Лабораторная работа №3	M3136	2022
ISA	Никитин Богдан Петрович	

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

Инструментарий и требования к работе: C++17.

Описание

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

Должен поддерживаться следующий набор команд: RISC-V RV32I, RV32M.

Кодирование: little endian.

Обрабатывать нужно только секции .text, .symtable.

3. Описание системы кодирования команд RISC-V

RISC-V представляет из себя открытую ISA. RISC-V имеет вид Reg-Reg.

RISC-V описывает наборы команд, способ их кодирования, как эти команды должны исполняться (результат работы команд), набор регистров, реализацию ISA в аппаратуре: аппаратные потоки (называемы hart), работа с памятью, исключения (exception, необычное состояние, возникающее во время выполнения, связанное с инструкцией в текущем потоке), прерывания (внешнее асинхронное событие, которое может заставить текущий поток передать управление), вводится понятие trap (передача управления обработчику trap в связи с исключением или прерыванием).

Первый том описывает непривилегированные инструкции, второй – привилегированные.

RISC-V включает в себя базовый набор команд для работы с целочисленными значениями, а также необязательные расширения для базового набора. Базовый набор обязан присутствовать в реализации. RISC-V скорее представляет из себя семейство ISA.

Существует четыре базовых набора, отличающихся размером целочисленных регистров, соответствующим размером адресного пространства (количество допустимых адресов) и числом целочисленных регистров. Основные базовые наборы – RV32I и RV64I, предоставляющие 32-битное и 64-битное адресные пространства соответственно.

Инструкции из базового расширения имею фиксированную длину. Некоторые расширения могут добавлять инструкции непостоянной длины.

Два младших бита инструкций длиной 32 бита должны быть равны 1, следующие три бита не должны быть равны 1 одновременно.

Стандартное расширение “A” добавляет атомарные операции.

Стандартное расширение для работы с числами с плавающей запятой одинарной точности, обозначаемое буквой “F”, добавляет регистры с плавающей запятой, вычислительные инструкции одинарной точности, а также инструкции чтения из/записи в память с одинарной точностью.

Стандартное расширение для работы с числами с плавающей запятой двойной точности, обозначаемое буквой “D”, расширяет регистры с плавающей запятой и добавляет вычислительные инструкции двойной точности, инструкции чтения из/записи в память.

Стандартное расширение сжатых инструкций “C” предоставляет более узкие 16-битные формы некоторых инструкций.

Порядок байт – little-endian; big-endian и bi-endian поддерживаются как расширения.

Инструкции хранятся в формате little-endian.

Базовый набор целочисленных инструкций RV32I

RV32I включает в себя 32 регистра. В регистре x0 все биты равны 0 (это обеспечивается аппаратной реализацией). Регистры общего назначения x1-x31 хранят значения, которые различные инструкции интерпретируют как набор логических значений, как целое число со знаком в форме дополнение до двух или как беззнаковое целое число. В таблице 1 приведены регистры и их ABI имена.

Регистр	ABI
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5	t0
x6-7	t1-2
x8	s0/fp
x9	s1
x10-11	a0-1
x12-17	a2-7
x18-27	s2-11
x28-31	t3-6

Таблица 1 – базовые регистры RISC-V

Помимо 32-ух регистров присутствует один дополнительный: программный счётчик pc, который содержит адрес текущей инструкции (доступ к этому регистру имеется лишь косвенный).

В базовом наборе присутствует 6 типов инструкций:

- R – register-register
- I – register-immediate
- S
- U
- B
- U
- J

Все они имеют фиксированную длину 32 бита (и их адрес в памяти должен быть кратен 4). Формат кодирования инструкции указан в таблице 2.

Биты																																Тип инструкции
31					30			25	24			21	20	19			15	14			12	11			8	7	6			0	
funct7									rs2					rs1				funct3				rd					opcode				R
imm[11:0]														rs1				funct3				rd					opcode				I
imm[11:5]									rs2					rs1				funct3				imm[4:0]					opcode				S
imm[12]		imm[10:5]							rs2					rs1				funct3				imm[4:1]			imm[11]		opcode				B
imm[31:12]																						rd					opcode				U
imm[20]		imm[10:1]											imm[11]	imm[19:12]								rd					opcode				J

Таблица 2 – кодирование инструкций разных типов

imm[x] – immediate – константа. В квадратных скобках указано, какой бит (или диапазон бит) константы данное поле инструкции формирует.

rd – регистр назначения, т.е. регистр, в который будет сохранён результат выполнения команды.

rs1, rs2 – аргументы (являющиеся регистрами) команды.

opcode, funct3, funct7 служат для указания самой инструкции.

Константы 32-ух битные, таблица 3 демонстрирует, из каких бит инструкции формируется константа.

Биты																				Вид константы
31			30			20	19			12	11			10			5	4			1	0	
inst[31]														inst[30:25]				inst[24:21]				inst[20]	I
inst[31]														inst[30:25]				inst[11:8]				inst[7]	S
inst[31]											inst[7]			inst[30:25]				inst[11:8]				0	B
inst[31]			inst[30:20]				inst[19:12]				0												U
inst[31]							inst[19:12]				inst[20]			inst[30:25]				inst[24:21]				0	J

Таблица 3 – виды констант, производимые инструкциями RISC-V

inst[y] – instruction – инструкция. В квадратных скобках указано, какой бит (или диапазон бит) инструкции использован для формирования указанных(ого) бит(а) константы. Один бит инструкции может заполнять диапазон бит в константе (например, бит знака).

За знак всегда отвечает 31-й бит (здесь и далее нумерация бит с 0) инструкции.

Вычислительные инструкции имеют тип I или R. Все такие инструкции имеют opcode равный OP-IMM (некоторая константа, значение которой будет указано позже). Конкретный тип инструкции определяется по значению funct3. Есть следующие вычислительные инструкции: ADDI, SLTI, SLTIU, ANDI, ORI, XORI. Присутствуют команды сдвигов SLLI, SRLI, SRAI, которые кодируются специальной версией типа I (старшие 7 бит также участвуют в определении типа инструкции, остальные биты константы формируют shamt – размер сдвига).

Инструкция LUI используется для формирования 32-битных констант и кодируется типом U.

Инструкция AUIPC используется для формирования адреса относительно pc и кодируется типом U.

В наборе присутствуют арифметические операции типа R. Операндами операции являются регистры rs1 и rs2, результат сохраняется в

регистр rd. funct7 и funct3 определяют тип операции. Есть следующие типы операций: ADD, SLT, SLTU, AND, OR, XOR, SLL, SRL, SUB, SRA. opcode у всех инструкций такого типа равен OP.

В наборе присутствует два типа инструкций передачи управления: безусловные переходы и условные ветки (branch).

