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.

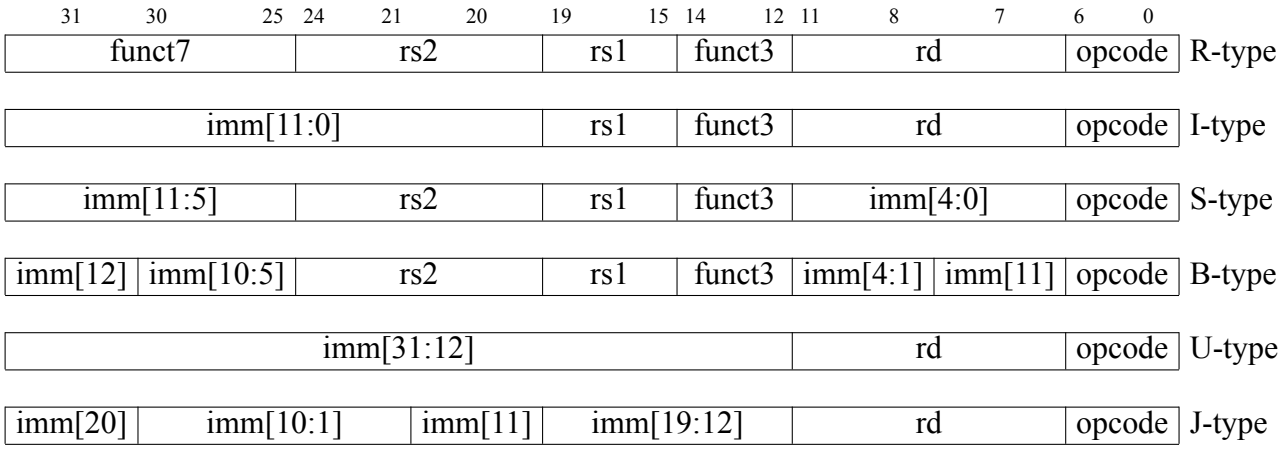

Рис. 1: Виды инструкций RISC-V

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

Инструкции управления исполнения

Инструкции для работы с памятью

Fence

Инструкции-подсказки исполнения

Расширение RV32M

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: **../src/elfheader.cpp**

```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 2: **../src/elfsectiontable.cpp**

```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 3: **../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 4: **../src/main.cpp**

```
#include ”elfheader.h”
#include ”elfsectiontable.h”
#include ”elfsymbtable.h”
#include ”typedefs.h”
#include ”output.h”

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

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

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

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

    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;
}

// Op codes
const unsigned char LUI    = 0b0110111;
const unsigned char AUIPC  = 0b0010111;
const unsigned char JAL    = 0b1101111;
const unsigned char JALR   = 0b1100111;
const unsigned char BRANCH = 0b1100011;
const unsigned char LOAD   = 0b0000011;
const unsigned char STORE  = 0b0100011;
const unsigned char ARITHI = 0b0010011;
const unsigned char ARITH  = 0b0110011;
const unsigned char FENCE  = 0b0001111;
const unsigned char EX_CTR = 0b1110011;

struct InstructionType {
    unsigned char opcode;
    InstructionType(const Elf32_Word& cmd) {
        opcode = cmd & 0x7f;
    }

    unsigned char get_rd(Elf32_Word x) {
        return get_reg(x, 7);
    }

    unsigned char get_rs1(Elf32_Word x) {
        return get_reg(x, 15);
    }

    unsigned char get_rs2(Elf32_Word x) {
        return get_reg(x, 20);
    }

    unsigned char get_funct3(Elf32_Word x) {
        return get_blk(x, 3, 12);
    }

    unsigned char get_funct7(Elf32_Word x) {
        return get_blk(x, 7, 25);
    }

    unsigned char get_reg(Elf32_Word x, unsigned char pos) {
        return get_blk(x, 5, pos);
    }

    unsigned char get_cmd(Elf32_Word x) {
        return get_blk(x, 7, 0);
    }

    Elf32_Word get_blk(Elf32_Word x, unsigned char len, unsigned char pos) {
        return (x>>pos)&((1ll << len) - 1);
    }

    virtual void print() = 0;

    virtual ~InstructionType() = default;
};

struct UType : public InstructionType {
    std::string cmd_name;
    unsigned char rd;
    Elf32_Word imm;

    UType(const Elf32_Word& cmd) : InstructionType(cmd) {
        rd = get_rd(cmd);
        imm = get_blk(cmd, 20, 12) << 12;
        cmd_name = (get_cmd(cmd) == LUI ? "lui" : "auipc");
    }
```

```
    void print() {
        printf("%s %02hhx %x", cmd_name.c_str(), rd, imm);
    }

    ~UType() = default;
};

struct RType : public InstructionType {
    std::string cmd_name;
    unsigned char rd;
    unsigned char rs1;
    unsigned char rs2;
    unsigned char funct3;
    unsigned char funct7;

    RType(const Elf32_Word& cmd) : InstructionType(cmd) {
        rd = get_rd(cmd);
        rs1 = get_rs1(cmd);
        rs2 = get_rs2(cmd);
        funct3 = get_funct3(cmd);
        funct7 = get_funct7(cmd);
        if ((funct7 & 1) == 0) {
            // R32I
            switch (funct3) {
                case 0b000: cmd_name = (funct7 ? "sub" : "add"); break;
                case 0b001: cmd_name = "sll"; break;
                case 0b010: cmd_name = "slt"; break;
                case 0b011: cmd_name = "sltu"; break;
                case 0b100: cmd_name = "xor"; break;
                case 0b101: cmd_name = (funct7 ? "srl" : "sra"); break;
                case 0b110: cmd_name = "or"; break;
                case 0b111: cmd_name = "and"; break;
            }
        } else {
            // R32M
            switch (funct3) {
                case 0b000: cmd_name = "mul";   break;
                case 0b001: cmd_name = "mulh";  break;
                case 0b010: cmd_name = "mulhsu";break;
                case 0b011: cmd_name = "mulhu"; break;
                case 0b100: cmd_name = "div";   break;
                case 0b101: cmd_name = "divu";  break;
                case 0b110: cmd_name = "rem";   break;
                case 0b111: cmd_name = "remu";  break;
            }
        }
    }

    void print() {
        printf("%s %02hhx %02hhx %02hhx", cmd_name.c_str(), rd, rs1, rs2);
    }

    ~RType() = default;
};

struct SType : public InstructionType {
    std::string cmd_name;
    unsigned char rs1;
    unsigned char rs2;
    unsigned char funct3;
    Elf32_Word imm;

    SType(const Elf32_Word& cmd) : InstructionType(cmd) {
        rs1 = get_rs1(cmd);
        rs2 = get_rs2(cmd);
        funct3 = get_funct3(cmd);
        imm = (get_blk(cmd, 7, 25) << 4) | get_blk(cmd, 4, 7);
        switch (funct3) {
            case 0b000: cmd_name = "sb"; break;
            case 0b001: cmd_name = "sh"; break;
            case 0b010: cmd_name = "sw"; break;
        }
    }

    void print() {
        printf("%s %02hhx,%d(%02hhx)", cmd_name.c_str(), rs1, imm, rs2);
    }

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

