

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 длиной 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.

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

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 исполнением

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 уровень 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 атрибутов.
 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(0xffff1) — 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/command_types.cpp

```

1  #include "command_types.h"
2
3  #include "typedefs.h"
4  #include <string>
5  #include <vector>
6  #include <stdexcept>
7
8  void extend_sign(int& x, int sz) {
9      for (; sz < 32; ++sz) {
10         if (x & (1ll << sz)) {
11             x |= (1ll << (sz + 1));
12         }
13     }
14 }
15
16 std::string prettify_reg(unsigned char reg) {
17     if (reg >= 32) throw std::logic_error("Accessing to non-existent register");
18     std::vector<std::string> names = {
19         "zero", "ra", "sp", "gp", "tp", "t0", "t1", "t2", "s0", "s1",
20     };
21     for (int i = 0; i <= 7; ++i) {
22         names.push_back("a" + std::to_string(i));
23     }
24     for (int i = 2; i <= 11; ++i) {
25         names.push_back("s" + std::to_string(i));
26     }
27     for (int i = 3; i <= 6; ++i) {
28         names.push_back("t" + std::to_string(i));
29     }
30     return names[reg];
31 }
32
33 InstructionType::InstructionType(const Elf32_Word& cmd) {
34     opcode = cmd & 0x7f;
35 }
36
37 unsigned char InstructionType::get_rd(Elf32_Word x) {
38     return get_reg(x, 7);
39 }
40
41 unsigned char InstructionType::get_rs1(Elf32_Word x) {
42     return get_reg(x, 15);
43 }
44
45 unsigned char InstructionType::get_rs2(Elf32_Word x) {
46     return get_reg(x, 20);
47 }
48
49 unsigned char InstructionType::get_funct3(Elf32_Word x) {
50     return get_blk(x, 3, 12);
51 }
52
53 unsigned char InstructionType::get_funct7(Elf32_Word x) {
54     return get_blk(x, 7, 25);
55 }
56
57 unsigned char InstructionType::get_reg(Elf32_Word x, unsigned char pos) {
58     return get_blk(x, 5, pos);
59 }
60
61 unsigned char InstructionType::get_cmd(Elf32_Word x) {
62     return get_blk(x, 7, 0);
63 }
64
65 Elf32_Word InstructionType::get_blk(Elf32_Word x, unsigned char len, unsigned char pos) {
66     return (x>>pos)&((1ll << len) - 1);
67 }
68
69 UType::UType(const Elf32_Word& cmd) : InstructionType(cmd) {
70     rd = get_rd(cmd);
71     imm = get_blk(cmd, 20, 12) << 12;
72     cmd_name = (get_cmd(cmd) == LUI ? "lui" : "auipc");
73 }
74
75 void UType::print() {
76     printf("%7s\t%s, %x", cmd_name.c_str(), prettify_reg(rd).c_str(), imm);
77 }
78
79 RType::RType(const Elf32_Word& cmd) : InstructionType(cmd) {