Инструкция перехода и связывания JAL кодируется J типом, константа имеет знаковый целочисленный тип. Для вычисления адреса перехода константа прибавляется к текущему адресу инструкции.

Инструкция JALR работает как JAL, только для вычисления адреса константа прибавляется к регистру rs1. Кодируется типом I.

Имеются следующие команды условных переходов: BEQ, BNE, BLT, BLTU, BGE, BGEU. Все имеют opcode равный BRANCH, для определения типа перехода используется funct3. Константа кодирует смещение относительно адреса команды ветвления.

RV32I типа Reg-Reg. Чтением из/записью в память занимаются только соответствующие инструкции. Адресное пространство 32-битно, адресация побайтовая (размер байта 8 бит).

Операции чтения кодируются типом I, операции записи – S. Все операции чтения имеют opcode равный LOAD, операции записи – STORE. Для вычисления адреса константа прибавляется к значению rs1. В память записывается значения регистра rs2. Операция определяется по funct3.

К операциям чтения относятся LW, LH, LHU, LB, LBU. К операциям записи – SW, SH, SB.

Инструкции с opcode равным SYSTEM используются для доступа к функциям системы, исполнение которых может требовать привилегии. Кодируются типом I.

В наборе есть две таких инструкции: ECALL, служащая для создания запроса к среде исполнения, и EBREAK, служащая для передачи управления среде отладки.

У обоих команд rs1 и rd должны быть равны 0, funct3 равен PRIV (который равен 0), funct12 (старшие 12 бит) используются для определения операции.

Набор включает инструкцию FENCE.

Расширение RV32M

Это расширение включает в себя целочисленные операции умножения и деления.

К операциям относятся MUL, MULH, MULHU, MULHSU, DIV, DIVU, REM, REMU, имеющие opcode раный OP. Тип операции определяется по funct7.

Типы команд

Команды кодируются следующими значениями (приведён код на языке C).

```
#define OP_IMM 0b0010011
#define OP 0b0110011
#define LOAD 0b0000011
#define STORE 0b0100011
#define JALR 0b1100111
#define LUI 0b0110111
#define AUIPC 0b0010111
#define JAL 0b1101111
#define BRANCH 0b1100011
#define SYSTEM 0b1110011
```

4. Описание структуры файла ELF

ELF файл может служить для разных целей. Далее последует описание ELF как двоичного исполняемого файла.

Структура файла:

- Заголовок ELF
- Таблица заголовков программ (необязательна)
- Сегменты
- Таблица заголовков секций (необязательна)

Типы данных

В файле используются типы данных, указанные в таблице 4.

Название	Размер	Выравнивание	Назначение
Elf32_Addr	4	4	Беззнаковый адрес
Elf32_Half	2	2	Беззнаковое среднее целое число

Elf32_Off	4	4	Беззнаковое смещение в файле
Elf32_Sword	4	4	Знаковое большое целое число
Elf32_Word	4	4	Беззнаковое большое целое число
unsigned char	1	1	Беззнаковое маленькое целое число

Таблица 4 – типы данных в ELF файле

Размер в байтах, выравнивание указывает, какому числу должен быть кратен адрес.

Заголовок ELF

Расположен в начале файла, содержит основные сведения о файле.

Заголовок имеет следующую структуру (в качестве описания приложен код на языке C).

```
#define EI_NIDENT 16


typedef struct {
    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;
} Elf32_Ehdr;
```

e_ident содержит общую характеристику файла, подробнее далее.

e_type – тип файла (исполняемый, объектный, перемещаемый или другой).

e_machine – архитектура, для которой создан файл. В нашем случае это RISC-V, поэтому значение должно быть равно 0xF3.

e_version – номер версии формата. На данный момент единственное корректное значение – 1.

e_entry – виртуальный адрес, которому система передаёт управление, тем самым начиная выполнение процесса.

e_phoff – смещение таблицы заголовков программы от начала файла в байтах. 0, если таблица отсутствует.

e_shoff – смещение таблицы заголовков секций. 0, если отсутствует.

e_flags – связанные с файлом флаги, зависящие от процессора. При их отсутствии это поле содержит 0.

e_ehsize – размер заголовка (в байтах).

e_phentsize – размер записи в таблице заголовков программ.

e_phnum – количество записей в таблице заголовков программ.

e_shentsize – размер записи в таблице заголовков секций.

e_shnum – количество записей в таблице заголовков секций.

e_shstrndx – индекс таблицы названий заголовков (таблицы строк) в таблице заголовков секций.

Идентификация ELF файла

e_ident указывает, как необходимо интерпретировать файл. Первые 4 байта содержат «волшебное число», определяющее тип файла. Они должны равняться 0x7f, ‘E’, ‘L’, ‘F’.

Далее используются следующие обозначения:

```
#define EI_CLASS 4

#define ELFCLASS32 1
#define ELFDATA2LSB 1
#define EV_CURRENT 1
#define EI_DATA 5
#define EI_VERSION 6
```

e_ident[EI_CLASS] обозначает разрядность файла. Для 32-битных значение ELFCLASS32.

e_ident[EI_DATA] обозначает порядок байт. Для little-endian значение ELFDATA2LSB.

e_ident[EI_VERSION] обозначает версию ELF файла. Единственное корректное значение EV_CURRENT.

Секции

Заголовок секции имеет следующий вид:

```
typedef struct {
  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;
} Elf32_Shdr;
```

sh_name – смещение в таблице строк, указывающее на начало строки с названием секции.

sh_type – тип секции. Далее будут использоваться значения:

```
#define SHT_PROGBITS 0x1
#define SHT_SYMTAB 0x2
#define SHT_STRTAB 0x3
```

sh_offset – смещение секции относительно начала файла.

sh_size – размер секции.

sh_link – индекс в таблице секций, интерпретация зависит от типа секции.

sh_entsize – для таблиц с записями фиксированного размера (например, для таблицы символов) хранит размер записи.

Таблица строк

sh_type равен SH_STRTAB.

Содержит нуль-терминированные строки (оканчивающиеся на символ `\0`).

Названия секций содержатся в таблице строк с индексом e_shstrndx.

Таблица символов .symtab

sh_type равен SHT_SYMTAB. Таблица символов объектного файла содержит информацию, необходимую для поиска и перемещения символьных определений и ссылок программы.

sh_link содержит индекс таблицы строк с названиями символов.

Записи имеют структуру:

```
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;
```

st_name – смещение в таблице строк с названиями символов, указывающее на начало названия символа.

st_value – значение символа.

st_size – размер символа.

st_info – указывает тип символа (TYPE) и атрибуты связывания (BIND). Значения извлекаются с помощью макросов:

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

st_shndx – все символы объявлены по отношению к какой-то секции. Содержит соответствующий номер секции.

st_other – содержит видимость (VIS) символа, извлекается с помощью макроса:

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

Таблицы 5, 6 и 7 содержат возможные значения BIND, TYPE и VIS соответственно.

