Лабораторная работа №3	Б10	2022
ISA	Хорохорин Андрей Сергеевич	

Цель работы

Знакомство с архитектурой набора команд RISC-V

Инструментарий

1. clang version 14.0.6
2. VIM - Vi IMproved 9.0 (2022 Jun 28, compiled Nov 19 2022 14:37:14)
3. GNU Make 4.3
4. XeTeX 3.141592653-2.6-0.999994 (TeX Live 2022/Arch Linux)

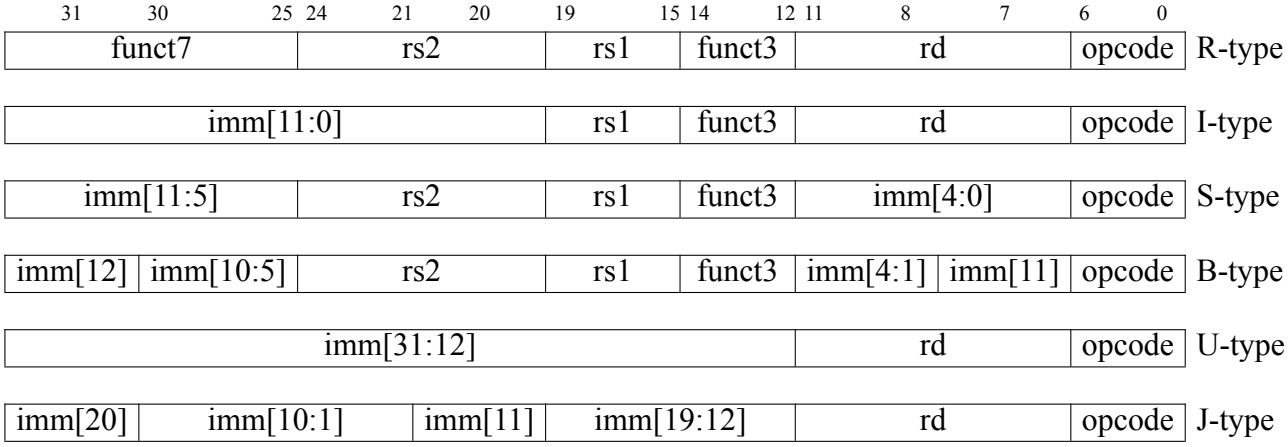
Описание системы кодирования команд RISC-V

Общая структура ISA

Стандарт RISC-V декларирует 4 минимальных ISA для 32, 64 и даже 128 битных систем, а также множество ISA-расширений для них. RISC-V может быть расширена и при помощи сторонних, не входящих в стандарт RISC-V. Это достигается некоторой избыточностью кодирования команд и резервированием некоторых кодов команд для сторонних команд-расширений.
Стандарт написан таким образом, чтобы RISC-V можно было реализовать не только в виде процессора, поддерживающего эту ISA, но и как программный эмулятор. Любую сущность поддерживающую описанный набор инструкций называют платформой RISC-V. Вне зависимости от платформы логически есть только одно адресное пространство, которое может как дизъюнктно объединять несколько присутствующих физических, так и объединять имея некоторое не пустое пересечение адресных пространств.
Базовая ISA предполагает 32 битное кодирование для всех команд, что оправдывает название архитектуры, но стандарт также позволят использовать кодирование с переменной длинной инструкции, но необходимо, чтобы длина каждой была кратна 16 битам.
Отдельно стоит отметить, что некоторые стандарты уже утверждены и не могут быть изменены в будущем, как например разрабатываемая нами RV32I, RV32M, в то же время есть стандарты, которые могут быть изменены в будущем.
Для примера устройства базовой ISA, к которой уже в дальнейшем будут накладываться улучшения, возьмём RV32I. Остальные базовые ISA отличаются главным образом количеством и размеров регистров.

Устройство базовой ISA на примере RV32I

В RV32I в пользование программиста предоставляется 32 регистра, каждый из которых размера 32 бита. Один из регистров имеет особенность: его содержимое всегда равно нулю, даже после записи в него. Помимо этого, есть ещё один регистр pc, основной целью которого является хранения места в памяти текущей исполняемой инструкции. Каждая из команд кодируется в двоичном виде. Для облегчения декодирования все команды разбиты на 6 типов, которые приведены в таблице ниже.



Заметим, что типы U,J и B,S почти совпадают. Они существуют для кодирования команд, где данные imm точно делится на 2, позволяет передать один дополнительный старший бит. А такая странная адресация в B, J сделана из-за того, что аппаратно легче записать один бит, чем записывать один бит и делать битовый сдвиг всего imm, чтобы кодировать в imm[12:6], imm[5:1].
В таблице выше за rd обозначен регистр куда сохранять результат команды, а за rs — из какого регистра считывать данные. Из такой системы кодирования следует, что данная архитектура относится к типу регистр-регистр. Помимо этого заметим, что команды построены так, чтобы rd и rs всегда были на одном и том же месте, для того, чтобы облегчить их декодирование.
Чтобы уже сейчас разобраться с типом нашей ISA скажу, что наша ISA является Load-Store, что значит, что каждая операция либо производит какое-либо вычисление на ALU, либо работает с памятью, но ни в коем виде не делает это одновременно.

Целочисленные арифметические инструкции

Арифметические команды по структуре делятся на два типа: на операции с двумя регистрами(Register-Register) и на операции с регистром и некоторой константой(Register-Immediate), которые имеют в конце своего называния I. Таким образом, большинство арифметических операций встречается в двух вариациях.

