Лабораторная Работа №3	M3100	2022
ISA	Ерёмин Владимир Ильич	

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

Инструментарий и требования к работе: C++20, MSVC *(19.34.31937.0, x64)* и CMake *(3.24.202208181-MSVC_2)*.

Описание: изучить систему кодирования команд RISC-V и структуру Elf файла. Решить предложенную задачу. Оформить отчет.

Задача: необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера.

Система кодирования команд RISC-V

Основная информация.

Систему кодирования команд описывает так называемая **ISA** (Instruction set architecture). В общем ISA задает поддерживаемые инструкции, типы данных, регистры. Описывает управление памятью, а также фундаментальный функционал. А также модель ввода/вывода..RISC-V — это, семейство связанных между собой ISA. На данных момент оно состоит из 4-основных ISA. Каждая из которых задает ширину регистров, размер адресного пространства и количество регистров, соответственно. Из них мы будем рассматривать **RV32I**.

RV32I имеет: 32-битное адресное пространство, 32-битные инструкции и 32-битные целочисленные регистры.

RISC-V была спроектирована с расчетом на расширяемость и специфику конкретного окружения. Так, каждая из базовых ISA может быть расширена с одним или более *“расширением набора инструкций”*. Причем, стандарт задает и специфику конкретной инструкции, это сделано для того, чтобы основные инструкции не были перекрыты расширениями. Мы рассматриваем стандартную **ISA - I**, и расширение **M -** для поддержки целочисленного деления и умножения.

Система также задает и модель памяти: в данном случае **RV32I** имеет зацикленное байт-адресное адресное пространство размером 2^{32} байт. Также задаются стандартные типы данных *(эти типы встретятся еще при описании elf-файла):* word (4 байта), halfword (2 байта), doubleword (8 байт) и quadword (16 байт).

Хоть мы требуем здесь 32-битных инструкции, в действительности размер инструкций в разных ISA может отличаться. Это характеризуется префиксом соответствующей команды. Сейчас, официально задокументированы только 16 и 32-битные инструкции.

Основные ISA RISC-V могут иметь как little-endian, так и big-endian модели памяти. Сами инструкции записаны последовательно так, чтобы при последовательном считывании по 32-х бит - команда не оказалась считанной частично.

Регистры.

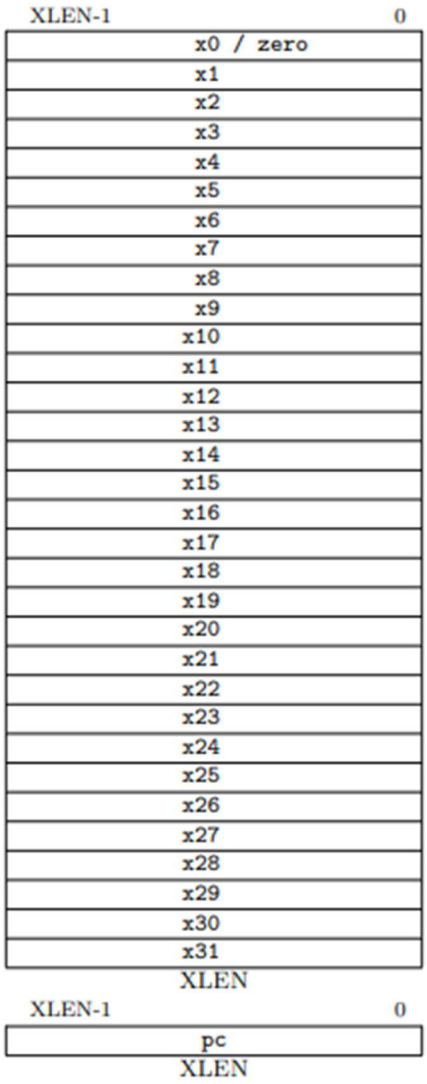


Рисунок 1 - Базовое состояние регистров.

Продолжая описывать ISA, мы переходим к регистрам. Аналогично предыдущим параметрам, количество регистров у RV32I также 32. В нулевом регистре **(zero)** все биты установлены в 0. Дополнительно имеется дополнительный регистр **pc** - содержащий адрес исполняемой инструкции.

Формат инструкций.

Все инструкции описываются через 6 возможных типов. Основными считаются 4 (R/I/S/U), с дополнительными 2 (B/J) - они незначительно отличаются, поэтому названия иногда даже совмещают (SB/UJ).

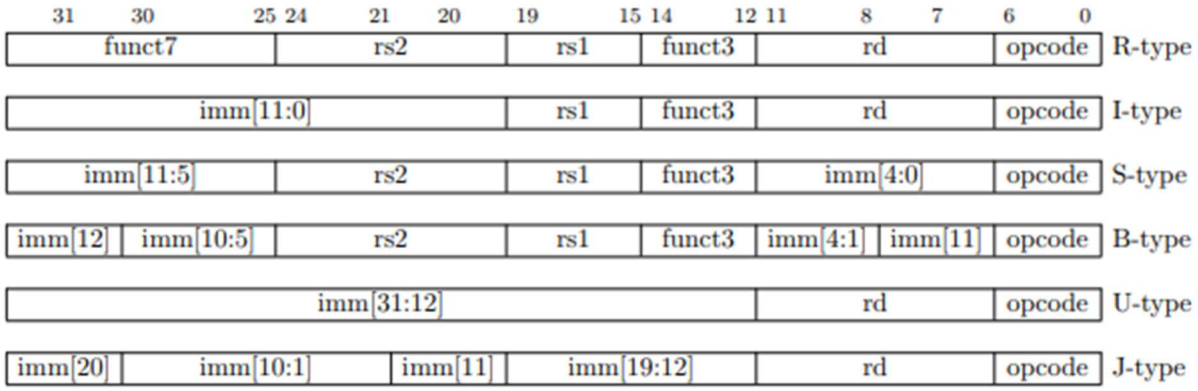


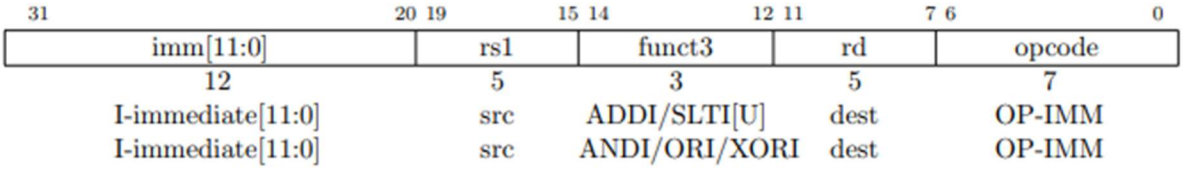
Рисунок 2 - Виды инструкций RISC-V ISA.

Можно заметить, что первые 7 бит *(opcode)* статичны независимо от вида инструкции. Верно! Именно по ним и можно различать вид, чтобы затем более детально ее разобрать. Также стоит заметить, что соответствующие поля *(rs1/rs2/rd) и (funct3/funct7)* остаются на одном месте независимо от вида.

Кстати, эти поля отвечают за регистры источника **(RS1/2 - Register Source 1/2)** и регистр назначения **(RD - Register Destination)**. А "*funct3/7*" отвечают за идентификацию инструкции.

Imm часто остается разбитым на кусочки, но в действительности — это одно дополнительное значение для инструкции: константа, адрес, офсет.

Пример.



31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Рисунок 3 - Пример инструкции вида I.

Здесь исходя из *opcode* - мы понимаем вид. Основываясь на поле *funct3* - мы узнаем инструкцию. Данная инструкции производят соответствующую операцию над регистром *rs1* и значением *imm,* и возвращают результат в *rd.*

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

Рисунок 4 - Список используемых инструкций (1/2).

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

Рисунок 5 - Список используемых инструкций (2/2).

Имея информацию про каждую конкретную инструкцию, можно сопоставить любой набор из 32-битов любой команде. Используя битовые маски и уникальное для каждой команды значение, мы прогоняем рассматриваемую команду через маску и сравниваем с конкретным значением, чтобы определить вид команды.

Структура elf файла.

