Лабораторная работа №3	M3136	2022
ISA	Багринцев Михаил Алексеевич	

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

Инструментарий и требования к работе: работа была выполнена на языке программирования Java.

Описание: Необходимо написать программу-транслятор (дизассемблер), с помощью которой можно преобразовывать машинный код в текст программы на языке ассемблера. Должен поддерживаться следующий набор команд: RISC-V RV32I, RV32M.

Кодирование: little endian.

Обрабатывать необходимо только секции .text, .symtab.

Для каждой строки кода указывается её адрес в hex формате.

Вариант: 1

Описание системы кодирования команд RISC-V

Архитектуру RISC-V опубликовали в 2010 году в Калифорнийском университете в Беркли. К её созданию были причастны Эндрю Уотерман, Дэвид Паттерсон, Крсте Асанович. Эти люди ещё в 80, 90-х годах прошлого века основали философию того, как нужно проектировать процессоры - RISC (Reduced Instruction Set Computer), которая, вопреки основному течению, преследовала принцип, состоящий в том, что одна ассемблерная инструкция соответствует наименьшему числу микропрограммных инструкций. Такое решение даёт возможность компилятору с высокоуровневого языка более эффективно организовывать вычисления. Например, CISC (Complex Instruction Set Computer) философия грешит присутствием сложных инструкций, функционально напоминающих операторы высокоуровневых языков программирования. Такие инструкции работают медленно относительно их реализации в RISC, так как в RISC эта инструкция представлена набором независимых команд и конвейер может выполнять некоторые из них одновременно.

Принципы RISC

1. Отсутствие инструкций со сложными вычислениями - каждая инструкция должна выполняться быстро.
2. Фиксированная длина инструкции - увеличивает скорость обработки инструкции, так как если инструкции будут переменной длины, то сначала необходимо будет прочитать и декодировать размер инструкции, а только потом саму инструкцию.
3. Много регистров общего назначения - относительно дёшево можно сделать большое количество таких регистров.

4. Ограниченный функционал работы с оперативной памятью непосредственно - доступны только простейшие команды на чтение и запись, потому что обращение к ОЗУ - дорогое удовольствие, мы хотим взаимодействовать с регистрами как можно больше.

RISC-V - это пятая версия архитектуры по принципам RISC. Она относится к Load-Store типу ISA, что означает, что инструкции делятся на тех, которые работают между регистрами и памятью, и тех, которые работают между регистрами с регистрами. Популярность RISC-V заслужила сочетанием высокой производительности и открытости (Open Hardware). Также RISC-V опционально позволяет использовать инструкции переменной длины для расширения доступного пространства для кодирования инструкций.

Давайте посмотрим какие типы команд предоставляет нам RISC-V.

Формат инструкций 32-х битной RISC-V

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

Рисунок 1 - Типы инструкций 32-х битной RISC-V

На рисунке 1 вы можете наблюдать представление различных типов инструкций 32-х битной RISC-V архитектуры. Давайте разберёмся с

каждым типом по отдельности. Нам будут интересны только наборы инструкций RV32I (Base Integer ISA) и RV32M (Integer Multiplication and Division ISA Extension). В первом наборе 47 команд (учитывая fence, которую мы не будем затрагивать), во втором - 8 команд.

- Register/Register (R) - инструкции вида Reg-Reg-Reg (регистр-регистр-регистр)

Инструкции этого типа используют 3 регистра: 2 регистра назначения и один регистр источника. Каждый регистр кодируется 5 битами, так как количество регистров равно 32. Конкретная инструкция кодируется полями funct7, funct3 и opcode одновременно. Например, команды сложения, вычитания и умножения имеют одинаковые opcode и funct3, но различные funct7:

```
add -     0000000_xxxxx_xxxxx_000_xxxxx_011011
sub -     0100000_xxxxx_xxxxx_000_xxxxx_011011
mul -     0000001_xxxxx_xxxxx_000_xxxxx_011011
```

Результат применения команды к регистрам rs1 и rs2 записывается в регистр rd, но стоит учесть, что регистры перечислены слева направо в инструкции ассемблера, но следуют справа налево в машинном коде.
Пример перевода машинной команды в язык ассемблера:

```
0000000_10110_10100_000_10010_011011
add x18, x20, x22
```

Помимо арифметических операций к типу R относятся логические операции и операции битовых сдвигов, однако это очень маленькая часть от возможного количества инструкций.

- Immediate (I) - инструкций вида Imm-Reg-Reg (непосредственное значение-регистр-регистр)

Этот тип команд оперирует двумя регистрами и 12-и битной знаковой (знак кодируется сдвигом на 2) константой (immediate). Тут уже нет битов, отвечающий за funct7 - только funct3. Сразу встаёт вопрос о том как записать 32 бита в регистр, если можно только первые 12. Ответ на этот вопрос даст следующий тип инструкций, а сейчас предлагаю рассмотреть пример команды типа I.

```
111111111111_00101_000_01100_0010011
addi x10, x5, -1
```

Среди инструкций этого типа есть несколько довольно интересных. Например, инструкции побитового сдвига на константу imm. Эта инструкция использует только 5 первых бит на определение регистра (так как их всего 32), остальные 6 бит (регистров может быть 64) используются как funct6. Посмотрим чем отличается битовый логический сдвиг налево и арифметический сдвиг вправо.

```
000000_shamt_rs1_001_rd_0010011
slli rd, rs1, shamt
010000_shamt_rs1_001_rd_0010011
srai rd, rs1, shamt
```

Кстати, среди этих инструкций мы можем найти и команды, работающие с памятью, а именно, читающие из неё: lb (читает 8 бит), lh (читает 16 бит) и lw (читает 32 бита).

```
offset_rs1_010_rd_0000011
lw rd, offset(rs1)
```

В регистре rs1 лежит абсолютный адрес в памяти. Чтение данных происходит по адресу, который получается после сдвига абсолютного адреса из регистра rs1 на offset.

- Upper Immediate (U) - инструкции вида Imm-Reg (непосредственное значение-регистр)

В этом типе инструкций у нас нет даже funct3 поля (напоминаю, что вид инструкций можно посмотреть на рисунке 1) поэтому все команды этого типа имеют различные opcode. Однако, до этого мы оставили один вопрос открытым, а именно “Как записать 32-х битное значение в регистр?” За одну настоящую инструкцию это сделать нельзя (зато за одну псевдоинструкцию можно) - нужно записать 12 первых бит в регистр с помощью команды типа I и остальные 20 бит с помощью команды типа U.

Пример такой инструкции:

```
01010101010101010101_rd_0110111
lui rd 349525
```

- Store (S) - инструкция вида Imm-Reg-Reg (непосредственное значение-регистр-регистр)

Инструкции этого типа предназначены для сохранения данных в память. Такие команды работают с двумя регистрами и одной константой. На самом деле, логика функционирования этих инструкций очень похожа на инструкции записи в память из типа I. Значительное отличие - поле константы разбивается внутри машинного слова на два блока, а именно первые 5 битов константы находятся на месте rd, то есть с 7-ого по 11-ый бит, а последние 6 битов константы находятся с 25 по 31 биты. Такое решение было принято в связи с тем, что в таком типе команд нет принимающего регистра rd.

```
1111111_00111_10011_001_11010_0100011
sh x7, -6(x19)
```

- Branch (B) - инструкция вида Imm-Reg-Reg (непосредственное значение-регистр-регистр)

Тип Branch предоставляет возможность прыгать в разные части программы, если выполнено условие condition. Например, команда beq совершает переход на offset относительно текущего положения (у этого типа команд адресация относительная, что делает команды более обособленными от конкретного железа), если значение в rs1 равно значению в rs2. У команды bne условием прыжка является неравенство регистров, инструкция bge совершит прыжок, если значение в регистре rs1 больше или равно значению в регистре rs2.

