Лабораторная работа №3	M3137	2022
ISA	Нартов Дмитрий Николаевич	

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

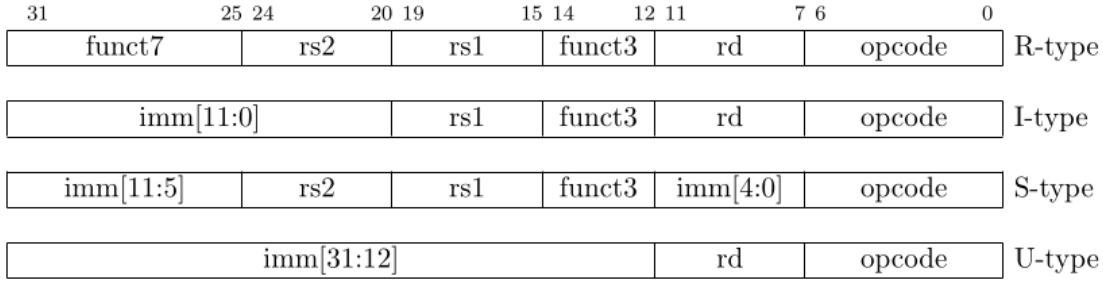
Инструментарий и требования к работе: C++ 20, Clang 14.0.5.

Описание: Разработать программу-транслятор, с помощью которой можно преобразовывать машинный код в текст программы на языке Ассемблера с набором команд RISC-V.

Система кодирования команд RISC-V.

Немного информации об этой ISA. RISC-V - это свободно распространяемая ISA, которая придерживается принципа RISC. Как известно существует два больших подхода к проектированию набора инструкции: RISC и CISC. RISC - reduced instruction set computer. Главная особенность заключается в максимальной упрощении инструкции, вследствие чего получаем небольшое ускорение, так как компьютеру проще обрабатывать инструкцию. CISC - complex instruction set computer. Здесь каждая команда может быть представлена как несколько низкоуровневых команд. Можно также подчеркнуть, что RISC-V является load/store архитектурой, что означает существование двух групп команд: load/store, которые обращаются к памяти, и ALU, которые используют, только регистры процессора.

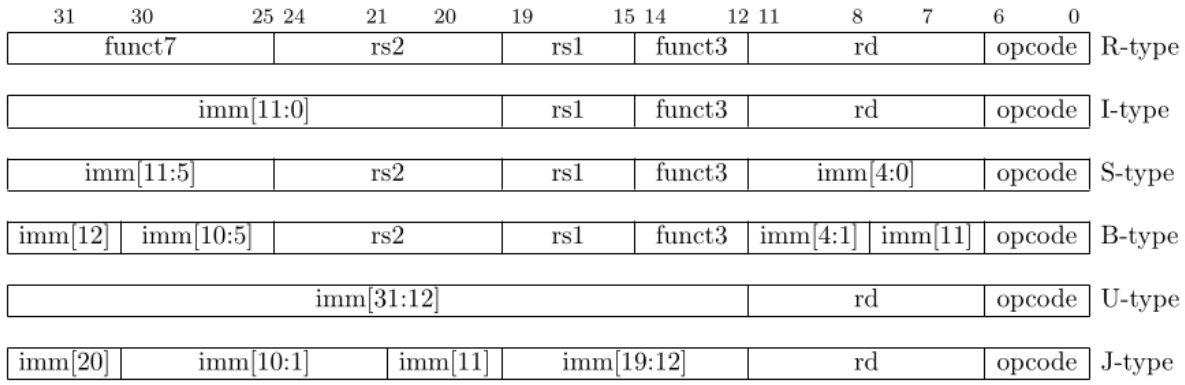
Перейдем к инструкциями, а именно их типам и кодировании. Так как мы рассматриваем только набор команд RV32I и стандартное расширение RV32M. Именно поэтому у нас каждая инструкции кодируется 32 битами. Выше было сказано что команды делятся на 2 группы, на самом деле их 4, к которым добавляется еще 2 типа, когда мы начинаем касаться кодирования констант.



Перед тем, как описывать каждый тип, нужно объяснить условные обозначения: opcode - код инструкции, funct3 и funct7 используются, чтобы

различить команды с одинаковым opcode. rd - номер регистра-приемника, rs1 - номер регистра источника, rs2 - номер регистра-операнда. imm - операнды-константы.

- R - register, “регистр, регистр, регистр”.
- I - immediate, “непосредственное значение, регистр, регистр”.
- S - store, “регистр, регистр, непосредственное значение”.
- U - uppper, “непосредственное значение, регистр”.



Добавились еще два типа B и J, как видно по рисунку, они различаются только кодирование imm.

Нужно сказать пару слов о соглашениях насчет регистров.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Соглашения следующие: x0(zero) - специальный регистр - он всегда хранит 0, x1(ra) - регистр для хранения адреса возврата из подпрограммы. x2(sp) - хранит ссылку на вершину стека, x3(gp) - хранит адрес области глобальных данных, регистры x5 - x7, x28 - x31 (t0 - t6) можно использовать свободно. x10 - x11(a0 - a1) - служат как аргументы функции и возвращаемые значения, x12 - x17 (a2 - a7) - используются только, как аргументы функции. x8(s0) - используется для хранения ссылки на область данных текущей подпрограммы. x8, x9, x18 - x27 (s0 - s11) - нужно восстановить в исходные значения, перед выходом из подпрограммы.

Описывать принцип и результат работы каждой инструкции не является целесообразным, так как задача работы не состоит в исполнении этих инструкций.

структуры файла ELF.

elf файл (executable b linkable format) - специальный файл, которые может использоваться для исполнения или сборки исполняемого файла. В данном файле нас интересуют лишь две секции: .symtab, .text. После ознакомления со спецификацией становится понятно, что в самом начале файла мы имеем некоторое количество байт, которые предоставляют нам информацию о всем остальном файле. Начнем с парсинга .symtab. Сначала нам нужно найти эту секцию, нужно распарсить заголовки секций, которые содержатся в таблице заголовков секций. Адрес на эту таблицу получаем из главного заголовка, который парсили в самом начале. Так мы найдем не только .symtab, но и .text. Конечно, для того чтобы найти, мы будем ориентироваться на имя, которое мы можем получить и таблицы строк файла, адрес на которую находиться в главном заголовке. Имя - это индекс в таблице строк. Сама таблица строк представляет собой скорее не таблицу, а список строк, которые разделены нуль-терминатором. Для того чтобы распарсить .symtab, нужно просто вывести все его элементы, которое являются структурами, которое содержат нужную нам информацию.