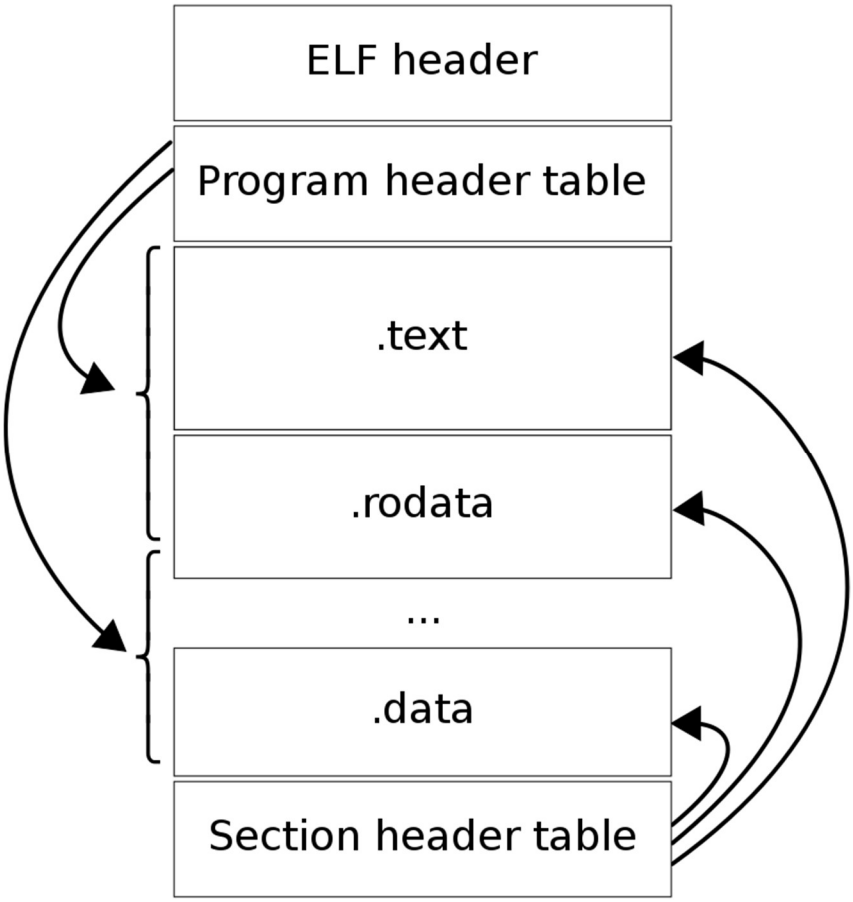


Рисунок 6 - Структура Elf файла.

Elf файл состоит из:

Заголовок файла (ELF Header) имеет фиксированное расположение в начале файла и содержит общее описание структуры файла и его основные характеристики, такие как: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры и смещения остальных частей файла.

Таблица заголовков программы - содержит заголовки, каждый из которых описывает отдельный сегмент программы и его атрибуты либо другую информацию, необходимую операционной системе для подготовки программы к исполнению. (но в данном случае мы ее будем игнорировать).

Таблица заголовков секций содержит атрибуты секций файла. Данная таблица необходима только компоновщику, исполняемые файлы в наличии этой таблицы не нуждаются (ELF загрузчик её игнорирует). Предоставленную в таблице заголовков секций информацию компоновщик использует для оптимального размещения данных секций по сегментам при сборке файла с учётом их атрибутов. Секции, которые мы обрабатываем — это: .text - подряд записанные инструкции. .symtab - таблица со статическими символами elf файла. .strtab - имена вхождений .symtab. И таблица с именами секций, чтобы иметь возможность их распознать.

Описание работы написанного кода.

disasm

Данный файл содержит констаты (маски, названия регистров и уникальные биты для каждой команды). С их помощью производится анализ каждой отдельной команды, а затем такая информация, как: имя, регистры и формат помещаются в массив в таком-же порядке.

/elf

Содержит магические байты, объявления функций для парсинга. Функции парсинга, пытаются разбить работу на секции, а затем собрать все в одну структуру которую в дальнейшем выведет основная программа.

/main

Для тестов – не иммет смысла в ходе данной работы.

/rv3

Исполняемый файл. Принимает аргументы, запускает парсеры. После получения структуры elf выводит ее. Чтобы расставить метки – делает двойной проход. Поддерживат std::map для контроля адресного положения при выводе.

Результат работы написанной программы.

/test/test_elf.disasm

```
SYMBOL TABLE

00000000 l       *UND* 00000000 *UND*
00010074 l    d  .text 00000000 .text
00011124 l    d  .bss 00000000 .bss
00000000 l    d  .comment 00000000 .comment
00000000 l    d  .riscv.attributes 00000000 .riscv.attributes
00000000 l    df *ABS* 00000000 test.c
00011924 g       *ABS* 00000000 __global_pointer$
000118f4 g     O .bss 00000320 b
00011124 g       .text 00000000 __SDATA_BEGIN__
000100ac g     F .text 00000078 mmul
00000000 g       *UND* 00000000 _start
00011124 g     O .bss 00000640 c
00011c14 g       .bss 00000000 __BSS_END__
00011124 g       .bss 00000000 __bss_start
00010074 g     F .text 0000001c main
00011124 g       .text 00000000 __DATA_BEGIN__
00011124 g       .text 00000000 _edata
00011c14 g       .bss 00000000 _end
00011764 g     O .bss 00000190 a

Disassembly of section .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           li     a0,0
       10088: 01010113           addi   sp,sp,16
       1008c: 00008067           ret
       10090: 00000013           nop
       10094: 00100137           lui    sp,0x100
       10098: fddff0ef           jal    ra,0x10074 <main>
       1009c: 00050593           mv     a1,a0
       100a0: 00a00893           li     a7,10
       100a4: 0ff0000f           fence
       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           li     t4,40
       100c8: fec50e13           addi   t3,a0,-20, L2
       100cc: 000f0313           mv     t1,t5
       100d0: 000f8893           mv     a7,t6
       100d4: 00000813           li     a6,0
       100d8: 00088693           mv     a3,a7, L1
       100dc: 000e0793           mv     a5,t3
       100e0: 00000613           li     a2,0
       100e4: 00078703           lb     a4,0(a5), L0
       100e8: 00069583           lh     a1,0(a3)
       100ec: 00178793           addi   a5,a5,1
```

```
59.       100f0: 02868693           addi   a3,a3,40
60.       100f4: 02b70733           mul    a4,a4,a1
61.       100f8: 00e60633           add    a2,a2,a4
62.       100fc: fea794e3           bne    a5,a0,0x100e4 <L3>
63.       10100: 00c32023           sw     a2,0(t1)
64.       10104: 00280813           addi   a6,a6,2
65.       10108: 00430313           addi   t1,t1,4
66.       1010c: 00288893           addi   a7,a7,2
67.       10110: fdd814e3           bne    a6,t4,0x100d8 <L3>
68.       10114: 050f0f13           addi   t5,t5,80
69.       10118: 01478513           addi   a0,a5,20
70.       1011c: fa5f16e3           bne    t5,t0,0x100c8 <L3>
71.       10120: 00008067           ret
72.
73.
```

Список источников.

Листинг кода.

/include/disasm.h

```
#pragma once

#include <array>
#include <bitset>

namespace elf {
    constexpr uint32_t MATCH_BEQ = 0x63;
    constexpr uint32_t MASK_BEQ = 0x707f;
    constexpr uint32_t MATCH_BNE = 0x1063;
    constexpr uint32_t MASK_BNE = 0x707f;
    constexpr uint32_t MATCH_BLT = 0x4063;
    constexpr uint32_t MASK_BLT = 0x707f;
    constexpr uint32_t MATCH_BGE = 0x5063;
    constexpr uint32_t MASK_BGE = 0x707f;
    constexpr uint32_t MATCH_BLTU = 0x6063;
    constexpr uint32_t MASK_BLTU = 0x707f;
    constexpr uint32_t MATCH_BGEU = 0x7063;
    constexpr uint32_t MASK_BGEU = 0x707f;
    constexpr uint32_t MATCH_JALR = 0x67;
    constexpr uint32_t MASK_JALR = 0x707f;
    constexpr uint32_t MATCH_JAL = 0x6f;
    constexpr uint32_t MASK_JAL = 0x7f;
    constexpr uint32_t MATCH_LUI = 0x37;
    constexpr uint32_t MASK_LUI = 0x7f;
    constexpr uint32_t MATCH_AUIPC = 0x17;
    constexpr uint32_t MASK_AUIPC = 0x7f;
    constexpr uint32_t MATCH_ADDI = 0x13;
    constexpr uint32_t MASK_ADDI = 0x707f;
    constexpr uint32_t MATCH_SLLI = 0x1013;
    constexpr uint32_t MASK_SLLI = 0xfc00707f;
    constexpr uint32_t MATCH_SLTI = 0x2013;
    constexpr uint32_t MASK_SLTI = 0x707f;
    constexpr uint32_t MATCH_SLTIU = 0x3013;
    constexpr uint32_t MASK_SLTIU = 0x707f;
    constexpr uint32_t MATCH_XORI = 0x4013;
    constexpr uint32_t MASK_XORI = 0x707f;
    constexpr uint32_t MATCH_SRLI = 0x5013;
    constexpr uint32_t MASK_SRLI = 0xfc00707f;
    constexpr uint32_t MATCH_SRAI = 0x40005013;
    constexpr uint32_t MASK_SRAI = 0xfc00707f;
    constexpr uint32_t MATCH_ORI = 0x6013;
    constexpr uint32_t MASK_ORI = 0x707f;
    constexpr uint32_t MATCH_ANDI = 0x7013;
    constexpr uint32_t MASK_ANDI = 0x707f;
    constexpr uint32_t MATCH_ADD = 0x33;
    constexpr uint32_t MASK_ADD = 0xfe00707f;
    constexpr uint32_t MATCH_SUB = 0x40000033;
    constexpr uint32_t MASK_SUB = 0xfe00707f;
    constexpr uint32_t MATCH_SLL = 0x1033;
    constexpr uint32_t MASK_SLL = 0xfe00707f;
    constexpr uint32_t MATCH_SLT = 0x2033;
    constexpr uint32_t MASK_SLT = 0xfe00707f;
    constexpr uint32_t MATCH_SLTU = 0x3033;
    constexpr uint32_t MASK_SLTU = 0xfe00707f;
    constexpr uint32_t MATCH_XOR = 0x4033;
    constexpr uint32_t MASK_XOR = 0xfe00707f;
    constexpr uint32_t MATCH_SRL = 0x5033;
    constexpr uint32_t MASK_SRL = 0xfe00707f;
    constexpr uint32_t MATCH_SRA = 0x40005033;
    constexpr uint32_t MASK_SRA = 0xfe00707f;
    constexpr uint32_t MATCH_OR = 0x6033;
    constexpr uint32_t MASK_OR = 0xfe00707f;
    constexpr uint32_t MATCH_AND = 0x7033;
    constexpr uint32_t MASK_AND = 0xfe00707f;
    constexpr uint32_t MATCH_ADDIW = 0x1b;
    constexpr uint32_t MASK_ADDIW = 0x707f;
```