Register-Immediate Начнём с рассмотрения Register-Immediate операций. Схематично она имеет следующий вид:

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	dest	OP-IMM

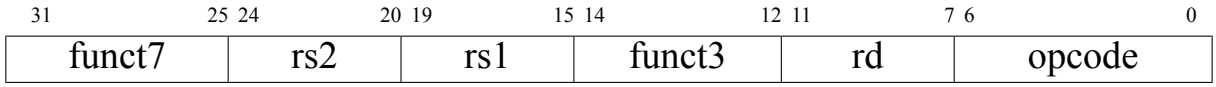
Данная команда задаёт запись результата некоторой бинарный операции заданной в поле funct3 от содержимого регистра rs1 и некоторого числа imm в регистр rd. Отметим, что всего регистров в нашей архитектуре 32, поэтому для кодирования rs1 и rd достаточно 5 бит. Стоит отметить, что 12 битное число imm является знаковым и закодировано при помощи дополнения до степени двойки, поэтому при написании компилятора приходится проделать некоторую дополнительную работу, называемую sign extension для аккуратной конвертации его в int32 с сохранением ожидаемого значения и знака.

```
void extend_sign(int& x, int sz) {
    for (; sz < 32; ++sz) {
        if (x & (1ll << sz)) {
            x |= (1ll << (sz + 1));
        }
    }
}
```

В качестве бинарной операции в ISA RV32I могут быть задействованы следующие функции(рассмотрим на примере команд Register-Immediate):

- ADDI — сложение
- SLTI — проверка простейшего неравенства: rs1 < imm
- SLTIU — проверка простейшего равенства: rs1 = imm
- ANDI — побитовое И
- ORI — побитовое ИЛИ
- XORI — побитовое Исключающее ИЛИ
- Различные битовые сдвиги: SLLI, SRLI, SRAI. Левый логический, правый логический и правый алгебраический соответственно. Команда подразумевает сдвиг значения регистра rs1 на imm. Отдельно стоит отметить, что кодирования этих сдвигов SRLI и SRAI используется одна и та же комбинация opcode, funct3, но они всё равно отличимы, так как imm в данной операции не может быть более 32, то старшие биты imm никак не используются, а это значит можно использовать их для различия этих двух видов сдвига.

Register-Register Для начала рассмотрим принцип кодирования данного вида инструкций. Схематично это выглядит следующим образом:



7	5	5	3	5	7
0000000	src2	src1	ADD/SLT[U]	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Заметим, что за счёт того, что второй регистр rs2 закодировать проще, чем значение imm, то в этом виде команд у нас всегда есть свободные для кодирования старшие байты, которые на схемы выше названы funct7.
Как следствие большего количества бит на кодирование команд, в Register-Register набор бинарных операций включает в себя аналоги всех операций из Register-Immediate и добавляет к ним ещё SUB(вычитание). Отдельно стоит отметить, что для операций SLL, SRL, SRA в качестве размера сдвига берутся младшие 5 бит регистра rs2.

Расширение RV32M

Данное расширение стандарта расширяет арифметические команды следующим набором, которые кодируются аналогичным Register-Register арифметике:

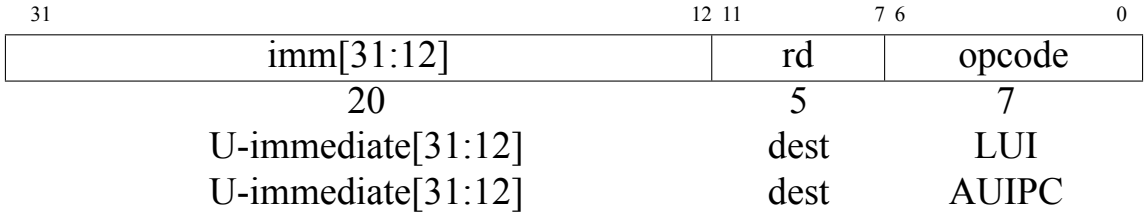
- MUL — умножение в результате которого младшие биты результата будут помещены в rd
- MULH — аналог команды MUL, но в rd будут записаны старшие биты умножения в предположении, что оба числа знаковые.
- MULHSU — аналог MULH, но предполагает, что оба числа беззнаковые.
- MULHU — аналог MULH, но предполагает, что rd1 знаковое, а rd2 — нет.
- DIV – деление чисел с знаком с округлением к нулю
- DIVU — деление чисел без знака с округлением к нулю
- REM — нахождение остатка от деления знаковых чисел. Знак результата совпадает со знаком rs1(то есть делимого).
- REMU — аналог REM для чисел без знака.

Поведение при делении на ноль указано в спецификации и задаётся следующим образом:

Condition	Dividend	Divisor	DIVU[W]	REMU[W]	DIV[W]	REM[W]
Division by zero	x	0	$2^L - 1$	x	-1	x
Overflow (signed only)	-2^{L-1}	-1	–	–	-2^{L-1}	0

Special

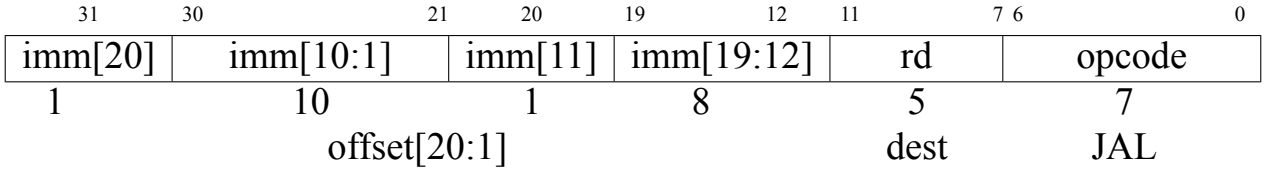
Далее будут перечислены команды LUI и AUIPC, которые я не смог явно выделить не в один из других типов. Обе эти инструкции кодируются в U типе, то есть следующим образом:



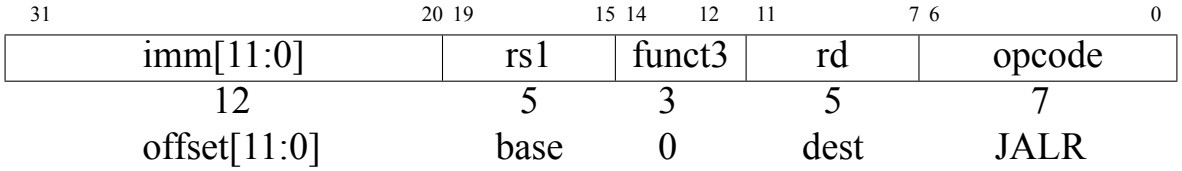
LUI rd imm записывает в регистр rd значение константы imm. Стоит обратить внимание на то, что команда задаёт у imm 20 старших бита, остальные 12 младших байт заполняются нулями. Данная инструкция необходима для быстрой загрузки в регистр 32 битных констант явным образом.
AUIPC rd imm записывает в регистр rd адрес инструкции, которая находится на смещении imm от текущей(той, которая хранится в pc). Использование данной команды в сочетании с JALR может позволять задавать произвольный 32 битный относительное смещение относительно текущего pc.