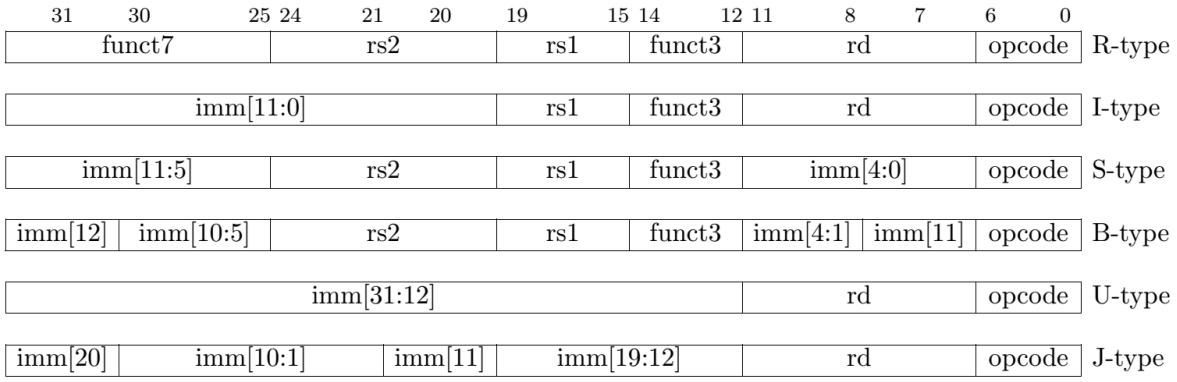

Рисунок 2 - Виды инструкций RISC-V32

Однако, может возникнуть вопрос: “Почему у команд типа Branch значение константы так странно разбросанно по машинному коду?” Результат, который мы сейчас наблюдаем является следствием попыток сохранить положение битов констант внутри кодов команд разных типов. Это делали для сведения к минимуму количества мультиплексоров и соединений, требуемых для дешифровки констант и дополнения их знаковым битом.

Также вы можете заметить, что у нас нигде не кодируется нулевой бит. Это связано с тем, что адресация относительная, длина инструкции фиксированная - 32 бита. Почему тогда не занулить последние 2 бита? Ответ заключается в существовании расширения RISV32C (Compressed ISA Extension), в котором команды кодируются 16 битами. Поэтому гарантировать можно только кратность адреса двум.

```
0_000000_rs2_rs1_000_1000_0_1100011
beq rs2, rs1, 16
```

Так или иначе, такой способ адресации не совсем удобен, так как если мы добавим пару новых строк ассемблерного кода между строкой, откуда мы совершаем прыжок и куда точка, куда мы прыгнем, сместится. Поэтому часто в коде используются метки. Вы ставите метку с человеко-читаемым названием на какой-то из строк и вместо сдвига в ассемблерном коде указываете название этой метки. Пример ниже.

```
L0: add 0x10, 0x0, 0x8
    beq 0x7, 0x4, L0
```

- Jump (J) - инструкция вида Imm-Reg (непосредственное значение-регистр)

В отличии от команд типа Branch, Jump - это прыжок в какую-то часть кода без проверки condition. Это может быть полезно, например, в последней строке функции, чтобы выйти из неё. Также, так как у нас осталось место от двух удалённых регистров, константа immediate теперь занимает все 20 битов, что позволяет прыгать на более далёкие расстояния. По аналогии с командами Branch, в инструкциях Jump нулевой бит всегда равен 0. Но возникает вопрос: “Зачем в этих командах регистр?”. В этот регистр сохраняется значение адреса возврат, чтобы, прыгнув куда-нибудь, мы смогли вернуться назад. Пример:

```
0_1000011111_1_00101011_00001_1101111
jal x1,0x000af03e
```

Стоит сказать, что помимо базового набора команд I, у RISC-V существует множество других (опциональных):

- E - базовый набор инструкций для работы с целыми числами, но количество регистров уменьшено до 16-ти;
- M - набор инструкций с операциями умножения и деления;
- A - набор инструкций с атомарными операциями для поддержки синхронизации между несколькими RISC-V harts, запущенными в одном пространстве памяти;
- F - набор инструкций для работы с Single-Precision Floating-Point;
- D - набор инструкций для работы с Double-Precision Floating Point;
- Q - набор инструкций для работы с Quad-Precision Floating Point;
- C - набор инструкций с уменьшенным статическим и динамическим размером кода, путём добавления 16-ти битных инструкций;

Теперь, самое время обсудить специальные названия регистров (так как их всего 32 штуки, то каждому давно дали название и правила использования), которые вы можете использовать вместо номера регистра в языке ассемблера. Начнём с регистра x0. Он имеет название zero и обладает уникальным свойством - в нём всегда лежит значение 0. То есть присвоить этому регистру данные равносильно потере данных.

Далее идёт регистр x1, он же ra. Его особенность в том, что при прыжках с записью адреса возврата в регистр можно не указывать регистр. Тогда он по умолчанию будет записан в регистр ra.

Регистр x2, он же sp - указатель стека. С помощью, например, инструкции sw можно класть новые значение в стек, обращаясь к регистру sp.

Регистр x3, он же gp, содержит адрес области глобальных данных. Он бывает необходим для передачи данных в программу со стороны ОС или для доступа к глобальным переменным из подпрограмм.

Регистр x4 also known as tp (thread pointer) содержит адрес набора переменных данного потока. Это имеет смысл в случае многопоточного приложения.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Рисунок 3 - Специальные названия регистров

Регистры x5, x6, x7, x28, x29, x30, x31 называются t%i, где i пробегает значения от 0 до 6 включительно. Они декларируются как временные и их можно использовать без каких-либо ограничений.

Регистр s0 иногда называют fp (frame pointer), так как иногда он может хранить адрес области данных текущей подпрограммы (зависит от конвенции организации подпрограмм).

Регистры x9, x18-x27 называются s%i (saved), где i пробегает от 1 до 11 включительно. Такие регистры следует восстанавливать в исходное значение перед тем как завершить выполнение подпрограммы.

Регистры x10-x17 называются a%i (argument), где i пробегает от 0 до 7 включительно. Эти регистры, как понятно из названия, передаются в качестве аргументов подпрограмме.

Описание структуры файла ELF

Начнём с определения, которое нам предоставляет английская википедия. В вычислительной технике, Executable and Linkable Format (ELF, прежнее название Extensible Linking Format) - это общий стандартный формат файлов для исполняемых файлов, объектного кода (результат компиляции исходного кода), разделяемых библиотек и дампов ядра. По задумке, ELF должен быть гибким и кроссплатформенным, поэтому его header часть хранит столько информации (об этом далее).

ELF-файл может интерпретироваться по-разному в зависимости от того, кто с ним работает. Взгляд со стороны исполнителя (execution view) требует наличия Program Header Table и сегментов. Если мы посмотрим на файл со стороны компоновщика (linking view), то у нас уже нет необходимости в Program Header Table, зато мы просматриваем все секции и заголовки секций.