/include/elf.h

```
#pragma once

#include <cstdint>
#include <filesystem>

#include "disasm.h"

namespace fs = std::filesystem;

namespace elf {
    constexpr char MAGIC[4] = {0x7f, 'E', 'L', 'F'};

    using Elf32_Half = uint16_t;

    using Elf32_Word = uint32_t;
    using Elf32_Sword = int32_t;

    using Elf32_Xword = uint64_t;
    using Elf32_Sxword = int64_t;

    using Elf32_Addr = int32_t;

    using Elf32_Off = uint32_t;

    using Elf32_Section = uint16_t;

    using Elf32_Versym = Elf32_Half;

    constexpr uint32_t EI_NIDENT = 16;

    struct Elf32_Ehdr {
        unsigned char e_ident[EI_NIDENT]; // Magic number and other info
        Elf32_Half e_type; // Object file type
        Elf32_Half e_machine; // Architecture
        Elf32_Word e_version; // Object file version
        Elf32_Addr e_entry; // Entry point virtual address
        Elf32_Off e_phoff; // Program header table file offset
        Elf32_Off e_shoff; // Section header table file offset
        Elf32_Word e_flags; // Processor-specific flags
        Elf32_Half e_ehsize; // ELF header size in bytes
        Elf32_Half e_phentsize; // Program header table entry size
        Elf32_Half e_phnum; // Program header table entry count
        Elf32_Half e_shentsize; // Section header table entry size
        Elf32_Half e_shnum; // Section header table entry count
        Elf32_Half e_shstrndx; // Section header string table index
    };

    struct Elf32_Shdr {
```

```
49.         Elf32_Word sh_name; // Section name (string tbl index)
50.         Elf32_Word sh_type; // Section type
51.         Elf32_Word sh_flags; // Section flags
52.         Elf32_Addr sh_addr; // Section virtual addr at execution
53.         Elf32_Off sh_offset; // Section file offset
54.         Elf32_Word sh_size; // Section size in bytes
55.         Elf32_Word sh_link; // Link to another section
56.         Elf32_Word sh_info; // Additional section information
57.         Elf32_Word sh_addralign; // Section alignment
58.         Elf32_Word sh_entsize; // Entry size if section holds table
59.     };
60.
61.     struct Elf32_Sym {
62.         Elf32_Word st_name;
63.         Elf32_Addr st_value;
64.         Elf32_Word st_size;
65.         unsigned char st_info;
66.         unsigned char st_other;
67.         Elf32_Half st_shndx;
68.     };
69.
70.     struct Elf {
71.         Elf32_Ehdr header;
72.         std::vector<Elf32_Shdr> sections;
73.         std::vector<Elf32_Sym> symtab;
74.         std::vector<char> shstr;
75.         std::vector<char> strtab;
76.         std::vector<std::pair<RawInst, Inst>> text;
77.     };
78.
79.     bool Validate(const fs::path& path);
80.
81.     Elf Parse(const fs::path& path);
82.
83.     Elf32_Ehdr ParseEhdr(std::istream& in);
84.     std::vector<Elf32_Shdr> ParseShdr(std::istream& in, Elf32_Off e_shoff,
    Elf32_Half e_shnum);
85.     std::vector<Elf32_Sym> ParseSym(std::istream& in, const Elf32_Shdr&
    shdr);
86.
87.     std::vector<char> ParseShstr(std::istream& in, const Elf32_Shdr&
    shstr);
88.     std::vector<char> ParseStrtab(std::istream& in, const Elf32_Shdr&
    strtab);
89.     std::vector<std::pair<RawInst, Inst>> ParseText(std::istream& in, const
    Elf32_Shdr& text);
90. };
91.
```