Figure 1-15. Symbol Table Entry

```
typedef struct {
        Elf32_Word    st_name;
        Elf32_Addr    st_value;
        Elf32_Word    st_size;
        unsigned char st_info;
        unsigned char st_other;
        Elf32_Half    st_shndx;
} Elf32_Sym;
```

name, value, size просто выводим. type, bind рассчитываются из st_info, vis из st_other. Здесь не обращаем внимание на #define, просто пользуемся следующими формулами:

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
```

Для vis используем следующее:

```
#define ELF32_ST_VISIBILITY(o) ((o)&0x3)
```

bind, type и vis ставятся в соответствие с следующими таблицами:

Figure 4-17: Symbol Binding

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

Figure 4-18: Symbol Types

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_HIPROC	15

Figure 4-19: Symbol Visibility

Name	Value
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3

Индекс получаем, как номер секции, к которой относится данный элемент, если элемент не относится ни к одной секции то выводим ABS, если имя секции пустое, то выводим UNDEF.

Имя самого элемента получаем таким же образом, как получали имя секции, только нужно обратиться к другой таблице строк, которая находится в секции .strtab.

Парсинг секции .text. Тут все проще, мы всего лишь берем по 4 байта и декодируем инструкцию, и так пока не пройдем всю секцию.

оаврфарвыраодОписание работы кода.

Описание работы кода не будет слишком длинным, так как почти все моменты были изложены ранее. Код написан на языке C++, и состоит из двух файлов: elf_data.h - это файл в котором находятся данные связанные с эльф файлом. main.cpp - основной код, которые производит парсинг и дизассемблирование. В начале производим проверки связанные с типом файла и ISA: проверяем, что данный нам файл elf-файл, что делается с помощью данных из главного заголовка, также проверяем, используя главный заголовок, что мы имеем 32 битный elf-файл и что ISA комманд из данного файла - это RISC-V. Дальше парсим таблицу заголовков секций и ищем три секции: .symtab, .text, .strtab. Далее происходит первичный парсинг .symtab, чтобы найти метки, а также вызываем функцию add_labels, которая добавляет недостающие метки(адреса, на которые происходят переходы). Дальше производим парсинг .text. Здесь просто вызываем функцию parse_command, которая просто определяет команду и выводит её. Парсинг происходит с использованием данной таблицы:

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Также не забываем выводить метки, когда попадаем на адрес, который связан с ними.

Наконец производим финальный парсинг .symtab, который уже выводит эту секцию. Значения каждого элемента получаем с помощью следующих функций, из названий которых понятно их предназначение: get_symtab_elem_bind, get_symtab_elem_type, get_symtab_elem_vis, get_symtab_elem_index. Забыл сказать, что весь наш файл читается в buffer из, которого мы достаем нужные данные.

Результат работы.

```
.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, 0x100
   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, 0x11
   100b0:    124f0513        addi      a0, t5, 292
   100b4:    65450513        addi      a0, a0, 1620
   100b8:    124f0f13        addi      t5, t5, 292
   100bc:    e4018293        addi      t0, gp, -448
   100c0:    fd018f93        addi      t6, gp, -48
   100c4:    02800e93        addi      t4, zero, 40
000100c8   <L2>:
   100c8:    fec50e13        addi      t3, a0, -20
   100cc:    000f0313        addi      t1, t5, 0
   100d0:    000f8893        addi      a7, t6, 0
   100d4:    00000813        addi      a6, zero, 0
000100d8   <L1>:
   100d8:    00088693        addi      a3, a7, 0
   100dc:    000e0793        addi      a5, t3, 0
   100e0:    00000613        addi      a2, zero, 0
000100e4   <L0>:
```

```
   100e4:    00078703          lb      a4, 0(a5)
   100e8:    00069583          lh      a1, 0(a3)
   100ec:    00178793        addi      a5, a5, 1
   100f0:    02868693        addi      a3, a3, 40
   100f4:    02b70733         mul      a4, a4, a1
   100f8:    00e60633         add      a2, a2, a4
   100fc:    fea794e3         bne      a5, a0, 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. https://en.wikipedia.org/wiki/Reduced_instruction_set_computer
2. https://en.wikipedia.org/wiki/Complex_instruction_set_computer
3. https://uneex.org/LecturesCMC/ArchitectureAssembler2022
4. https://refspecs.linuxfoundation.org/elf/elf.pdf
5. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf
6. https://refspecs.linuxbase.org/elf/gabi4+/ch4.symtab.html
7. https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-elf.adoc

Листинг.

elf_data.h

```
#ifndef DISASSEMBLER_ELF_DATA_H
#define DISASSEMBLER_ELF_DATA_H

#include <cstdint>
#include <fstream>
#include <algorithm>

typedef uint32_t Elf32_Addr;
typedef uint16_t Elf32_Half;
typedef uint32_t Elf32_Off;
typedef int32_t  Elf32_Sword;
typedef uint32_t Elf32_Word;

constexpr int EI_NIDENT = 16;
constexpr int ELF_HEADER_SIZE = 52;
constexpr int EI_MAG0 = 0;
constexpr int EI_MAG1 = 1;
constexpr int EI_MAG2 = 2;
constexpr int EI_MAG3 = 3;
constexpr int EI_CLASS = 4;
constexpr unsigned char EM_RISCV = 243;
constexpr unsigned char ELFCLASS32 = 2;

struct Elf32_Ehdr {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half      e_type;
    Elf32_Half      e_machine;
    Elf32_Word      e_version;
    Elf32_Addr      e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word      e_flags;
    Elf32_Half      e_ehsize;
    Elf32_Half      e_phentsize;
    Elf32_Half      e_phnum;
    Elf32_Half      e_shentsize;
    Elf32_Half      e_shnum;
    Elf32_Half      e_shstrndx;
};

struct Elf32_Shdr {
    Elf32_Word      sh_name;
    Elf32_Word      sh_type;
    Elf32_Word      sh_flags;
    Elf32_Addr      sh_addr;
    Elf32_Off sh_offset;
    Elf32_Word      sh_size;
    Elf32_Word      sh_link;
    Elf32_Word      sh_info;
    Elf32_Word      sh_addralign;
    Elf32_Word      sh_entsize;
};

struct Elf32_Sym{
    Elf32_Word      st_name;
    Elf32_Addr      st_value;
    Elf32_Word      st_size;
    unsigned char st_info;
```

```
    unsigned char st_other;
    Elf32_Half      st_shndx;
};

#endif //DISASSEMBLER_ELF_DATA_H
```

