

Лабораторная работа №3	Б10	2022
ISA	Хорохорин Андрей Сергеевич	

## Цель работы

Знакомство с архитектурой набора команд RISC-V

## Инструментарий

Секция генерируется автоматически при компиляции

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.

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		

Заметим, что типы U, J и B, S почти совпадают. Они существуют для кодирования команд, где данные *imm* точно делится на 2 (так как размер любой команды кратен 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 названия 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 бинарной 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 до-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) {
    sz = 32 - sz;
    x = (x << sz) >> sz;
}
```

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:

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
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:

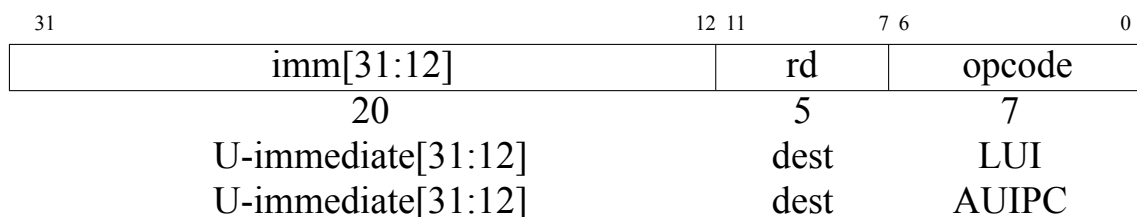
- MUL — умножение в результате которого младшие биты результата будут помещены в rd
- MULH — аналог команды MUL, но в rd будут записаны старшие биты умножения в предположении, что оба числа знаковые.
- MULHSU — аналог MULH, но предполагает, что оба числа беззнаковые.
- MULHU — аналог MULH, но предполагает, 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:



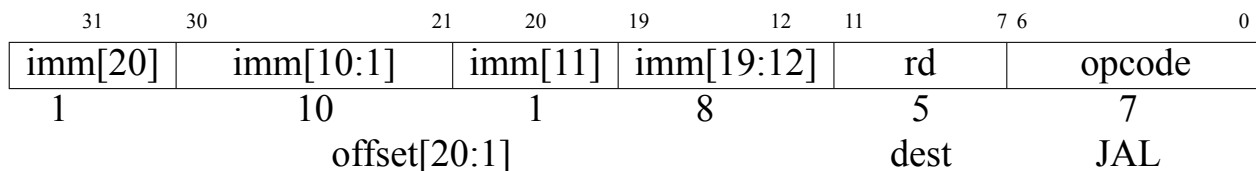
LUI rd imm записывает в регистр rd значение константы imm. Стоит обратить внимание на то, что команда задаёт у imm 20 старших бита, остальные 12 младших байт заполняются нулями. Данная инструкция необходима для быстрой загрузки в регистр 32 битных констант явным образом.

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.

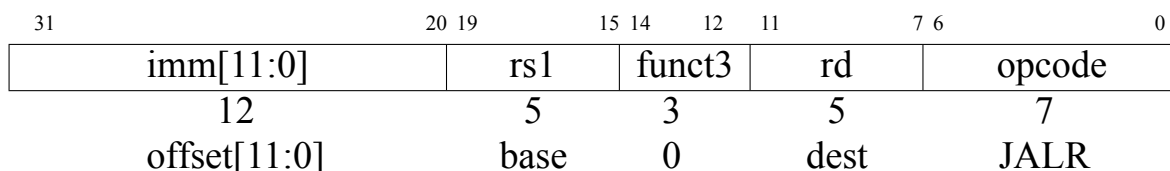
## Инструкции управления исполнением

**Безусловные переходы** В данной архитектуре реализовано 2 безусловных перехода: JAL и JALR. Первый из которых кодируется в J type, а второй в I type.

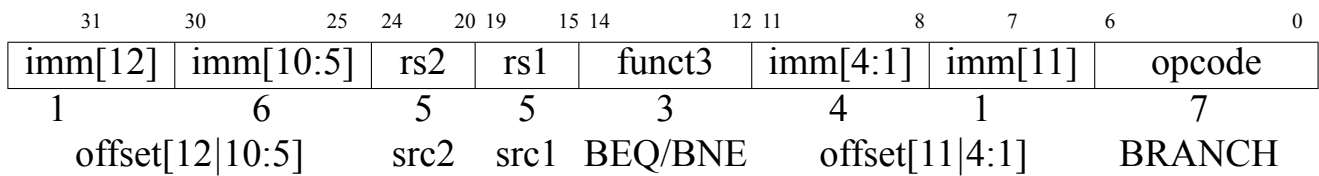
JAL работает следующим образом. В imm с 1 по 20 бит кодируется смещение относительно адреса текущей команды pc(смещение конечно же со знаком, нулевой бит imm равен 0, так как размер наших команд кратен 2, а значит последний бит можно сэкономить), после чего совершается переход на указанное смещение imm 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 отличие от безусловных в нашей архитектуре все относительно текущей инструкции и кодируются в B type.

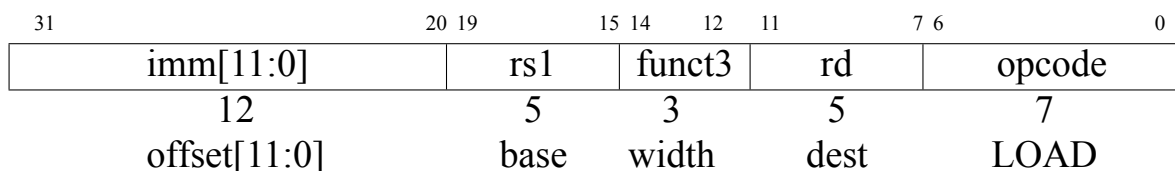


К тому же, как следствие того, что приходится кодировать ещё целых два регистра, на которые будет накладываться условие перехода, диапазон смещения достаточно маленький, всего  $\pm 4\text{KiB}$ . Но из-за того, что все переходы относительно текущей позиции, то можно не кодировать последний бит смещения.