/lib/disasm.cpp

```cpp
#include <disasm.h>

#include <stdexcept>

uint32_t elf::GetCsrImm(const RawInst& inst) {
    return (inst >> 15) & 0x1f;
}

uint32_t elf::GetRD(const RawInst& inst) {
    return (inst >> 7) & 0x1f;
}

uint32_t elf::GetRS1(const RawInst& inst) {
    return (inst >> 15) & 0x1f;
}

uint32_t elf::GetRS2(const RawInst& inst) {
    return (inst >> 20) & 0x1f;
}

uint32_t elf::Bextr(
    const uint32_t src, const uint32_t start,
    const uint32_t len
) {
    return (src >> start) & ((1 << len) - 1);
}

int32_t elf::Shamt(const uint32_t value) {
    return Bextr(value, 20, 6);
}

uint32_t elf::ImmSign(const uint32_t value) {
    const int sign = Bextr(value, 31, 1);
    return sign == 1 ? static_cast<uint32_t>(-1) : 0;
}

int32_t elf::GetBImm(const uint32_t value) {
    return (Bextr(value, 8, 4) << 1) + (Bextr(value, 25, 6) << 5) +
        (Bextr(value, 7, 1) << 11) + (ImmSign(value) << 12);
}

uint32_t elf::GetIImmUnsigned(const uint32_t value) {
    return value >> 20;
}

int32_t elf::GetIImm(const uint32_t value) {
    int val = Bextr(value, 20, 12);
    int sign = Bextr(value, 31, 1) == 1 ? -1 : 1;

    if (sign == -1)
        val |= (0xfffff000);
    return val;
}

int32_t elf::GetSImm(const uint32_t value) {
```

```
56.    int sign = Bextr(value, 31, 1) == 1 ? -1 : 1;
57.    int val = Bextr(value, 7, 5) + (Bextr(value, 25, 7) << 5);
58.
59.    if (sign == -1)
60.        val |= (0xfffff000);
61.    return val;
62.}
63.
64.int32_t elf::GetJImm(const uint32_t value) {
65.    int sign = Bextr(value, 31, 1) == 1 ? -1 : 1;
66.    int val = (Bextr(value, 21, 10) << 1) + (Bextr(value, 20, 1) << 11) +
67.        (Bextr(value, 12, 8) << 12) + (Bextr(value, 31, 1) << 20);
68.
69.    if (sign == -1)
70.        val |= (0xfff00000);
71.    return val;
72.}
73.
74.elf::Inst elf::Add(const RawInst& inst) {
75.    return {
76.        "add", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
77.        REGISTERS[GetRD(inst)], 0, "add\t%R,%1S,%2S"
78.    };
79.}
80.
81.elf::Inst elf::Addi(const RawInst& inst) {
82.    const int32_t imm = GetIImm(inst);
83.    if (GetRS1(inst) == 0) {
84.        if (GetRD(inst) == 0 && imm == 0) {
85.            return {"nop", "", "", "", 0, "nop"};
86.        }
87.
88.        return {"li", "", "", REGISTERS[GetRD(inst)], imm, "li\t%R,%I"};
89.    }
90.
91.    if (imm == 0) {
92.        return {"mv", REGISTERS[GetRS1(inst)], "", REGISTERS[GetRD(inst)],
    0, "mv\t%R,%1S"};
93.    }
94.
95.    return {"addi", REGISTERS[GetRS1(inst)], "", REGISTERS[GetRD(inst)],
    imm, "addi\t%R,%1S,%I"};
96.}
97.
98.elf::Inst elf::And(const RawInst& inst) {
99.    return {
100.            "and", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
101.            REGISTERS[GetRD(inst)], 0, "and\t%R,%1S,%2S"
102.        };
103.    }
104.
105.    elf::Inst elf::Andi(const RawInst& inst) {
106.        const int32_t imm = GetIImm(inst);
107.
108.        return {"andi", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], imm, "andi\t%R,0x%X"};
```

```
109.      }
110.
111.      elf::Inst elf::Auipc(const RawInst& inst) {
112.         const auto* in = reinterpret_cast<const InstructionU*>(&inst);
113.         return {
114.             "auipc", "", "", REGISTERS[GetRD(inst)],
115.             (in->data << 12 >> 12) & 0xfffff, "auipc\t%R,0x%X"
116.         };
117.      }
118.
119.      elf::Inst elf::Beq(const RawInst& inst, const int32_t pc) {
120.         if (GetRS2(inst) == 0) {
121.             return {
122.                 "begz", REGISTERS[GetRS1(inst)], "", "", GetBImm(inst) +
    pc, "beqz\t%1S,0x%X"
123.
124.             };
125.         }
126.
127.         return {
128.             "beq", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)], "",
129.             GetBImm(inst) + pc, "beqz\t%1S,%2S,0x%X"
130.         };
131.      }
132.
133.      elf::Inst elf::Bge(const RawInst& inst, const int32_t pc) {
134.         if (GetRS2(inst) == 0) {
135.             return {"bgez", REGISTERS[GetRS1(inst)], "", "",
    GetBImm(inst) + pc, "bgez\t%1S,0x%X"};
136.         }
137.         if (GetRS1(inst) == 0) {
138.             return {"blez", "", REGISTERS[GetRS2(inst)], "",
    GetBImm(inst) + pc, "blez\t%2S,0x%X"};
139.         }
140.         return {
141.             "bge", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)], "",
142.             GetBImm(inst) + pc, "bge\t%1S,%2S,0x%X"
143.         };
144.      }
145.
146.      elf::Inst elf::Bgeu(const RawInst& inst, const int32_t pc) {
147.         return {
148.             "bgeu", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
    "",
149.             GetBImm(inst) + pc, "bgeu\t%1S,%2S,0x%X"
150.         };
151.      }
152.
153.      elf::Inst elf::Blt(const RawInst& inst, const int32_t pc) {
154.         if (GetRS2(inst) == 0) {
155.             return {"bltz", REGISTERS[GetRS1(inst)], "", "",
    GetBImm(inst) + pc, "bltz\t%1S,0x%X"};
156.         }
157.
158.         if (GetRS1(inst) == 0) {
159.             return {"bgtz", "", REGISTERS[GetRS2(inst)], "",
    GetBImm(inst) + pc, "bgtz\t%2S,0x%X"};
```

```
160.         }
161.
162.         return {
163.             "blt", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)], "",
164.             GetBImm(inst) + pc, "blt\t%1S,%2S,0x%X"
165.         };
166.     }
167.
168.     elf::Inst elf::Bltu(const RawInst& inst, const int32_t pc) {
169.         return {
170.             "bltu", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
    "",
171.             GetBImm(inst) + pc, "bltu\t%1S,%2S,0x%X"
172.         };
173.     }
174.
175.     elf::Inst elf::Bne(const RawInst& inst, const int32_t pc) {
176.         if (GetRS2(inst) == 0) {
177.             return {"bnez", REGISTERS[GetRS1(inst)], "", "",
    GetBImm(inst) + pc, "bnez\t%1S,0x%X"};
178.         }
179.
180.         return {
181.             "bne", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)], "",
182.             GetBImm(inst) + pc, "bne\t%1S,%2S,0x%0X"
183.         };
184.     }
185.
186.     elf::Inst elf::Fence(const RawInst& inst) {
187.         return {"fence", "", "", "", 0, "fence"};
188.     }
189.
190.     elf::Inst elf::Fencei(const RawInst& inst) {
191.         return {"fence.i", "", "", "", 0, "fence.i"};
192.     }
193.
194.     elf::Inst elf::Jal(const RawInst& inst, const int32_t pc) {
195.         const int32_t offset = GetJImm(inst);
196.         const int32_t rd = GetRD(inst);
197.         if (rd == 0) {
198.             return {"j", "", "", "", offset + pc, "j\t0x%0X"};
199.         }
200.
201.         return {"jal", "", "", REGISTERS[rd], offset + pc,
    "jal\t%R,0x%0X"};
202.     }
203.
204.     elf::Inst elf::Jalr(const RawInst& inst) {
205.         const int32_t offset = GetIImm(inst);
206.         if (offset == 0 && GetRD(inst) == 0 && GetRS1(inst) == 1) {
207.             return {"ret", "", "", "", 0, "ret"};
208.         }
209.
210.         if (offset == 0 && GetRD(inst) == 1) {
211.             return {"jalr", REGISTERS[GetRS1(inst)], "", "", 0,
    "jalr\t%1S"};
```

```
212.         }
213.         if (offset == 0 && GetRD(inst) == 0) {
214.             return {"jr", REGISTERS[GetRS1(inst)], "", "", 0,
    "jr\t%1S"};
215.         }
216.         if (GetRD(inst) == GetRS1(inst)) {
217.             return {"jalr", REGISTERS[GetRS1(inst)], "", "", offset,
    "jalr\t%I(%1S)"};
218.         }
219.
220.         return {"jalr", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "jalr\t%R,%1S,%I"};
221.     }
222.
223.     elf::Inst elf::Lb(const RawInst& inst) {
224.         const int32_t offset = GetIImm(inst);
225.
226.         return {"lb", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "lb\t%R,%I(%1S)"};
227.     }
228.
229.     elf::Inst elf::Lbu(const RawInst& inst) {
230.         const int32_t offset = GetIImm(inst);
231.
232.         return {"lbu", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "lbu\t%R,%I(%1S)"};
233.     }
234.
235.     elf::Inst elf::Lh(const RawInst& inst) {
236.         const int32_t offset = GetIImm(inst);
237.
238.         return {"lh", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "lh\t%R,%I(%1S)"};
239.     }
240.
241.     elf::Inst elf::Lhu(const RawInst& inst) {
242.         const int32_t offset = GetIImm(inst);
243.