```

```

81     rd = get_rd(cmd);
82     rs1 = get_rs1(cmd);
83     rs2 = get_rs2(cmd);
84     funct3 = get_funct3(cmd);
85     funct7 = get_funct7(cmd);
86     if ((funct7 & 1) == 0) {
87         // R32I
88         switch (funct3) {
89             case 0b000: cmd_name = (funct7 ? "sub" : "add"); break;
90             case 0b001: cmd_name = "sll"; break;
91             case 0b010: cmd_name = "slt"; break;
92             case 0b011: cmd_name = "sltu"; break;
93             case 0b100: cmd_name = "xor"; break;
94             case 0b101: cmd_name = (funct7 ? "srl" : "sra"); break;
95             case 0b110: cmd_name = "or"; break;
96             case 0b111: cmd_name = "and"; break;
97         }
98     } else {
99         // R32M
100        switch (funct3) {
101            case 0b000: cmd_name = "mul"; break;
102            case 0b001: cmd_name = "mulh"; break;
103            case 0b010: cmd_name = "mulhsu"; break;
104            case 0b011: cmd_name = "mulhu"; break;
105            case 0b100: cmd_name = "div"; break;
106            case 0b101: cmd_name = "divu"; break;
107            case 0b110: cmd_name = "rem"; break;
108            case 0b111: cmd_name = "remu"; break;
109        }
110    }
111 }
112
113 void RType::print() {
114     printf("%7s\t%s, %s, %s", cmd_name.c_str(), prettify_reg(rd).c_str(),
115           prettify_reg(rs1).c_str(), prettify_reg(rs2).c_str());
116 }
117
118
119 SType::SType(const Elf32_Word& cmd) : InstructionType(cmd) {
120     rs1 = get_rs1(cmd);
121     rs2 = get_rs2(cmd);
122     funct3 = get_funct3(cmd);
123     imm = (get_blk(cmd, 7, 25) << 4) | get_blk(cmd, 4, 7);
124     extend_sign(imm, 11);
125     switch (funct3) {
126         case 0b000: cmd_name = "sb"; break;
127         case 0b001: cmd_name = "sh"; break;
128         case 0b010: cmd_name = "sw"; break;
129     }
130 }
131
132 void SType::print() {
133     printf("%7s\t%s, %d(%s)", cmd_name.c_str(), prettify_reg(rs2).c_str(), imm,
134           prettify_reg(rs1).c_str());
135 }
136
137 IType::IType(const Elf32_Word& cmd) : InstructionType(cmd) {
138     rd = get_rd(cmd);
139     rs1 = get_rs1(cmd);
140     funct3 = get_funct3(cmd);
141     imm = get_blk(cmd, 12, 20);
142     extend_sign(imm, 11);
143     if ((cmd & 0x7f) == EX_CTR) {
144         is_exec = true;
145         cmd_name = (imm & 1) ? "ebreak" : "ecall";
146     } else if (cmd & 0x40) {
147         is_load = true;
148         cmd_name = "jalr";
149     } else if (cmd & 0x10) {
150         switch (funct3) {
151             case 0b000: cmd_name = "addi"; break;
152             case 0b010: cmd_name = "slti"; break;
153             case 0b011: cmd_name = "sltiu"; break;
154             case 0b100: cmd_name = "xori"; break;
155             case 0b110: cmd_name = "ori"; break;
156             case 0b111: cmd_name = "andi"; break;
157             case 0b001: cmd_name = "slli"; break;
158             case 0b101: cmd_name = (imm & 0x20) ? "srai" : "srli"; break;
159         }
160     } else {

```

```

161         is_load = true;
162         switch (funct3) {
163             case 0b000: cmd_name = "lb"; break;
164             case 0b001: cmd_name = "lh"; break;
165             case 0b010: cmd_name = "lw"; break;
166             case 0b100: cmd_name = "lbu"; break;
167             case 0b101: cmd_name = "lhu"; break;
168         }
169     }
170 }
171
172 void IType::print() {
173     if (is_exec) {
174         printf("%7s", cmd_name.c_str());
175     } else if (is_load) {
176         printf("%7s\t%s, %d(%s)", cmd_name.c_str(), prettify_reg(rd).c_str(),
177             imm, prettify_reg(rs1).c_str());
178     } else {
179         // ARITHI
180         printf("%7s\t%s, %s, %d", cmd_name.c_str(), prettify_reg(rd).c_str(),
181             prettify_reg(rs1).c_str(), imm);
182     }
183 }
184
185 JType::JType(const Elf32_Word& cmd) : InstructionType(cmd) {
186     rd = get_rd(cmd);
187     imm = (get_blk(cmd, 1, 31) << 20) |
188         (get_blk(cmd, 10, 21) << 1) |
189         (get_blk(cmd, 1, 20) << 11) |
190         (get_blk(cmd, 8, 12) << 12);
191     cmd_name = "jal";
192 }
193
194 void JType::print() {
195     printf("%7s\t%s, %x", cmd_name.c_str(), prettify_reg(rd).c_str(), imm);
196 }
197
198 BType::BType(const Elf32_Word& cmd) : InstructionType(cmd) {
199     rs1 = get_rs1(cmd);
200     rs2 = get_rs2(cmd);
201     funct3 = get_funct3(cmd);
202     imm = (get_blk(cmd, 1, 31) << 12) |
203         (get_blk(cmd, 6, 25) << 5) |
204         (get_blk(cmd, 4, 8) << 1) |
205         (get_blk(cmd, 1, 7) << 11);
206     switch (funct3) {
207         case 0b000: cmd_name = "beq"; break;
208         case 0b001: cmd_name = "bne"; break;
209         case 0b100: cmd_name = "blt"; break;
210         case 0b101: cmd_name = "bge"; break;
211         case 0b110: cmd_name = "bltu"; break;
212         case 0b111: cmd_name = "bgeu"; break;
213     }
214 }
215
216 void BType::print() {
217     printf("%7s\t%s, %s, %x", cmd_name.c_str(), prettify_reg(rs2).c_str(),
218         prettify_reg(rs1).c_str(), imm);
219 }
220