Название	Значение

STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2

Таблица 5 – BIND

Название	Значение
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6

Таблица 6 – TYPE

Название	Значение
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3

Таблица 7 – VIS

Список инструкций .text

sh_type равен SHT_PROGBITS.

Содержит список исполняемых инструкций программы. Размер секции равен sh_size.

5. Описание работы написанного кода.

Программа написана на языке C++, стандарт 17. Компиляция производилась с помощью компилятора clang. Компиляцию можно произвести командой:

```
clang++ --std=c++17 main.cpp disasm.cpp elfutil.cpp riscvutil.cpp
```

Часть кода написана в C-стиле (т.е. с использованием функций из языка C).

Требования к системе:

Порядок байт little-endian, представление отрицательных чисел в форме дополнения до двух.

Код разделён на несколько файлов:

- main.cpp
- disasm.h
- disasm.cpp
- elfutil.h
- elfutil.cpp
- riscvutil.h
- riscvutil.cpp

Далее следует описание файлов.

elfutil

Содержит объявления структур ELF файла, объявления типов, констант, макросов, а также функции для работы со структурами. Описания типов, констант и макросов присутствует в описании ELF файла.

Функции get_index, get_type, get_vis, get_bind служат для получения строкового представления индекса, типа, видимости, связывания соответственно. Тип, видимость, связывания извлекаются из st_info с помощью макросов.

riscvutil

Содержит объявления типов (typedef), констант, связанных с обработкой инструкций RISC-V, а также функции для обработки инструкций.

ILEN_BYTE – размер инструкции в байтах.

Функции, извлекающие значения из инструкции, применяют к инструкции битовую маску и сдвиг.

Функции get_rd, get_rs1, get_rs2 извлекают из инструкции соответствующие регистры.

get_funct3, get_funct7, get_funct12 извлекают соответствующие значения funct.

get_reg_name возвращает имя регистра. По номеру регистра возвращается значения из массива REG_ABI, объявленного в .cpp файле (в .cpp, т.к. массив используется только для реализации одной функции).

get_*t*_immediate извлекает из инструкции константу типа *T*. Константы конструируются так же, как указано в таблице 3, за исключением U типа, которому не добавляются 0 в начале. Из инструкции извлекаются части константы с помощью битовых масок, с помощью сдвигов части помещаются на нужное место.

get_*t*_cmd, get_load_jalr_cmd возвращают название инструкции типа *T* в виде строки. На вход принимают opcode, funct3, funct7, funct12, shift_type (тип сдвига, старшие 7 бит) по необходимости. Реализованы как switch по значению аргумента или набор if-else.

is_i_shift проверяет, является ли инструкция сдвигом.

get_shift_type возвращает тип сдвига (slli, srli, srai).

get_shamt возвращает размер сдвига.

is_valid_b_instruction проверяет, является ли инструкция переходом.

disasm

Содержит класс Disasm, в котором и находится основная логика программы.

process обрабатывает ELF файл с названием input_file_name, и пишет результат в output_file_name:

```
void Disasm::process(const char *input_file_name, const char *output_file_name) {
    if (!read_input_file(elf_file_content, input_file_name)) {
        return;
    }
    elf_ptr = &elf_file_content[0];
    if (!process_header()) {
        return;
    }
    if (!process_section_header_table()) {
        return;
    }
    if (!process_symtab()) {
        return;
    }
```

```
    collect_l_labels();
    if (!open_write_file(output_file_name)) {
        return;
    }
    print_text();
    print("\n");
    print_symtab();
    if (write_error != 0) {
        report_error("Errors occurred while writing to the output file, the output
file is incorrect");
    }
    if (fclose(output_file) != 0) {
        perror("Error. Couldn't close the output file");
    }
}
```

Сначала целиком считывается входной файл, затем происходит обработка заголовка (метод process_header), после чего обработка таблицы заголовков секций (process_section_header_table), затем обработка .symtab (process_symtab), после чего происходит проход по файлу, собирающий L метки в переходах, затем в выходной файл выводится содержимое .text (print_text) и содержимое .symtab (print_symtab).

Содержимое файла хранится в векторе elf_file_content, на начало данных вектора указывает указатель elf_ptr.

print_symtab пишет данные строго по формату из условия, поэтому заголовок таблицы получается немного смещённым.

Работа с файлами происходит в C-стиле для возможности использования функции форматированного вывода в файл – fprintf.

Методы process_* возвращают bool – успешно ли произошло выполнение метода. Если false, то работа программы завершается (в process при вызове каждого такого метода происходит проверка возвращённого значения, при false просходит return из process). Помимо этого, данные методы выводят сообщения об ошибке с помощью методов report_error (принимает на вход строку с форматом printf, выводит сообщение в stderr) и perror (функция из cstdio, выводит на экран переданную строку и сообщение о последней произошедшей ошибки, например, ошибки при работе с фалом).

Разбор заголовка и секций происходит с помощью указателей: указатель на место в файле, где находится секция или заголовок, приводится к типу указателя на соответствующую структуру.

Перед приведением типа указателя вызывается функция in_file, принимающая указатель и размер типа данных. Она возвращает true, если содержимое типа не выходит за границы файла и false в ином случае. Пример:

```
if (!in_file(section_names_strtab_ptr, sizeof(Elf32_Shdr))) {
    report_error("No section header table");
    return false;
}
```

В методе process_header происходит проверка на тип файла (проверка «магического числа»), разрядность, архитектуру и т.п. Заголовок хранится в переменной header.

В методе process_section_header_table смещение таблицы названий секций находится следующим образом:

```
char *section_names_strtab_ptr = elf_ptr + header->e_shoff + header->e_shstrndx * header->e_shentsize;
```

Затем идёт проход по всем записям таблицы секций для поиска .symtab и .text. В ELF может присутствовать несколько секций SHT_PROGBITS, поэтому также происходит сравнение названия секции со строкой .text. Перед сравнением проверяется, что при сравнении функция не выйдет за границы файла:

```
        if (elf_file_content.size() - get_file_offset(section_name) > 5 && strcmp(section_name, ".text") == 0) {
            text = section;
        }
```

get_file_offset возвращает смещение указателя на данные в файле относительно указателя на начало файла.

Указатель на таблицу названий символов получается следующим образом:

```
char *strtab_ptr = elf_ptr + header->e_shoff + symtab->sh_link * header->e_shentsize;
```

Вывод инструкций в файл происходит с помощью метода print_instruction, который делает switch по opcode инструкции и вызывает соответствующий метод обработчик.