Инструкции управления исполнения

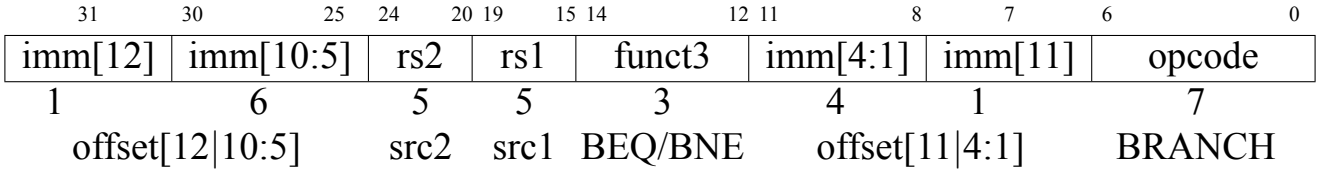
Безусловные переходы В данной архитектуре реализовано 2 безусловных перехода: JAL и JALR. Первый из которых кодируется в J type, а второй в I type.
JAL работает следующим образом. В imm с 1 по 20 бит кодируется смещение относительно адреса текущей команды pc(смещение конечно же со знаком, нулевой бит imm равен 0, так как размер наших команд кратен 2, а значит последний бит можно сэкономить), после чего совершается переход на указанное смещение imm и в rd записывается адрес newpc + 4, то есть адрес инструкции, которая будет выполнена через одну после перехода.



Поведение JALR отличается лишь способом кодирования адреса для перемещения. В этом случае оно вычисляется как rs1 + imm, причём imm уже задан с нулевого байта(так как о значении в rs1 мы ничего не знаем и может понадобится сместиться на нечётное количество байт).



Условные переходы Условные переходы в отличии от безусловных в нашей архитектуре все относительны текущей инструкции и кодируются в B type.



К тому же, как следствие того, что приходится кодировать ещё целых два регистра, на которые будет накладываться условие перехода, диапазон смещения достаточно маленький, всего $\pm$4KiB. Но из-за того, что все переходы относительны текущей позиции, то можно не кодировать последний бит смещения.

1. BEQ — совершает переход, если rs1 = rs2
2. BNE — совершает переход, если rs1 $\neq$ rs2
3. BLT — совершает переход, если rs1 < rs2 причём оба числа воспринимаются как знаковые.
4. BGE — совершает переход, если rs1 > rs2 причём оба числа воспринимаются как знаковые.
5. BLTU — совершает переход, если rs1 < rs2 причём оба числа воспринимаются как беззнаковые.
6. BGEU — совершает переход, если rs1 > rs2 причём оба числа воспринимаются как беззнаковые.

Инструкции для работы с памятью

Fence

Инструкции-подсказки исполнения

Описание структуры файла ELF

ELF(Executable and Linkable Format) — бинарный формат файла предназначенный как для хранения как целых исполняемых файлов с машинным кодом и дополнительной

информацией для его запуска, так и для хранения отдельных его частей, которые предварительно должны быть слинкованы, чтобы получить полноценный исполняемый файл.
Сам ELF файл состоит из 4 частей:

- **Заголовок файла.** Хранит в себе метаданные о файле и предполагаемом исполнителе, а также информацию о расположении заголовков программ и заголовков секций.

- **Заголовки программ.** Служат для описания процесса выделения памяти до запуска программы. Может выставлять некоторые флаги, описывающие уровен доступа к тому или иному участку памяти.

- **Секции.** Служат для хранения произвольных данных, начиная от кода программы и таблицы символов, заканчивая инициализированными переменными. Каждая из секций может иметь какую-либо структуру или не имеет её вовсе. Это зависит от самой секции и прописано в документации. Порядок секций внутри ELF файла не задан. Есть как обязательные секции, без которых программа не запустится, так и опциональные.

- **Заголовки секций.** Задают расположение непосредственно секций внутри ELF файла, а также их тип, для того, чтобы не определять его по ходу.

Для данной работы необходимо работать с двумя видами секций и заголовком файла, поэтому опишем их подробнее.

- **Заголовок файла**
  Всегда находится в самом начале файла и имеет следующие поля, которые идут именно в перечисленном ниже порядке.

  1. Первые 4 байта всегда хранят значение `7f 45 4c 46`, которые служат сигналом того, что данный файл следует считать ELF файлом.
  2. Следующий байт хранит информацию о разрядности архитектуры, где значения 1 и 2 означают 32 и 64 битные архитектуры соответственно.
  3. Следующий байт аналогично предыдущему может иметь значение либо 1 либо 2, означающие тип кодирования little endian и big endian соответственно.
  4. Следующий байт содержит версию стандарта файла ELF, но на данный момент есть только одна версия. Поэтому значение этого байта всегда будет равно 1.
  5. Следующие 2 байт указывают на тип ABI(Application Binary Interface) и его версию целевой операционной системы.
  6. Следующие 7 байт зарезервированы под дальнейшее расширение стандарта и не используются в данный момент.
  7. 2 байта по адресу `0x10` указывают на тип объектного файла ELF. Это поле необходимо, так как требования к наличию различных секций у исполняемого файла и, например, динамической библиотеки различные.

8. 2 байта по адресу `0x12` указывают архитектуру набора инструкций. RISC-V соответствует код `0xf3`.
9. 4 байта по адресу `0x14` указывает на версию ELF и равно 1 для актуальной версии ELF.
10. 4 байта по адресу `0x18` задаёт точку входа в программу, то есть адрес первой исполняемой инструкции.
11. 4 байта по адресу `0x1C` задаёт адрес начала таблицы заголовков программ.
12. 4 байта по адресу `0x20` задаёт адрес начала таблицы заголовков секций.
13. 4 байта по адресу `0x24` оставлены для использования в нуждах архитектуры и их содержимое зависит от неё.
14. 2 байта по адресу `0x28` хранят суммарный размера заголовка файла. В 32 битном случае он равен 52.
15. 2 байта по адресу `0x2A` хранят размер одной записи в таблице заголовков программ.
16. 2 байта по адресу `0x2C` хранят количество записей в таблицу заголовков программ.
17. 2 байта по адресу `0x2E` хранят размер одной записи в таблице заголовков секций.
18. 2 байта по адресу `0x30` хранят количество записей в таблицу заголовков секций.
19. 2 байта по адресу `0x32` хранят индекс секции с именами секций в таблице заголовков секций.