```

Listing 2: ../src/elfheader.cpp

```

1  #include "elfheader.h"
2  #include "elfsectiontable.h"
3
4  #include <exception>
5
6  Header parse_header(std::istream &ss) {
7
8      const unsigned int expected_magic = 0x464C457F;
9      const unsigned char expected_arch = 1;
10     const unsigned char expected_endian = 1;
11     const unsigned char SysV_ABI_code = 0x00;
12     const unsigned short RiscV_ISA_code = 0x00F3;
13     const unsigned short header_expected_size = 52;

```



```

14
15     Header header;
16
17     ss.read((char*)&header, sizeof(header));
18     uint32_t magic = *(uint32_t*)header.e_ident;
19     if (magic != expected_magic) {
20         throw std::invalid_argument("Invalid ELF magic bits");
21     }
22     if (header.e_ident[5] != expected_arch) {
23         throw std::invalid_argument("Unsupported architecture. 32bit only");
24     }
25     if (header.e_ident[6] != expected_endian) {
26         throw std::invalid_argument("Big endian is no supported");
27     }
28     if (header.e_ident[7] != SysV_ABI_code) {
29         throw std::invalid_argument("Unsupported ABI. System V only");
30     }
31     if (header.e_machine != RiscV_ISA_code) {
32         throw std::invalid_argument("Unsupported ISA. RISC-V only");
33     }
34     if (header.e_ehsize != header_expected_size) {
35         throw std::invalid_argument("Illegal elf header size");
36     }
37     if (header.e_shentsize != sizeof(SectionTableEntry)) {
38         throw std::invalid_argument("Unexpected section header table entry size");
39     }
40     return header;
41 };
42

```

Listing 3: ../src/elfsectiontable.cpp

```

1  #include "elfsectiontable.h"
2  #include "typedefs.h"
3
4  #include <vector>
5  #include <istream>
6
7  std::vector<SectionTableEntry> parse_section_table (
8      std::istream &ss,
9      Elf32_Addr addr,
10     unsigned int entries
11 ) {
12     ss.seekg(addr);
13     std::vector<SectionTableEntry> res(entries);
14     for (SectionTableEntry& i : res) {
15         ss.read((char*)&i, sizeof(SectionTableEntry));
16     }
17     return res;
18 }
19
20 std::string get_section_name (
21     std::istream &ss,
22     const SectionTableEntry& shstrEntry,
23     const SectionTableEntry& section)
24 {
25     ss.seekg(shstrEntry.sh_offset + section.sh_name);
26     char res[MAX_NAME];
27     ss.getline(res, MAX_NAME, '\0');
28     return (std::string) res;
29 };
30
31

```

Listing 4: ../src/elfsymtable.cpp

```

1  #include "elfsymtable.h"
2  #include "elfsectiontable.h"
3  #include "typedefs.h"
4
5  #include <istream>
6  #include <vector>

```

```

7
8 std::vector<SymTableEntry> parse_symtable(
9     std::istream &ss,
10     Elf32_Addr offset,
11     unsigned int entries) {
12     ss.seekg(offset);
13     std::vector<SymTableEntry> res(entries);
14     for (SymTableEntry &i : res) {
15         ss.read((char*) &i, sizeof(SymTableEntry));
16     }
17     return res;
18 }
19