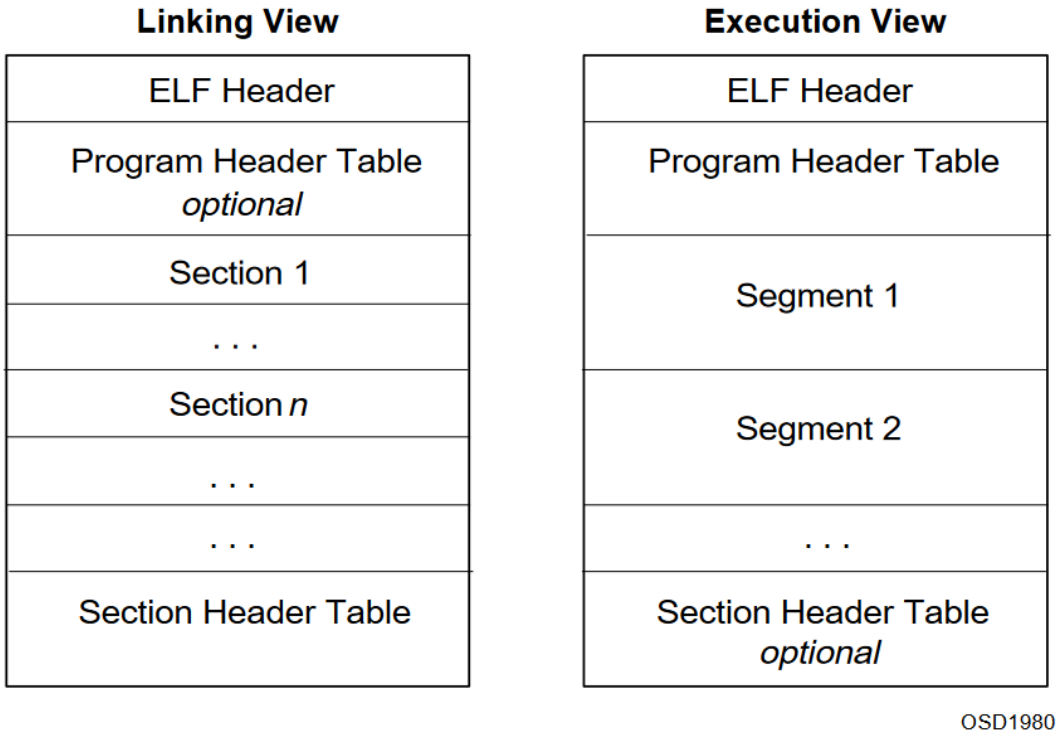

Рисунок 4 - Структура ELF-файла

ELF Header находится в начале и содержит п. Секции содержат информацию, необходимую компоновщику, а именно код (секция .text), таблица символов (секция .symtab), область для глобальных и статических локальных переменных и кучи других данных программы (секция .data) и так далее. Заголовки секций дают краткую информацию, такую как адрес имени секции (есть отдельная секция, которая хранит имена всех секций), размер секции, адрес начала секции и так далее.

Так как мы рассматриваем программу с позиции компоновщика, нет необходимости много писать про сегменты и Program Header Table, поэтому уделим им буквально один абзац. Сегмент обычно состоит из нескольких секций - .init, .plt, .text и .finit. Сегмент является непрерывной областью адресного пространства, с разными атрибутами доступа (может быть доступ на чтение, а может на исполнение). Program Header Table содержит информацию для создания образа процесса системой.

Рассмотрим последовательно структуру ELF-файла (нас интересует первый столбец рисунка 4).

ELF Header

Для 32-х битных систем размер заголовка файла составляет 52 байта.

```
#define EI_NIDENT       16

typedef struct {
          unsigned char  e_ident[EI_NIDENT];
          Elf32_Half     e_type;
          Elf32_Half     e_machine;
          Elf32_Word     e_version;
          Elf32_Addr     e_entry;
          Elf32_Off      e_phoff;
          Elf32_Off      e_shoff;
          Elf32_Word     e_flags;
          Elf32_Half     e_ehsize;
          Elf32_Half     e_phentsize;
          Elf32_Half     e_phnum;
          Elf32_Half     e_shentsize;
          Elf32_Half     e_shnum;
          Elf32_Half     e_shstrndx;
} Elf32_Ehdr;
```

Рисунок 5 - Структура ELF Header

Рассмотрим поля заголовка файла поподробнее:

- e_ident - 10 байт, определяющих тип файла как ELF (через магические числа), 32/64-х битный формат, little/big endian, ABI целевой ОС.
- e_type - тип объектного файла (executable, shared, etc…);
- e_machine - хранит целевое ISA (в нашем случае RISC-V);
- e_version - версия ELF;
- e_entry - адрес в виртуальной памяти, на который система сначала передаёт управление, а потом запускает с него процесс (по умолчанию 0);
- e_phoff - сдвиг в байтах от начала файла, по которому лежит Program Header Table (по умолчанию 0);
- e_shoff - сдвиг в байтах от начала файла, по которому лежит таблица заголовков секций (по умолчанию 0);
- e_flags - специфические для процессора флаги;
- e_ehsize - размер ELF Header;
- e_phentsize - размер одной записи в Program Header Table;
- e_phnum - количество записей в Program Header Table;
- e_shentsize - размер записи в таблице заголовков секций;
- e_shnum - количество записей в таблице заголовков секций;
- e_shstrndx - индекс в таблице заголовков секций, указывающий на заголовок, принадлежащий секции, которая хранит названия других секций.

Теперь, прочитав заголовок файла, мы знаем все необходимые данные про таблицу заголовков секций (по ней мы узнаем всю информацию о каждой секции) - где её искать, сколько занимает один заголовок, сколько заголовков всего. Теперь можно рассматривать структуру таблицы заголовков секций.

Section Header Table

Так как, прочитав заголовок файла, мы знаем только адрес таблицы заголовков секций, но не самих секций, то следующим этапом разбора ELF-файла нужно сделать разбор таблицы заголовков секций.

Информация о каждой секции в таблице заголовков секций называется записью (entry). Запись в таблице состоит из e_shentsize байтов, количество записей содержит параметр e_shnum, каждая запись имеет структуру, представленную на рисунке 6.

```
typedef struct {
            Elf32_Word    sh_name;
            Elf32_Word    sh_type;
            Elf32_Word    sh_flags;
            Elf32_Addr    sh_addr;
            Elf32_Off     sh_offset;
            Elf32_Word    sh_size;
            Elf32_Word    sh_link;
            Elf32_Word    sh_info;
            Elf32_Word    sh_addralign;
            Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

Рисунок 6 - Структура записи в таблице заголовков секций

- sh_name - индекс в таблице названий секций (.shstrtab), указывающий на название данной секции;
- sh_type - классифицирует содержимое секции
    - SHT_NULL - неактивная секция;
    - SHT_SYMTAB, SHT_DYNSYM - таблица символов;
    - SHT_STRTAB - таблица названий секций;
    - SHT_HASH - таблица хэш-символов;
    - SHT_SHLIB - зарезервированная секция без определённого типа;

    - SHT_NOBITS - секция не занимающая места в файле;
    - и остальные …
- sh_flags - если установлено ненулевое значение, то для данной секции включается один из атрибутов
    - SHF_WRITE - во время исполнения программы разрешается перезаписать данные в этой секции;
    - SHF_ALLOC - секция будет занимать место в памяти во время исполнения;
    - SHF_EXECINSTR - секция содержит исполняемый машинный код
- sh_addr - указывает адрес на начало секции в памяти или 0, если секции в памяти не будет;
- sh_offset - смещение относительно файла на начало данной секции;
- sh_size - размер данной секции в байтах (секция типа SHT_NOBITS не обязательно имеет размер 0, но она не будет занимать место в файле)
- sh_link - индексная ссылка, интерпретация которой зависит от типа секции;
- sh_info - дополнительная информация, интерпретация которой зависит от типа секции;
- sh_addralign - выравнивание адреса секции. Если установлено значение большее единицы, то поле sh_addr должно делиться на sh_addralign;
- sh_entsize - если данная секция тоже является таблицей, то поле указывает размер одной записи этой таблицы. Если таблицей не является, то поле равно нулю.

Секции

Для дизассемблирования нам нужны секции .text, в которой собственно и хранится программный код, .symtab, из которой мы будем доставать метки для языка ассемблера. Так как нам нужно доставать секции по названиям, придётся воспользоваться секцией .strtab, в которой хранятся названия, ассоциированные с записями таблицы символов, и .shstrtab. В формате RISC-V32 двоичное представление ассемблерного кода выглядит несложно - инструкции длинной в 32 бита, которые читаются последовательно - одна за другой. Интерес вызывает таблица символов - у этой секции есть своя структура, которую мы и рассмотрим далее.