rv3.cpp

```
#include <filesystem>
#include <iostream>
#include <fstream>
#include <cstring>
#include <vector>
#include <cstdio>
#include <array>
#include <map>

#include "elf_data.h"

std::array<std::string, 32> reg_names = {
        "zero",
        "ra",
        "sp",
        "gp",
        "tp",
        "t0",
        "t1",
        "t2",
        "s0",
        "s1",
        "a0",
        "a1",
        "a2",
        "a3",
        "a4",
        "a5",
        "a6",
        "a7",
        "s2",
        "s3",
        "s4",
        "s5",
        "s6",
        "s7",
        "s8",
        "s9",
        "s10",
        "s11",
        "t3",
        "t4",
        "t5",
        "t6"
};

constexpr unsigned int rd = 0xF80;
constexpr unsigned int rs1 = 0xF8000;
constexpr unsigned int rs2 = 0x1F00000;

std::map<Elf32_Addr, std::string> labels;
unsigned int cnt = 0;
FILE* out;

std::string get_string(Elf32_Word index, char* buffer) {
```

```
56.    size_t i = 0;
57.    while (*(buffer + index + i) != '\0') {
58.        ++i;
59.    }
60.    return std::string(buffer + index, i);
61.}
62.
63.//parse symbtab element type
64.inline std::string get_symtab_elem_type(unsigned char info) {
65.    switch(info & 0xf) {
66.        case 0: return "NOTYPE";
67.        case 1: return "OBJECT";
68.        case 2: return "FUNC";
69.        case 3: return "SECTION";
70.        case 4: return "FILE";
71.        case 13: return "LOPROC";
72.        case 15: return "HIPROC";
73.    }
74.}
75.
76.//parse symbtab element bind
77.inline std::string get_symtab_elem_bind(unsigned char info) {
78.    switch(info >> 4) {
79.        case 0: return "LOCAL";
80.        case 1: return "GLOBAL";
81.        case 2: return "WEAK";
82.        case 13: return "LOPROC";
83.        case 15: return "HIPROC";
84.    }
85.}
86.
87.//parse symbtab element visibility
88.inline std::string get_symtab_elem_vis(unsigned char other) {
89.    switch(other & 0x3) {
90.        case 0: return "DEFAULT";
91.        case 1: return "INTERNAL";
92.        case 2: return "HIDDEN";
93.        case 3: return "PROTECTED";
94.    }
95.}
96.
97.//parse symbtab element index
98.inline std::string get_symtab_elem_indx(
99.        unsigned int shndx,
100.          std::vector <Elf32_Shdr*>& elf_sections,
101.          char* &buffer,
102.          unsigned int tab_start
103.      ) {
104.      if (shndx >= elf_sections.size()) {
105.          return "ABS";
106.      }
107.      if (get_string(tab_start + elf_sections[shndx] -> sh_name,
    buffer).empty()) {
108.          return "UNDEF";
109.      }
110.      return std::to_string(shndx);
111.  }
112.
113.  //get instruction opcode
114.  unsigned int get_opcode(Elf32_Word instruction) {
115.      return (instruction & 0x7f);
116.  }
117.
```

```
118.  //get instruction func3
119.  unsigned int get_func3(Elf32_Word instruction) {
120.     return (instruction & 0x3800) >> 12;
121.  }
122.
123.  //get instruction func7
124.  unsigned int get_funct7(Elf32_Word instruction) {
125.     return (instruction & 0x7F000000) >> 25;
126.  }
127.
128.  //get instruction part
129.  int get_part(Elf32_Word instruction, int mask, int offset) {
130.     return (static_cast<int>(instruction) & mask) >> offset;
131.  }
132.
133.  // Add extra labels to map labels
134.  void add_labels(Elf32_Addr addr, Elf32_Word instruction) {
135.     if (get_opcode(instruction) == 0b1101111) { // JAL
136.         Elf32_Addr tmp_addr = addr +
137.                               (get_part(instruction, 0x7FE00000, 21)
   << 1) +
138.                               (get_part(instruction, 0x100000, 20)
   << 11) +
139.                               (get_part(instruction, 0xFF000, 12) <<
   12) +
140.                               (get_part(instruction, 0x80000000, 31)
   << 20);
141.         if (labels.find(tmp_addr) == labels.end()) {
142.             labels.insert({tmp_addr, "L" + std::to_string(cnt)});
143.             cnt++;
144.         }
145.     }
146.
147.     else if (get_opcode(instruction) == 0b1100011 &&
   get_func3(instruction) == 0b000) { // BEQ
148.         Elf32_Addr tmp_addr = addr +
149.                               (get_part(instruction, 0xF00, 8) << 1)
   +
150.                               (get_part(instruction, 0x7E000000, 25)
   << 5) +
151.                               (get_part(instruction, 0x80, 7) << 11)
   +
152.                               (get_part(instruction, 0x80000000, 31)
   << 12);
153.         if (labels.find(tmp_addr) == labels.end()) {
154.             labels.insert({tmp_addr, "L" + std::to_string(cnt)});
155.             cnt++;
156.         }
157.     } else if (get_opcode(instruction) == 0b1100011 &&
   get_func3(instruction) == 0b001) { // BNE
158.         Elf32_Addr tmp_addr = addr +
159.                               (get_part(instruction, 0xF00, 8) << 1)
   +
160.                               (get_part(instruction, 0x7E000000, 25)
   << 5) +
161.                               (get_part(instruction, 0x80, 7) << 11)
   +
162.                               (get_part(instruction, 0x80000000, 31)
   << 12);
163.         if (labels.find(tmp_addr) == labels.end()) {
164.             labels.insert({tmp_addr, "L" + std::to_string(cnt)});
165.             cnt++;
166.         }
```

```
    } else if (get_opcode(instruction) == 0b1100011 &&
get_func3(instruction) == 0b100) { // BLT
        Elf32_Addr tmp_addr = addr +
                              (get_part(instruction, 0xF00, 8) << 1)
+
                              (get_part(instruction, 0x7E000000, 25)
<< 5) +
                              (get_part(instruction, 0x80, 7) << 11)
+
                              (get_part(instruction, 0x80000000, 31)
<< 12);
        if (labels.find(tmp_addr) == labels.end()) {
            labels.insert({tmp_addr, "L" + std::to_string(cnt)});
            cnt++;
        }
    } else if (get_opcode(instruction) == 0b1100011 &&
get_func3(instruction) == 0b101) {
        Elf32_Addr tmp_addr = addr +
                              (get_part(instruction, 0xF00, 8) << 1)
+
                              (get_part(instruction, 0x7E000000, 25)
<< 5) +
                              (get_part(instruction, 0x80, 7) << 11)
+
                              (get_part(instruction, 0x80000000, 31)
<< 12);
        if (labels.find(tmp_addr) == labels.end()) {
            labels.insert({tmp_addr, "L" + std::to_string(cnt)});
            cnt++;
        }
    } else if (get_opcode(instruction) == 0b1100011 &&
get_func3(instruction) == 0b110) { // BLTU
        Elf32_Addr tmp_addr = addr +
                              (get_part(instruction, 0xF00, 8) << 1)
+
                              (get_part(instruction, 0x7E000000, 25)
<< 5) +
                              (get_part(instruction, 0x80, 7) << 11)
+
                              (get_part(instruction, 0x80000000, 31)
<< 12);
        if (labels.find(tmp_addr) == labels.end()) {
            labels.insert({tmp_addr, "L" + std::to_string(cnt)});
            cnt++;
        }
    } else if (get_opcode(instruction) == 0b1100011 &&
get_func3(instruction) == 0b111) { // BGEU
        Elf32_Addr tmp_addr = addr +
                              (get_part(instruction, 0xF00, 8) << 1)
+
                              (get_part(instruction, 0x7E000000, 25)
<< 5) +
                              (get_part(instruction, 0x80, 7) << 11)
+
                              (get_part(instruction, 0x80000000, 31)
<< 12);
        if (labels.find(tmp_addr) == labels.end()) {
            labels.insert({tmp_addr, "L" + std::to_string(cnt)});
            cnt++;
        }
    }
}

