Лабораторная работа №3	39	2022
ISA	Ляпин Дмитрий Романович	

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

Инструментарий и требования к работе: Java, JDK-17.0.2.

Описание

Написать дизассемблер – транслятор из машинного кода в текст программы на языке ассемблера. Требуется поддержка набора команд RISC-V RV32I и RV32M. Кодирование little-endian. Требуется обработка секций .text и .symtab (также, чтобы интерпретировать их содержимое, придётся обработать .strtab и .shstrtab)

Вариант

Без вариантов.

Разбор ELF-файла.

Для этой части использовались спецификации с linuxbase[1] и oracle[2], а также для уточнения некоторых моментов – статья Executable and Linkable Format в английской Википедии[3].

1) Заголовок файла.

В отличие от оригинальной спецификации, я буду здесь пользоваться терминами языка программирования java:

Название типа	бит
`byte`	8
`short`	16
`int`	32
`long`	64

Также буду пользоваться `` `word` `` для обозначения слова, соответствующего архитектуре (32-битное для x32 и 64-битное для x64). Все `word`'ы хранятся в типе `long`, чтобы избежать генерификации.

Так как не было явных запретов на поддержку x64 и bigEndian, я сделал и её. Правда, ввиду того, что создавать в java массивы длины более `Integer.MAX_VALUE` невозможно (только списки со специальными костылями), поддержка x64 выражается только в том, что не будет путаницы с длиной слов. Павел Сергеевич заверил меня, что файлы весом более 2Гб = 2^{31}б обрабатывать не обязательно.

Первые четыре байта заголовка - `0x7f, 0x45, 0x4c, 0x46` (DEL, E, L, F) – маркируют начало именно этого формата. Затем идёт байт архитектуры (`0x01` = x32, `0x02` = x64) и байт кодировки (0x`01` = little-endian, `0x02` = big-endian)

Что означает архитектура – указано выше. x64, в отличие от x32, позволяет файлы длиной больше 4Гб (до 16Эб) с корректными ссылками внутри, а также адресацию в памяти 64-битными адресами вместо 32-битных. Кодировка отвечает за то, в каком порядке последовательно идущие байты собираются в `short`, `int`, `long` или `word`.

Пусть последовательно в файле идут байты a, b, c, d.

Little-endian: `int i = a + b << 8 + c << 16 + d << 24`

Big-endian: `int i = a << 24 + b << 16 + c << 8 + d`

[1] https://refspecs.linuxbase.org/elf/elf.pdf
[2] https://docs.oracle.com/cd/E19683-01/816-1386/6m7qcoblj/index.html
[3] https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Затем в файле идут:

byte	version	Версия ELF заголовка. Проверяется, что равна 1
byte	OSABI	Специфичные для операционной системы или ABI расширения, используемые в файле. Описаны в `extenum OS_ABI`.
byte	ABIVERSION	Версия ABI

Вскоре стало ясно, что очень часто встречаются различные перечисления, где некоторые значения имеют фиксированный смысл, а некоторые промежутки – специально зарезервированы для частного использования. Например, упомянутое выше OSABI:

```
OS_ABI values {
   NONE : 0
   HPUX : 1
   NETBSD : 2
   GNU : 3
   SOLARIS : 6
   AIX : 7
   IRIX : 8
   FREEBSD : 9
   TRU64 : 10
   MODESTO : 11
   OPENBSD : 12
   OPENVMS : 13
   NSK : 14
   AROS : 15
   FENIXOS : 16
   CLOUDABI : 17
   OPENVOS : 18
   ProcSpecific : 64..255
}
```

Встроенной конструкции для такого нет, поэтому я решил её создать. Я написал ExtendedEnumPreprocessor на Kotlin 1.5 (файл `preproc/ExtenumPreprocessor.kt`), ищущий в папке проекта файлы расширения .extenum и создающие из них структуру по шаблону. Таким образом описаны `EMachine`, `EType`, `OS_ABI`, `PType`, `SHType`, `SymbolBinding`, `SymbolType`, `SymbolVisibility`.