В функция обработчиках происходит извлечение информации из инструкции с помощью функций из riscutil.h, и последующий вывод их в файл в указанном в условии формате.

check_text проверяет, что размер секции кратен размеру инструкции и что секция не выходит за границы файла.

format_target возвращает строковое представление адреса перехода.

Вспомогательная статическая функция (не видно за пределами .cpp файла) to_hex конвертирует число в hex строку.

main.cpp

Происходит создание экземпляра Disasm и передача методу process аргументов командной строки, а также обработка ошибок, связанных с неверным число аргументов.

6. Результат работы написанной программы

```
.text
00010074 <main>:
   10074:    ff010113        addi sp, sp, -16
   10078:    00112623         sw ra, 12(sp)
   1007c:    030000ef         jal  ra, 0x100ac <mmul>
   10080:    00c12083         lw ra, 12(sp)
   10084:    00000513        addi a0, zero, 0
   10088:    01010113        addi sp, sp, 16
   1008c:    00008067        jalr  zero, 0(ra)
   10090:    00000013        addi zero, zero, 0
   10094:    00100137         lui  sp, 256
   10098:    fddff0ef         jal  ra, 0x10074 <main>
   1009c:    00050593        addi a1, a0, 0
   100a0:    00a00893        addi a7, zero, 10
   100a4:    0ff0000f      unknown_instruction
   100a8:    00000073       ecall
000100ac <mmul>:
   100ac:    00011f37         lui  t5, 17
   100b0:    124f0513        addi a0, t5, 292
   100b4:    65450513        addi a0, a0, 1620
   100b8:    124f0f13        addi t5, t5, 292
```

```
  100bc:      e4018293        addit0, gp, -448
  100c0:      fd018f93        addit6, gp, -48
  100c4:      02800e93        addit4, zero, 40
000100c8  <L2>:
  100c8:      fec50e13        addit3, a0, -20
  100cc:      000f0313        addit1, t5, 0
  100d0:      000f8893        addia7, t6, 0
  100d4:      00000813        addia6, zero, 0
000100d8  <L1>:
  100d8:      00088693        addia3, a7, 0
  100dc:      000e0793        addia5, t3, 0
  100e0:      00000613        addia2, zero, 0
000100e4  <L0>:
  100e4:      00078703         lb  a4, 0(a5)
  100e8:      00069583         lh  a1, 0(a3)
  100ec:      00178793        addia5, a5, 1
  100f0:      02868693        addia3, a3, 40
  100f4:      02b70733         mula4, a4, a1
  100f8:      00e60633         adda2, a2, a4
  100fc:      fea794e3         bnea5, a0, 0x100e4 <L0>
  10100:      00c32023         sw a2, 0(t1)
  10104:      00280813        addia6, a6, 2
  10108:      00430313        addit1, t1, 4
  1010c:      00288893        addia7, a7, 2
  10110:      fdd814e3         bnea6, t4, 0x100d8 <L1>
  10114:      050f0f13        addit5, t5, 80
  10118:      01478513        addia0, a5, 20
  1011c:      fa5f16e3         bnet5, 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 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
```

7. Список источников

- http://www.skyfree.org/linux/references/ELF_Format.pdf
- https://refspecs.linuxbase.org/elf/gabi4+/ch4.symtab.html
- https://riscv.org/technical/specifications/

8. Листинг кода

main.cpp

```
#include <iostream>


#include "disasm.h"

#include "elfutil.h"

#include "riscvutil.h"



int main(int argc, char* argv[]) {
   if (argc != 3) {
      std::cout << "Specify input and output files and only" << std::endl;
      return 0;
   }
   Disasm disasm{};
   disasm.process(argv[1], argv[2]);
```

}

disasm.h

```
#ifndef DISASM_H
#define DISASM_H

#include <unordered_map>
#include <vector>

#include "riscvutil.h"
#include "elfutil.h"



class Disasm {
public:
    void process(const char *input_file_name, const char *output_file_name);
private:
    long get_file_offset(const char *ptr);
    bool in_file(const char *ptr, long size);
    bool has_symtab_label(Elf32_Addr addr);
    bool has_l_label(Elf32_Addr addr);
    bool has_label(Elf32_Addr addr);
    void print_unknown(Elf32_Addr addr, Instruction instruction);
    void print_r(Elf32_Addr addr, Instruction instruction);
    void print_s(Elf32_Addr addr, Instruction instruction);
    void print_u(Elf32_Addr addr, Instruction instruction, Opcode opcode);
    void print_i(Elf32_Addr addr, Instruction instruction, Opcode opcode);
```

```
        void print_load_jalr(Elf32_Addr addr, Instruction instruction, Opcode
opcode);
        std::string get_label(Elf32_Addr addr);
        std::string format_target(Elf32_Addr addr, Immediate immediate);
        void print_j(Elf32_Addr addr, Instruction instruction);
        void print_b(Elf32_Addr addr, Instruction instruction);
        const char * get_system_cmd(Instruction instruction);
        void print_system(Elf32_Addr addr, Instruction instruction);
        void extract_l_label(Elf32_Addr addr, Instruction instruction);
        void print_instruction(Elf32_Addr addr, Instruction instruction);
        void print(const char *format, ...);
        void report_error(const char *format, ...);
        bool read_input_file(std::vector<char> &dest, const char
*input_file_name);
        void collect_l_labels();
        bool process_section_header_table();
        bool process_symtab();
        void print_text();
        void print_symtab();
        bool process_header();
        bool check_text();
        bool open_write_file(const char *output_file_name);

        std::vector<char> elf_file_content;
        std::unordered_map<Elf32_Addr, const char *> symtab_labels;
        std::unordered_map<Elf32_Addr, Elf32_Addr> l_labels;
        Elf32_Shdr *text = nullptr;
        Elf32_Shdr *symtab = nullptr;
```

```
    Elf32_Shdr *strtab;
    char *elf_ptr;
    Elf32_Ehdr *header;
    FILE *output_file;
    int write_error = 0;
};

#endif
```