```
typedef struct {
          Elf32_Word    st_name;
          Elf32_Addr    st_value;
          Elf32_Word    st_size;
          unsigned char st_info;
          unsigned char st_other;
          Elf32_Half    st_shndx;
} Elf32_Sym;
```

Рисунок 7 - Структура записи в таблице символов

- st_name - сдвиг внутри секции .strtab, с которой нужно начать читать название;
- st_value - интерпретируется в зависимости от типа файла
    - Relocatable file: хранит сдвиг от начала секции, которую определяет st_shndx;
    - Executable & Shared object files: хранит виртуальный адрес в памяти.
- st_size - ассоциированный размер символа;
- st_info - хранит одновременно тип символа (первый байт) и binding attribute (второй байт). Binding attribute может быть локальным - не

виден за пределами файла, в котором объявлен символ, глобальным - символ виден всем скомбинированным файлам, слабым (weak) - глобальные символы, но с меньшим приоритетом и зарезервированным для нужд процессора. Типы представим в виде enam-а:

  - STT_NOTYPE - тип символа не специфицирован;
  - STT_OBJECT - символ связан с данными (массив, переменная…);
  - STT_FUNC - символ связан с функцией;
  - STT_SECTION - символ связан с секцией;
  - STT_FILE - символ связан с файлом;

    ещё есть типы, зарезервированные процессором.
- st_other - поле для будущих свойств символа, сейчас установлено нулевое значение;
- st_shndx - индекс секции, с которой связан данный символ.

Теперь мы знаем всё необходимое, что можно было получить из других секций, время приступать к трансляции кода в язык ассемблера RISC-V32, бегая по секции .text.

Описание работы написанного кода

Выбор языка высокого уровня для написания дизассемблера я остановил на Java. Запустить программу вы можете, скомпилировав файл Main.java и передав ему в качестве первого аргумента название входного файла, в качестве второго - выходного.

Главных классов два - RISCV32Disassembler и ELFParser. Класс Main создаёт экземпляр класса RISCV32Disassembler и вызывая на нём метод parse передаёт InputStream и PrintStream. RISCV32Disassembler в свою очередь создаёт экземпляр ELFParser, сохраняет его себе как поле и вызывает на нём метод parse, передавая в качестве аргументы массив байтов, прочитанный с InputStream.

```
public void parse( final InputStream input,
     final PrintStream out) throws FileHeaderException,
     ELFException, IOException {

     data = input.readAllBytes();
     elfParser.parse(data);

     printTextSegment(out,
        elfParser.getELFSection(".text"),
        elfParser.getSymTabMap());
     out.println();
     elfParser.printSymTab(out);

     System.out.println("Success disassembling.");
}
```

Метод parse класса RISCV32Disassembler

ELFParser вызывает метод parse у nested класса - FileHeaderParser. FileHeaderParser проверяет корректность заголовка файла (в случае некорректного заголовка вылетает исключение FileHeaderException с информацией об ошибке, которое будет поймано классом Main) и

устанавливает публичные поля типа int, которые содержат атрибуты заголовка (e_entry, e_phoff, e_shoff…). Потом, ELFParser начинает парсить таблицу заголовков, секций.

```
public void parse(final byte[] elf)
     throws FileHeaderException, ELFException {
     this.elf = elf;
     fileHeaderParser.parse(elf);
     parseAndFillSections();
     parseSymTab();
}
```

Метод parse класса ELFParser

Перед парсингом таблицы заголовков секций я присваиваю полям, относящимся к секциям пустые значения коллекций.

```
private Map<String, ELFSection> sectionsMap;
private List<ELFSection> sectionsList;
```

Поля, относящиеся к секциям

Класс ELFSection - обычный геттер с final полями (s_name, sh_name, sh_type…) примитивного типа int.

Для поиска названия секции мне было необходимо знать адрес .shstrtab в файле, так как в таблице заголовков секций лежит только offset от начала .shstrtab. Ниже показано как я нашёл адрес этой таблицы и использую его для получения имён секций.

```
final int shstrtabIdx = readNBytes(
     fileHeaderParser.e_shoff +
     fileHeaderParser.e_shstrndx *
fileHeaderParser.e_shentsize + 4 * 4,
     2);
...
...
int sh_name = readNBytes(shIdx, 4);
final String name = readName(shstrtabIdx + sh_name);
```

Код, работающий с названиями секций

Функция readNBytes читает N байт в порядке Little Endian, функция readName читает символы до тех пор, пока на встретит символ 0x0 (документация гласит, что все имена разделяются таким служебным именем).

Далее, читая поля секции с помощью метода readNBytes, заполняем sectionsMap и sectionsList (так как заполняются ссылки на объекты, по памяти всё обходится очень дёшево). Может возникнуть вопрос: “Зачем вообще нужен List?” Дело в том, что при выводе таблицы символов, мы должны сохранять порядок, которой был в таблице заголовков секций, поэтому в случае Symbol Table используется List.

После того, как мы запарсили таблицу заголовков секций, начинается парсинг таблицы символов.

```
this.symtab = new ArrayList<>();
final ELFSection symTabSec = sectionsMap.get(".symtab");
final ELFSection strTabSec = sectionsMap.get(".strtab");
int idx = symTabSec.sh_offset;
```

Первые строки парсинга таблицы символов

Начинаем парсить Symbol Table с idx и после каждого символа увеличиваем значение idx на значение symTabSec.sh_entsize. Собираем список symtab из объектов класса SymbolDescription аналогичного ELFSection. Для примера приведу парсинг bind-а:

```
switch (readNBytes(idx + 12, 1) >> 4) {
     case 0x0 -> "LOCAL";
     case 0x1 -> "GLOBAL";
     case 0x2 -> "WEAK";
     case 0xd, 0xe, 0xf -> "<processor specific>: " +
(readNBytes(idx + 12, 1) >> 4);
     default -> throw new ELFException("Unexpected Symbol
```

```
Binding while parsing .symtab section (name='" + name +
"'):" + (readNBytes(idx + 12, 1) >> 4), idx + 12);
},
```

Парсинг Bind в Symbol Table

На этом метод parse класса ELFParser заканчивает свою работу и мы возвращаемся в RISCV32Disassembler.

Следующее, что мы тут делаем это парсим и выводим ассемблерные инструкции в дизассемблированном виде. Для этого мы запускаем метод printTextSegment. Так как от нас просят расставлять L-метки, которых изначально нет в Symbol Table - переходы на инструкции без меток, то перед тем как парсить все команды нужно пройтись и собрать все метки.

Сейчас самое лучшее время, чтобы объяснить как происходит парсинг инструкций. Есть метод, который, разбирая 32-х битный код возвращает строковое представление команды - getInstruction(word).

```
final int word = readNBytes(idx, 4);
final String instruction = getInstruction(word);
```

Начало парсинга инструкции RISC-V

Внутри этого метода лежит огромный switch-case, который порой опускается до 3-го уровня вложенности (opcode -> funct3 -> funct7). Значение по умолчанию в каждом switch-case это “unknown_instruction”.

В классе RISCV32Disassembler есть поле,

```
private static Map<String, InstructionTypes>
instructionType;
```

которое инициализируется и заполняется значениями в статическом инициализаторе класса (там же и заполняется отображения список, сопоставляющий номеру регистра его название). Там мы составляем отображение из строки в enum.

```
public enum InstructionTypes {
    UPPER_IMMEDIATE, // lui rd, imm
```

```
    STORE, // sb rs2, offset(rs1)
    BRANCH, // beq rs1, rs2, offset
    JUMP, // jal rd, offset
    JUMP_WITH_REG, // jalr rd, rs1, offset
    REGISTER_REGISTER, // add rd, rs1, rs2
    IMMEDIATE_L, // lb rd, offset(rs1)
    IMMEDIATE_CSRR, // csrrw rd, offset, rs1
    IMMEDIATE, // addi rd, rs1, imm
    I_SHAMT, // slli rd, rs1, shamt
    SPECIAL, // ecall
    UNKNOWN_INSTRUCTION;
}
```

Токены инструкция RISC-V32

Как вы можете заметить, некоторые типы команд RISC-V у меня разбиты на подтипы. Помните особенность инструкций побитового сдвига? Именно из-за таких “радостей” было принято решение токенизировать команды более узко. Для каждого типа инструкции, слева в комментарии находится один из представителей.

Поэтому после того как мы имеем название команды, полученного от метода getInstruction(), можно легко токенизировать инструкцию и добавить новую метку (если метки с таким адресом ещё нет) при условии, что мы нашли инструкцию типов BRANCH или JUMP. Собрали метки в копию Map из ELFParser, теперь можно возвращаться к парсингу всех типов команд.

```
final var label = symTabMap.get(text.sh_addr + i * 4);
if (label != null) {
    out.printf("%08x   <%s>:\n", text.sh_addr + i * 4,
label.name);
}
final int word = readNBytes(idx, 4);

final String instruction = getInstruction(word);
```

Вывод меток при парсинге инструкций

Для примера приведу парсинг инструкций типа UPPER_IMMEDIATE.