```

```
210.   //parse command
211.   void parse_command(Elf32_Addr addr, Elf32_Word instruction) {
212.      if (get_opcode(instruction) == 0b0110111) { // LUI
213.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, 0x%x\n",
214.                 addr,
215.                 instruction,
216.                 "lui",
217.                 reg_names[get_part(instruction, rd, 7)].c_str(),
218.                 get_part(instruction, 0x7FFFF800, 12)
219.          );
220.      }
221.
222.      else if (get_opcode(instruction) == 0b0010111) { // AUIPC
223.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, 0x%x\n",
224.                 addr,
225.                 instruction,
226.                 "auipc",
227.                 reg_names[get_part(instruction, rd, 7)].c_str(),
228.                 get_part(instruction, 0x7FFFF800, 12)
229.          );
230.      }
231.
232.      else if (get_opcode(instruction) == 0b1100111 &&
    get_func3(instruction) == 0b000) { // JALR
233.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
234.                 addr,
235.                 instruction,
236.                 "jalr",
237.                 reg_names[get_part(instruction, rd, 7)].c_str(),
238.                 get_part(instruction, 0xFFF00000, 20),
239.                 reg_names[get_part(instruction, rs1, 15)].c_str()
240.          );
241.      }
242.
243.      else if (get_opcode(instruction) == 0b1101111) { // JAL
244.          Elf32_Addr label_addr = addr +
245.                                  (get_part(instruction, 0x7FE00000,
    21) << 1) +
246.                                  (get_part(instruction, 0x100000, 20)
    << 11) +
247.                                  (get_part(instruction, 0xFF000, 12)
    << 12) +
248.                                  (get_part(instruction, 0x80000000,
    31) << 20);
249.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, 0x%x <%s>\n",
250.                 addr,
251.                 instruction,
252.                 "jal",
253.                 reg_names[get_part(instruction, rd, 7)].c_str(),
254.                 label_addr,
255.                 labels[label_addr].c_str()
256.          );
257.      }
258.
259.      else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b000) { // BEQ
260.          Elf32_Addr label_addr = addr +
261.                                  (get_part(instruction, 0xF00, 8) <<
    1) +
262.                                  (get_part(instruction, 0x7E000000,
    25) << 5) +
263.                                  (get_part(instruction, 0x80, 7) <<
    11) +
```

```
264.                                    (get_part(instruction, 0x80000000,
    31) << 12);
265.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
266.                 addr,
267.                 instruction,
268.                 "beq",
269.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
270.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
271.                 label_addr,
272.                 labels[label_addr].c_str()
273.          );
274.      } else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b001) { // BNE
275.          Elf32_Addr label_addr = addr +
276.                                    (get_part(instruction, 0xF00, 8) <<
    1) +
277.                                    (get_part(instruction, 0x7E000000,
    25) << 5) +
278.                                    (get_part(instruction, 0x80, 7) <<
    11) +
279.                                    (get_part(instruction, 0x80000000,
    31) << 12);
280.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
281.                 addr,
282.                 instruction,
283.                 "bne",
284.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
285.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
286.                 label_addr,
287.                 labels[label_addr].c_str()
288.          );
289.      } else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b100) { // BLT
290.          Elf32_Addr label_addr = addr +
291.                                    (get_part(instruction, 0xF00, 8) <<
    1) +
292.                                    (get_part(instruction, 0x7E000000,
    25) << 5) +
293.                                    (get_part(instruction, 0x80, 7) <<
    11) +
294.                                    (get_part(instruction, 0x80000000,
    31) << 12);
295.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
296.                 addr,
297.                 instruction,
298.                 "blt",
299.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
300.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
301.                 label_addr,
302.                 labels[label_addr].c_str()
303.          );
304.      } else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b101) { // BGE
305.          Elf32_Addr label_addr = addr +
306.                                    (get_part(instruction, 0xF00, 8) <<
    1) +
307.                                    (get_part(instruction, 0x7E000000,
    25) << 5) +
308.                                    (get_part(instruction, 0x80, 7) <<
    11) +
309.                                    (get_part(instruction, 0x80000000,
    31) << 12);
310.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
```

```
311.                 addr,
312.                 instruction,
313.                 "bge",
314.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
315.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
316.                 label_addr,
317.                 labels[label_addr].c_str()
318.         );
319.     } else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b110) { // BLTU
320.         Elf32_Addr label_addr = addr +
321.                                 (get_part(instruction, 0xF00, 8) <<
    1) +
322.                                 (get_part(instruction, 0x7E000000,
    25) << 5) +
323.                                 (get_part(instruction, 0x80, 7) <<
    11) +
324.                                 (get_part(instruction, 0x80000000,
    31) << 12);
325.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
326.                 addr,
327.                 instruction,
328.                 "bltu",
329.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
330.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
331.                 label_addr,
332.                 labels[label_addr].c_str()
333.         );
334.     } else if (get_opcode(instruction) == 0b1100011 &&
    get_func3(instruction) == 0b111) { // BGEU
335.         Elf32_Addr label_addr = addr +