В дальнейшем поясненения будут приведены только к используемым полям (хотя интерпретированы как отдельные extenum'ы и некоторые неиспользуемые – так как сначала я делал elf-парсер, а потом уже дизассемблер)

Итак, после вышеописанных значений до 16 байт идёт «набивка» - незначащие байты, зарезервированные для будущего использования. После этого:

битность	тип	поле	значение
`short`	`EType`	`eType`	
`short`	`EMachine`	`eMachine`	
`int`		`eVersion`	Версия формата (= 1)
`word`		`eEntry`	Вирт. адрес точки входа
`word`		`ePhOff`	
`word`		`eShOff`	Смещение таблицы section headers
`int`		`eFlags`	
`short`		`eEhSize`	Размер заголовка файла (для проверки)
`short`		`ePhEntSize`	
`short`		`ePhNum`	
`short`		`eShEntSize`	Размер одного section header'а
`short`		`eShNum`	Количество section header'ов
`short`		`eShStrNdx`	Номер section header'а, описывающего таблицу имён `.shstrtab` (см. далее)

Всё вышеперечисленное составляет информацию о файле и хранится в `record MetaInf`.

2) Таблица заголовков секций

Довольно очевидно, что она хранится в файле, начиная с байта `eShOff`, содержит eShNum заголовков секций длиной `eShEntSize` и занимает соответственно `eShNum*eShEntSize` байт. Каждый заголовок секции содержит следующую информацию:

битность	тип	поле	значение
`int`		`shName`	Индекс имени в таблице имён
`int`	`SHType`	`shType`	Тип секции
`word`	`SectionFlags`	`shFlags`	
`word`		`shAddr`	
`word`		`shOffset`	Положение секции в файле
`word`		`shSize`	Размер секции
`int`		`shLink`	Индекс заголовка ассоциированной таблицы (например, таблица имён символов для таблицы символов)
`int`		`shInfo`	
`word`		`shAddrAlign`	
`word`		`shEntSize`	Размер «элемента» секции

3) Секции

Для каждой секции находится имя в таблице имён (`.shstrtab`), которая является StringTable (см. далее), описываемой заголовком секции с индексом `eShStrNdx` (считая с нуля).

Секция с типом NOBITS не занимает места в файле, хотя может иметь shSize, отличный от нуля. Кроме этого, нас интересуют секции типов `PROGBITS`, `SYMTAB` и `STRTAB`.

4) StringTable, в том числе таблица имён.

Здесь посимвольно хранятся строки, разделённые `\u0000`. Например, таблица имён в примере выглядит так:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2FX		Ø	.	s	y	m	t	a	b	Ø	.	s	t	r	t	a
30X	b	Ø	.	s	h	s	t	r	t	a	b	Ø	.	t	e	x
31X	t	Ø	.	b	s	s	Ø	.	c	o	m	m	e	n	t	Ø
32X	.	r	i	s	c	v	.	a	t	t	r	i	b	u	t	e
33X	s	Ø														

Здесь символ `NULL` для простоты восприятия таблицы обозначен через Ø.

По индексу 1 здесь лежит строка `.symtab`, по индексу 9 – `.strtab`, и так далее. В целом, никто не мешает обратиться посреди строки, например, по индексу 22 получить строку `rtab`, но это вносит путаницу, и так лучше не делать.

5) SymbolTable

Эта секция должна иметь тип `SYMTAB`, и в её поле `shLink` должна быть прописана ссылка на таблицу имён символов (соответственно `STRTAB`). Но в целом реализовывать подобное линкование, да и ещё по возможности с сохранением неизменяемости, задача нетривиальная, поэтому просто воспользуемся знанием, что к таблице `.symtab`, которая нас интересует привязана таблица имён `.strtab`.

Таблица символов состоит из `shSize/shEntSize` символов, каждый из которых описывается `shEntSize` байтами:

битность	тип	поле
`int`	Ссылка на имя	`name`
`word`		`value`
`word`		`size`
`byte`	`SymbolType` и `SymbolBinding`	`info`
`byte`	`SymbolVisibility`	`other`
`short`	`SymbolIndex`	`shndx`

`info` описывает атрибуты `type` и `binding`:

```
bind = info >> 4
type = info & 0xf
```

`other` содержит информацию о видимости: `vis = other & 0b11`, и, возможно, какую-то ещё информацию.

shndx содержит либо ссылку на таблицу, к которой привязан символ, либо специальное зарезервированное значение.