- **.text** Секция непосредственно хранящая код программы. В стандарте ELF ничего не декларируется о его структуре.

- **.symtab** Таблица с метками, необходимая для линковки, а именно для замены ссылок, оставленных на этапе компиляции. Представляет собой некоторое количество идущих подряд записей, каждая из которых имеет следующий вид:

  1. **st_name** — хранит индекс внутри таблицы **.strtab**, в которой по этому индексу написано имя данной метки.
  2. **st_value** — хранит адрес того объекта, на который указывает данная метка.
  3. **st_size** — хранит размер объекта, на который ссылается метка. Может быть равен 0, что значит что объект либо не имеет размера, либо его размер не известен.
  4. **st_info** — хранит тип объекта и связанные с ним атрибуты, на который указывает метка. Типичные объекты, на которые ссылается метка: структура данных, функция, файл или даже секция ELF файла. Атрибуты указывают на видимость данной метки при линковке текущего файла с другими. Тип и атрибуты специфицируется согласно стандарту ELF. Объект может не иметь ни типа ни аттрибутов.

5. **st_other** — используется для указания области видимости данной метки.
6. **st_shndx** — указывает секцию в которой используется данная ссылка. Помимо обычных значений может встретиться два следующих значения, определённых в стандарте ELF
   - `SHN_ABS(0xfff1)` — данная метка является глобальной.
   - `SHN_UNDEF(0)` — информация отсутствует.

Описание работы написанного кода

Результат работы написанной программы

Список источников

- Спецификация RISC-V
- Статья на википедии про адресацию в машинных языках
- Краткое описание структуры ELF файла
- Спецификация ELF файла

Листинг кода

Listing 1: **../include/command_types.h**

```
#ifndef COMMAND_TYPES_GUARD
#define COMMAND_TYPES_GUARD

#include "typedefs.h"
#include "elfsymbtable.h"

#include <string>
#include <vector>

struct InstructionType {
    unsigned char opcode;
    Elf32_Addr pc;
    const std::vector<SymTableEntry>& functions;
    const std::string& symbols;

    InstructionType(const Elf32_Word& cmd, const Elf32_Addr& addr,
            const std::vector<SymTableEntry>& functions, const std::string& symbols);

    unsigned char get_rd(Elf32_Word x);

    unsigned char get_rs1(Elf32_Word x);

    unsigned char get_rs2(Elf32_Word x);

    unsigned char get_funct3(Elf32_Word x);

    unsigned char get_funct7(Elf32_Word x);

    unsigned char get_reg(Elf32_Word x, unsigned char pos);

    unsigned char get_cmd(Elf32_Word x);

```

```
33      std::string format_addr(const Elf32_Addr &x);
34
35      Elf32_Word get_blk(Elf32_Word x, unsigned char len, unsigned char pos);
36
37      virtual void print() = 0;
38
39      virtual ~InstructionType() = default;
40  };
41
42  struct UType : public InstructionType {
43      std::string cmd_name;
44      unsigned char rd;
45      int imm;
46
47      UType(const Elf32_Word& cmd, const Elf32_Addr& addr,
48              const std::vector<SymTableEntry>& functions, const std::string& symbols);
49
50      void print();
51
52      ~UType() = default;
53  };
54
55  struct RType : public InstructionType {
56      std::string cmd_name;
57      unsigned char rd;
58      unsigned char rs1;
59      unsigned char rs2;
60      unsigned char funct3;
61      unsigned char funct7;
62
63      RType(const Elf32_Word& cmd, const Elf32_Addr& addr,
64              const std::vector<SymTableEntry>& functions, const std::string& symbols);
65
66      void print();
67
68      ~RType() = default;
69  };
70
71  struct SType : public InstructionType {
72      std::string cmd_name;
73      unsigned char rs1;
74      unsigned char rs2;
75      unsigned char funct3;
76      int imm;
77
78      SType(const Elf32_Word& cmd, const Elf32_Addr& addr,
79              const std::vector<SymTableEntry>& functions, const std::string& symbols);
80
81      void print();
82
83      ~SType() = default;
84  };
85
86  struct IType : public InstructionType {
87      std::string cmd_name;
88      unsigned char rs1;
89      unsigned char funct3;
90      unsigned char rd;
91      int imm;
92      bool is_load = false;
93      bool is_jump = false;
94      bool is_exec = false;
95
96      IType(const Elf32_Word& cmd, const Elf32_Addr& addr,
97              const std::vector<SymTableEntry>& functions, const std::string& symbols);
98
99      void print();
100
101     ~IType() = default;
102 };
103
104 struct JType : public InstructionType {
105     std::string cmd_name;
106     unsigned char rd;
107     int imm;
108
109     JType(const Elf32_Word& cmd, const Elf32_Addr& addr,
110             const std::vector<SymTableEntry>& functions, const std::string& symbols);
111
112     void print();
```

```
    ~JType() = default;
};

struct BType : public InstructionType {
    std::string cmd_name;
    unsigned char rs1;
    unsigned char rs2;
    unsigned char funct3;
    int imm;

    BType(const Elf32_Word& cmd, const Elf32_Addr& addr,
            const std::vector<SymTableEntry>& functions, const std::string& symbols);

    void print();

    ~BType() = default;
};

#endif
```

Listing 2: **../include/elfheader.h**

```
#ifndef ELFHEADER_GUARDS
#define ELFHEADER_GUARDS

#include "typedefs.h"

#include <istream>

struct Header {
    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;
};

Header parse_header(std::istream &ss);

#endif
```

Listing 3: **../include/elfsectiontable.h**

```
#ifndef ELFSECTIONTABLE_GUARDS
#define ELFSECTIONTABLE_GUARDS

#include "typedefs.h"

#include <vector>
#include <istream>

struct SectionTableEntry {
    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;
```

```
};


std::vector<SectionTableEntry> parse_section_table
    (std::istream &ss, Elf32_Addr addr, unsigned int entries);

std::string get_section_name (
        std::istream &ss,
        const SectionTableEntry& shstrEntry,
        const SectionTableEntry& section);

#endif
```