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

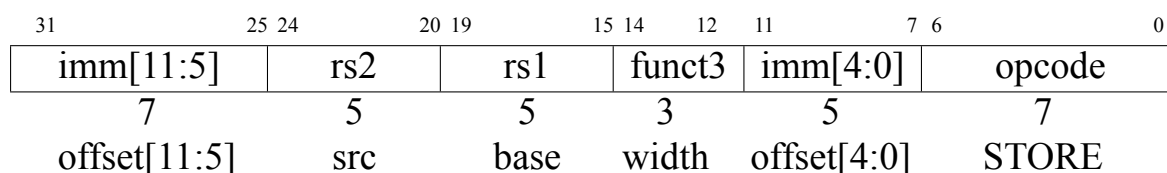
### Load



- LW — загружает из оперативной памяти 32 битное число по адресу  $base + offset$  в регистр rd.
- LH — загружает из оперативной памяти 16 битное число по адресу  $base + offset$ , а затем дополняет до 32 битного сохраняя и знак и значение по модулю и записывает результат в rd.
- LHU — загружает из оперативной памяти 16 битное число по адресу  $base + offset$ , а затем дополняет до 32 путём дописывания слева нулей и записывает результат в rd.

- LB — 8 битный аналог LH.
- LBU — 8 битный аналог LHU.

**Store** Все 3 Store инструкции делают одно и то же: берут некоторое число младших бит из регистра rs2 и помещают их в память по адресу offset + base, где base берётся из регистра rs1, а offset передаётся в виде константы. SW, SH и SB соответствуют 32, 16 и 8 битам.



## Fence

Не смотря на то, что в моём декомпиляторе данная команда не поддерживается, считаю нужным сказать пару слов про её работу. Данная команда необходимая для синхронизации потоков. Когда на процессоре работает несколько потоков, то для повышения производительности операция записи какого-то потока в память может быть видна остальным потокам не сразу. Но, если добавить после некоторых операций с памятью FENCE, то все последующие операции с памятью данного потока станут видны остальным потокам не раньше, чем команды отправленные до FENCE.

## Инструкции-подсказки исполнения

В отличии от всех вышеперечисленных инструкций, поведение которых указано в спецификации, функцию всех перечисленных далее инструкций сторона, которая реализует ISA RISC-V может задать любым удобным для себя образом.

Кодируются эти команды весьма странно: если в одной из стандартных команд rd=x0, то она воспринимается не как обычно, а как команда-подсказка. Тут стоит напомнить, что x0 это регистр, содержимое которого всего равно 0, поэтому реальной функциональности кодирования от такого трюка мы не потеряем.

Видимо, в будущем планируется внедрение подсказок в ISA RISC-V, но в текущий момент они отсутствуют.

## 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 выставять 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 размер 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 таблице 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 таблице 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.

- **.strtab** Таблица хранит все строковые значения, которые есть в elf файле в виде записанных в памяти подряд строк, каждая из которых оканчивается нулевым символом.

