info = byteReader.getByte(currentOffset);
    currentOffset += 1;
    symbol.st_other = byteReader.getByte(currentOffset);
    currentOffset += 1;
    symbol.st_shndx = byteReader.getByte2(currentOffset);

    symbol.name = getSectionName(symbol.st_name, e_strtabndx);

    if (ELF32_ST_TYPE(symbol.st_info) == STT_FUNC)
        if (!symbol.name.empty())
instructionParser.address2FunctionName[symbol.st_value] = symbol.name;
}

void ElfReader::parseSymbols() {
    uint amountOfSymbols = sectionHeaders[e_symtabndx].sh_size / symbolSizeBits;
    symbols.resize(amountOfSymbols);
    for (int i = 0; i < amountOfSymbols; i++) {
        parseSymbol(i);
    }
}

void ElfReader::processText() {
    uint index = 0;
    fprintf(out, ".text\n");
    while (index < textSectionHeader.sh_size) {

instructionParser.parsInstruction(byteReader.getByte4(textSectionHeader.sh_offset
+ index),
                                          textSectionHeader.sh_addr + index);
        index += instructionSize;
    }
}

void ElfReader::printSymbol(uint index) {
    Symbol &symbol = symbols[index];
    fprintf(out, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n", index,
symbol.st_value, symbol.st_size,
            symbol.getTypeName().c_str(), symbol.getBindName().c_str(),
symbol.getVisName().c_str(),
            symbol.getIndexName().c_str(),
            symbol.name.c_str());
}

void ElfReader::printSymbols() {
    uint amountOfSymbols = sectionHeaders[e_symtabndx].sh_size / symbolSizeBits;
    fprintf(out, "\n.symtab\nSymbol Value              Size Type     Bind     Vis
Index Name\n");
    symbols.resize(amountOfSymbols);
    for (int i = 0; i < amountOfSymbols; i++) {
        printSymbol(i);
    }
}

void ElfReader::process() {
    std::string error = isValid();
    if (!error.empty()) {
        std::cout << "Error!" + error;
        return;
```

```
    }

    parseSectionHeaders();

    e_strtabndx = findTabIndex(".strtab");
    e_symtabndx = findTabIndex(".symtab");
    textSectionHeader = sectionHeaders[findTabIndex(".text")];

    parseSymbols();

    processText();

    printSymbols();
}
```