```
case UPPER_IMMEDIATE: {
     out.printf(
     "   %05x:\t%08x\t%7s\t%s, %s\n",
     text.sh_addr + i * 4,
     word,
     instruction,
     registers.get((word >> 7) & 0b11111),
     String.valueOf((word >>> 12))
     );
     break;
}
```

Парсинг инструкции типа UPPER_IMMEDIATE

Таким образом парсятся команды RISC-V32I и RISCV-32M. Осталось вывести таблицу символов. Это мы делегируем ELFParser-у, передав методу printSymTab ссылку на PrintStream. Метод выглядит довольно минималистично, поэтому покажу его в полном объёме.

```
public void printSymTab(final PrintStream out) {
     out.println(".symtab");
     out.println("Symbol Value              Size Type
Bind      Vis         Index Name");
     int i = 0;
     for (SymbolDescription e : symtab) {
     out.printf("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s
%s\n", i, e.st_value, e.st_size, e.st_type, e.st_bind,
e.st_vis, e.st_shndx, e.name);
     i++;
     }
}
```

Метод printSymTab в классе ELFParser

Завершающая команда вывод в консоль “Success disassembling”.

Результат работы написанной программы на приложенном к заданию файле

```
.text
00010074     <main>:
   10074:    ff010113     addi    sp, sp, -16
   10078:    00112623     sw    ra, 12(sp)
   1007c:    030000ef     jal    ra, 100ac <mmul>
   10080:    00c12083     lw    ra, 12(sp)
   10084:    00000513     addi    a0, zero, 0
   10088:    01010113     addi    sp, sp, 16
   1008c:    00008067     jalr    zero, 0(ra)
   10090:    00000013     addi    zero, zero, 0
   10094:    00100137     lui    sp, 256
   10098:    fddff0ef     jal    ra, 10074 <main>
   1009c:    00050593     addi    a1, a0, 0
   100a0:    00a00893     addi    a7, zero, 10
   100a4:    0ff0000f    unknown_instruction
   100a8:    00000073      ecall
000100ac     <mmul>:
   100ac:    00011f37     lui    t5, 17
   100b0:    124f0513     addi    a0, t5, 292
   100b4:    65450513     addi    a0, a0, 1620
   100b8:    124f0f13     addi    t5, t5, 292
   100bc:    e4018293     addi    t0, gp, -448
   100c0:    fd018f93     addi    t6, gp, -48
   100c4:    02800e93     addi    t4, zero, 40
000100c8     <L2>:
   100c8:    fec50e13     addi    t3, a0, -20
   100cc:    000f0313     addi    t1, t5, 0
   100d0:    000f8893     addi    a7, t6, 0
   100d4:    00000813     addi    a6, zero, 0
000100d8     <L1>:
   100d8:    00088693     addi    a3, a7, 0
   100dc:    000e0793     addi    a5, t3, 0
   100e0:    00000613     addi    a2, zero, 0
000100e4     <L0>:
   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    a5, a0, 100e4 <L0>
   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    a6, t4, 100d8 <L1>
   10114:    050f0f13     addi    t5, t5, 80
   10118:    01478513     addi    a0, a5, 20
   1011c:    fa5f16e3     bne    t5, t0, 100c8 <L2>
   10120:    00008067     jalr    zero, 0(ra)

.symtab
Symbol Value                 Size Type Bind      Vis    Index Name
```

```
[   0] 0x0                0 NOTYPE   LOCAL    DEFAULT   UNDEF
[   1] 0x10074            0 SECTION  LOCAL    DEFAULT      1 .text
[   2] 0x11124            0 SECTION  LOCAL    DEFAULT      2 .bss
[   3] 0x0                0 SECTION  LOCAL    DEFAULT      3 .comment
[   4] 0x0                0 SECTION  LOCAL    DEFAULT      4 .riscv.attributes
[   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

1. https://uneex.org/LecturesCMC/ArchitectureAssembler2022
2. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf
3. https://en.wikipedia.org/wiki/RISC-V
4. https://shakti.org.in/docs/risc-v-asm-manual.pdf
5. https://en.wikipedia.org/wiki/Executable_and_Linkable_Format
6. https://ru.bmstu.wiki/ELF_(Executable_and_Linkable_Format)
7. https://refspecs.linuxfoundation.org/elf/elf.pdf
8. https://docs.oracle.com/cd/E19683-01/817-3677/817-3677.pdf

Листинг кода

Main.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintStream;

public class Main {
	public static void main(String[] args) {
	final String inputFileName = args[0];
	final String outputFileName = args[1];

	try (
		final var input = new FileInputStream(new File(inputFileName));
		final var output = new PrintStream(new File(outputFileName))
	) {
		new RISCV32Disassembler().parse(input, output);
	} catch (FileNotFoundException e) {
		System.out.println(e.getMessage());
	} catch (IOException e) {
		System.out.println(e.getMessage());
	} catch (FileHeaderException e) {
		System.out.println("FileHeaderException while trying to parse ELF
file '" + inputFileName + "': " + e.getMessage());
		System.out.println("Byte position: " + e.getPos());
	} catch (ELFException e) {
		System.out.println("ELFException while trying to parse ELF file '" +
inputFileName + "': " + e.getMessage());
		System.out.println("Byte position: " + e.getPos());
	}
	}
}
```

ELFParser.java

```
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public final class ELFParser {
	private byte[] elf;
	private final FileHeaderParser fileHeaderParser;
	private List<SymbolDescription> symtab;
	private Map<String, ELFSection> sectionsMap;
```

```
	private List<ELFSection> sectionsList;

	public ELFParser() {
	this.fileHeaderParser = new FileHeaderParser();
	}

	public void parse(final byte[] elf) throws FileHeaderException,
ELFException {
	this.elf = elf;
	fileHeaderParser.parse(elf);
	parseAndFillSections();
	parseSymTab();
	}

	public void printSymTab(final PrintStream out) {
	out.println(".symtab");
	out.println("Symbol Value                  Size Type Bind      Vis
Index Name");
	int i = 0;
	for (SymbolDescription e : symtab) {
		out.printf("[%4d] 0x%-15X %5d %-8s %-8s %-8s %6s %s\n", i,
e.st_value, e.st_size, e.st_type, e.st_bind, e.st_vis, e.st_shndx, e.name);
		i++;
	}
	}

	public Map<Integer, SymbolDescription> getSymTabMap() {
	final Map<Integer, SymbolDescription> symTabMap = new
HashMap<>(symtab.size() * 2);
	for (SymbolDescription symDesc : symtab) {
		symTabMap.put(symDesc.st_value, symDesc);
	}
	return symTabMap;
	}

	public ELFSection getELFSection(final String name) {
	return sectionsMap.get(name);
	}

	private void parseSymTab() throws ELFException {
	this.symtab = new ArrayList<>();
	final ELFSection symTabSec = sectionsMap.get(".symtab");
	final ELFSection strTabSec = sectionsMap.get(".strtab");
	int idx = symTabSec.sh_offset;

	if (symTabSec == null || strTabSec == null) {
		throw new ELFException("The ELF file does not contain sections
.symtab or .strtab", idx);
	}