disasm.cpp

```
#include <string>
#include <ios>
#include <vector>
#include <cstring>
#include <string>
#include <sstream>
#include <unordered_map>
#include <cstdio>
#include <cstdarg>
#include <cstdlib>
#include <cerrno>
#include <algorithm>

#include "disasm.h"
#include "riscvutil.h"
#include "elfutil.h"
```

```
template <class T>
static std::string to_hex(T value)
{
    std::ostringstream stream;
    stream << "0x" << std::hex << value;
    return stream.str();
}


long Disasm::get_file_offset(const char *ptr) {
    return ptr - elf_ptr;
}


bool Disasm::in_file(const char *ptr, long size) {
    long offset = get_file_offset(ptr);
    return offset >= 0 && offset + size <= elf_file_content.size();
}


bool Disasm::has_symtab_label(Elf32_Addr addr) {
    return symtab_labels.find(addr) != symtab_labels.end();
}


bool Disasm::has_l_label(Elf32_Addr addr) {
```

```
    return l_labels.find(addr) != l_labels.end();
}


bool Disasm::has_label(Elf32_Addr addr) {
    return has_symtab_label(addr) || has_l_label(addr);
}


void Disasm::print_unknown(Elf32_Addr addr, Instruction instruction) {
    print("   %05x:\t%08x\tunknown_instruction\n", addr, instruction);
}


void Disasm::print_r(Elf32_Addr addr, Instruction instruction) {
    const char * const cmd = get_r_cmd(get_funct7(instruction), get_funct3(instruction));
    if (cmd == nullptr) {
        print_unknown(addr, instruction);
    }
    else {
        print("    %05x:\t%08x\t%7s\t%s, %s, %s\n", addr, instruction, cmd, get_reg_name(get_rd(instruction)), get_reg_name(get_rs1(instruction)), get_reg_name(get_rs2(instruction)));
    }
}
```

```
void Disasm::Disasm::print_s(Elf32_Addr addr, Instruction instruction) {
    const char * cmd = get_s_cmd(get_funct3(instruction));
    if (cmd == nullptr) {
        print_unknown(addr, instruction);
    }
    else {
        std::string immediate = std::to_string(get_s_immediate(instruction));
        print("    %05x:\t%08x\t%7s\t%s, %s(%s)\n", addr, instruction, cmd,
get_reg_name(get_rs2(instruction)), immediate.c_str(),
get_reg_name(get_rs1(instruction)));
    }
}


void Disasm::print_u(Elf32_Addr addr, Instruction instruction, Opcode
opcode) {
    const char * const cmd = get_u_cmd(opcode);
    std::string immediate = std::to_string(get_u_immediate(instruction));
    print("    %05x:\t%08x\t%7s\t%s, %s\n", addr, instruction, cmd,
get_reg_name(get_rd(instruction)), immediate.c_str());
}


void Disasm::print_i(Elf32_Addr addr, Instruction instruction, Opcode
opcode) {
    Funct3 funct3 = get_funct3(instruction);
    const char * cmd;
    std::string arg;
    if (is_i_shift(funct3, opcode)) {
```

```
            cmd = get_shift_cmd(get_shift_type(instruction), funct3);
            arg = std::to_string(get_shamt(instruction));
        }
        else {
            cmd = get_i_cmd(funct3, opcode);
            arg = std::to_string(get_i_immediate(instruction));
        }
        if (cmd == nullptr) {
            print_unknown(addr, instruction);
        } else {
            print("    %05x:\t%08x\t%7s\t%s, %s, %s\n", addr, instruction, cmd,
get_reg_name(get_rd(instruction)),          get_reg_name(get_rs1(instruction)),
arg.c_str());
        }
    }


    void Disasm::print_load_jalr(Elf32_Addr addr, Instruction instruction,
Opcode opcode) {
        const char * cmd = get_load_jalr_cmd(get_funct3(instruction), opcode);
        if (cmd == nullptr) {
            print_unknown(addr, instruction);
        }
        else {
            std::string immediate = std::to_string(get_i_immediate(instruction));
            print("    %05x:\t%08x\t%7s\t%s, %s(%s)\n", addr, instruction, cmd,
get_reg_name(get_rd(instruction)),                immediate.c_str(),
get_reg_name(get_rs1(instruction)));
        }
```

```
}


std::string Disasm::get_label(Elf32_Addr addr) {
    std::string label;
    if (has_symtab_label(addr)) {
        label = symtab_labels[addr];
    }
    else {
        label = "L" + std::to_string(l_labels[addr]);
    }
    return label;
}


std::string Disasm::format_target(Elf32_Addr addr, Immediate immediate) {
    Elf32_Addr target = addr + immediate;
    return to_hex(target) + " <" + get_label(target) + ">";
}


void Disasm::print_j(Elf32_Addr addr, Instruction instruction) {
    std::string target = format_target(addr, get_j_immediate(instruction));
    print("    %05x:\t%08x\t%7s\t%s, %s\n", addr, instruction, "jal", get_reg_name(get_rd(instruction)), target.c_str());
}
```

```
void Disasm::print_b(Elf32_Addr addr, Instruction instruction) {
    const char * cmd = get_b_cmd(get_funct3(instruction));
    if (cmd == nullptr) {
        print_unknown(addr, instruction);
    }
    else {
        std::string target = format_target(addr, get_b_immediate(instruction));
        print("    %05x:\t%08x\t%7s\t%s, %s, %s\n", addr, instruction, cmd, get_reg_name(get_rs1(instruction)), get_reg_name(get_rs2(instruction)), target.c_str());
    }
}



const char * Disasm::get_system_cmd(Instruction instruction) {
    if (get_funct3(instruction) == PRIV && get_rd(instruction) == 0 && get_rs1(instruction) == 0) {
        switch (get_funct12(instruction)) {
            case ECALL:
                return "ecall";
            case EBREAK:
                return "ebreak";
            default:
                return nullptr;
        }
    }
    return nullptr;
```

```
}


void Disasm::print_system(Elf32_Addr addr, Instruction instruction) {
    const char * cmd = get_system_cmd(instruction);
    if (cmd == nullptr) {
        print_unknown(addr, instruction);
    }
    else {
        print("   %05x:\t%08x\t%7s\n", addr, instruction, cmd);
    }
}


void Disasm::extract_l_label(Elf32_Addr addr, Instruction instruction) {
    Opcode opcode = instruction & 0b1111111;
    Immediate immediate;
    if (opcode == JAL) {
        immediate = get_j_immediate(instruction);
    }
    else if (opcode == BRANCH && is_valid_b_instruction(get_funct3(instruction))) {
        immediate = get_b_immediate(instruction);
    } else {
        return;
    }
    Elf32_Addr target = addr + immediate;
    if (!has_label(target) && !has_l_label(target)) {
```

```
            l_labels[target] = l_labels.size();
        }
    }


void Disasm::print_instruction(Elf32_Addr addr, Instruction instruction) {
    Opcode opcode = instruction & 0b1111111;
    switch (opcode) {
        case LOAD:
        case JALR:
            print_load_jalr(addr, instruction, opcode);
            break;
        case LUI:
        case AUIPC:
            print_u(addr, instruction, opcode);
            break;
        case JAL:
            print_j(addr, instruction);
            break;
        case OP_IMM:
            print_i(addr, instruction, opcode);
            break;
        case BRANCH:
            print_b(addr, instruction);
            break;
        case STORE:
            print_s(addr, instruction);
```

```
            break;
        case OP:
            print_r(addr, instruction);
            break;
        case SYSTEM:
            print_system(addr, instruction);
            break;
        default:
            print_unknown(addr, instruction);
            break;
    }
}


void Disasm::report_error(const char *format, ...) {
    fprintf(stderr, "Error. ");
    va_list ptr;
    va_start(ptr, format);
    vfprintf(stderr, format, ptr);
    va_end(ptr);
    printf("\n");
}


void Disasm::print(const char *format, ...) {
    va_list ptr;
    va_start(ptr, format);
```

```
    write_error = std::min(write_error, vfprintf(output_file, format, ptr));
    va_end(ptr);
}

bool Disasm::read_input_file(std::vector<char> &dest, const char *input_file_name) {
    FILE *elf_file = fopen(input_file_name, "rb");
    if (elf_file == NULL) {
        perror("Error. Couldn't open input file");
        return false;
    }
    fseek(elf_file, 0, SEEK_END);
    long length = ftell(elf_file);
    dest.resize(length);
    fseek(elf_file, 0, SEEK_SET);
    fread(&dest[0], length, 1, elf_file);
    if (fclose(elf_file) != 0) {
        perror("Error. Couldn't close input file");
        return false;
    }
    if (length == 0) {
        report_error("Input file is empty");
        return false;
    }
    return true;
}
```

```
void Disasm::collect_l_labels() {
    for (Elf32_Word i = 0; i < text->sh_size; i += ILEN_BYTE) {
        extract_l_label(header->e_entry + i, *((Instruction *) (elf_ptr + text->sh_offset + i)));
    }
}


bool Disasm::process_section_header_table() {
    char *section_names_strtab_ptr = elf_ptr + header->e_shoff + header->e_shstrndx * header->e_shentsize;
    if (!in_file(section_names_strtab_ptr, sizeof(Elf32_Shdr))) {
        report_error("No section header table");
        return false;
    }
    Elf32_Shdr *section_names_strtab = (Elf32_Shdr *) (section_names_strtab_ptr);
    char *section_names_ptr = elf_ptr + section_names_strtab->sh_offset;
    for (Elf32_Half i = 0; i < header->e_shnum; i++) {
        char *section_ptr = elf_ptr + header->e_shoff + i * header->e_shentsize;
        if (!in_file(section_ptr, sizeof(Elf32_Shdr))) {
            report_error("No section %d", i);
        }
        Elf32_Shdr *section = (Elf32_Shdr *) (section_ptr);
        switch (section->sh_type) {
            case SHT_PROGBITS:
            {
```

```
                const char *section_name = section_names_ptr + section->sh_name;
                if (elf_file_content.size() - get_file_offset(section_name) > 5 && strcmp(section_name, ".text") == 0) {
                    text = section;
                }
                break;
            }
            case SHT_SYMTAB:
                symtab = section;
                break;
        }
    }
    if (text == nullptr) {
        report_error(".text not found");
        return false;
    }
    if (!check_text()) {
        return false;
    }
    else if (symtab == nullptr) {
        report_error(".symtab not found");
        return false;
    }
    char *strtab_ptr = elf_ptr + header->e_shoff + symtab->sh_link * header->e_shentsize;
    if (!in_file(strtab_ptr, sizeof(Elf32_Shdr))) {
        report_error("No .strtab");
```

```
            return false;
        }
        strtab = (Elf32_Shdr *) (strtab_ptr);
        return true;
    }


    bool Disasm::process_symtab() {
        for (Elf32_Word i = 0; i < symtab->sh_size / symtab->sh_entsize; i++)
{
            char *sym_ptr = elf_ptr + symtab->sh_offset + i * symtab-
>sh_entsize;
            if (!in_file(sym_ptr, sizeof(Elf32_Sym))) {
                report_error("No .symtab entry %d", i);
                return false;
            }
            Elf32_Sym *sym = (Elf32_Sym *) (sym_ptr);
            long name_offset = strtab->sh_offset + sym->st_name;
            const char *name = elf_ptr + name_offset;
            long max_length = elf_file_content.size() - name_offset;
            if (strnlen(name, max_length) == max_length) {
                report_error("Invalid .symtab (name of entry %ld not null
terminated)");
                return false;
            }
            symtab_labels[sym->st_value] = name;
        }
        return true;
```

```
}


void Disasm::print_text() {
    print(".text\n");
    for (Elf32_Word i = 0; i < text->sh_size; i += ILEN_BYTE) {
        Elf32_Addr addr = header->e_entry + i;
        if (has_label(addr)) {
            std::string label = get_label(addr);
            print("%08x   <%s>:\n", addr, label.c_str());
        }
        print_instruction(header->e_entry + i, *((Instruction *) (elf_ptr + text->sh_offset + i)));
    }
}


void Disasm::print_symtab() {
    print(".symtab\n");
    print("Symbol Value              Size Type    Bind   Vis     Index Name\n");
    for (Elf32_Word i = 0; i < symtab->sh_size / symtab->sh_entsize; i++) {
        Elf32_Sym *sym = (Elf32_Sym *) (elf_ptr + symtab->sh_offset + i * symtab->sh_entsize);
        std::string index = get_index(sym->st_shndx);
        const char * name = elf_ptr + strtab->sh_offset + sym->st_name;
```

```
            print("[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n", i, sym->st_value, sym->st_size, get_type(sym->st_info), get_bind(sym->st_info), get_vis(sym->st_other), index.c_str(), name);
        }
    }


    bool Disasm::process_header() {
        if (!in_file(elf_ptr, sizeof(Elf32_Ehdr))) {
            report_error("No file header");
            return false;
        }
        header = (Elf32_Ehdr *) elf_ptr;
        if (header->e_ident[EI_MAG0] != 0x7f ||
                header->e_ident[EI_MAG1] != 0x45 ||
                header->e_ident[EI_MAG2] != 0x4c ||
                header->e_ident[EI_MAG3] != 0x46) {
            report_error("Input file is not ELF file");
            return false;
        }
        if (header->e_ident[EI_CLASS] != ELFCLASS32) {
            report_error("Only 32 bits files supported");
            return false;
        }
        if (header->e_ident[EI_DATA] != ELFDATA2LSB) {
            report_error("Only little-endian files supported");
            return false;
        }
```

```
    if (header->e_ident[EI_VERSION] != EV_CURRENT) {
        report_error("Incorrect ELF version");
        return false;
    }
    if (header->e_machine != EM_RISCV) {
        report_error("Not RISC-V file");
        return false;
    }
    if (header->e_version != EV_CURRENT) {
        report_error("Incorrect format version");
        return false;
    }
    if (header->e_entry == 0) {
        report_error("No entry point");
        return false;
    }
    return true;
}


bool Disasm::check_text() {
    if (text->sh_size % ILEN_BYTE != 0) {
        report_error("Invalid .text size");
        return false;
    }
    if (text->sh_offset + text->sh_size > elf_file_content.size()) {
        report_error("End of .text beyond file boundaries");
```

```
        return false;
    }
    return true;
}


bool Disasm::open_write_file(const char *output_file_name) {
    output_file = fopen(output_file_name, "wb");
    if (output_file == NULL) {
        perror("Error. Couldn't open the output file");
        return false;
    }
    return true;
}


void Disasm::process(const char *input_file_name, const char *output_file_name) {
    if (!read_input_file(elf_file_content, input_file_name)) {
        return;
    }
    elf_ptr = &elf_file_content[0];
    if (!process_header()) {
        return;
    }
    if (!process_section_header_table()) {
        return;
    }
```

```
    if (!process_symtab()) {
        return;
    }
    collect_1_labels();
    if (!open_write_file(output_file_name)) {
        return;
    }
    print_text();
    print("\n");
    print_symtab();
    if (write_error != 0) {
        report_error("Errors occurred while writing to the output file, the output file is incorrect");
    }
    if (fclose(output_file) != 0) {
        perror("Error. Couldn't close the output file");
    }
}
```