## Описание работы написанного кода

В начале написания лабораторной мной было принято неверное решение: писать на C++ вместо C. В данном случае, это привело к смешению C и C++ стилей и на код без слёз не взглянешь.

Файловая структура:

- **elfsectiontable.cpp**, **elfsymtable.cpp** и **elfheader.cpp** и их заголовочные файлы хранят в себе структуры и вспомогательные функции для считывания elf файлов.
- **output\_code.cpp** и **output\_symb.cpp** содержат непосредственно функции вывода содержания секций **.text** и **.symtab** соответственно.
- **command\_types.cpp** описывает каждый из типов команд и то, как их распознавать и выводить, а также общий интерфейс для типа команды. Всё это используется в **output\_code.cpp**
- **main.cpp** — точка входа в программу. Помимо этого занимается некоторой обработкой данных перед тем, как передать их в **output\_code.cpp**.

Отдельно стоит пояснить работу функции **parse\_code** из файла **output\_code.cpp**. В данной функции из таблицы меток выделяются только метки с типом "FUNC", которые затем упорядочиваются по возрастанию их адреса. Это позволяет при обработке команд из секции **.text** использовать метод двух указателей для того, чтобы выводить места начала функций-меток.

Внутри цикла очередная команда обрабатывается следующим образом: создаётся ссылка на абстрактный класс **InstructionType**, в котором реализованы общие функции для всех типов команд(таких получилось достаточно много, благодаря удобному стандарту RISC). Далее по **opcode** определяется тип команды и создаётся соответствующая этому типу класс-наследник **InstructionType**, в конструкторе которого уже идёт разбор команды на части. В конце вызывается **print** от данного объекта, причём эта функция объявлена виртуальной для **InstructionType**, а значит реализуется для каждого наследника в отдельности. Сами классы типов команд задаются в **command\_types.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, v_addr, functions, symbols);
45         } else if (opcode == ARITH) {
46             parsed_cmd = new RType(cmd, v_addr, functions, symbols);
47         } else if (opcode == STORE) {
48             parsed_cmd = new SType(cmd, v_addr, functions, symbols);
49         } else if (opcode == EX_CTR || opcode == ARITHI || opcode == LOAD ||
50             opcode == JALR) {
51             parsed_cmd = new IType(cmd, v_addr, functions, symbols);
52         } else if (opcode == BRANCH) {
53             parsed_cmd = new BType(cmd, v_addr, functions, symbols);
54         } else if (opcode == JAL) {
55             parsed_cmd = new JType(cmd, v_addr, functions, symbols);
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     }
67     return;
68 }