При дизассемблировании нас будут интересовать только символы типа `FUNC`, обозначающие соответствующие метки в программе.

Дизассемблирование

Для этой части использовалась, по-видимому, официальная спецификация[1], а также некоторая дополнительная информация со стороннего сайта[2].

Требуется поддержка наборов команд RV32I и RV32M, кодируемых 32-битными словами. В спецификации приведено шесть типов команд, отличаемых по `opcode`:

31 25	24 20	19 15	14 12	11 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|10:5]`	`rs2`	`rs1`	`funct3`	`imm[4:1|11]`	`opcode`	B-type
`imm[31:12]`				`rd`	`opcode`	U-type
`imm[20|10:1|11|19:12]`				`rd`	`opcode`	J-type

Здесь `opcode`, `func3` и `func7` – маркеры команд, `rs1` и `rs2` – регистры аргументов, `rd` – регистр назначения, а `imm` – константа, собираемая по-разному в зависимости от типа команды. Все неуказанные биты `imm` принимаются за `0`.

Полный листинг команд есть в Приложении 1. При разборе нас интересует следующее:

- Команды с двумя младшими битами, не равными `11` – 16-битные. С пятью младшими битами, равными `11111` – 48-битные или больше. Их нам разбирать не нужно.
- Команды с opcode = 1110011 нужно обрабатывать отдельно – это команды `ECALL` и `EBREAK`.
- `opcode` определяет тип:

0110011	R
1100111	I
0000011	I
0010011	I
0100011	S
1100011	B
0110111	U
0010111	U
1101111	J

- Команды с opcode = `0010011` (отнесённые к I-типу) и `funct3 = 001, 101` нужно обрабатывать отдельно – их интерпретация imm существенно отличается.
- Все imm, если не указано остального, воспринимаются как знаковые числа в коде дополнения до двух. Старшим из известных бит `imm` заполняются все остальные до 31 (функция `sext` – sign extension).

[1] https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf
[2] https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html

Обрабатываемые команды

1) R-type

funct7	funct3	opcode		Реализация
0000000	000	0110011	ADD	`x[rd] = x[rs1] + x[rs2]`
0100000	000	0110011	SUB	`x[rd] = x[rs1] - x[rs2]`
0000000	001	0110011	SLL	`x[rd] = x[rs1] << x[rs2]`
0000000	010	0110011	SLT	`x[rd] = x[rs1] <s x[rs2]`
0000000	011	0110011	SLTU	`x[rd] = x[rs1] <u x[rs2]`
0000000	100	0110011	XOR	`x[rd] = x[rs1] ^ x[rs2]`
0000000	101	0110011	SRL	`x[rd] = x[rs1] >>u x[rs2]`
0100000	101	0110011	SRA	`x[rd] = x[rs1] >>s x[rs2]`
0000000	110	0110011	OR	`x[rd] = x[rs1] | x[rs2]`
0000000	111	0110011	AND	`x[rd] = x[rs1] & x[rs2]`
0000001	000	0110011	MUL	`x[rd] = x[rs1] s*s x[rs2]`
0000001	001	0110011	MULH	`x[rd] = (x[rs1] s*s x[rs2]) >>s XLEN`
0000001	010	0110011	MULHSU	`x[rd] = (x[rs1] s*u x[rs2]) >>s XLEN`
0000001	011	0110011	MULHU	`x[rd] = (x[rs1] u*u x[rs2]) >>u XLEN`
0000001	100	0110011	DIV	`x[rd] = x[rs1] /s x[rs2]`
0000001	101	0110011	DIVU	`x[rd] = x[rs1] /u x[rs2]`
0000001	110	0110011	REM	`x[rd] = x[rs1] %s x[rs2]`
0000001	111	0110011	REMU	`x[rd] = x[rs1] %u x[rs2]`

Все эти команды представляют из себя арифметические действия и битовые операции.

Здесь и далее операторы `/s`, `%s`, `>>s` и т.д. означают действия над числами как над знаковыми, `/u`, `%u`, `>>u` и т.д. – как над беззнаковыми. `s*s`, `s*u`, `u*s`, `u*u` указывают на то, что операнды могут быть разных типов (`s` и `u`). `XLEN` означает длину слова и в нашем случае всюду равно 32. Результат логических операций интерпретируется как `1` – истина, `0` – ложь, что позволяет оперировать логическими значениями с помощью битовых операций над ними как над числами. `x[·]` означает запись или чтение в/из регистров.