riscvutil.h

```
#ifndef RISCVUTIL_H
#define RISCVUTIL_H

#include <cstdint>

#define ILEN_BYTE 4

#define OP_IMM 0b0010011
```

```
#define OP 0b0110011
#define LOAD 0b0000011
#define STORE 0b0100011
#define JALR 0b1100111
#define LUI 0b0110111
#define AUIPC 0b0010111
#define JAL 0b1101111
#define BRANCH 0b1100011
#define SYSTEM 0b1110011

#define PRIV 0b000
#define ECALL 0b000000000000
#define EBREAK 0b000000000001


typedef uint32_t Instruction;
typedef uint8_t Opcode;
typedef uint8_t Register;
typedef uint8_t Funct3;
typedef uint8_t Funct7;
typedef uint16_t Funct12;
typedef int32_t Immediate;
typedef uint8_t Shamt;
typedef uint8_t ShiftType;


Register get_rd(Instruction instruction);
```

```
Register get_rs1(Instruction instruction);
Register get_rs2(Instruction instruction);

Funct3 get_funct3(Instruction instruction);
Funct7 get_funct7(Instruction instruction);
Funct12 get_funct12(Instruction instruction);

const char * get_reg_name(Register reg);

Immediate get_i_immediate(Instruction instruction);
Immediate get_u_immediate(Instruction instruction);
Immediate get_s_immediate(Instruction instruction);
Immediate get_j_immediate(Instruction instruction);
Immediate get_b_immediate(Instruction instruction);

const char * get_r_cmd(Funct7 funct7, Funct3 funct3);
const char * get_u_cmd(Opcode opcode);
const char * get_s_cmd(Funct3 funct3);
const char * get_i_cmd(Funct3 funct3, Opcode opcode);
const char * get_b_cmd(Funct3 funct3);
const char * get_load_jalr_cmd(Funct3 funct3, Opcode opcode);
const char * get_shift_cmd(ShiftType shift_type, Funct3 funct3);

bool is_i_shift(Funct3 funct3, Opcode opcode);

ShiftType get_shift_type(Instruction instruction);
```

```
Shamt get_shamt(Instruction instruction);

bool is_valid_b_instruction(Funct3 funct3);

#endif
```