```

## Результат работы программы

Секция генерируется автоматически при компиляции

```

00010074 <main>:
    10074: ff010113    addi sp, sp, -16
    10078: 00112623    sw ra, 12(sp)
    1007c: 030000ef    jal ra, 0x100ac <mmul>
    10080: 00c12083    lw ra, 12(sp)
    10084: 00000513    addi a0, zero, 0
    10088: 01010113    addi sp, sp, 16
    1008c: 00008067    jalr zero, 0(ra)
    10090: 00000013    addi zero, zero, 0
    10094: 00100137    lui sp, 100000
    10098: fddff0ef    jal ra, 0x10074 <main>
    1009c: 00050593    addi a1, a0, 0
    100a0: 00a00893    addi a7, zero, 10
    100a4: 0fff000f    unknown_instruction
    100a8: 00000073    ecall
000100ac <mmul>:
    100ac: 00011f37    lui t5, 11000
    100b0: 124f0513    addi a0, t5, 292
    100b4: 65450513    addi a0, a0, 1620
    100b8: 124f0f13    addi t5, t5, 292
    100bc: e4018293    addi t0, gp, -448
    100c0: fd018f93    addi t6, gp, -48
    100c4: 02800e93    addi t4, zero, 40
    100c8: fec50e13    addi t3, a0, -20
    100cc: 000f0313    addi t1, t5, 0
    100d0: 000f8893    addi a7, t6, 0
    100d4: 00000813    addi a6, zero, 0
    100d8: 00088693    addi a3, a7, 0
    100dc: 000e0793    addi a5, t3, 0
    100e0: 00000613    addi a2, zero, 0
    100e4: 00078703    lb a4, 0(a5)
    100e8: 00069583    lh a1, 0(a3)
    100ec: 00178793    addi a5, a5, 1
    100f0: 02868693    addi a3, a3, 40
    100f4: 02b70733    mul a4, a4, a1
    100f8: 00e60633    add a2, a2, a4
    100fc: fea794e3    bne a0, a5, 0x100e4
    10100: 00c32023    sw a2, 0(t1)
    10104: 00280813    addi a6, a6, 2
    10108: 00430313    addi t1, t1, 4
    1010c: 00288893    addi a7, a7, 2
    10110: fdd814e3    bne t4, a6, 0x100d8
    10114: 050f0f13    addi t5, t5, 80
    10118: 01478513    addi a0, a5, 20
    1011c: fa5f16e3    bne t0, t5, 0x100c8
    10120: 00008067    jalr zero, 0(ra)

```

Symbol	Value	Size	Type	Bind	Vis	Index	Name
[ 0]	0x0	0	NOTYPE	LOCAL	DEFAULT	UNDEF	
[ 1]	0x10074	0	SECTION	LOCAL	DEFAULT	1	
[ 2]	0x11124	0	SECTION	LOCAL	DEFAULT	2	

[ 3]	0x0	0	SECTION	LOCAL	DEFAULT	3
[ 4]	0x0	0	SECTION	LOCAL	DEFAULT	4
[ 5]	0x0	0	FILE	LOCAL	DEFAULT	ABS test.c
[ 6]	0x11924	0	NOTYPE	GLOBAL	DEFAULT	ABS __global_pointer\$
[ 7]	0x118F4	800	OBJECT	GLOBAL	DEFAULT	2 b
[ 8]	0x11124	0	NOTYPE	GLOBAL	DEFAULT	1 __SDATA_BEGIN__
[ 9]	0x100AC	120	FUNC	GLOBAL	DEFAULT	1 mmul
[10]	0x0	0	NOTYPE	GLOBAL	DEFAULT	UNDEF _start
[11]	0x11124	1600	OBJECT	GLOBAL	DEFAULT	2 c
[12]	0x11C14	0	NOTYPE	GLOBAL	DEFAULT	2 __BSS_END__
[13]	0x11124	0	NOTYPE	GLOBAL	DEFAULT	2 __bss_start
[14]	0x10074	28	FUNC	GLOBAL	DEFAULT	1 main
[15]	0x11124	0	NOTYPE	GLOBAL	DEFAULT	1 __DATA_BEGIN__
[16]	0x11124	0	NOTYPE	GLOBAL	DEFAULT	1 _edata
[17]	0x11C14	0	NOTYPE	GLOBAL	DEFAULT	2 _end
[18]	0x11764	400	OBJECT	GLOBAL	DEFAULT	2 a