```
210
211 struct IType : public InstructionType {
212     std::string cmd_name;
213     unsigned char rs1;
214     unsigned char funct3;
215     unsigned char rd;
216     Elf32_Word imm;
217     bool is_load = false;
218     bool is_exec = false;
219
220     IType(const Elf32_Word& cmd) : InstructionType(cmd) {
221         rd = get_rd(cmd);
222         rs1 = get_rs1(cmd);
223         funct3 = get_funct3(cmd);
224         imm = get_blk(cmd, 12, 20);
225         if ((cmd & 0x7f) == EX_CTR) {
226             is_exec = true;
227             cmd_name = (imm & 1) ? "ebreak" : "ecall";
228         } else if (cmd & 0x40) {
229             is_load = true;
230             cmd_name = "jalr";
231         } else if (cmd & 0x10) {
232             switch (funct3) {
233                 case 0b000: cmd_name = "addi"; break;
234                 case 0b010: cmd_name = "slti"; break;
235                 case 0b011: cmd_name = "sltiu"; break;
236                 case 0b100: cmd_name = "xori"; break;
237                 case 0b110: cmd_name = "ori"; break;
238                 case 0b111: cmd_name = "andi"; break;
239                 case 0b001: cmd_name = "slli"; break;
240                 case 0b101: cmd_name = (imm & 0x20) ? "srai" : "srli"; break;
241             }
242         } else {
243             is_load = true;
244             switch (funct3) {
245                 case 0b000: cmd_name = "lb"; break;
246                 case 0b001: cmd_name = "lh"; break;
247                 case 0b010: cmd_name = "lw"; break;
248                 case 0b100: cmd_name = "lbu"; break;
249                 case 0b101: cmd_name = "lhu"; break;
250             }
251         }
252     }
253
254     void print() {
255         if (is_exec) {
256             printf("%s", cmd_name.c_str());
257         } else if (is_load) {
258             printf("%s %02hhx,%d(%02hhx)", cmd_name.c_str(), rs1, imm, rd);
259         } else {
260             printf("%s %02hhx %02hhx %d", cmd_name.c_str(), rd, rs1, imm);
261         }
262     }
263
264     ~IType() = default;
265 };
266
267 struct JType : public InstructionType {
268     std::string cmd_name;
269     unsigned char rd;
270     Elf32_Word imm;
271
272     JType(const Elf32_Word& cmd) : InstructionType(cmd) {
273         rd = get_rd(cmd);
274         imm = (get_blk(cmd,  1, 31) << 20) |
275               (get_blk(cmd, 10, 21) <<  1) |
276               (get_blk(cmd,  1, 20) << 11) |
277               (get_blk(cmd,  8, 12) << 12);
278         cmd_name = "jal";
279     }
280
281     void print() {
282         printf("%s %02hhx %x", cmd_name.c_str(), rd, imm);
283     }
284
285     ~JType() = default;
286 };
287
288 struct BType : public InstructionType {
289     std::string cmd_name;
```