```

Listing 5: ../src/main.cpp

```

1  #include "elfheader.h"
2  #include "elfsectiontable.h"
3  #include "elfsymtable.h"
4  #include "typedefs.h"
5  #include "output_symtable.h"
6  #include "output_code.h"
7
8  #include <iostream>
9  #include <algorithm>
10 #include <fstream>
11 #include <stdexcept>
12 #include <vector>
13 #include <string>
14 #include <iterator>
15
16 int main(int argc, char* argv[]) {
17
18     if (argc != 2) {
19         std::cout << "Usage: rv3 <elf_input_file_name> <output_file_name>\n";
20         return 1;
21     }
22     std::ifstream fin((std::string) argv[1], std::ios::binary);
23
24     Header header = parse_header(fin);
25     std::vector<SectionTableEntry> sectionTable =
26         parse_section_table(fin, header.e_shoff, header.e_shnum);
27     SectionTableEntry &shstrHeader = sectionTable[header.e_shstrndx];
28
29     auto find_tab_by_name = [&] (std::string name) {
30         return std::find_if(
31             sectionTable.begin(),
32             sectionTable.end(),
33             [&] (SectionTableEntry& el) {
34                 return get_section_name(fin, shstrHeader, el) == name;
35             }
36         );
37     };
38     std::vector<SectionTableEntry>::iterator symtabHeader = find_tab_by_name(".symtab");
39     std::vector<SectionTableEntry>::iterator strtabHeader = find_tab_by_name(".strtab");
40     std::vector<SectionTableEntry>::iterator textHeader = find_tab_by_name(".text");
41     std::vector<SymTableEntry> symtab =
42         parse_symtable(fin, symtabHeader->sh_offset,
43             symtabHeader->sh_size / symtabHeader->sh_entsize);
44     std::string symbols(strtabHeader->sh_size, 0);
45     fin.seekg(strtabHeader->sh_offset);
46     fin.read(&symbols.front(), symbols.size());
47
48     std::string code(textHeader->sh_size, 0);
49     fin.seekg(textHeader->sh_offset);
50     fin.read(&code.front(), code.size());
51     print_code(code, symbols, symtab, textHeader->sh_addr);
52     printf("\n");
53     print_symtable(symtab, symbols);
54     return 0;
55 }

```

Listing 6: ../src/output_code.cpp

```

1  #include "elfsymtable.h"
2  #include "command_types.h"
3  #include "output_symtable.h"
4
5  #include <string>
6  #include <algorithm>
7  #include <vector>
8
9  void print_code(
10     const std::string& code,
11     const std::string& symbols,
12     std::vector<SymTableEntry> symtab,
13     Elf32_Addr v_addr) {
14
15
16     std::vector<SymTableEntry> functions;
17     std::copy_if(symtab.begin(), symtab.end(), std::back_inserter(functions),
18         [](const SymTableEntry& el) {
19             return type_by_info(el.st_info) == STT_FUNC;
20         }
21     );
22     std::sort(functions.begin(), functions.end(),
23         [](const SymTableEntry& a, const SymTableEntry& b) {
24             return a.st_value < b.st_value;
25         }
26     );
27
28     auto it = functions.begin();
29
30     for (size_t i = 0; i < code.size(); i += 4, v_addr += 4) {
31         if (it->st_value == v_addr) {
32             printf("%08x <%=s>:\n",
33                 it->st_value, format_name(it->st_name, symbols).c_str());
34             it++;
35         }
36         std::string cmd_str(code.begin() + i, code.begin() + i + 4);
37         const char* buff = cmd_str.c_str();
38         Elf32_Word cmd = *((Elf32_Word*)(buff));
39         printf("  %05x:\t%08x\t", v_addr, cmd);
40
41         unsigned char opcode = cmd & 0x7f;
42         InstructionType* parsed_cmd = nullptr;
43         if (opcode == LUI || opcode == AUIPC) {
44             parsed_cmd = new UType(cmd);
45         } else if (opcode == ARITH) {
46             parsed_cmd = new RType(cmd);
47         } else if (opcode == STORE) {
48             parsed_cmd = new SType(cmd);
49         } else if (opcode == EX_CTR || opcode == ARITHI || opcode == LOAD ||
50             opcode == JALR) {
51             parsed_cmd = new IType(cmd);
52         } else if (opcode == BRANCH) {
53             parsed_cmd = new BType(cmd);
54         } else if (opcode == JAL) {
55             parsed_cmd = new JType(cmd);
56         }
57
58         if (parsed_cmd != nullptr) {
59             parsed_cmd->print();
60             delete parsed_cmd;
61         } else {
62             printf("unknown_instruction");
63         }
64         printf("\n");
65     }
66     return;
67 }
68