## Список источников

- [Спецификация RISC-V](#)
- [Статья на википедии про адресацию в машинных языках](#)
- [Краткое описание структуры ELF файла](#)
- [Спецификация ELF файла](#)

## Листинг кода

Секция генерируется автоматически при компиляции

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

```

1  #ifndef COMMAND_TYPES_GUARD
2  #define COMMAND_TYPES_GUARD
3
4  #include "typedefs.h"
5  #include "elfsymtable.h"
6
7  #include <string>
8  #include <vector>
9
10 struct InstructionType {
11     unsigned char opcode;
12     Elf32_Addr pc;
13     const std::vector<SymTableEntry>& functions;
14     const std::string& symbols;
15
16     InstructionType(const Elf32_Word& cmd, const Elf32_Addr& addr,
17                   const std::vector<SymTableEntry>& functions, const std::string& symbols);
18
19     unsigned char get_rd(Elf32_Word x);
20
21     unsigned char get_rs1(Elf32_Word x);
22

```

```

23     unsigned char get_rs2(Elf32_Word x);
24
25     unsigned char get_funct3(Elf32_Word x);
26
27     unsigned char get_funct7(Elf32_Word x);
28
29     unsigned char get_reg(Elf32_Word x, unsigned char pos);
30
31     unsigned char get_cmd(Elf32_Word x);
32
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();
113
114     ~JType() = default;
115 };
116
117 struct BType : public InstructionType {
118     std::string cmd_name;
119     unsigned char rs1;
120     unsigned char rs2;
121     unsigned char funct3;
122     int imm;
123
124     BType(const Elf32_Word& cmd, const Elf32_Addr& addr,
125           const std::vector<SymTableEntry>& functions, const std::string& symbols);
126
127     void print();
128
129     ~BType() = default;
130 };
131
132 #endif

```

## Listing 2: ../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 3: ../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 4: ../include/elfsymtable.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 5: ../include/output\_code.h

```

1  #ifndef OUTPUTCODE_GUARDS
2  #define OUTPUTCODE_GUARDS
3
4  #include "typedefs.h"
5  #include "elfsymbtable.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 6: ../include/output\_symbtable.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 7: ../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

```

## Listing 8: ../src/command\_types.cpp

```

1  #include "command_types.h"
2
3  #include "elfsymtable.h"
4  #include "output_symtable.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 ? "sr1" : "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) |
220         (get_blk(cmd, 10, 21) << 1) |
221         (get_blk(cmd, 1, 20) << 11) |
222         (get_blk(cmd, 8, 12) << 12);
223     extend_sign(imm, 21);
224     cmd_name = "jal";
225 }
226
227 void JType::print() {
228     printf("%7s\t%s, 0x%x", cmd_name.c_str(), prettify_reg(rd).c_str(), imm + pc);