336.                                 (get_part(instruction, 0xF00, 8) <<
    1) +
337.                                 (get_part(instruction, 0x7E000000,
    25) << 5) +
338.                                 (get_part(instruction, 0x80, 7) <<
    11) +
339.                                 (get_part(instruction, 0x80000000,
    31) << 12);
340.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, 0x%x <%s>\n",
341.                 addr,
342.                 instruction,
343.                 "bgeu",
344.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
345.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
346.                 label_addr,
347.                 labels[label_addr].c_str()
348.         );
349.     }
350.
351.     else if (get_opcode(instruction) == 0b0000011 &&
    get_func3(instruction) == 0b000) { // LB
352.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
353.                 addr,
354.                 instruction,
355.                 "lb",
356.                 reg_names[get_part(instruction, rd, 7)].c_str(),
357.                 get_part(instruction, 0xFFF00000, 20),
358.                 reg_names[get_part(instruction, rs1, 15)].c_str()
359.         );
360.     } else if (get_opcode(instruction) == 0b0000011 &&
    get_func3(instruction) == 0b001) { // LH
361.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
```

```
362.                addr,
363.                instruction,
364.                "lh",
365.                reg_names[get_part(instruction, rd, 7)].c_str(),
366.                get_part(instruction, 0xFFF00000, 20),
367.                reg_names[get_part(instruction, rs1, 15)].c_str()
368.          );
369.      } else if (get_opcode(instruction) == 0b0000011 &&
   get_func3(instruction) == 0b010) { // LW
370.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
371.                addr,
372.                instruction,
373.                "lw",
374.                reg_names[get_part(instruction, rd, 7)].c_str(),
375.                get_part(instruction, 0xFFF00000, 20),
376.                reg_names[get_part(instruction, rs1, 15)].c_str()
377.          );
378.      } else if (get_opcode(instruction) == 0b0000011 &&
   get_func3(instruction) == 0b100) { // LBU
379.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
380.                addr,
381.                instruction,
382.                "lbu",
383.                reg_names[get_part(instruction, rd, 7)].c_str(),
384.                get_part(instruction, 0xFFF00000, 20),
385.                reg_names[get_part(instruction, rs1, 15)].c_str()
386.          );
387.      } else if (get_opcode(instruction) == 0b0000011 &&
   get_func3(instruction) == 0b101) { // LHU
388.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
389.                addr,
390.                instruction,
391.                "lhu",
392.                reg_names[get_part(instruction, rd, 7)].c_str(),
393.                get_part(instruction, 0xFFF00000, 20),
394.                reg_names[get_part(instruction, rs1, 15)].c_str()
395.          );
396.      }
397.
398.      else if (get_opcode(instruction) == 0b0100011 &&
   get_func3(instruction) == 0b000) { // SB
399.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
400.                 addr,
401.                 instruction,
402.                 "sb",
403.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
404.                 (get_part(instruction, 0xFE000000, 25) << 5) +
   get_part(instruction, 0xF80, 7),
405.                 reg_names[get_part(instruction, rs1, 15)].c_str()
406.          );
407.      } else if (get_opcode(instruction) == 0b0100011 &&
   get_func3(instruction) == 0b001) { // SH
408.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
409.                 addr,
410.                 instruction,
411.                 "sh",
412.                 reg_names[get_part(instruction, rs2, 20)].c_str(),
413.                 (get_part(instruction, 0xFE000000, 25) << 5) +
   get_part(instruction, 0xF80, 7),
414.                 reg_names[get_part(instruction, rs1, 15)].c_str()
415.          );
416.      } else if (get_opcode(instruction) == 0b0100011 &&
   get_func3(instruction) == 0b010) { // SW
```

```
417.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %i(%s)\n",
418.                 addr,
419.                 instruction,
420.                 "sw",
421.                  reg_names[get_part(instruction, rs2, 20)].c_str(),
422.                 (get_part(instruction, 0xFE000000, 25) << 5) +
   get_part(instruction, 0xF80, 7),
423.                  reg_names[get_part(instruction, rs1, 15)].c_str()
424.          );
425.      }
426.
427.      else if ( // SLLI
428.              get_opcode(instruction) == 0b0010011 &&
429.              get_func3(instruction) == 0b001 &&
430.              get_funct7(instruction) == 0b0000000
431.      ) {
432.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
433.                 addr,
434.                 instruction,
435.                 "slli",
436.                 reg_names[get_part(instruction, rd, 7)].c_str(),
437.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
438.                 get_part(instruction, rs2, 20)
439.          );
440.      } else if ( // SRLI
441.              get_opcode(instruction) == 0b0010011 &&
442.              get_func3(instruction) == 0b101 &&
443.              get_funct7(instruction) == 0b0000000
444.      ) {
445.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
446.                 addr,
447.                 instruction,
448.                 "srli",
449.                 reg_names[get_part(instruction, rd, 7)].c_str(),
450.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
451.                 get_part(instruction, rs2, 20)
452.          );
453.      } else if ( // SRAI
454.              get_opcode(instruction) == 0b0010011 &&
455.              get_func3(instruction) == 0b101 &&
456.              get_funct7(instruction) == 0b0100000
457.      ) {
458.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
459.                 addr,
460.                 instruction,
461.                 "slli",
462.                 reg_names[get_part(instruction, rd, 7)].c_str(),
463.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
464.                 get_part(instruction, rs2, 20)
465.          );
466.      } else if ( // ADD
467.              get_opcode(instruction) == 0b0110011 &&
468.              get_func3(instruction) == 0b000 &&
469.              get_funct7(instruction) == 0b0000000
470.      ) {
471.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