Listing 4: **../include/elfsymbtable.h**

```
#ifndef ELFSYMBTABLE_GUARDS
#define ELFSYMBTABLE_GUARDS

#include "typedefs.h"

#include <vector>
#include <istream>

struct SymTableEntry {
    Elf32_Word    st_name;
    Elf32_Addr    st_value;
    Elf32_Word    st_size;
    unsigned char st_info;
    unsigned char st_other;
    Elf32_Half    st_shndx;
};

std::vector<SymTableEntry> parse_symtable(
        std::istream&,
        Elf32_Addr offset,
        unsigned int entries);


#endif
```

Listing 5: **../include/output_code.h**

```
#ifndef OUTPUTCODE_GUARDS
#define OUTPUTCODE_GUARDS

#include "typedefs.h"
#include "elfsymbtable.h"

#include <string>
#include <vector>

void print_code(
        const std::string& code,
        const std::string& symbols,
        std::vector<SymTableEntry> symtab,
        Elf32_Addr v_addr
    );


#endif
```

Listing 6: **../include/output_symbtable.h**

```
#ifndef OUTPUT_GUARDS
#define OUTPUT_GUARDS

#include "typedefs.h"
#include "elfsymbtable.h"
```

```

#include <vector>
#include <string>

void print_symtable(const std::vector<SymTableEntry>&, const std::string&);

std::string format_bind(const unsigned char& info);

std::string format_type(const unsigned char& info);

std::string format_index(const Elf32_Half& idx);

std::string format_vis(const unsigned char& st_other);

std::string format_name(const Elf32_Word& name_offset, const std::string& symbols);

inline unsigned char type_by_info(const unsigned char& info) {
    return (info&0xf);
}

inline unsigned char bind_by_info(const unsigned char& info) {
    return (info>>4);
}

#endif
```

Listing 7: **../include/typedefs.h**

```
#ifndef TYPEDEFS_GUARD
#define TYPEDEFS_GUARD

typedef unsigned int   Elf32_Addr;
typedef unsigned short Elf32_Half;
typedef unsigned int   Elf32_Off;
typedef int            Elf32_Sword;
typedef unsigned int   Elf32_Word;

const unsigned short EI_NIDENT = 16;

const unsigned int MAX_NAME = 256;

const unsigned short SHN_ABS = 0xfff1;

const unsigned short SHN_UNDEF = 0;

#define STB_LOCAL      0
#define STB_GLOBAL     1
#define STB_WEAK       2
#define STB_NUM        3

#define STB_LOPROC    13
#define STB_HIPROC    15

#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 STT_NUM        7

#define STT_LOPROC    13
#define STT_HIPROC    15


#define STV_DEFAULT    0
#define STV_INTERNAL   1
#define STV_HIDDEN     2
#define STV_PROTECTED  3
#define STV_EXPORTED   4
#define STV_SINGLETON  5
#define STV_ELIMINATE  6

#define STV_NUM        7

```

```
49  // Op codes
50  const unsigned char LUI    = 0b0110111;
51  const unsigned char AUIPC  = 0b0010111;
52  const unsigned char JAL    = 0b1101111;
53  const unsigned char JALR   = 0b1100111;
54  const unsigned char BRANCH = 0b1100011;
55  const unsigned char LOAD   = 0b0000011;
56  const unsigned char STORE  = 0b0100011;
57  const unsigned char ARITHI = 0b0010011;
58  const unsigned char ARITH  = 0b0110011;
59  const unsigned char FENCE  = 0b0001111;
60  const unsigned char EX_CTR = 0b1110011;
61
62  #endif
```