244.         return {"lhu", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "lhu\t%R,%I(%1S)"};
245.     }
246.
247.     elf::Inst elf::Lui(const RawInst& inst) {
248.         const auto* in = reinterpret_cast<const InstructionU*>(&inst);
249.         return {
250.             "lui", "", "", REGISTERS[GetRD(inst)],
251.             in->data << 12 >> 12 & 0xfffff, "lui\t%R,0x%X"
252.         };
253.     }
254.
255.     elf::Inst elf::Lw(const RawInst& inst) {
256.         const int32_t offset = GetIImm(inst);
257.
258.         return {"lw", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], offset, "lw\t%R,%I(%1S)"};
259.     }
```

```
260.
261.      elf::Inst elf::Or(const RawInst& inst) {
262.         return {
263.             "or", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
264.             REGISTERS[GetRD(inst)], 0, "or\t%R,%1S,%2S"
265.         };
266.      }
267.
268.      elf::Inst elf::Ori(const RawInst& inst) {
269.         const auto* in = reinterpret_cast<const InstructionI*>(&inst);
270.
271.         return {"ori", REGISTERS[GetRS1(inst)], "",
    REGISTERS[GetRD(inst)], in->imm, "ori\t%R,%1S,%I"};
272.      }
273.
274.      elf::Inst elf::Sb(const RawInst& inst) {
275.         const int32_t offset = GetSImm(inst);
276.
277.         return {"sb", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
    "", offset, "sb\t%2S,%I(%1S)"};
278.      }
279.
280.      elf::Inst elf::Sh(const RawInst& inst) {
281.         const int32_t offset = GetSImm(inst);
282.
283.         return {"sh", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
    "", offset, "sh\t%2S,%I(%1S)"};
284.      }
285.
286.      elf::Inst elf::Sll(const RawInst& inst) {
287.         return {
288.             "sll", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
289.             REGISTERS[GetRD(inst)], 0, "sll\t%R,%1S,%2S"
290.         };
291.      }
292.
293.      elf::Inst elf::Slli(const RawInst& inst) {
294.         const auto* in = reinterpret_cast<const
    InstructionIShift*>(&inst);
295.
296.         return {
297.             "slli", REGISTERS[GetRS1(inst)], "", REGISTERS[GetRD(inst)],
298.             in->shamt & 0x1F, "slli\t%R,%1S,0x%X"
299.         };
300.      }
301.
302.      elf::Inst elf::Slt(const RawInst& inst) {
303.         if (GetRS1(inst) == 0) {
304.             return {"sgtz", "", REGISTERS[GetRS2(inst)],
    REGISTERS[GetRD(inst)], 0, "sgtz\t%R,%2S"};
305.         }
306.
307.         return {
308.             "slt", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
309.             REGISTERS[GetRD(inst)], 0, "slt\t%R,%1S,%2S"
310.         };
```

```
311.      }
312.
313.      elf::Inst elf::Sltu(const RawInst& inst) {
314.         if (GetRS1(inst) == 0) {
315.             return {"snez", "", REGISTERS[GetRS2(inst)],
   REGISTERS[GetRD(inst)], 0, "snez\t%R,%2S"};
316.         }
317.         return {
318.             "sltu", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
319.             REGISTERS[GetRD(inst)], 0, "sltu\t%R,%1S,%2S"
320.         };
321.      }
322.
323.      elf::Inst elf::Slti(const RawInst& inst) {
324.         const auto* in = reinterpret_cast<const InstructionI*>(&inst);
325.
326.         return {"slti", REGISTERS[GetRS1(inst)], "",
   REGISTERS[GetRD(inst)], in->imm, "slti\t%R,%1S,%I"};
327.      }
328.
329.      elf::Inst elf::Sltiu(const RawInst& inst) {
330.         const auto* in = reinterpret_cast<const InstructionI*>(&inst);
331.         if (in->imm == 1) {
332.             return {"seqz", REGISTERS[GetRS1(inst)], "",
   REGISTERS[GetRD(inst)], 0, "seqz\t%R,%1S"};
333.         }
334.
335.         return {
336.             "sltiu", REGISTERS[GetRS1(inst)], "",
   REGISTERS[GetRD(inst)],
337.             in->imm, "sltiu\t%R,%1S,%I"
338.         };
339.      }
340.
341.      elf::Inst elf::Sra(const RawInst& inst) {
342.         return {
343.             "sra", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
344.             REGISTERS[GetRD(inst)], 0, "sra\t%R,%1S,%2S"
345.         };
346.      }
347.
348.      elf::Inst elf::Srai(const RawInst& inst) {
349.         const auto* in = reinterpret_cast<const
   InstructionIShift*>(&inst);
350.
351.         return {
352.             "srai", REGISTERS[GetRS1(inst)], "", REGISTERS[GetRD(inst)],
353.             in->shamt & 0x1F, "srai\t%R,%1S,0x%X"
354.         };
355.      }
356.
357.      elf::Inst elf::Srl(const RawInst& inst) {
358.         return {
359.             "srl", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
360.             REGISTERS[GetRD(inst)], 0, "srl\t%R,%1S,%2S"
361.         };
```

```
362.      }
363.
364.      elf::Inst elf::Srli(const RawInst& inst) {
365.         const auto* in = reinterpret_cast<const
   InstructionIShift*>(&inst);
366.
367.         return {
368.             "srli", REGISTERS[GetRS1(inst)], "", REGISTERS[GetRD(inst)],
369.             in->shamt & 0x1F, "srli\t%R,%1S,0x%X"
370.         };
371.      }
372.
373.      elf::Inst elf::Sub(const RawInst& inst) {
374.         if (GetRS1(inst) == 0) {
375.             return {"neg", "", REGISTERS[GetRS2(inst)],
   REGISTERS[GetRD(inst)], 0, "neg\t%R,%2S"};
376.         }
377.
378.         return {
379.             "sub", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
380.             REGISTERS[GetRD(inst)], 0, "sub\t%R,%1S,%2S"
381.         };
382.      }
383.
384.      elf::Inst elf::Sw(const RawInst& inst) {
385.         int32_t offset = GetSImm(inst);
386.
387.         return {"sw", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
   "", offset, "sw\t%2S,%I(%1S)"};
388.      }
389.
390.      elf::Inst elf::Xor(const RawInst& inst) {
391.         return {
392.             "xor", REGISTERS[GetRS1(inst)], REGISTERS[GetRS2(inst)],
393.             REGISTERS[GetRD(inst)], 0, "xor\t%R,%1S,%2S"
394.         };
395.      }
396.
397.      elf::Inst elf::Xori(const RawInst& inst) {
398.         const auto* in = reinterpret_cast<const InstructionI*>(&inst);
399.         if (in->imm == -1) {
400.             return {"not", REGISTERS[GetRS1(inst)], "",
   REGISTERS[GetRD(inst)], 0, "not\t%R,%1S"};
401.         }
402.
403.         return {"xori", REGISTERS[GetRS1(inst)], "",
   REGISTERS[GetRD(inst)], in->imm, "xori\t%R,%1S,%I"};
404.      }
405.
406.      elf::Inst elf::Mret(const RawInst& inst) {
407.         return {"mret", "", "", "", 0, "mret"};
408.      }
409.
410.      elf::Inst elf::Sret(const RawInst& inst) {
411.         return {"sret", "", "", "", 0, "sret"};
412.      }
```

```
413.
414.      elf::Inst elf::Uret(const RawInst& inst) {
415.         return {"uret", "", "", "", 0, "uret"};
416.      }
417.
418.      elf::Inst elf::SfenceVma(const RawInst& inst) {
419.         return {"sfence.vma", "", "", "", 0, "sfence.vma"};
420.      }
421.
422.      elf::Inst elf::Wfi(const RawInst& inst) {
423.         return {"wfi", "", "", "", 0, "wfi"};
424.      }
425.
426.      elf::Inst elf::ECall(const RawInst& inst) {
427.         return {
428.              "ecall",
429.              "",
430.              "",
431.              "",
432.              0,
433.              "ecall"
434.         };
435.      }
436.
437.      elf::Inst elf::EBreak(const RawInst& inst) {
438.         return {
439.              "ebreak",
440.              "",
441.              "",
442.              "",
443.              0,
444.              "ebreak"
445.         };
446.      }
447.
448.      elf::Inst elf::Mul(const RawInst& inst) {
449.         return {
450.              "mul",
451.              REGISTERS[GetRS1(inst)],
452.              REGISTERS[GetRS2(inst)],
453.              REGISTERS[GetRD(inst)],
454.              0,
455.              "mul\t%R,%1S,%2S"
456.         };
457.      }
458.
459.      elf::Inst elf::Mulh(const RawInst& inst) {
460.         return {
461.              "mulh",
462.              REGISTERS[GetRS1(inst)],
463.              REGISTERS[GetRS2(inst)],
464.              REGISTERS[GetRD(inst)],
465.              0,
466.              "mulh\t%R,%1S,%2S"
467.         };
468.      }
469.
```

```
470.      elf::Inst elf::Mulhu(const RawInst& inst) {
471.         return {
472.             "mulhu",
473.             REGISTERS[GetRS1(inst)],
474.             REGISTERS[GetRS2(inst)],