```

```

229     printf("%s", format_addr(imm + pc).c_str());
230 }
231
232 BType::BType(const Elf32_Word& cmd, const Elf32_Addr& addr,
233             const std::vector<SymTableEntry>& f, const std::string& symbols) :
234     InstructionType(cmd, addr, f, symbols) {
235     rs1 = get_rs1(cmd);
236     rs2 = get_rs2(cmd);
237     funct3 = get_funct3(cmd);
238     imm = (get_blk(cmd, 1, 31) << 12) |
239           (get_blk(cmd, 6, 25) << 5) |
240           (get_blk(cmd, 4, 8) << 1) |
241           (get_blk(cmd, 1, 7) << 11);
242     extend_sign(imm, 13);
243     switch (funct3) {
244     case 0b000: cmd_name = "beq"; break;
245     case 0b001: cmd_name = "bne"; break;
246     case 0b100: cmd_name = "blt"; break;
247     case 0b101: cmd_name = "bge"; break;
248     case 0b110: cmd_name = "bltu"; break;
249     case 0b111: cmd_name = "bgeu"; break;
250     }
251 }
252
253 void BType::print() {
254     printf("%7s\t%s, %s, 0x%x", cmd_name.c_str(), prettify_reg(rs2).c_str(),
255           prettify_reg(rs1).c_str(), imm + pc);
256     printf("%s", format_addr(imm + pc).c_str());
257 }
258

```

## Listing 9: ../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 not 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 10: ../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 11: ../src/elfsymbtable.cpp

```

1  #include "elfsymbtable.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 12: ../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 #include <cstdio>
16
17 int main(int argc, char* argv[]) {
18
19     if (argc != 3) {
20         std::cout << "Usage: rv3 <elf_input_file_name> <output_file_name>\n";
21         return 1;
22     }
23     try {
24         std::ifstream fin((std::string) argv[1], std::ios::binary);
25         freopen(argv[2], "w", stdout);
26
27         Header header = parse_header(fin);
28         std::vector<SectionTableEntry> sectionTable =
29             parse_section_table(fin, header.e_shoff, header.e_shnum);
30         SectionTableEntry &shstrHeader = sectionTable[header.e_shstrndx];
31
32         auto find_tab_by_name = [&] (std::string name) {
33             return std::find_if(
34                 sectionTable.begin(),
35                 sectionTable.end(),
36                 [&] (SectionTableEntry& el) {
37                     return get_section_name(fin, shstrHeader, el) == name;
38                 }
39             );
40         };
41         std::vector<SectionTableEntry>::iterator symtabHeader = find_tab_by_name(".symtab");
42         std::vector<SectionTableEntry>::iterator strtabHeader = find_tab_by_name(".strtab");
43         std::vector<SectionTableEntry>::iterator textHeader = find_tab_by_name(".text");
44         std::vector<SymTableEntry> symtab =
45             parse_symtable(fin, symtabHeader->sh_offset,
46                 symtabHeader->sh_size / symtabHeader->sh_entsize);

```

```

47     std::string symbols(strtabHeader->sh_size, 0);
48     fin.seekg(strtabHeader->sh_offset);
49     fin.read(&symbols.front(), symbols.size());
50
51     std::string code(textHeader->sh_size, 0);
52     fin.seekg(textHeader->sh_offset);
53     fin.read(&code.front(), code.size());
54     print_code(code, symbols, symtab, textHeader->sh_addr);
55     printf("\n");
56     print_symtable(symtab, symbols);
57 } catch (std::logic_error err) {
58     std::cerr << "ERROR! Reason: " << ' ' << err.what() << '\n';
59     return 1;
60 }
61 return 0;
62 }

```

### Listing 13: ../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, v_addr, functions, symbols);
45     } else if (opcode == ARITH) {
46         parsed_cmd = new RType(cmd, v_addr, functions, symbols);
47     } else if (opcode == STORE) {
48         parsed_cmd = new SType(cmd, v_addr, functions, symbols);
49     } else if (opcode == EX_CTR || opcode == ARITHI || opcode == LOAD ||
50         opcode == JALR) {
51         parsed_cmd = new IType(cmd, v_addr, functions, symbols);
52     } else if (opcode == BRANCH) {
53         parsed_cmd = new BType(cmd, v_addr, functions, symbols);
54     } else if (opcode == JAL) {
55         parsed_cmd = new JType(cmd, v_addr, functions, symbols);
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 }
67 return;
68 }

```

Listing 14: ../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 }