Listing 8: **../src/command_types.cpp**

```
1   #include "command_types.h"
2
3   #include "elfsymbtable.h"
4   #include "output_symbtable.h"
5   #include "typedefs.h"
6   #include <string>
7   #include <algorithm>
8   #include <vector>
9   #include <stdexcept>
10
11  namespace {
12
13  void extend_sign(int& x, int sz) {
14      sz = 32 - sz;
15      x = (x << sz) >> sz;
16  }
17
18  std::string prettify_reg(unsigned char reg) {
19      if (reg >= 32) throw std::logic_error("Accessing to non-existent register");
20      std::vector<std::string> names = {
21          "zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2", "s0", "s1",
22      };
23      for (int i = 0; i <= 7; ++i) {
24          names.push_back("a" + std::to_string(i));
25      }
26      for (int i = 2; i <= 11; ++i) {
27          names.push_back("s" + std::to_string(i));
28      }
29      for (int i = 3; i <= 6; ++i) {
30          names.push_back("t" + std::to_string(i));
31      }
32      return names[reg];
33  }
34
35  }
36
37  InstructionType::InstructionType(const Elf32_Word& cmd, const Elf32_Addr& addr,
38      const std::vector<SymTableEntry>& f, const std::string& symbols)
39      : pc(addr), functions(f), symbols(symbols) {
40      opcode = cmd & 0x7f;
41  }
42
43  std::string InstructionType::format_addr(const Elf32_Addr& x) {
44      auto it = std::find_if(functions.begin(), functions.end(),
45              [&] (const SymTableEntry& f) {
46                  return f.st_value == x;
47              }
48      );
49      if (it == functions.end()) {
50          return "";
51      } else {
52          return " <" + format_name(it->st_name, symbols) + ">";
53      }
54  }
55
56  unsigned char InstructionType::get_rd(Elf32_Word x) {
57      return get_reg(x, 7);
58  }
59
```

```
60  unsigned char InstructionType::get_rs1(Elf32_Word x) {
61      return get_reg(x, 15);
62  }
63
64  unsigned char InstructionType::get_rs2(Elf32_Word x) {
65      return get_reg(x, 20);
66  }
67
68  unsigned char InstructionType::get_funct3(Elf32_Word x) {
69      return get_blk(x, 3, 12);
70  }
71
72  unsigned char InstructionType::get_funct7(Elf32_Word x) {
73      return get_blk(x, 7, 25);
74  }
75
76  unsigned char InstructionType::get_reg(Elf32_Word x, unsigned char pos) {
77      return get_blk(x, 5, pos);
78  }
79
80  unsigned char InstructionType::get_cmd(Elf32_Word x) {
81      return get_blk(x, 7, 0);
82  }
83
84  Elf32_Word InstructionType::get_blk(Elf32_Word x, unsigned char len, unsigned char pos) {
85      return (x>>pos)&((1ll << len) - 1);
86  }
87
88  UType::UType(const Elf32_Word& cmd, const Elf32_Addr& addr,
89          const std::vector<SymTableEntry>& f, const std::string& symbols) :
90      InstructionType(cmd, addr, f, symbols) {
91      rd = get_rd(cmd);
92      imm = get_blk(cmd, 20, 12) << 12;
93      cmd_name = (get_cmd(cmd) == LUI ? "lui" : "auipc");
94  }
95
96  void UType::print() {
97      printf("%7s\t%s, %x", cmd_name.c_str(), prettify_reg(rd).c_str(), imm);
98  }
99
100 RType::RType(const Elf32_Word& cmd, const Elf32_Addr& addr,
101         const std::vector<SymTableEntry>& f, const std::string& symbols) :
102     InstructionType(cmd, addr, f, symbols) {
103     rd = get_rd(cmd);
104     rs1 = get_rs1(cmd);
105     rs2 = get_rs2(cmd);
106     funct3 = get_funct3(cmd);
107     funct7 = get_funct7(cmd);
108     if ((funct7 & 1) == 0) {
109         // R32I
110         switch (funct3) {
111             case 0b000: cmd_name = (funct7 ? "sub" : "add"); break;
112             case 0b001: cmd_name = "sll"; break;
113             case 0b010: cmd_name = "slt"; break;
114             case 0b011: cmd_name = "sltu"; break;
115             case 0b100: cmd_name = "xor"; break;
116             case 0b101: cmd_name = (funct7 ? "srl" : "sra"); break;
117             case 0b110: cmd_name = "or"; break;
118             case 0b111: cmd_name = "and"; break;
119         }
120     } else {
121         // R32M
122         switch (funct3) {
123             case 0b000: cmd_name = "mul";   break;
124             case 0b001: cmd_name = "mulh";  break;
125             case 0b010: cmd_name = "mulhsu";break;
126             case 0b011: cmd_name = "mulhu"; break;
127             case 0b100: cmd_name = "div";   break;
128             case 0b101: cmd_name = "divu";  break;
129             case 0b110: cmd_name = "rem";   break;
130             case 0b111: cmd_name = "remu";  break;
131         }
132     }
133 }
134
135 void RType::print() {
136     printf("%7s\t%s, %s, %s", cmd_name.c_str(), prettify_reg(rd).c_str(),
137             prettify_reg(rs1).c_str(), prettify_reg(rs2).c_str());
138 }
139
```

```
140
141 SType::SType(const Elf32_Word& cmd, const Elf32_Addr& addr,
142         const std::vector<SymTableEntry>& f, const std::string& symbols) :
143         InstructionType(cmd, addr, f, symbols) {
144     rs1 = get_rs1(cmd);
145     rs2 = get_rs2(cmd);
146     funct3 = get_funct3(cmd);
147     imm = (get_blk(cmd, 7, 25) << 4) | get_blk(cmd, 4, 7);
148     extend_sign(imm, 12);
149     switch (funct3) {
150         case 0b000: cmd_name = "sb"; break;
151         case 0b001: cmd_name = "sh"; break;
152         case 0b010: cmd_name = "sw"; break;
153     }
154 }
155
156 void SType::print() {
157     printf("%7s\t%s, %d(%s)", cmd_name.c_str(), prettify_reg(rs2).c_str(),
158             imm,
159             prettify_reg(rs1).c_str());
160 }
161
162 IType::IType(const Elf32_Word& cmd, const Elf32_Addr& addr,
163         const std::vector<SymTableEntry>& f, const std::string& symbols) :
164         InstructionType(cmd, addr, f, symbols) {
165     rd = get_rd(cmd);
166     rs1 = get_rs1(cmd);
167     funct3 = get_funct3(cmd);
168     imm = get_blk(cmd, 12, 20);
169     extend_sign(imm, 12);
170     if ((cmd & 0x7f) == EX_CTR) {
171         is_exec = true;
172         cmd_name = (imm & 1) ? "ebreak" : "ecall";
173     } else if (cmd & 0x40) {
174         is_jump = true;
175         cmd_name = "jalr";
176     } else if (cmd & 0x10) {
177         switch (funct3) {
178             case 0b000: cmd_name = "addi"; break;
179             case 0b010: cmd_name = "slti"; break;
180             case 0b011: cmd_name = "sltiu"; break;
181             case 0b100: cmd_name = "xori"; break;
182             case 0b110: cmd_name = "ori"; break;
183             case 0b111: cmd_name = "andi"; break;
184             case 0b001: cmd_name = "slli"; break;
185             case 0b101: cmd_name = (imm & 0x20) ? "srai" : "srli"; break;
186         }
187     } else {
188         is_load = true;
189         switch (funct3) {
190             case 0b000: cmd_name = "lb"; break;
191             case 0b001: cmd_name = "lh"; break;
192             case 0b010: cmd_name = "lw"; break;
193             case 0b100: cmd_name = "lbu"; break;
194             case 0b101: cmd_name = "lhu"; break;
195         }
196     }
197 }
198
199 void IType::print() {
200     if (is_exec) {
201         printf("%7s", cmd_name.c_str());
202     } else if (is_load) {
203         printf("%7s\t%s, %d(%s)", cmd_name.c_str(), prettify_reg(rd).c_str(),
204                 imm, prettify_reg(rs1).c_str());
205     } else if (is_jump) {
206         printf("%7s\t%s, %x(%s)", cmd_name.c_str(), prettify_reg(rd).c_str(),
207                 imm, prettify_reg(rs1).c_str());
208     } else {
209         // ARITHI
210         printf("%7s\t%s, %s, %d", cmd_name.c_str(), prettify_reg(rd).c_str(),
211                 prettify_reg(rs1).c_str(), imm);
212     }
213 }
214
215 JType::JType(const Elf32_Word& cmd, const Elf32_Addr& addr,
216         const std::vector<SymTableEntry>& f, const std::string& symbols) :
217         InstructionType(cmd, addr, f, symbols) {
218     rd = get_rd(cmd);
219     imm = (get_blk(cmd,  1, 31) << 20) |
```

```
            (get_blk(cmd, 10, 21) <<  1) |
            (get_blk(cmd,  1, 20) << 11) |
            (get_blk(cmd,  8, 12) << 12);
    extend_sign(imm, 21);
    cmd_name = "jal";
}

void JType::print() {
    printf("%7s\t%s, 0x%x", cmd_name.c_str(), prettify_reg(rd).c_str(), imm + pc);
    printf("%s", format_addr(imm + pc).c_str());
}

BType::BType(const Elf32_Word& cmd, const Elf32_Addr& addr,
        const std::vector<SymTableEntry>& f, const std::string& symbols) :
        InstructionType(cmd, addr, f, symbols) {
    rs1 = get_rs1(cmd);
    rs2 = get_rs2(cmd);
    funct3 = get_funct3(cmd);
    imm = (get_blk(cmd, 1, 31) << 12) |
          (get_blk(cmd, 6, 25) <<  5) |
          (get_blk(cmd, 4,  8) <<  1) |
          (get_blk(cmd, 1,  7) << 11);
    extend_sign(imm, 13);
    switch (funct3) {
        case 0b000: cmd_name = "beq"; break;
        case 0b001: cmd_name = "bne"; break;
        case 0b100: cmd_name = "blt"; break;
        case 0b101: cmd_name = "bge"; break;
        case 0b110: cmd_name = "bltu"; break;
        case 0b111: cmd_name = "bgeu"; break;
    }
}

void BType::print() {
    printf("%7s\t%s, %s, 0x%x", cmd_name.c_str(), prettify_reg(rs2).c_str(),
            prettify_reg(rs1).c_str(), imm + pc);
    printf("%s", format_addr(imm + pc).c_str());
}
```