472.                 addr,
473.                 instruction,
474.                 "add",
475.                 reg_names[get_part(instruction, rd, 7)].c_str(),
476.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
477.                 reg_names[get_part(instruction, rs2, 20)].c_str()
478.          );
```

```
479.      } else if ( // SUB
480.              get_opcode(instruction) == 0b0110011 &&
481.              get_func3(instruction) == 0b000 &&
482.              get_funct7(instruction) == 0b0100000
483.      ) {
484.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
485.                  addr,
486.                  instruction,
487.                  "sub",
488.                  reg_names[get_part(instruction, rd, 7)].c_str(),
489.                  reg_names[get_part(instruction, rs1, 15)].c_str(),
490.                  reg_names[get_part(instruction, rs2, 20)].c_str()
491.          );
492.      } else if ( // SLL
493.              get_opcode(instruction) == 0b0110011 &&
494.              get_func3(instruction) == 0b001 &&
495.              get_funct7(instruction) == 0b0000000
496.      ) {
497.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
498.                  addr,
499.                  instruction,
500.                  "sll",
501.                  reg_names[get_part(instruction, rd, 7)].c_str(),
502.                  reg_names[get_part(instruction, rs1, 15)].c_str(),
503.                  reg_names[get_part(instruction, rs2, 20)].c_str()
504.          );
505.      } else if ( // SLT
506.              get_opcode(instruction) == 0b0110011 &&
507.              get_func3(instruction) == 0b010 &&
508.              get_funct7(instruction) == 0b0000000
509.      ) {
510.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
511.                  addr,
512.                  instruction,
513.                  "slt",
514.                  reg_names[get_part(instruction, rd, 7)].c_str(),
515.                  reg_names[get_part(instruction, rs1, 15)].c_str(),
516.                  reg_names[get_part(instruction, rs2, 20)].c_str()
517.          );
518.      } else if ( // SLTU
519.              get_opcode(instruction) == 0b0110011 &&
520.              get_func3(instruction) == 0b011 &&
521.              get_funct7(instruction) == 0b0000000
522.      ) {
523.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
524.                  addr,
525.                  instruction,
526.                  "sltu",
527.                  reg_names[get_part(instruction, rd, 7)].c_str(),
528.                  reg_names[get_part(instruction, rs1, 15)].c_str(),
529.                  reg_names[get_part(instruction, rs2, 20)].c_str()
530.          );
531.      } else if ( // XOR
532.              get_opcode(instruction) == 0b0110011 &&
533.              get_func3(instruction) == 0b100 &&
534.              get_funct7(instruction) == 0b0000000
535.      ) {
536.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
537.                  addr,
538.                  instruction,
539.                  "xor",
540.                  reg_names[get_part(instruction, rd, 7)].c_str(),
541.                  reg_names[get_part(instruction, rs1, 15)].c_str(),
```

```
542.                 reg_names[get_part(instruction, rs2, 20)].c_str()
543.             );
544.         } else if ( // SRL
545.                 get_opcode(instruction) == 0b0110011 &&
546.                 get_func3(instruction) == 0b101 &&
547.                 get_funct7(instruction) == 0b0000000
548.         ) {
549.             fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
550.                     addr,
551.                     instruction,
552.                     "srl",
553.                     reg_names[get_part(instruction, rd, 7)].c_str(),
554.                     reg_names[get_part(instruction, rs1, 15)].c_str(),
555.                     reg_names[get_part(instruction, rs2, 20)].c_str()
556.             );
557.         } else if ( // SRA
558.                 get_opcode(instruction) == 0b0110011 &&
559.                 get_func3(instruction) == 0b101 &&
560.                 get_funct7(instruction) == 0b0100000
561.         ) {
562.             fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
563.                     addr,
564.                     instruction,
565.                     "sra",
566.                     reg_names[get_part(instruction, rd, 7)].c_str(),
567.                     reg_names[get_part(instruction, rs1, 15)].c_str(),
568.                     reg_names[get_part(instruction, rs2, 20)].c_str()
569.             );
570.         } else if ( // OR
571.                 get_opcode(instruction) == 0b0110011 &&
572.                 get_func3(instruction) == 0b110 &&
573.                 get_funct7(instruction) == 0b0000000
574.         ) {
575.             fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
576.                     addr,
577.                     instruction,
578.                     "or",
579.                     reg_names[get_part(instruction, rd, 7)].c_str(),
580.                     reg_names[get_part(instruction, rs1, 15)].c_str(),
581.                     reg_names[get_part(instruction, rs2, 20)].c_str()
582.             );
583.         } else if ( // AND
584.                 get_opcode(instruction) == 0b0110011 &&
585.                 get_func3(instruction) == 0b111 &&
586.                 get_funct7(instruction) == 0b0000000
587.         ) {
588.             fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
589.                     addr,
590.                     instruction,
591.                     "and",
592.                     reg_names[get_part(instruction, rd, 7)].c_str(),
593.                     reg_names[get_part(instruction, rs1, 15)].c_str(),
594.                     reg_names[get_part(instruction, rs2, 20)].c_str()
595.             );
596.         } else if ( // MUL
597.                 get_opcode(instruction) == 0b0110011 &&
598.                 get_func3(instruction) == 0b000 &&
599.                 get_funct7(instruction) == 0b0000001
600.         ) {
601.             fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
602.                     addr,
603.                     instruction,
604.                     "mul",
```

```
                reg_names[get_part(instruction, rd, 7)].c_str(),
                reg_names[get_part(instruction, rs1, 15)].c_str(),
                reg_names[get_part(instruction, rs2, 20)].c_str()
        );
    } else if ( // MULH
            get_opcode(instruction) == 0b0110011 &&