475.             REGISTERS[GetRD(inst)],
476.             0,
477.             "mulhu\t%R,%1S,%2S"
478.         };
479.      }
480.
481.      elf::Inst elf::Mulhsu(const RawInst& inst) {
482.         return {
483.             "mulhsu",
484.             REGISTERS[GetRS1(inst)],
485.             REGISTERS[GetRS2(inst)],
486.             REGISTERS[GetRD(inst)],
487.             0,
488.             "mulhsu\t%R,%1S,%2S"
489.         };
490.      }
491.
492.      elf::Inst elf::Rem(const RawInst& inst) {
493.         return {
494.             "rem",
495.             REGISTERS[GetRS1(inst)],
496.             REGISTERS[GetRS2(inst)],
497.             REGISTERS[GetRD(inst)],
498.             0,
499.             "rem\t%R,%1S,%2S"
500.         };
501.      }
502.
503.      elf::Inst elf::Remu(const RawInst& inst) {
504.         return {
505.             "remu",
506.             REGISTERS[GetRS1(inst)],
507.             REGISTERS[GetRS2(inst)],
508.             REGISTERS[GetRD(inst)],
509.             0,
510.             "remu\t%R,%1S,%2S"
511.         };
512.      }
513.
514.      elf::Inst elf::OpDiv(const RawInst& inst) {
515.         return {
516.             "div",
517.             REGISTERS[GetRS1(inst)],
518.             REGISTERS[GetRS2(inst)],
519.             REGISTERS[GetRD(inst)],
520.             0,
521.             "div\t%R,%1S,%2S"
522.         };
523.      }
524.
525.      elf::Inst elf::Divu(const RawInst& inst) {
526.         return {
```

```
527.             "divu",
528.             REGISTERS[GetRS1(inst)],
529.             REGISTERS[GetRS2(inst)],
530.             REGISTERS[GetRD(inst)],
531.             0,
532.             "divu\t%R,%1S,%2S"
533.         };
534.     }
535.
536.     elf::Inst elf::DiInst(const RawInst& inst) {
537.         if ((inst & MASK_BEQ) == MATCH_BEQ) {
538.             return Beq(inst, 0);
539.         }
540.         if ((inst & MASK_BNE) == MATCH_BNE) {
541.             return Bne(inst, 0);
542.         }
543.         if ((inst & MASK_BLT) == MATCH_BLT) {
544.             return Blt(inst, 0);
545.         }
546.         if ((inst & MASK_BGE) == MATCH_BGE) {
547.             return Bge(inst, 0);
548.         }
549.         if ((inst & MASK_BLTU) == MATCH_BLTU) {
550.             return Bltu(inst, 0);
551.         }
552.         if ((inst & MASK_BGEU) == MATCH_BGEU) {
553.             return Bgeu(inst, 0);
554.         }
555.         if ((inst & MASK_JALR) == MATCH_JALR) {
556.             return Jalr(inst);
557.         }
558.         if ((inst & MASK_JAL) == MATCH_JAL) {
559.             return Jal(inst, 0);
560.         }
561.         if ((inst & MASK_LUI) == MATCH_LUI) {
562.             return Lui(inst);
563.         }
564.         if ((inst & MASK_AUIPC) == MATCH_AUIPC) {
565.             return Auipc(inst);
566.         }
567.         if ((inst & MASK_ADDI) == MATCH_ADDI) {
568.             return Addi(inst);
569.         }
570.         if ((inst & MASK_SLLI) == MATCH_SLLI) {
571.             return Slli(inst);
572.         }
573.         if ((inst & MASK_SLTI) == MATCH_SLTI) {
574.             return Slti(inst);
575.         }
576.         if ((inst & MASK_SLTIU) == MATCH_SLTIU) {
577.             return Sltu(inst);
578.         }
579.         if ((inst & MASK_XORI) == MATCH_XORI) {
580.             return Xori(inst);
581.         }
582.         if ((inst & MASK_SRLI) == MATCH_SRLI) {
583.             return Srli(inst);
584.         }
```

```
585.         if ((inst & MASK_SRAI) == MATCH_SRAI) {
586.             return Srai(inst);
587.         }
588.         if ((inst & MASK_ORI) == MATCH_ORI) {
589.             return Ori(inst);
590.         }
591.         if ((inst & MASK_ANDI) == MATCH_ANDI) {
592.             return Andi(inst);
593.         }
594.         if ((inst & MASK_ADD) == MATCH_ADD) {
595.             return Add(inst);
596.         }
597.         if ((inst & MASK_SUB) == MATCH_SUB) {
598.             return Sub(inst);
599.         }
600.         if ((inst & MASK_SLL) == MATCH_SLL) {
601.             return Sll(inst);
602.         }
603.         if ((inst & MASK_SLT) == MATCH_SLT) {
604.             return Slt(inst);
605.         }
606.         if ((inst & MASK_SLTU) == MATCH_SLTU) {
607.             return Sltu(inst);
608.         }
609.         if ((inst & MASK_XOR) == MATCH_XOR) {
610.             return Xor(inst);
611.         }
612.         if ((inst & MASK_SRL) == MATCH_SRL) {
613.             return Srl(inst);
614.         }
615.         if ((inst & MASK_SRA) == MATCH_SRA) {
616.             return Sra(inst);
617.         }
618.         if ((inst & MASK_OR) == MATCH_OR) {
619.             return Or(inst);
620.         }
621.         if ((inst & MASK_AND) == MATCH_AND) {
622.             return And(inst);
623.         }
624.         if ((inst & MASK_ADDIW) == MATCH_ADDIW) {
625.             return Addi(inst);
626.         }
627.         if ((inst & MASK_SLLIW) == MATCH_SLLIW) {
628.             return Sll(inst);
629.         }
630.         if ((inst & MASK_SRLIW) == MATCH_SRLIW) {
631.             return Srli(inst);
632.         }
633.         if ((inst & MASK_SRAIW) == MATCH_SRAIW) {
634.             return Srai(inst);
635.         }
636.         if ((inst & MASK_ADDW) == MATCH_ADDW) {
637.             return Add(inst);
638.         }
639.         if ((inst & MASK_SUBW) == MATCH_SUBW) {
640.             return Sub(inst);
641.         }
642.         if ((inst & MASK_SLLW) == MATCH_SLLW) {
643.             return Sll(inst);
```

```
644.         }
645.         if ((inst & MASK_SRLW) == MATCH_SRLW) {
646.             return Srl(inst);
647.         }
648.         if ((inst & MASK_SRAW) == MATCH_SRAW) {
649.             return Sra(inst);
650.         }
651.         if ((inst & MASK_LB) == MATCH_LB) {
652.             return Lb(inst);
653.         }
654.         if ((inst & MASK_LH) == MATCH_LH) {
655.             return Lh(inst);
656.         }
657.         if ((inst & MASK_LW) == MATCH_LW) {
658.             return Lw(inst);
659.         }
660.
661.         /*if ((inst & MASK_LD) == MATCH_LD) {
662.              return func(inst);
663.          }*/
664.
665.         if ((inst & MASK_LBU) == MATCH_LBU) {
666.             return Lbu(inst);
667.         }
668.
669.         if ((inst & MASK_LHU) == MATCH_LHU) {
670.             return Lhu(inst);
671.         }
672.
673.         /*if ((inst & MASK_LWU) == MATCH_LWU) {
674.              return func(inst);
675.          }*/
676.
677.         if ((inst & MASK_SB) == MATCH_SB) {
678.             return Sb(inst);
679.         }
680.         if ((inst & MASK_SH) == MATCH_SH) {
681.             return Sh(inst);
682.         }
683.         if ((inst & MASK_SW) == MATCH_SW) {
684.             return Sw(inst);
685.         }
686.
687.         /*if ((inst & MASK_SD) == MATCH_SD) {
688.              return func(inst);
689.          }*/
690.
691.         if ((inst & MASK_FENCE) == MATCH_FENCE) {
692.             return Fence(inst);
693.         }
694.         if ((inst & MASK_FENCE_I) == MATCH_FENCE_I) {
695.             return Fencei(inst);
696.         }
697.         if ((inst & MASK_ECALL) == MATCH_ECALL) {
698.             return ECall(inst);
699.         }
700.         if ((inst & MASK_EBREAK) == MATCH_EBREAK) {
```

```
701.             return EBreak(inst);
702.         }
703.
704.         if ((inst & MASK_MUL) == MATCH_MUL) {
705.             return Mul(inst);
706.         }
707.         if ((inst & MASK_MULH) == MATCH_MULH) {
708.             return Mulh(inst);
709.         }
710.         if ((inst & MASK_MULHSU) == MATCH_MULHSU) {
711.             return Mulhsu(inst);
712.         }
713.         if ((inst & MASK_MULHU) == MATCH_MULHU) {
714.             return Mulhu(inst);
715.         }
716.         if ((inst & MASK_DIV) == MATCH_DIV) {
717.             return OpDiv(inst);
718.         }
719.         if ((inst & MASK_DIVU) == MATCH_DIVU) {
720.             return Divu(inst);
721.         }
722.         if ((inst & MASK_REM) == MATCH_REM) {
723.             return Rem(inst);
724.         }
725.         if ((inst & MASK_REMU) == MATCH_REMU) {
726.             return Remu(inst);
727.         }
728.
729.         throw std::invalid_argument("elf::DiInst error: Passed
    instruction cannot be recognized.");
730.     }
731.
```