riscvutil.cpp

```
#include "riscvutil.h"


const char * const REG_ABI[] = {
    "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",
};


const char * get_load_jalr_cmd(Funct3 funct3, Opcode opcode) {
    if (opcode == JALR && funct3 == 0b000) {
        return "jalr";
    } else if (opcode != LOAD) {
        return nullptr;
    }
    switch (funct3) {
        case 0b000:
```

```
            return "lb";
        case 0b001:
            return "lh";
        case 0b010:
            return "lw";
        case 0b100:
            return "lbu";
        case 0b101:
            return "lhu";
        default:
            return nullptr;
    }
}

Register get_rd(Instruction instruction) {
    return (instruction >> 7) & 0b11111;
}

Register get_rs1(Instruction instruction) {
    return (instruction >> 15) & 0b11111;
}

Register get_rs2(Instruction instruction) {
    return (instruction >> 20) & 0b11111;
}

Funct3 get_funct3(Instruction instruction) {
```

```
    return (instruction >> 12) & 0b111;
}

Funct7 get_funct7(Instruction instruction) {
    return (instruction >> 25) & 0b1111111;
}

const char * get_reg_name(Register reg) {
    return REG_ABI[reg];
}

Funct12 get_funct12(Instruction instruction) {
    return instruction >> 20;
}

const char * get_r_cmd(Funct7 funct7, Funct3 funct3) {
    if (funct7 == 0b0000000 && funct3 == 0b000) {
        return "add";
    }
    else if (funct7 == 0b0100000 && funct3 == 0b000) {
        return "sub";
    }
    else if (funct7 == 0b0000000 && funct3 == 0b001) {
        return "sll";
    }
    else if (funct7 == 0b0000000 && funct3 == 0b010) {
        return "slt";
```

```
}
else if (funct7 == 0b0000000 && funct3 == 0b011) {
    return "sltu";
}
else if (funct7 == 0b0000000 && funct3 == 0b100) {
    return "xor";
}
else if (funct7 == 0b0000000 && funct3 == 0b101) {
    return "srl";
}
else if (funct7 == 0b0100000 && funct3 == 0b101) {
    return "sra";
}
else if (funct7 == 0b0000000 && funct3 == 0b110) {
    return "or";
}
else if (funct7 == 0b0000000 && funct3 == 0b111) {
    return "and";
}
// RV32M
else if (funct7 == 0b0000001 && funct3 == 0b000) {
    return "mul";
}
else if (funct7 == 0b0000001 && funct3 == 0b001) {
    return "mulh";
}
else if (funct7 == 0b0000001 && funct3 == 0b010) {
```

```
            return "mulhsu";
        }
        else if (funct7 == 0b0000001 && funct3 == 0b011) {
            return "mulhu";
        }
        else if (funct7 == 0b0000001 && funct3 == 0b100) {
            return "div";
        }
        else if (funct7 == 0b0000001 && funct3 == 0b101) {
            return "divu";
        }
        else if (funct7 == 0b0000001 && funct3 == 0b110) {
            return "rem";
        }
        else if (funct7 == 0b0000001 && funct3 == 0b111) {
            return "remu";
        }
        return nullptr;
    }

    Immediate get_i_immediate(Instruction instruction) {
        return ((instruction >> 20) & 0b11111111111) | ((instruction >> 31) ?
0b11111111111111111111100000000000 : 0);
    }

    Immediate get_u_immediate(Instruction instruction) {
        return instruction >> 12;
    }
```

```
Immediate get_s_immediate(Instruction instruction) {
    return ((instruction >> 7 & 0b11111) | (((instruction >> 25) & 0b111111) << 5)) | ((instruction >> 31) ? 0b11111111111111111111100000000000 : 0);
}

Immediate get_j_immediate(Instruction instruction) {
    return (((instruction >> 21) & 0b1111111111) << 1) | (((instruction >> 20) & 1) << 11) | (instruction & 0b11111111000000000000) | ((instruction >> 31) ? 0b11111111111100000000000000000000 : 0);
}


const char * get_u_cmd(Opcode opcode) {
    switch (opcode) {
        case LUI:
            return "lui";
        case AUIPC:
            return "auipc";
        default:
            return nullptr;
    }
}

const char * get_s_cmd(Funct3 funct3) {
    switch (funct3) {
        case 0b000:
            return "sb";
```

```
        case 0b001:
            return "sh";
        case 0b010:
            return "sw";
        default:
            return nullptr;
    }
}

const char * get_i_cmd(Funct3 funct3, Opcode opcode) {
    if (funct3 == 0b000 && opcode == OP_IMM) {
        return "addi";
    }
    else if (funct3 == 0b010 && opcode == OP_IMM) {
        return "slti";
    }
    else if (funct3 == 0b011 && opcode == OP_IMM) {
        return "sltiu";
    }
    else if (funct3 == 0b100 && opcode == OP_IMM) {
        return "xori";
    }
    else if (funct3 == 0b110 && opcode == OP_IMM) {
        return "ori";
    }
    else if (funct3 == 0b111 && opcode == OP_IMM) {
        return "andi";
```

```
    }
    return nullptr;
}

bool is_i_shift(Funct3 funct3, Opcode opcode) {
    return opcode == OP_IMM && (funct3 == 0b001 || funct3 == 0b101);
}

ShiftType get_shift_type(Instruction instruction) {
    return instruction >> 25;
}

Shamt get_shamt(Instruction instruction) {
    return (instruction >> 20) & 0b11111;
}

const char * get_shift_cmd(ShiftType shift_type, Funct3 funct3) {
    if (shift_type == 0b0000000 && funct3 == 0b001) {
        return "slli";
    }
    else if (shift_type == 0b0000000 && funct3 == 0b101) {
        return "srli";
    }
    else if (shift_type == 0b0100000 && funct3 == 0b101) {
        return "srai";
    }
    return nullptr;
```

```
}


Immediate get_b_immediate(Instruction instruction) {
    return (((instruction >> 8) & 0b1111) << 1) | (((instruction >> 25) & 0b111111) << 5) | (((instruction >> 7) & 1) << 11) | ((instruction >> 31) ? 0b11111111111111111111000000000000 : 0);
}


const char * get_b_cmd(Funct3 funct3) {
    switch (funct3) {
        case 0b000:
            return "beq";
        case 0b001:
            return "bne";
        case 0b100:
            return "blt";
        case 0b101:
            return "bge";
        case 0b110:
            return "bltu";
        case 0b111:
            return "bgeu";
        default:
            return nullptr;
    }
}
```

```
bool is_valid_b_instruction(Funct3 funct3) {
    switch (funct3) {
        case 0b000:
        case 0b001:
        case 0b100:
        case 0b101:
        case 0b110:
        case 0b111:
            return true;
        default:
            return false;
    }
}
```