            get_func3(instruction) == 0b001 &&
            get_funct7(instruction) == 0b0000001
    ) {
        fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                addr,
                instruction,
                "mulh",
                reg_names[get_part(instruction, rd, 7)].c_str(),
                reg_names[get_part(instruction, rs1, 15)].c_str(),
                reg_names[get_part(instruction, rs2, 20)].c_str()
        );
    } else if ( // MULHSU
            get_opcode(instruction) == 0b0110011 &&
            get_func3(instruction) == 0b010 &&
            get_funct7(instruction) == 0b0000001
    ) {
        fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                addr,
                instruction,
                "mulhsu",
                reg_names[get_part(instruction, rd, 7)].c_str(),
                reg_names[get_part(instruction, rs1, 15)].c_str(),
                reg_names[get_part(instruction, rs2, 20)].c_str()
        );
    } else if ( // MULHU
            get_opcode(instruction) == 0b0110011 &&
            get_func3(instruction) == 0b011 &&
            get_funct7(instruction) == 0b0000001
    ) {
        fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                addr,
                instruction,
                "mulhu",
                reg_names[get_part(instruction, rd, 7)].c_str(),
                reg_names[get_part(instruction, rs1, 15)].c_str(),
                reg_names[get_part(instruction, rs2, 20)].c_str()
        );
    } else if ( // DIV
            get_opcode(instruction) == 0b0110011 &&
            get_func3(instruction) == 0b100 &&
            get_funct7(instruction) == 0b0000001
    ) {
        fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                addr,
                instruction,
                "div",
                reg_names[get_part(instruction, rd, 7)].c_str(),
                reg_names[get_part(instruction, rs1, 15)].c_str(),
                reg_names[get_part(instruction, rs2, 20)].c_str()
        );
    } else if ( // DIVU
            get_opcode(instruction) == 0b0110011 &&
            get_func3(instruction) == 0b101 &&
            get_funct7(instruction) == 0b0000001
    ) {
        fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                addr,
```

```
668.                 instruction,
669.                 "divu",
670.                 reg_names[get_part(instruction, rd, 7)].c_str(),
671.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
672.                 reg_names[get_part(instruction, rs2, 20)].c_str()
673.         );
674.     } else if ( // REM
675.             get_opcode(instruction) == 0b0110011 &&
676.             get_func3(instruction) == 0b110 &&
677.             get_funct7(instruction) == 0b0000001
678.     ) {
679.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
680.                 addr,
681.                 instruction,
682.                 "rem",
683.                 reg_names[get_part(instruction, rd, 7)].c_str(),
684.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
685.                 reg_names[get_part(instruction, rs2, 20)].c_str()
686.         );
687.     } else if ( // REMU
688.             get_opcode(instruction) == 0b0110011 &&
689.             get_func3(instruction) == 0b111 &&
690.             get_funct7(instruction) == 0b0000001
691.     ) {
692.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
693.                 addr,
694.                 instruction,
695.                 "remu",
696.                 reg_names[get_part(instruction, rd, 7)].c_str(),
697.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
698.                 reg_names[get_part(instruction, rs2, 20)].c_str()
699.         );
700.     }
701.
702.     else if (get_opcode(instruction) == 0b0010011 &&
    get_func3(instruction) == 0b000) { // ADDI
703.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
704.                 addr,
705.                 instruction,
706.                 "addi",
707.                 reg_names[get_part(instruction, rd, 7)].c_str(),
708.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
709.                 get_part(instruction, 0xFFF00000, 20)
710.         );
711.     } else if (get_opcode(instruction) == 0b0010011 &&
    get_func3(instruction) == 0b010) { // SLTI
712.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
713.                 addr,
714.                 instruction,
715.                 "slti",
716.                 reg_names[get_part(instruction, rd, 7)].c_str(),
717.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
718.                 get_part(instruction, 0xFFF00000, 20)
719.         );
720.     } else if (get_opcode(instruction) == 0b0010011 &&
    get_func3(instruction) == 0b011) { // SLTIU
721.         fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
722.                 addr,
723.                 instruction,
724.                 "sltiu",
725.                 reg_names[get_part(instruction, rd, 7)].c_str(),
726.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
727.                 get_part(instruction, 0xFFF00000, 20)
```

```
728.          );
729.      } else if (get_opcode(instruction) == 0b0010011 &&
   get_func3(instruction) == 0b100) { // XORI
730.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
731.                 addr,
732.                 instruction,
733.                 "xori",
734.                 reg_names[get_part(instruction, rd, 7)].c_str(),
735.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
736.                 get_part(instruction, 0xFFF00000, 20)
737.          );
738.      } else if (get_opcode(instruction) == 0b0010011 &&
   get_func3(instruction) == 0b110) { // ORI
739.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
740.                 addr,
741.                 instruction,
742.                 "ori",
743.                 reg_names[get_part(instruction, rd, 7)].c_str(),
744.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
745.                 get_part(instruction, 0xFFF00000, 20)
746.          );
747.      } else if (get_opcode(instruction) == 0b0010011 &&
   get_func3(instruction) == 0b111) { // ANDI
748.          fprintf(out,"   %05x:\t%08x\t%7s\t%s, %s, %i\n",
749.                 addr,
750.                 instruction,
751.                 "andi",
752.                 reg_names[get_part(instruction, rd, 7)].c_str(),
753.                 reg_names[get_part(instruction, rs1, 15)].c_str(),
754.                 get_part(instruction, 0xFFF00000, 20)
755.          );
756.      }
757.