/lib/elf.cpp

```
1. #include <array>
2. #include <elf.h>
3. #include <fstream>
4.
5. #include <disasm.h>
6.
7. bool elf::Validate(const fs::path& path) {
8.     const std::string error = "elf::Validate error: ";
9.
10.    constexpr char kElfClass = 1;
11.    constexpr Elf32_Half kEMachine = 243;
12.
13.    const Elf elf = Parse(path);
14.
```

```
15.     if (memcmp(MAGIC, elf.header.e_ident, sizeof MAGIC) != 0) {
16.         throw std::invalid_argument(error + "Magic number is not
    compatible.");
17.     }
18.
19.     if (elf.header.e_ident[4] != kElfClass) {
20.         throw std::invalid_argument(error + "Elf file should be '32-bit
    object'.");
21.     }
22.
23.     if (elf.header.e_machine != kEMachine) {
24.         throw std::invalid_argument(error + "Elf file is not 'RISC-V ELF'
    compatible.");
25.     }
26.
27.     return is_regular_file(path);
28. }
29.
30. elf::Elf elf::Parse(const fs::path& path) {
31.     std::ifstream in{path, std::ios::in | std::ios::binary};
32.     in.exceptions(std::ios::failbit);
33.
34.     Elf result;
35.
36.     Elf32_Ehdr header = ParseEhdr(in);
37.     std::vector<Elf32_Shdr> sections = ParseShdr(in, header.e_shoff,
    header.e_shnum);
38.     std::vector<char> shstr = ParseShstr(in, sections[header.e_shstrndx]);
39.     std::vector<Elf32_Sym> symtab;
40.     std::vector<char> strtab;
41.     std::vector<std::pair<RawInst, Inst>> text;
42.
43.     for (const auto& shdr : sections) {
44.         if (std::string(shstr.data() + shdr.sh_name) == ".symtab") {
45.             symtab = ParseSym(in, shdr);
46.         }
47.
48.         if (std::string(shstr.data() + shdr.sh_name) == ".strtab") {
49.             strtab = ParseStrtab(in, shdr);
50.         }
51.
52.         if (std::string(shstr.data() + shdr.sh_name) == ".text") {
53.             text = ParseText(in, shdr);
54.         }
55.     }
56.
57.     return {header, sections, symtab, shstr, strtab, text};
58. }
59.
60. elf::Elf32_Ehdr elf::ParseEhdr(std::istream& in) {
61.     std::array<char, sizeof(Elf32_Ehdr)> buffer{};
62.
63.     in.read(buffer.data(), buffer.size());
64.
65.     const Elf32_Ehdr& result =
```

```
    *reinterpret_cast<Elf32_Ehdr*>(buffer.data());
66.
67.     return result;
68. }
69.
70. std::vector<elf::Elf32_Shdr> elf::ParseShdr(std::istream& in, const
    Elf32_Off e_shoff, const Elf32_Half e_shnum) {
71.     in.seekg(e_shoff);
72.
73.     std::vector<Elf32_Shdr> result;
74.     result.reserve(e_shnum);
75.
76.     std::array<char, sizeof(Elf32_Shdr)> buffer{};
77.
78.     for (Elf32_Half i = 0; i < e_shnum; ++i) {
79.         in.read(buffer.data(), buffer.size());
80.
81.         result.push_back(*reinterpret_cast<Elf32_Shdr*>(buffer.data()));
82.     }
83.
84.     return result;
85. }
86.
87. std::vector<elf::Elf32_Sym> elf::ParseSym(std::istream& in, const
    Elf32_Shdr& shdr) {
88.     in.seekg(shdr.sh_offset);
89.
90.     std::vector<Elf32_Sym> result;
91.
92.     std::array<char, sizeof(Elf32_Sym)> buffer{};
93.
94.     for (Elf32_Word i = 0; i < shdr.sh_size / sizeof(Elf32_Sym); ++i) {
95.         in.read(buffer.data(), buffer.size());
96.
97.         result.push_back(*reinterpret_cast<Elf32_Sym*>(buffer.data()));
98.     }
99.
100.        return result;
101.    }
102.
103.    std::vector<char> elf::ParseShstr(std::istream& in, const
    Elf32_Shdr& shstr) {
104.        in.seekg(shstr.sh_offset);
105.
106.        std::vector<char> result(shstr.sh_size);
107.
108.        in.read(result.data(), shstr.sh_size);
109.
110.        return {result};
111.    }
112.
113.    std::vector<char> elf::ParseStrtab(std::istream& in, const
    Elf32_Shdr& strtab) {
114.        in.seekg(strtab.sh_offset);
```

```
115.
116.          std::vector<char> result(strtab.sh_size);
117.
118.          in.read(result.data(), strtab.sh_size);
119.
120.          return {result};
121.      }
122.
123.      std::vector<std::pair<elf::RawInst, elf::Inst>>
    elf::ParseText(std::istream& in, const Elf32_Shdr& text) {
124.          in.seekg(text.sh_offset);
125.          RawInst inst = 0;
126.
127.          std::vector<std::pair<RawInst, Inst>> result;
128.          result.reserve(text.sh_size / sizeof(inst));
129.
130.          for (Elf32_Word i = 0; i < result.capacity(); ++i) {
131.              in.read(reinterpret_cast<char*>(&inst), sizeof(inst));
132.
133.              result.emplace_back(inst, DiInst(inst));
134.          }
135.
136.          return result;
137.      }
138.      #include <array>
139.      #include <elf.h>
140.      #include <fstream>
141.
142.      #include <disasm.h>
143.
144.      bool elf::Validate(const fs::path& path) {
145.          const std::string error = "elf::Validate error: ";
146.
147.          constexpr char kElfClass = 1;
148.          constexpr Elf32_Half kEMachine = 243;
149.
150.          const Elf elf = Parse(path);
151.
152.          if (memcmp(MAGIC, elf.header.e_ident, sizeof MAGIC) != 0) {
153.              throw std::invalid_argument(error + "Magic number is not
    compatible.");
154.          }
155.
156.          if (elf.header.e_ident[4] != kElfClass) {
157.              throw std::invalid_argument(error + "Elf file should be '32-
    bit object'.");
158.          }
159.
160.          if (elf.header.e_machine != kEMachine) {
161.              throw std::invalid_argument(error + "Elf file is not 'RISC-V
    ELF' compatible.");
162.          }
163.
164.          return is_regular_file(path);
165.      }
```

```
166.
167.      elf::Elf elf::Parse(const fs::path& path) {
168.          std::ifstream in{path, std::ios::in | std::ios::binary};
169.          in.exceptions(std::ios::failbit);
170.
171.          Elf result;
172.
173.          Elf32_Ehdr header = ParseEhdr(in);
174.          std::vector<Elf32_Shdr> sections = ParseShdr(in, header.e_shoff,
    header.e_shnum);
175.          std::vector<char> shstr = ParseShstr(in,
    sections[header.e_shstrndx]);
176.          std::vector<Elf32_Sym> symtab;
177.          std::vector<char> strtab;
178.          std::vector<std::pair<RawInst, Inst>> text;
179.
180.          for (const auto& shdr : sections) {
181.              if (std::string(shstr.data() + shdr.sh_name) == ".symtab") {
182.                  symtab = ParseSym(in, shdr);
183.              }
184.
185.              if (std::string(shstr.data() + shdr.sh_name) == ".strtab") {
186.                  strtab = ParseStrtab(in, shdr);
187.              }
188.
189.              if (std::string(shstr.data() + shdr.sh_name) == ".text") {
190.                  text = ParseText(in, shdr);
191.              }
192.          }
193.
194.          return {header, sections, symtab, shstr, strtab, text};
195.      }
196.
197.      elf::Elf32_Ehdr elf::ParseEhdr(std::istream& in) {
198.          std::array<char, sizeof(Elf32_Ehdr)> buffer{};
199.
200.          in.read(buffer.data(), buffer.size());
201.
202.          const Elf32_Ehdr& result =
    *reinterpret_cast<Elf32_Ehdr*>(buffer.data());
203.
204.          return result;
205.      }
206.
207.      std::vector<elf::Elf32_Shdr> elf::ParseShdr(std::istream& in, const
    Elf32_Off e_shoff, const Elf32_Half e_shnum) {
208.          in.seekg(e_shoff);
209.
210.          std::vector<Elf32_Shdr> result;
211.          result.reserve(e_shnum);
212.
213.          std::array<char, sizeof(Elf32_Shdr)> buffer{};
214.
215.          for (Elf32_Half i = 0; i < e_shnum; ++i) {
216.              in.read(buffer.data(), buffer.size());
```

```
217.
218.
    result.push_back(*reinterpret_cast<Elf32_Shdr*>(buffer.data()));
219.         }
220.
221.         return result;
222.     }
223.
224.     std::vector<elf::Elf32_Sym> elf::ParseSym(std::istream& in, const
    Elf32_Shdr& shdr) {
225.         in.seekg(shdr.sh_offset);
226.
227.         std::vector<Elf32_Sym> result;
228.
229.         std::array<char, sizeof(Elf32_Sym)> buffer{};
230.
231.         for (Elf32_Word i = 0; i < shdr.sh_size / sizeof(Elf32_Sym);
    ++i) {
232.             in.read(buffer.data(), buffer.size());
233.
234.
    result.push_back(*reinterpret_cast<Elf32_Sym*>(buffer.data()));
235.         }
236.
237.         return result;
238.     }
239.
240.     std::vector<char> elf::ParseShstr(std::istream& in, const
    Elf32_Shdr& shstr) {
241.         in.seekg(shstr.sh_offset);
242.
243.         std::vector<char> result(shstr.sh_size);
244.
245.         in.read(result.data(), shstr.sh_size);
246.
247.         return {result};
248.     }
249.
250.     std::vector<char> elf::ParseStrtab(std::istream& in, const
    Elf32_Shdr& strtab) {
251.         in.seekg(strtab.sh_offset);
252.
253.         std::vector<char> result(strtab.sh_size);
254.
255.         in.read(result.data(), strtab.sh_size);
256.
257.         return {result};
258.     }
259.
260.     std::vector<std::pair<elf::RawInst, elf::Inst>>
    elf::ParseText(std::istream& in, const Elf32_Shdr& text) {
261.         in.seekg(text.sh_offset);
262.         RawInst inst = 0;
263.
```

```
264.         std::vector<std::pair<RawInst, Inst>> result;
265.         result.reserve(text.sh_size / sizeof(inst));
266.
267.         for (Elf32_Word i = 0; i < result.capacity(); ++i) {
268.             in.read(reinterpret_cast<char*>(&inst), sizeof(inst));
269.
270.             result.emplace_back(inst, DiInst(inst));
271.         }
272.
273.         return result;
274.     }
275.
```