Listing 9: **../src/elfheader.cpp**

```
#include "elfheader.h"
#include "elfsectiontable.h"

#include <exception>

Header parse_header(std::istream &ss) {

    const unsigned int expected_magic = 0x464C457F;
    const unsigned char expected_arch = 1;
    const unsigned char expected_endian = 1;
    const unsigned char SysV_ABI_code = 0x00;
    const unsigned short RiscV_ISA_code = 0x00F3;
    const unsigned short header_expected_size = 52;

    Header header;

    ss.read((char*)&header, sizeof(header));
    uint32_t magic = *(uint32_t*)header.e_ident;
    if (magic != expected_magic) {
        throw std::invalid_argument("Invalid ELF magic bits");
    }
    if (header.e_ident[5] != expected_arch) {
        throw std::invalid_argument("Unsupported architecture. 32bit only");
    }
    if (header.e_ident[6] != expected_endian) {
        throw std::invalid_argument("Big endian is no supported");
    }
    if (header.e_ident[7] != SysV_ABI_code) {
        throw std::invalid_argument("Unsupported ABI. System V only");
    }
    if (header.e_machine != RiscV_ISA_code) {
        throw std::invalid_argument("Unsupported ISA. RISC-V only");
    }
    if (header.e_ehsize != header_expected_size) {
```

```
        throw std::invalid_argument("Illegal elf header size");
    }
    if (header.e_shentsize != sizeof(SectionTableEntry)) {
        throw std::invalid_argument("Unexpected section header table entry size");
    }
    return header;
};

```

Listing 10: **../src/elfsectiontable.cpp**

```
#include "elfsectiontable.h"
#include "typedefs.h"

#include <vector>
#include <istream>

std::vector<SectionTableEntry> parse_section_table (
     std::istream &ss,
     Elf32_Addr addr,
     unsigned int entries
 ) {
    ss.seekg(addr);
    std::vector<SectionTableEntry> res(entries);
    for (SectionTableEntry& i : res) {
        ss.read((char*)&i, sizeof(SectionTableEntry));
    }
    return res;
}

std::string get_section_name (
        std::istream &ss,
        const SectionTableEntry& shstrEntry,
        const SectionTableEntry& section)
{
    ss.seekg(shstrEntry.sh_offset + section.sh_name);
    char res[MAX_NAME];
    ss.getline(res, MAX_NAME, '\0');
    return (std::string) res;
};

```

Listing 11: **../src/elfsymbtable.cpp**

```
#include "elfsymbtable.h"
#include "elfsectiontable.h"
#include "typedefs.h"

#include <istream>
#include <vector>

std::vector<SymTableEntry> parse_symtable(
        std::istream &ss,
        Elf32_Addr offset,
        unsigned int entries) {
    ss.seekg(offset);
    std::vector<SymTableEntry> res(entries);
    for (SymTableEntry &i : res) {
        ss.read((char*) &i, sizeof(SymTableEntry));
    }
    return res;
}

```

Listing 12: **../src/main.cpp**

```cpp
#include "elfheader.h"
#include "elfsectiontable.h"
#include "elfsymbtable.h"
#include "typedefs.h"
#include "output_symbtable.h"
#include "output_code.h"

#include <iostream>
#include <algorithm>
#include <fstream>
#include <stdexcept>
#include <vector>
#include <string>
#include <iterator>
#include <cstdio>

int main(int argc, char* argv[]) {

    if (argc != 3) {
        std::cout << "Usage: rv3 <elf_input_file_name> <output_file_name>\n";
        return 1;
    }
    std::ifstream fin((std::string) argv[1], std::ios::binary);
    freopen(argv[2], "w", stdout);

    Header header = parse_header(fin);
    std::vector<SectionTableEntry> sectionTable =
        parse_section_table(fin, header.e_shoff, header.e_shnum);
    SectionTableEntry &shstrHeader = sectionTable[header.e_shstrndx];

    auto find_tab_by_name = [&] (std::string name) {
        return std::find_if(
            sectionTable.begin(),
            sectionTable.end(),
            [&] (SectionTableEntry& el) {
                return get_section_name(fin, shstrHeader, el) == name;
            }
        );
    };
    std::vector<SectionTableEntry>::iterator symtabHeader = find_tab_by_name(".symtab");
    std::vector<SectionTableEntry>::iterator strtabHeader = find_tab_by_name(".strtab");
    std::vector<SectionTableEntry>::iterator textHeader = find_tab_by_name(".text");
    std::vector<SymTableEntry> symtab =
        parse_symtable(fin, symtabHeader->sh_offset,
                symtabHeader->sh_size / symtabHeader->sh_entsize);
    std::string symbols(strtabHeader->sh_size, 0);
    fin.seekg(strtabHeader->sh_offset);
    fin.read(&symbols.front(), symbols.size());

    std::string code(textHeader->sh_size, 0);
    fin.seekg(textHeader->sh_offset);
    fin.read(&code.front(), code.size());
    print_code(code, symbols, symtab, textHeader->sh_addr);
    printf("\n");
    print_symtable(symtab, symbols);
    return 0;
}
```