758.      else if (instruction == 0b00000000000000000000000001110011) { //
   ECALL
759.          fprintf(out,"   %05x:\t%08x\t%7s\n", addr, instruction,
   "ecall");
760.      } else if (instruction == 0b00000000000100000000000001110011) {
   // EBREAK
761.          fprintf(out,"   %05x:\t%08x\t%7s\n", addr, instruction,
   "ebreak");
762.      } else { // unknown_operation
763.          fprintf(out,"   %05x:\t%08x\t%7s\n", addr, instruction,
   "unknown_instruction");
764.      }
765.
766.   }
767.
768.   int main(int argc, char *argv[]) {
769.      std::ifstream in;
770.      std::filesystem::path input_file_name, output_file_name;
771.
772.      if (argc == 3) {
773.          input_file_name = argv[1];
774.          output_file_name = argv[2];
775.      } else {
776.          std::cerr << "Expected 2 arguments, found " << argc - 1 <<
   "." << std::endl;
777.          return 0;
778.      }
779.
780.      try {
781.          in = std::ifstream(input_file_name, std::ios::in |
```

```
    std::ios::binary); // construct byte stream
782.          in.exceptions(std::ifstream::failbit);
783.
784.          uintmax_t size_of_file =
    std::filesystem::file_size(input_file_name);
785.          char* buffer = new char[size_of_file];
786.          in.read(buffer, static_cast<long>(size_of_file)); // read
    file
787.
788.
789.          auto elf_header = reinterpret_cast<Elf32_Ehdr*>(buffer); //
    parse elf header
790.
791.          // check that we have elf
792.          if (
793.                  elf_header -> e_ident[EI_MAG0] != 0x7f ||
794.                  elf_header -> e_ident[EI_MAG1] != 'E' ||
795.                  elf_header -> e_ident[EI_MAG2] != 'L' ||
796.                  elf_header -> e_ident[EI_MAG3] != 'F'
797.          ) {
798.              std::cout << "Provided for input file is not a elf
    file." << std::endl;
799.              return 0;
800.          }
801.
802.          if (elf_header -> e_ident[EI_CLASS] == ELFCLASS32) {
803.              std::cout << "Provided file is not a elf 32 bit file."
    << std::endl;
804.              return 0;
805.          }
806.
807.          // check that we have RISC-V
808.          if (elf_header -> e_machine != EM_RISCV) {
809.              std::cout << "Provided file does not support RISC-V." <<
    std::endl;
810.              return 0;
811.          }
812.
813.          // parse section header table
814.          std::vector<Elf32_Shdr*> elf_section_header_table(elf_header
    -> e_shnum);
815.          for (size_t i = 0; i < elf_header -> e_shnum; ++i) {
816.              elf_section_header_table[i] =
817.                      reinterpret_cast<Elf32_Shdr*>(buffer +
    elf_header -> e_shoff + elf_header -> e_shentsize * i);
818.          }
819.
820.          // find .symtab and .text section
821.          Elf32_Shdr* symtab = nullptr;
822.          Elf32_Shdr* text = nullptr;
823.          Elf32_Shdr* strtab = nullptr;
824.          unsigned int string_table_entry =
    elf_section_header_table[elf_header -> e_shstrndx] -> sh_offset;
825.          for (Elf32_Shdr* &i : elf_section_header_table) {
826.              if (get_string(string_table_entry + i -> sh_name,
    buffer) == ".symtab") {
827.                  symtab = i;
828.              }
829.              if (get_string(string_table_entry + i -> sh_name,
    buffer) == ".text") {
830.                  text = i;
831.              }
832.              if (get_string(string_table_entry + i -> sh_name,
```

```
    buffer) == ".strtab") {
833.                 strtab = i;
834.             }
835.         }
836.
837.         //parse .symtab
838.         std::vector <Elf32_Sym*> elf_symtab(symtab -> sh_size /
    sizeof(Elf32_Sym));
839.         for (size_t i = 0; i < elf_symtab.size(); ++i) {
840.             elf_symtab[i] = reinterpret_cast<Elf32_Sym*>(buffer +
    symtab -> sh_offset + sizeof(Elf32_Sym) * i);
841.         }
842.         for (Elf32_Sym* &i : elf_symtab) {
843.             if (!get_string(strtab->sh_offset + i->st_name,
    buffer).empty()) {
844.                 labels.insert({i->st_value,
    get_string(strtab->sh_offset + i->st_name, buffer)});
845.             }
846.         }
847.
848.         // parse .text
849.         if (text -> sh_size % 4 != 0) {
850.             std::cout << "Expected file that contain 4 byte RISC-V
    commands." << std::endl;
851.             return 0;
852.         }
853.
854.         out = fopen(argv[2], "w");
855.         fprintf(out,".text\n");
856.
857.         for (size_t i = 0; i < text -> sh_size; i += 4) {
858.             add_labels(text -> sh_addr + i,
859.                        *reinterpret_cast<Elf32_Word*>(buffer + text
    -> sh_offset + i)
860.             );
861.         }
862.
863.         for (size_t i = 0; i < text -> sh_size; i += 4) {
864.             if (labels.find(text -> sh_addr + i) != labels.end()) {
865.                 fprintf(out,"%08x   <%s>:\n",
866.                        text -> sh_addr + i,
867.                        labels[text -> sh_addr + i].c_str()
868.                 );
869.             }
870.             parse_command(text -> sh_addr + i,
    *reinterpret_cast<Elf32_Word*>(buffer + text -> sh_offset + i));
871.         }
872.
873.
874.
875.         // printing .symtab
876.         fprintf(out,"\n.symtab\n");
877.         fprintf(out,"Symbol Value              Size Type     Bind
    Vis       Index Name\n");
878.         int num = 0;
879.         for (Elf32_Sym* &i : elf_symtab) {
880.             fprintf(out,
881.                     "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n",
882.                     num,
883.                     i -> st_value,
884.                     i -> st_size,
885.                     get_symtab_elem_type(i -> st_info).c_str(),
886.                     get_symtab_elem_bind(i -> st_info).c_str(),
```

```
887.                     get_symtab_elem_vis(i -> st_other).c_str(),
888.                     get_symtab_elem_indx(
889.                             i -> st_shndx,
890.                             elf_section_header_table,
891.                             buffer,
892.                             string_table_entry
893.                             ).c_str(),
894.                     get_string(strtab -> sh_offset + i -> st_name,
   buffer).c_str()
895.                     );
896.                 ++num;
897.             }
898.             delete[]buffer;
899.         } catch (const std::ios_base::failure& e) {
900.             std::cout << e.what() << std::endl;
901.         }
902.         return 0;
903.     }
904.
```