```
    unsigned char rs1;
    unsigned char rs2;
    unsigned char funct3;
    Elf32_Word imm;

    BType(const Elf32_Word& cmd) : InstructionType(cmd) {
        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);
        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 print() {
        printf("%s %02hhx,%02hhx,%d", cmd_name.c_str(), rs2, rs1, imm);
    }

    ~BType() = default;
};

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));

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

        if (parsed_cmd != nullptr) {
            parsed_cmd->print();
```

```
            delete parsed_cmd;
        } else {
            printf("undef");
        }
        printf("\n");

    }
    return;
}
```

Listing 5: **../src/output.cpp**

```
#include "output.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());
    }
}
```

Listing 6: **../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 7: **../include/elfsectiontable.h**

```
#ifndef ELFSECTIONTABLE_GUARDS
#define ELFSECTIONTABLE_GUARDS

#include ”typedefs.h”
```

```
5
6   #include <vector>
7   #include <istream>
8
9   struct SectionTableEntry {
10      Elf32_Word    sh_name;
11      Elf32_Word    sh_type;
12      Elf32_Word    sh_flags;
13      Elf32_Addr    sh_addr;
14      Elf32_Off     sh_offset;
15      Elf32_Word    sh_size;
16      Elf32_Word    sh_link;
17      Elf32_Word    sh_info;
18      Elf32_Word    sh_addralign;
19      Elf32_Word    sh_entsize;
20  };
21
22
23  std::vector<SectionTableEntry> parse_section_table
24      (std::istream &ss, Elf32_Addr addr, unsigned int entries);
25
26  std::string get_section_name (
27          std::istream &ss,
28          const SectionTableEntry& shstrEntry,
29          const SectionTableEntry& section);
30
31  #endif
```

Listing 8: **../include/elfsymbtable.h**

```
1   #ifndef ELFSYMBTABLE_GUARDS
2   #define ELFSYMBTABLE_GUARDS
3
4   #include "typedefs.h"
5
6   #include <vector>
7   #include <istream>
8
9   struct SymTableEntry {
10      Elf32_Word    st_name;
11      Elf32_Addr    st_value;
12      Elf32_Word    st_size;
13      unsigned char st_info;
14      unsigned char st_other;
15      Elf32_Half    st_shndx;
16  };
17
18  std::vector<SymTableEntry> parse_symtable(
19          std::istream&,
20          Elf32_Addr offset,
21          unsigned int entries);
22
23
24  #endif
```

Listing 9: **../include/output.h**

```
1   #ifndef OUTPUT_GUARDS
2   #define OUTPUT_GUARDS
3
4   #include "typedefs.h"
5   #include "elfsymbtable.h"
6
7   #include <vector>
8   #include <string>
9
10  void print_symtable(const std::vector<SymTableEntry>&, const std::string&);
11
12  std::string format_bind(const unsigned char& info);
13
14  std::string format_type(const unsigned char& info);
15
```

```
16  std::string format_index(const Elf32_Half& idx);
17
18  std::string format_vis(const unsigned char& st_other);
19
20  std::string format_name(const Elf32_Word& name_offset, const std::string& symbols);
21
22  inline unsigned char type_by_info(const unsigned char& info) {
23      return (info&0xf);
24  }
25
26  inline unsigned char bind_by_info(const unsigned char& info) {
27      return (info>>4);
28  }
29
30  #endif
```

Listing 10: **../include/typedefs.h**

```
1   #ifndef TYPEDEFS_GUARD
2   #define TYPEDEFS_GUARD
3
4   typedef unsigned int   Elf32_Addr;
5   typedef unsigned short Elf32_Half;
6   typedef unsigned int   Elf32_Off;
7   typedef int            Elf32_Sword;
8   typedef unsigned int   Elf32_Word;
9
10  const unsigned short EI_NIDENT = 16;
11
12  const unsigned int MAX_NAME = 256;
13
14  const unsigned short SHN_ABS = 0xfff1;
15
16  const unsigned short SHN_UNDEF = 0;
17
18  #define STB_LOCAL      0
19  #define STB_GLOBAL     1
20  #define STB_WEAK       2
21  #define STB_NUM        3
22
23  #define STB_LOPROC    13
24  #define STB_HIPROC    15
25
26  #define STT_NOTYPE     0
27  #define STT_OBJECT     1
28  #define STT_FUNC       2
29  #define STT_SECTION    3
30  #define STT_FILE       4
31  #define STT_COMMON     5
32  #define STT_TLS        6
33  #define STT_NUM        7
34
35  #define STT_LOPROC    13
36  #define STT_HIPROC    15
37
38
39  #define STV_DEFAULT    0
40  #define STV_INTERNAL   1
41  #define STV_HIDDEN     2
42  #define STV_PROTECTED  3
43  #define STV_EXPORTED   4
44  #define STV_SINGLETON  5
45  #define STV_ELIMINATE  6
46
47  #define STV_NUM        7
48
49
50  #endif
```