```

Listing 7: ../src/output_symtable.cpp

```

1  #include "output_symtable.h"
2  #include "elfsymtable.h"
3  #include "typedefs.h"
4
5  #include <stdexcept>

```

```

6  #include <vector>
7  #include <string>
8  #include <tuple>
9  #include <algorithm>
10 #include <exception>
11
12 std::string format_bind(const unsigned char& info) {
13     unsigned char bind = (info>>4);
14     if (bind == STB_LOCAL)
15         return "LOCAL";
16     else if (bind == STB_GLOBAL)
17         return "GLOBAL";
18     else if (bind == STB_WEAK)
19         return "WEAK";
20     else if (bind == STB_LOPROC)
21         return "LOPROC";
22     else if (bind == STB_HIPROC)
23         return "HIPROC";
24     throw std::invalid_argument("Undefined bind value");
25 }
26
27 std::string format_type(const unsigned char& info) {
28     unsigned char type = (info&0xf);
29     if (type == STT_NOTYPE)
30         return "NOTYPE";
31     else if (type == STT_OBJECT)
32         return "OBJECT";
33     else if (type == STT_FUNC)
34         return "FUNC";
35     else if (type == STT_SECTION)
36         return "SECTION";
37     else if (type == STT_FILE)
38         return "FILE";
39     else if (type == STT_LOPROC)
40         return "LOPROC";
41     else if (type == STT_HIPROC)
42         return "HIPROC";
43     throw std::invalid_argument("Undefined type value");
44 }
45
46 std::string format_index(const Elf32_Half& idx) {
47     if (idx == SHN_ABS)
48         return "ABS";
49     else if (idx == SHN_UNDEF)
50         return "UNDEF";
51     else
52         return std::to_string(idx);
53 }
54
55 std::string format_vis(const unsigned char& st_other) {
56     int vis = (st_other&0x7);
57     if (vis == STV_DEFAULT)
58         return "DEFAULT";
59     else if (vis == STV_INTERNAL)
60         return "INTERNAL";
61     else if (vis == STV_HIDDEN)
62         return "HIDDEN";
63     else if (vis == STV_EXPORTED)
64         return "EXPORTED";
65     else if (vis == STV_SINGLETON)
66         return "SINGLETON";
67     else if (vis == STV_ELIMINATE)
68         return "ELIMINATE";
69     else if (vis == STV_NUM)
70         return "NUM";
71     throw std::invalid_argument("Undefined visibility value");
72 }
73
74 std::string format_name(const Elf32_Word& name_offset, const std::string& symbols) {
75     return
76         std::string (
77             symbols.begin() + name_offset,
78             find(symbols.begin() + name_offset, symbols.end(), '\0')
79         );
80 }
81
82 void print_symtable(
83     const std::vector<SymTableEntry>& symtable,
84     const std::string& symbols) {
85     printf("Symbol Value          Size Type      Bind      Vis      Index Name\n");

```

```

86     for (size_t i = 0; i < symtable.size(); ++i) {
87         const SymTableEntry &el = symtable[i];
88         std::string bind = format_bind(el.st_info);
89         std::string type = format_type(el.st_info);
90         std::string index(format_index(el.st_shndx));
91         std::string name = format_name(el.st_name, symbols);
92         std::string vis(format_vis(el.st_other));
93         printf("[%4zu] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n",
94             i, el.st_value, el.st_size, type.c_str(), bind.c_str(),
95             vis.c_str(), index.c_str(), name.c_str());
96     }
97 }
98