Listing 13: **../src/output_code.cpp**

```cpp
#include "elfsymbtable.h"
#include "command_types.h"
#include "output_symbtable.h"

#include <string>
#include <algorithm>
#include <vector>

void print_code(
        const std::string& code,
        const std::string& symbols,
        std::vector<SymTableEntry> symtab,
        Elf32_Addr v_addr) {


    std::vector<SymTableEntry> functions;
```

```
    std::copy_if(symtab.begin(), symtab.end(), std::back_inserter(functions),
        [] (const SymTableEntry& el) {
            return type_by_info(el.st_info) == STT_FUNC;
        }
    );
    std::sort(functions.begin(), functions.end(),
            [](const SymTableEntry& a, const SymTableEntry& b) {
                return a.st_value < b.st_value;
            }
    );

    auto it = functions.begin();

    for (size_t i = 0; i < code.size(); i += 4, v_addr += 4) {
        if (it->st_value == v_addr) {
            printf("%08x   <%s>:\n",
                    it->st_value, format_name(it->st_name, symbols).c_str());
            it++;
        }
        std::string cmd_str(code.begin() + i, code.begin() + i + 4);
        const char* buff = cmd_str.c_str();
        Elf32_Word cmd = *((Elf32_Word*)(buff));
        printf("   %05x:\t%08x\t", v_addr, cmd);

        unsigned char opcode = cmd & 0x7f;
        InstructionType* parsed_cmd = nullptr;
        if (opcode == LUI || opcode == AUIPC) {
            parsed_cmd = new UType(cmd, v_addr, functions, symbols);
        } else if (opcode == ARITH) {
            parsed_cmd = new RType(cmd, v_addr, functions, symbols);
        } else if (opcode == STORE) {
            parsed_cmd = new SType(cmd, v_addr, functions, symbols);
        } else if (opcode == EX_CTR || opcode == ARITHI || opcode == LOAD ||
                opcode == JALR) {
            parsed_cmd = new IType(cmd, v_addr, functions, symbols);
        } else if (opcode == BRANCH) {
            parsed_cmd = new BType(cmd, v_addr, functions, symbols);
        } else if (opcode == JAL) {
            parsed_cmd = new JType(cmd, v_addr, functions, symbols);
        }

        if (parsed_cmd != nullptr) {
            parsed_cmd->print();
            delete parsed_cmd;
        } else {
            printf("unknown_instruction");
        }
        printf("\n");

    }
    return;
}
```

Listing 14: **../src/output_symbtable.cpp**

```
#include "output_symbtable.h"
#include "elfsymbtable.h"
#include "typedefs.h"

#include <stdexcept>
#include <vector>
#include <string>
#include <tuple>
#include <algorithm>
#include <exception>

std::string format_bind(const unsigned char& info) {
    unsigned char bind = (info>>4);
    if (bind == STB_LOCAL)
        return "LOCAL";
    else if (bind == STB_GLOBAL)
        return "GLOBAL";
    else if (bind == STB_WEAK)
        return "WEAK";
    else if (bind == STB_LOPROC)
        return "LOPROC";
```

```
    else if (bind == STB_HIPROC)
        return "HIPROC";
    throw std::invalid_argument("Undefined bind value");
}

std::string format_type(const unsigned char& info) {
    unsigned char type = (info&0xf);
    if (type == STT_NOTYPE)
        return "NOTYPE";
    else if (type == STT_OBJECT)
        return "OBJECT";
    else if (type == STT_FUNC)
        return "FUNC";
    else if (type == STT_SECTION)
        return "SECTION";
    else if (type == STT_FILE)
        return "FILE";
    else if (type == STT_LOPROC)
        return "LOPROC";
    else if (type == STT_HIPROC)
        return "HIPROC";
    throw std::invalid_argument("Undefined type value");
}

std::string format_index(const Elf32_Half& idx) {
    if (idx == SHN_ABS)
        return "ABS";
    else if (idx == SHN_UNDEF)
        return "UNDEF";
    else
        return std::to_string(idx);
}

std::string format_vis(const unsigned char& st_other) {
    int vis = (st_other&0x7);
    if (vis == STV_DEFAULT)
        return "DEFAULT";
    else if (vis == STV_INTERNAL)
        return "INTERNAL";
    else if (vis == STV_HIDDEN)
        return "HIDDEN";
    else if (vis == STV_EXPORTED)
        return "EXPORTED";
    else if (vis == STV_SINGLETON)
        return "SINGLETON";
    else if (vis == STV_ELIMINATE)
        return "ELIMINATE";
    else if (vis == STV_NUM)
        return "NUM";
    throw std::invalid_argument("Undefined visibility value");
}

std::string format_name(const Elf32_Word& name_offset, const std::string& symbols) {
    return
        std::string (
            symbols.begin() + name_offset,
            find(symbols.begin() + name_offset, symbols.end(), '\0')
        );
}

void print_symtable(
        const std::vector<SymTableEntry>& symtable,
        const std::string& symbols) {
    printf("Symbol Value              Size Type     Bind     Vis       Index Name\n");
    for (size_t i = 0; i < symtable.size(); ++i) {
        const SymTableEntry &el = symtable[i];
        std::string bind  = format_bind(el.st_info);
        std::string type  = format_type(el.st_info);
        std::string index(format_index(el.st_shndx));
        std::string name = format_name(el.st_name, symbols);
        std::string vis(format_vis(el.st_other));
        printf("[%4zu] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n",
                i, el.st_value, el.st_size, type.c_str(), bind.c_str(),
                vis.c_str(), index.c_str(), name.c_str());
    }
}
```