	for (int i = 0; i < symTabSec.sh_size / symTabSec.sh_entsize; i++) {
```

```
            final String name;
            if ((readNBytes(idx + 12, 1) & 0xf) == 0x3) { // SECTION type
            name = sectionsList.get(readNBytes(idx + 14, 2)).s_name;
            } else { // other types
            name = readName(readNBytes(idx, 4) + strTabSec.sh_offset);
            }
            symtab.add(
            new SymbolDescription(
                  name,
                  readNBytes(idx, 4),
                  readNBytes(idx + 4, 4),
                  readNBytes(idx + 8, 4),
                  switch (readNBytes(idx + 12, 1) >> 4) {
                        case 0x0 -> "LOCAL";
                        case 0x1 -> "GLOBAL";
                        case 0x2 -> "WEAK";
                        case 0xd, 0xe, 0xf -> "<processor specific>: " +
(readNBytes(idx + 12, 1) >> 4);
                        default -> throw new ELFException("Unexpected Symbol
Binding while parsing .symtab section (name='" + name + "'):" + (readNBytes(idx +
12, 1) >> 4), idx + 12);
                  },
                  switch (readNBytes(idx + 12, 1) & 0xf) {
                        case 0x0 -> "NOTYPE";
                        case 0x1 -> "OBJECT";
                        case 0x2 -> "FUNC";
                        case 0x3 -> "SECTION";
                        case 0x4 -> "FILE";
                        case 0xd, 0xe, 0xf -> "<processor specific>: " +
(readNBytes(idx + 12, 1) & 0xf);
                        default -> throw new ELFException("Unexpected Symbol
Types while parsing .symtab section (name='" + name + "'):" + (readNBytes(idx +
12, 1) & 0xf), idx + 12);
                  },
                  switch (readNBytes(idx + 13, 1)) {
                        case 0x0 -> "DEFAULT";
                        case 0x1 -> "INTERNAL";
                        case 0x2 -> "HIDDEN";
                        case 0x3 -> "PROTECTED";
                        default -> throw new ELFException("Unexpected Symbol
Visibilty while parsing .symtab section (name='" + name + "'):" + (readNBytes(idx
+ 13, 1) & 0xf), idx + 13);
                  },
                  switch (readNBytes(idx + 14, 2)) {
                        case 0x0 -> "UNDEF";
                        case 0xff00 -> "LORESERVE";
                        case 0xfff1 -> "ABS";
                        case 0xfff2 -> "COMMON";
                        case 0xffff -> "HIRESERVE";
                        default -> {
                        final int specialSectionIdx = readNBytes(idx + 14, 2);
```

```
                        if (0xff00 <= specialSectionIdx && specialSectionIdx <=
0xff1f) {
                              yield "<processor specific>:" +
specialSectionIdx;
                        } else {
                              yield String.valueOf(specialSectionIdx);
                        }
                        }
                  }
            )
            );
            idx += symTabSec.sh_entsize;
      }
      }

      private void parseAndFillSections() {
      this.sectionsMap = new HashMap<>();
      this.sectionsList = new ArrayList<>();

      final int shstrtabIdx = readNBytes(
            fileHeaderParser.e_shoff +
            fileHeaderParser.e_shstrndx * fileHeaderParser.e_shentsize + 4 * 4,
            2);

      int shIdx = fileHeaderParser.e_shoff;
      for (int i = 0; i < fileHeaderParser.e_shnum; i++) {
            int sh_name = readNBytes(shIdx, 4);

            final String name = readName(shstrtabIdx + sh_name);
            sectionsList.add(
            new ELFSection(name,
                  sh_name,
                  readNBytes(shIdx + 4, 4),
                  readNBytes(shIdx + 8, 4),
                  readNBytes(shIdx + 12, 4),
                  readNBytes(shIdx + 16, 4),
                  readNBytes(shIdx + 20, 4),
                  readNBytes(shIdx + 24, 4),
                  readNBytes(shIdx + 28, 4),
                  readNBytes(shIdx + 32, 4),
                  readNBytes(shIdx + 36, 4)
            )
            );
            sectionsMap.put(name, sectionsList.get(i));
            shIdx += fileHeaderParser.e_shentsize;
      }
      }

      private String readName(int idx) {
      final StringBuilder sb = new StringBuilder();
      while (readNBytes(idx, 1) != 0x00) {
```

```
            sb.append((char) readNBytes(idx++, 1));
        }
        return sb.toString();
        }

        private int readNBytes(int idx, final int n) {
        int res = 0;
        for (int i = 0; i < n; i++) {
            res += Byte.toUnsignedInt(elf[idx++]) << (8 * i);
        }
        return res;
        }

        final class FileHeaderParser {
        private byte[] elf;
        private int pointer;

        public int e_entry;
        public int e_phoff;
        public int e_shoff;
        public int e_flags;
        public int e_ehsize;
        public int e_phentsize;
        public int e_phnum;
        public int e_shentsize;
        public int e_shnum;
        public int e_shstrndx;

        public void parse(final byte[] elf) throws FileHeaderException {
            this.elf = elf;
            checkFileHeader();
            setHeaderFilds();
        }

        private void setHeaderFilds() {
            e_entry = readNBytes(4);
            e_phoff = readNBytes(4);
            e_shoff = readNBytes(4);
            e_flags = readNBytes(4);
            e_ehsize = readNBytes(2);
            e_phentsize = readNBytes(2);
            e_phnum = readNBytes(2);
            e_shentsize = readNBytes(2);
            e_shnum = readNBytes(2);
            e_shstrndx = readNBytes(2);
        }

        private void checkFileHeader() throws FileHeaderException {
            // e_ident[16]
            if (readByte() != 0x7f || readByte() != 0x45 || readByte() != 0x4c ||
readByte() != 0x46) {
```

```
            throw new FileHeaderException("Illegal file signature. Magic number
must be 0x7f 0x45 0x4c 0x46.", pointer);
            }
            if (readByte() != 0x01) {
            throw new FileHeaderException("Illegal EI_CLASS. Supported only
32-bit.", pointer);
            }
            if (readByte() != 0x01) {
            throw new FileHeaderException("Illegal endianess(EI_DATA). Supported
only Little Endian.", pointer);
            }
            if (readByte() != 0x01) {
            throw new FileHeaderException("Illegal EI_VERSION.", pointer);
            }
            pointer = 16;
            // After this part data will be represented in little endian

            // e_type
            pointer += 2;
            // e_machine
            if (readByte() != 0xf3 || readByte() != 0x00) {
            throw new FileHeaderException("Illegal ISA. Supported only RISC-V",
pointer);
            }
            // e_version
            pointer += 4;
      }

      private int readByte() {
            return Byte.toUnsignedInt(elf[pointer++]);
      }

      private int readNBytes(final int n) {
            int res = 0;
            for (int i = 0; i < n; i++) {
            res += Byte.toUnsignedInt(elf[pointer++]) << (8 * i);
            }
            return res;
      }
      }
}
```

RISCV32Disassembler.java

```
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
```

```
public class RISCV32Disassembler {
	private byte[] data;
	private static Map<String, InstructionTypes> instructionType;
	private static List<String> registers;
	private ELFParser elfParser;

	public RISCV32Disassembler() {
	elfParser = new ELFParser();
	}

	static {
	instructionType = new HashMap<>();

	instructionType.put("lui", InstructionTypes.UPPER_IMMEDIATE);
	instructionType.put("auipc", InstructionTypes.UPPER_IMMEDIATE);
	instructionType.put("addi", InstructionTypes.IMMEDIATE);
	instructionType.put("slti", InstructionTypes.IMMEDIATE);
	instructionType.put("sltiu", InstructionTypes.IMMEDIATE);
	instructionType.put("xori", InstructionTypes.IMMEDIATE);
	instructionType.put("ori", InstructionTypes.IMMEDIATE);
	instructionType.put("andi", InstructionTypes.IMMEDIATE);
	instructionType.put("slli", InstructionTypes.I_SHAMT);
	instructionType.put("srli", InstructionTypes.I_SHAMT);
	instructionType.put("srai", InstructionTypes.I_SHAMT);
	instructionType.put("add", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("sub", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("sll", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("slt", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("sltu", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("xor", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("srl", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("or", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("and", InstructionTypes.REGISTER_REGISTER);
	instructionType.put("csrrw", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("csrrs", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("scrrc", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("csrrwi", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("csrrsi", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("csrrci", InstructionTypes.IMMEDIATE_CSRR);
	instructionType.put("ecall", InstructionTypes.SPECIAL);
	instructionType.put("ebreak", InstructionTypes.SPECIAL);
	instructionType.put("uret", InstructionTypes.SPECIAL);
	instructionType.put("sret", InstructionTypes.SPECIAL);
	instructionType.put("mret", InstructionTypes.SPECIAL);
	instructionType.put("wfi", InstructionTypes.SPECIAL);
	instructionType.put("lb", InstructionTypes.IMMEDIATE_L);
	instructionType.put("lh", InstructionTypes.IMMEDIATE_L);
	instructionType.put("lw", InstructionTypes.IMMEDIATE_L);
	instructionType.put("lbu", InstructionTypes.IMMEDIATE_L);
	instructionType.put("lhu", InstructionTypes.IMMEDIATE_L);
	instructionType.put("sb", InstructionTypes.STORE);
```