elfutil.h

```
#ifndef ELFUTIL_H
#define ELFUTIL_H

#include <string>

#define EI_MAG0 0
#define EI_MAG1 1
#define EI_MAG2 2
#define EI_MAG3 3
#define EI_CLASS 4
```

```
#define ELFCLASS32 1
#define ELFDATA2LSB 1
#define EV_CURRENT 1
#define EI_DATA 5
#define EI_VERSION 6
#define EM_RISCV 0xf3

#define SHT_PROGBITS 0x1
#define SHT_SYMTAB 0x2
#define SHT_STRTAB 0x3

#define SHN_UNDEF 0
#define SHN_LORESERVE 0xff00
#define SHN_LOPROC 0xff00
#define SHN_HIPROC 0xff1f
#define SHN_LIVEPATCH 0xff20
#define SHN_ABS 0xfff1
#define SHN_COMMON 0xfff2
#define SHN_HIRESERVE 0xffff

#define STT_NOTYPE 0
#define STT_OBJECT 1
#define STT_FUNC 2
#define STT_SECTION 3
#define STT_FILE 4
#define STT_COMMON 5
#define STT_TLS 6
```

```
#define ELF32_ST_TYPE(i) ((i)&0xf)

#define ELF32_ST_BIND(i) ((i)>>4)

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

#define STV_DEFAULT 0
#define STV_INTERNAL 1
#define STV_HIDDEN 2
#define STV_PROTECTED 3

#define STB_LOCAL 0
#define STB_GLOBAL 1
#define STB_WEAK 2
#define STB_LOOS 10
#define STB_HIOS 12
#define STB_LOPROC 13
#define STB_HIPROC 15

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;
```

```
#define EI_NIDENT 16


typedef struct {
    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_phum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;


typedef struct {
  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;
} Elf32_Shdr;


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;


std::string get_index(Elf32_Half st_shndx);

const char * get_type(unsigned char st_info);

const char * get_vis(unsigned char st_other);

const char * get_bind(unsigned char st_info);

#endif
```

elfutil.cpp

```
#include "elfutil.h"



std::string get_index(Elf32_Half st_shndx) {
    switch (st_shndx) {
        case 0:
            return "UNDEF";
        case 0xfff1:
            return "ABS";
        case 0xfff2:
            return "COMMON";
        default:
            return std::to_string(st_shndx);
    }
}

const char * get_type(unsigned char st_info) {
    switch(ELF32_ST_TYPE(st_info)) {
        case 0:
            return "NOTYPE";
        case 1:
            return "OBJECT";
        case 2:
            return "FUNC";
        case 3:
```

```
            return "SECTION";
        case 4:
            return "FILE";
        case 5:
            return "COMMON";
        case 6:
            return "TLS";
        default:
            return nullptr;
    }
}

const char * get_vis(unsigned char st_other) {
    switch(ELF32_ST_VISIBILITY(st_other)) {
        case 0:
            return "DEFAULT";
        case 1:
            return "INTERNAL";
        case 2:
            return "HIDDEN";
        case 3:
            return "PROTECTED";
        default:
            return nullptr;
    }
}
```

```
const char * get_bind(unsigned char st_info) {
    switch(ELF32_ST_BIND(st_info)) {
        case 0:
            return "LOCAL";
        case 1:
            return "GLOBAL";
        case 2:
            return "WEAK";
        default:
            return nullptr;
    }
}
```