/bin/rv3.cpp

```
1.  #include "rv3.h"
2.
3.  #include <fstream>
4.  #include <map>
5.
6.  std::pair<fs::path, fs::path> ParseArguments(const int argc, char** argv)
    {
7.      if (argc != 3) {
8.          throw std::runtime_error("rv3 error: Wrong amount of arguments
    (should be 3): " + std::to_string(argc));
9.      }
10.
11.     std::pair<fs::path, fs::path> result = {argv[1], argv[2]};
12.
13.     if (!elf::Validate(result.first)) {
14.         throw std::invalid_argument("rv3 error: Invalid argument was
    passed: " + result.first.string());
15.     }
16.
17.     return result;
18. }
19.
20. std::string ToHex(const int32_t value, const bool padding = true) {
21.     std::stringstream ss;
22.
23.     ss << std::hex << value;
24.
25.     std::string hex = ss.str();
26.
27.     return padding ? std::string(8 - ss.str().size(), '0') + ss.str() :
    ss.str();
28. }
```

```
29.
30.std::string BindType(const unsigned char info) {
31.    constexpr auto bind = [](const unsigned char info) {
32.        switch (info >> 4) {
33.            case 0:
34.                return 'l';
35.            case 1:
36.                return 'g';
37.            case 2:
38.                return 'w';
39.            case 13:
40.                return 'L';
41.            case 15:
42.                return 'H';
43.            default:
44.                return ' ';
45.        }
46.    };
47.    constexpr auto type = [](const unsigned char info) -> std::string {
48.        switch (info & 0xf) {
49.            case 0:
50.                return "  ";
51.            case 1:
52.                return " O";
53.            case 2:
54.                return " F";
55.            case 3:
56.                return "d ";
57.            case 4:
58.                return "df";
59.            case 13:
60.                return "LO";
61.            case 15:
62.                return "HI";
63.            default:
64.                return "  ";
65.        }
66.    };
67.
68.    std::stringstream ss;
69.
70.    ss << bind(info) << std::string(4, ' ') << type(info);
71.
72.    return ss.str();
73.}
74.
75.std::string Visibility(const unsigned char other) {
76.    switch (other & 0x3) {
77.        case 0:
78.            return "Default";
79.        case 1:
80.            return "Internal";
81.        case 2:
82.            return "Hidden";
83.        case 3:
84.            return "Protected";
85.        default:
```

```
86.             return "";
87.     }
88. }
89.
90. std::string Association(const elf::Elf& elf, const elf::Elf32_Half
    st_shndx) {
91.     if (st_shndx >= elf.sections.size() || st_shndx < 0) {
92.         return "*ABS*";
93.     }
94.
95.     std::string result{elf.shstr.data() + elf.sections[st_shndx].sh_name};
96.
97.     return result.empty() ? "*UND*" : result;
98. }
99.
100.      std::string StringifyIter(
101.          const elf::Inst& inst, const elf::Elf32_Addr addr, const
    std::map<elf::Elf32_Addr, std::string>& symtab,
102.          std::map<elf::Elf32_Addr, std::string>& marks
103.      ) {
104.          std::string result = inst.fmt;
105.
106.          if (const size_t pos = result.find("%R"); pos !=
    std::string::npos) {
107.              result.replace(pos, 2, inst.rd);
108.          }
109.
110.          if (const size_t pos = result.find("%1S"); pos !=
    std::string::npos) {
111.              result.replace(pos, 3, inst.rs1);
112.          }
113.
114.          if (const size_t pos = result.find("%2S"); pos !=
    std::string::npos) {
115.              result.replace(pos, 3, inst.rs2);
116.          }
117.
118.          if (const size_t pos = result.find("%I"); pos !=
    std::string::npos) {
119.              result.replace(pos, 2, std::to_string(inst.imm));
120.          }
121.
122.          if (const size_t pos = result.find("%0X"); pos !=
    std::string::npos) {
123.              std::stringstream ss;
124.
125.              ss << std::hex << addr + inst.imm;
126.
127.              std::string associate;
128.              if (symtab.contains(addr + inst.imm)) {
129.                  associate = symtab.at(addr + inst.imm);
130.              }
131.              else {
132.                  associate = "L" + std::to_string(marks.size());
133.                  marks[addr + inst.imm] = associate;
134.              }
```

```
135.
136.             result.replace(pos, 3, ss.str() + " <" + associate + '>');
137.         }
138.
139.         if (const size_t pos = result.find("%X"); pos !=
    std::string::npos) {
140.             std::stringstream ss;
141.
142.             ss << std::hex << inst.imm;
143.
144.             result.replace(pos, 2, ss.str());
145.         }
146.
147.         return result;
148.     }
149.
150.     void SymtabOut(std::ostream& out, const elf::Elf& elf) {
151.         out << "SYMBOL TABLE\n\n";
152.
153.         for (const auto& sym : elf.symtab) {
154.             out << ToHex(sym.st_value) << ' ';
155.             out << BindType(sym.st_info) << ' ';
156.             out << Association(elf, sym.st_shndx) << ' ';
157.             out << ToHex(sym.st_size) << ' ';
158.             out << (sym.st_name == 0 ? Association(elf, sym.st_shndx) :
    std::string{elf.strtab.data() + sym.st_name}) <<
159.                 '\n';
160.         }
161.
162.         out << std::endl;
163.     }
164.
165.     void DisAsmOut(std::ostream& out, const elf::Elf& elf) {
166.         out << "Disassembly of section .text:\n";
167.
168.         std::map<elf::Elf32_Addr, std::string> functions;
169.         std::map<elf::Elf32_Addr, std::string> symtab;
170.         std::map<elf::Elf32_Addr, std::string> marks;
171.
172.         for (const auto& sym : elf.symtab) {
173.             if (BindType(sym.st_info).back() == 'F') {
174.                 functions[sym.st_value] = std::string{elf.strtab.data()
    + sym.st_name};
175.             }
176.
177.             symtab[sym.st_value] = std::string{elf.strtab.data() +
    sym.st_name};
178.         }
179.
180.         auto addr_iter = functions.begin();
181.         elf::Elf32_Addr addr = addr_iter->first;
182.
183.         for (const auto& inst : elf.text) {
184.             StringifyIter(inst.second, addr, symtab, marks);
185.
```

```
186.             addr += sizeof(elf::RawInst);
187.         }
188.
189.         addr = addr_iter->first;
190.
191.         for (const auto& inst : elf.text) {
192.             if (addr_iter != functions.end() && addr_iter->first ==
    addr) {
193.                 out << '\n';
194.                 out << ToHex(addr_iter->first) << " <" << addr_iter-
    >second << ">:\n";
195.
196.                 ++addr_iter;
197.             }
198.
199.             out << '\t' << ToHex(addr, false) << ":\t" <<
    ToHex(inst.first) << "\t\t" << StringifyIter(
200.                 inst.second, addr, symtab, marks
201.             );
202.
203.             if (marks.contains(addr)) {
204.                 out << ", " << marks[addr];
205.             }
206.
207.             out << '\n';
208.
209.             addr += sizeof(elf::RawInst);
210.         }
211.
212.         out << std::endl;
213.     }
214.
215.     void Print(const elf::Elf& data, const fs::path& path) {
216.         std::ofstream out{path, std::ios::out};
217.         out.exceptions(std::ios::failbit);
218.
219.         SymtabOut(out, data);
220.         DisAsmOut(out, data);
221.     }
222.
223.     int main(int argc, char** argv) {
224.         std::pair<fs::path, fs::path> args;
225.
226.         try {
227.             args = ParseArguments(argc, argv);
228.         }
229.         catch (std::exception& err) {
230.             std::cout << err.what();
231.         }
232.
233.         auto data = elf::Parse(args.first);
234.
235.         try {
236.             Print(data, args.second);
237.         }
```

```
238.         catch (std::exception& err) {
239.             std::cout << err.what();
240.         }
241.
242.         return 0;
243.     }
244.
```