```
        instructionType.put("sh", InstructionTypes.STORE);
        instructionType.put("sw", InstructionTypes.STORE);
        instructionType.put("jal", InstructionTypes.JUMP);
        instructionType.put("jalr", InstructionTypes.JUMP_WITH_REG);
        instructionType.put("beq", InstructionTypes.BRANCH);
        instructionType.put("bne", InstructionTypes.BRANCH);
        instructionType.put("blt", InstructionTypes.BRANCH);
        instructionType.put("bge", InstructionTypes.BRANCH);
        instructionType.put("bltu", InstructionTypes.BRANCH);
        instructionType.put("bgeu", InstructionTypes.BRANCH);
        instructionType.put("mul", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("mulh", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("mulhsu", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("mulhu", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("div", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("divu", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("rem", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("remu", InstructionTypes.REGISTER_REGISTER);
        instructionType.put("unknown_instruction",
InstructionTypes.UNKNOWN_INSTRUCTION);


        registers = new ArrayList<>();
        registers.add(0, "zero");
        registers.add(1, "ra");
        registers.add(2, "sp");
        registers.add(3, "gp");
        registers.add(4, "tp");
        registers.add(5, "t0");
        registers.add(6, "t1");
        registers.add(7, "t2");
        registers.add(8, "s0");
        registers.add(9,  "s1");
        registers.add(10, "a0");
        registers.add(11, "a1");
        registers.add(12, "a2");
        registers.add(13, "a3");
        registers.add(14, "a4");
        registers.add(15, "a5");
        registers.add(16, "a6");
        registers.add(17, "a7");
        registers.add(18, "s2");
        registers.add(19, "s3");
        registers.add(20, "s4");
        registers.add(21, "s5");
        registers.add(22, "s6");
        registers.add(23, "s7");
        registers.add(24, "s8");
        registers.add(25, "s9");
        registers.add(26, "s10");
        registers.add(27, "s11");
```

```
	registers.add(28, "t3");
	registers.add(29, "t4");
	registers.add(30, "t5");
	registers.add(31, "t6");
	}

	public void parse(final InputStream input, final PrintStream out) throws
FileHeaderException, ELFException, IOException {
	data = input.readAllBytes();
	elfParser.parse(data);

	printTextSegment(out, elfParser.getELFSection(".text"),
elfParser.getSymTabMap());
	out.println();
	elfParser.printSymTab(out);

	System.out.println("Success disassembling.");
	}

	private void printTextSegment(final PrintStream out, final ELFSection text,
final Map<Integer, SymbolDescription> symTabMap) {
	int idx = text.sh_offset;
	fillSymTabWithLLabels(text, symTabMap);

	out.println(".text");
	for (int i = 0; i < text.sh_size / 4; i++) {
		final var label = symTabMap.get(text.sh_addr + i * 4);
		if (label != null) {
		out.printf("%08x    <%s>:\n", text.sh_addr + i * 4, label.name);
		}
		final int word = readNBytes(idx, 4);

		final String instruction = getInstruction(word);

		switch (instructionType.get(instruction)) {
		case REGISTER_REGISTER: {
			out.printf(
				"   %05x:\t%08x\t%7s\t%s, %s, %s\n",
				text.sh_addr + i * 4,
				word,
				instruction,
				registers.get((word >> 7) & 0b11111),
				registers.get((word >> 15) & 0b11111),
				registers.get((word >> 20) & 0b11111)
			);
			break;
		}
		case UPPER_IMMEDIATE: {
			out.printf(
				"   %05x:\t%08x\t%7s\t%s, %s\n",
				text.sh_addr + i * 4,
```

```
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        String.valueOf((word >>> 12))
                );
                break;
        }
        case IMMEDIATE: {
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        registers.get((word >> 15) & 0b11111),
                        String.valueOf((word >> 20)) // не менять, иначе
отрицательные значения не появляются
                );
                break;
        }
        case IMMEDIATE_CSRR: {
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        Integer.toString(word >> 20),
                        registers.get((word >> 15) & 0b11111)
                );
                break;
        }
        case IMMEDIATE_L: {
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s(%s)\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        Integer.toString(word >> 20),
                        registers.get((word >> 15) & 0b11111)
                );
                break;
        }
        case STORE: {
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s(%s)\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 20) & 0b11111),
```

```
                        Integer.toString(
                        ((word >> 25) << 5) |
                        ((word >> (11 - 4)) & 0x0000001f)
                        ),
                        registers.get((word >> 15) & 0b11111)
                );
                break;
        }
        case BRANCH: {
                final int jAddr = ((
                        (((word >>> 25) & 0x0000003f) << 5)  |
                        (((word >>> 7) & 0x00000001) << 11)  |
                        (((word >>> 8) & 0x0000000f) << 1)) + text.sh_addr + i
* 4 -
                        (((word >>> 31) == 1) ? ((int) Math.pow(2, 12)) : 0)
                );
                var symDesc = symTabMap.get(jAddr);
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s, %s <%s>\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 15) & 0b11111),
                        registers.get((word >> 20) & 0b11111),
                        Integer.toHexString(jAddr),
                        symDesc.name
                );
                break;
        }
        case JUMP: {
                final int jAddr = ((
                        (((word >>> 21) & 0x000003ff) << 1)  |
                        (((word >>> 20) & 0x00000001) << 11)  |
                        (((word >>> 12) & 0x000000ff) << 12)) + text.sh_addr +
i * 4 -
                        (((word >>> 31) == 1) ? ((int) Math.pow(2, 20)) : 0)
                );
                var symDesc = symTabMap.get(jAddr);
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s <%s>\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        Integer.toHexString(jAddr),
                        symDesc.name
                );
                break;
        }
        case JUMP_WITH_REG: {
                out.printf(
```

```
                        "   %05x:\t%08x\t%7s\t%s, %s(%s)\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        Integer.toHexString(word >> 20),
                        registers.get((word >> 15) & 0b11111)
                );
                break;
            }
            case I_SHAMT: {
                out.printf(
                        "   %05x:\t%08x\t%7s\t%s, %s, %s\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction,
                        registers.get((word >> 7) & 0b11111),
                        registers.get((word >> 15) & 0b11111),
                        String.valueOf((word >>> 20) & 0b11111)
                );
                break;
            }
            case SPECIAL, UNKNOWN_INSTRUCTION: {
                out.printf(
                        "   %05x:\t%08x\t%7s\n",
                        text.sh_addr + i * 4,
                        word,
                        instruction
                );
                break;
            }
            default:
                throw new IllegalStateException("IllegalStateException: Token
for '0x" + Integer.toHexString(word) + "' wasn't created.");
            }

            idx += 4;
        }
        }

        private void fillSymTabWithLLabels(final ELFSection text, final
Map<Integer, SymbolDescription> symTabMap) {
        int idx = text.sh_offset;
        int numOfUndefLabels = 0;

        for (int i = 0; i < text.sh_size / 4; i++) {
            final int word = readNBytes(idx, 4);
            final String instruction = getInstruction(word);

            switch (instructionType.get(instruction)) {
            case BRANCH: {
```

```
                  final int jAddr = ((
                        (((word >>> 25) & 0x0000003f) << 5)  |
                        (((word >>> 7) & 0x00000001) << 11)  |
                        (((word >>> 8) & 0x0000000f) << 1)) + text.sh_addr + i
* 4 -
                        (((word >>> 31) == 1) ? ((int) Math.pow(2, 12)) : 0)
                  );
                  var symDesc = symTabMap.get(jAddr);
                  if (symDesc == null) {
                        symDesc = new SymbolDescription("L" + numOfUndefLabels,
-1, jAddr, 0, "LOCAL", "NOTYPE", "DEFAULT", "UNDEF");
                        symTabMap.put(jAddr, symDesc);
                        numOfUndefLabels++;
                  }
                  break;
            }
            case JUMP: {
                  final int jAddr = ((
                        (((word >>> 21) & 0x000003ff) << 1)  |
                        (((word >>> 20) & 0x00000001) << 11)  |
                        (((word >>> 12) & 0x000000ff) << 12)) + text.sh_addr +
i * 4 -
                        (((word >>> 31) == 1) ? ((int) Math.pow(2, 20)) : 0)
                  );
                  var symDesc = symTabMap.get(jAddr);
                  if (symDesc == null) {
                        symDesc = new SymbolDescription("L" + numOfUndefLabels,
-1, jAddr, 0, "LOCAL", "NOTYPE", "DEFAULT", "UNDEF");
                        symTabMap.put(jAddr, symDesc);
                        numOfUndefLabels++;
                  }
                  break;
            }
            default:
                  break;
            }