```

Listing 8: ../include/command_types.h

```

1  #ifndef COMMAND_TYPES_GUARD
2  #define COMMAND_TYPES_GUARD
3
4  #include "typedefs.h"
5  #include <string>
6
7  struct InstructionType {
8      unsigned char opcode;
9      InstructionType(const Elf32_Word& cmd);
10
11      unsigned char get_rd(Elf32_Word x);
12
13      unsigned char get_rs1(Elf32_Word x);
14
15      unsigned char get_rs2(Elf32_Word x);
16
17      unsigned char get_funct3(Elf32_Word x);
18
19      unsigned char get_funct7(Elf32_Word x);
20
21      unsigned char get_reg(Elf32_Word x, unsigned char pos);
22
23      unsigned char get_cmd(Elf32_Word x);
24
25      Elf32_Word get_blk(Elf32_Word x, unsigned char len, unsigned char pos);
26
27      virtual void print() = 0;
28
29      virtual ~InstructionType() = default;
30 };
31
32 struct UType : public InstructionType {
33     std::string cmd_name;
34     unsigned char rd;
35     int imm;
36
37     UType(const Elf32_Word& cmd);
38
39     void print();
40
41     ~UType() = default;
42 };
43
44 struct RType : public InstructionType {
45     std::string cmd_name;
46     unsigned char rd;
47     unsigned char rs1;
48     unsigned char rs2;
49     unsigned char funct3;
50     unsigned char funct7;
51
52     RType(const Elf32_Word& cmd);
53
54     void print();
55
56     ~RType() = default;
57 };
58
59 struct SType : public InstructionType {
60     std::string cmd_name;

```

```

61     unsigned char rs1;
62     unsigned char rs2;
63     unsigned char funct3;
64     int imm;
65
66     SType(const Elf32_Word& cmd);
67
68     void print();
69
70     ~SType() = default;
71 };
72
73 struct IType : public InstructionType {
74     std::string cmd_name;
75     unsigned char rs1;
76     unsigned char funct3;
77     unsigned char rd;
78     int imm;
79     bool is_load = false;
80     bool is_exec = false;
81
82     IType(const Elf32_Word& cmd);
83
84     void print();
85
86     ~IType() = default;
87 };
88
89 struct JType : public InstructionType {
90     std::string cmd_name;
91     unsigned char rd;
92     Elf32_Word imm;
93
94     JType(const Elf32_Word& cmd);
95
96     void print();
97
98     ~JType() = default;
99 };
100
101 struct BType : public InstructionType {
102     std::string cmd_name;
103     unsigned char rs1;
104     unsigned char rs2;
105     unsigned char funct3;
106     Elf32_Word imm;
107
108     BType(const Elf32_Word& cmd);
109
110     void print();
111
112     ~BType() = default;
113 };
114
115 #endif

```

Listing 9: ../include/elfheader.h

```

1  #ifndef ELFHEADER_GUARDS
2  #define ELFHEADER_GUARDS
3
4  #include "typedefs.h"
5
6  #include <istream>
7
8  struct Header {
9      unsigned char e_ident[EI_NIDENT];
10     Elf32_Half e_type;
11     Elf32_Half e_machine;
12     Elf32_Word e_version;
13     Elf32_Addr e_entry;
14     Elf32_Off e_phoff;
15     Elf32_Off e_shoff;
16     Elf32_Word e_flags;
17     Elf32_Half e_ehsize;
18     Elf32_Half e_phentsize;

```

```

19     Elf32_Half    e_phnum;
20     Elf32_Half    e_shentsize;
21     Elf32_Half    e_shnum;
22     Elf32_Half    e_shstrndx;
23 };
24
25 Header parse_header(std::istream &ss);
26
27 #endif

```

Listing 10: ../include/elfsectiontable.h

```

1  #ifndef ELFSECTIONTABLE_GUARDS
2  #define ELFSECTIONTABLE_GUARDS
3
4  #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 11: ../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 12: ../include/output_code.h

```
1  #ifndef OUTPUTCODE_GUARDS
2  #define OUTPUTCODE_GUARDS
3
4  #include "typedefs.h"
5  #include "elfsymtable.h"
6
7  #include <string>
8  #include <vector>
9
10 void print_code(
11     const std::string& code,
12     const std::string& symbols,
13     std::vector<SymTableEntry> symtab,
14     Elf32_Addr v_addr
15 );
16
17
18 #endif
```

Listing 13: ../include/output_symtable.h

```
1  #ifndef OUTPUT_GUARDS
2  #define OUTPUT_GUARDS
3
4  #include "typedefs.h"
5  #include "elfsymtable.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 14: ../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 = 0xffff1;
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  // 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

```