            idx += 4;
      }
      }

      private int readNBytes(int idx, final int n) {
      int res = 0;
      for (int i = 0; i < n; i++) {
            res += Byte.toUnsignedInt(data[idx++]) << (8 * i);
      }
      return res;
      }

      private String getInstruction(final int word) {
```

```
return switch (word & 0b1111111) {
	case 0b0110111 -> "lui";
	case 0b0010111 -> "auipc";
	case 0b0010011 -> switch ((word >> 12) & 0b111) {
	case 0b000 -> "addi";
	case 0b010 -> "slti";
	case 0b011 -> "sltiu";
	case 0b100 -> "xori";
	case 0b110 -> "ori";
	case 0b111 -> "andi";
	case 0b001 -> "slli";
	case 0b101 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "srli";
		case 0b0100000 -> "srai";
		default -> "unknown_instruction";
	};
	default -> "unknown_instruction";
	};
	case 0b0110011 -> switch ((word >> 12) & 0b111) {
	case 0b000 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "add";
		case 0b0100000 -> "sub";
		case 0b0000001 -> "mul";
		default -> "unknown_instruction";
	};
	case 0b001 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "sll";
		case 0b0000001 -> "mulh";
		default -> "unknown_instruction";
	};
	case 0b010 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "slt";
		case 0b0000001 -> "mulhsu";
		default -> "unknown_instruction";
	};
	case 0b011 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "sltu";
		case 0b0000001 -> "mulhsu";
		default -> "unknown_instruction";
	};
	case 0b100 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "xor";
		case 0b0000001 -> "div";
		default -> "unknown_instruction";
	};
	case 0b101 -> switch ((word >> 25) & 0b1111111) {
		case 0b0000000 -> "srl";
		case 0b0100000 -> "sra";
		case 0b0000001 -> "divu";
		default -> "unknown_instruction";
	};
```

```
case 0b110 -> switch ((word >> 25) & 0b1111111) {
       case 0b0000000 -> "or";
       case 0b0000001 -> "rem";
       default -> "unknown_instruction";
};
case 0b111 -> switch ((word >> 25) & 0b1111111) {
       case 0b0000000 -> "and";
       case 0b0000001 -> "remu";
       default -> "unknown_instruction";
};
default -> "unknown_instruction";
};
case 0b1110011 -> switch ((word >> 12) & 0b111) {
case 0b001 -> "csrrw";
case 0b010 -> "csrrs";
case 0b011 -> "csrrc";
case 0b101 -> "csrrwi";
case 0b110 -> "csrrsi";
case 0b111 -> "csrrci";
case 0b000 -> switch ((word >> 20) & 0b111111111111) {
       case 0b000000000000 -> "ecall";
       case 0b000000000001 -> "ebreak";
       case 0b000000000010 -> "uret";
       case 0b000100000010 -> "sret";
       case 0b001100000010 -> "mret";
       case 0b000100000101 -> "wfi";
       default -> "unknown_instruction";
};
default -> "unknown_instruction";
};
case 0b0000011 -> switch ((word >> 12) & 0b111) {
case 0b000 -> "lb";
case 0b001 -> "lh";
case 0b010 -> "lw";
case 0b100 -> "lbu";
case 0b101 -> "lhu";
default -> "unknown_instruction";
};
case 0b0100011 -> switch ((word >> 12) & 0b111) {
case 0b000 -> "sb";
case 0b001 -> "sh";
case 0b010 -> "sw";
default -> "unknown_instruction";
};
case 0b1101111 -> "jal";
case 0b1100111 -> "jalr";
case 0b1100011 -> switch ((word >> 12) & 0b111) {
case 0b000 -> "beq";
case 0b001 -> "bne";
case 0b100 -> "blt";
case 0b101 -> "bge";
```

```
                case 0b110 -> "bltu";
                case 0b111 -> "bgeu";
                default -> "unknown_instruction";
                };
                default -> "unknown_instruction";
        };
        }

}
```

SymbolDescription.java

```
public class SymbolDescription {
        public final String name;
        public final int st_name;
        public final int st_value;
        public final int st_size;
        public final String st_bind;
        public final String st_type;
        public final String st_vis;
        public final String st_shndx;

        public SymbolDescription(String name, int st_name, int st_value, int
st_size, String st_bind, String st_type, String st_vis, String st_shndx) {
        this.name = name;
        this.st_name = st_name;
        this.st_value = st_value;
        this.st_size = st_size;
        this.st_bind = st_bind;
        this.st_type = st_type;
        this.st_vis = st_vis;
        this.st_shndx = st_shndx;
        }

        @Override
        public String toString() {
        return "SymbolDescription [name=" + name + ", st_name=" + st_name + ",
st_value=" + st_value + ", st_size="
                + st_size + ", st_bind=" + st_bind + ", st_type=" + st_type + ",
st_vis=" + st_vis + ", st_shndx="
                + st_shndx + "]";
        }
}
```

ParserException.java

```
public abstract class ParserException extends Exception {
        private final int pos;

        public ParserException(final String message, final int pos) {
        super(message);
        this.pos = pos;
        }
```

```
	public int getPos() {
	return pos;
	}
}
```

FileHeaderException.java

```
public class FileHeaderException extends ParserException {
	public FileHeaderException(final String message, final int pos) {
	super(message, pos);
	}
}
```

ELFException.java

```
public class ELFException extends ParserException {
	ELFException(final String message, final int pos) {
	super(message, pos);
	}
}
```

ELFSection.java

```
public class ELFSection {
	public final String s_name;
	public final int sh_name;
	public final int sh_type;
	public final int sh_flags;
	public final int sh_addr;
	public final int sh_offset;
	public final int sh_size;
	public final int sh_link;
	public final int sh_info;
	public final int sh_addralign;
	public final int sh_entsize;

	public ELFSection(String s_name, int sh_name, int sh_type, int sh_flags,
int sh_addr, int sh_offset, int sh_size,
	int sh_link, int sh_info, int sh_addralign, int sh_entsize) {
	this.s_name = s_name;
	this.sh_name = sh_name;
	this.sh_type = sh_type;
	this.sh_flags = sh_flags;
	this.sh_addr = sh_addr;
	this.sh_offset = sh_offset;
	this.sh_size = sh_size;
	this.sh_link = sh_link;
	this.sh_info = sh_info;
	this.sh_addralign = sh_addralign;
	this.sh_entsize = sh_entsize;
	}

	@Override
```

```
	public String toString() {
	return "ELFSection [s_name=" + s_name + ", sh_name=" + sh_name + ",
sh_type=" + sh_type + ", sh_flags="
			+ sh_flags + ", sh_addr=" + sh_addr + ", sh_offset=" + sh_offset + ",
sh_size=" + sh_size + ", sh_link="
			+ sh_link + ", sh_info=" + sh_info + ", sh_addralign=" + sh_addralign
+ ", sh_entsize=" + sh_entsize
			+ "]";
	}
}
```

InstructionTypes.java

```
public enum InstructionTypes {
	UPPER_IMMEDIATE, // lui rd, imm
	STORE, // sb rs2, offset(rs1)
	BRANCH, // beq rs1, rs2, offset
	JUMP, // jal rd, offset
	JUMP_WITH_REG, // jalr rd, rs1, offset
	REGISTER_REGISTER, // add rd, rs1, rs2
	IMMEDIATE_L, // lb rd, offset(rs1)
	IMMEDIATE_CSRR, // csrrw rd, offset, rs1
	IMMEDIATE, // addi rd, rs1, imm
	I_SHAMT, // slli rd, rs1, shamt
	SPECIAL, // ecall
	UNKNOWN_INSTRUCTION;
}
```