В выводе все команды этого типа форматируются как `command rd, rs1, rs2`.

2) I-type

funct3	opcode		Реализация

000	1100111	JALR	`t =pc+4;`
			`pc=(x[rs1]+ imm)&~1;`
			`x[rd]=t;`

000	0000011	LB	`x[rd] = sext(M[x[rs1] + imm][7:0])`
001	0000011	LH	`x[rd] = sext(M[x[rs1] + imm][15:0])`
010	0000011	LW	`x[rd] = sext(M[x[rs1] + imm][31:0])`
100	0000011	LBU	`x[rd] = M[x[rs1] + imm][7:0]`
101	0000011	LHU	`x[rd] = M[x[rs1] + imm 15:0]`
000	0010011	ADDI	`x[rd] = x[rs1] + imm`
010	0010011	SLTI	`x[rd] = x[rs1] <s imm`
011	0010011	SLTIU	`x[rd] = x[rs1] <u imm`
100	0010011	XORI	`x[rd] = x[rs1] ^ imm`
110	0010011	ORI	`x[rd] = x[rs1] | imm`
111	0010011	ANDI	`x[rd] = x[rs1] & imm`

`M[·]` – обращение к памяти,

`number[x:y]` – взятие бит `x..y` (включительно) числа `number`,

`pc` – указатель на текущую команду.

`JALR` и load-команды выводятся в формате `command rd, imm(rs1)`.

Остальные – `command rd, rs1, imm`.

К этому типу я отнёс и команды `slli`, `srli`, srai, поскольку у них в битах `24..20` содержится не `rs2`, а константа-аргумент:

31 25	24 20	19 15	14 12	11 7	6 0		Реализация
0000000	shamt	rs1	001	rd	0010011	SLLI	`x[rd] = x[rs1] << shamt`
0000000	shamt	rs1	101	rd	0010011	SRLI	`x[rd] = x[rs1] >>u shamt`
0100000	shamt	rs1	101	rd	0010011	SRAI	`x[rd] = x[rs1] >>s shamt`

Вывод: `command rd, rs, shamt`

3) S-type

К этому типу относятся store-команды:

funct3	opcode		Реализация	Комментарий
000	0100011	SB	`M[x[rs1] + imm] = x[rs2][7:0]`	Выгрузка в память 8-битного числа
001	0100011	SH	`M[x[rs1] + imm] = x[rs2][15:0]`	Выгрузка в память 16-битного числа
010	0100011	SW	`M[x[rs1] + imm] = x[rs2][31:0]`	Выгрузка в память 32-битного числа

Вывод: `command rs2, imm(rs1)`

4) U-type

opcode		Реализация
0110111	LUI	`x[rd] = imm`
0010111	AUIPC	`x[rd] = pc + imm`

Вывод:

`command rd, 0xImmHex`

`imm` выводится в hex формате, как того требует ТЗ. Фактически, я всё равно вывожу `imm`, а не то, что дано (`imm[31:12]`), так что это, на мой взгляд, особого значения не имеет.

5) B-type

Это команды перехода:

funct3	opcode		Реализация
000	1100011	BEQ	`if (x[rs1] == x[rs2]) pc += offset`
001	1100011	BNE	`if (x[rs1] != x[rs2]) pc += offset`
100	1100011	BLT	`if (x[rs1] <s x[rs2]) pc += offset`
101	1100011	BGE	`if (x[rs1] >=s x[rs2]) pc += offset`
110	1100011	BLTU	`if (x[rs1] >u x[rs2]) pc += offset`
111	1100011	BGEU	`if (x[rs1] >=u x[rs2]) pc += offset`

Вывод: `command rs1, rs2, addr <label>`

6) J-type

opcode		Реализация
1101111	JAL	`x[rd] = pc+4;` `pc += offset`

Вывод: `jal rd, addr <label>`

Работа программы

Программа разбита на две глобальных части: разбор elf-файла и разбор команд.

Разбором отдельных байт и конкатенацией их в нужного размера числа производится классом `BytearrParser`. `Byte`, `short` и `int` хранятся в типе `int` – это связано с тем, что многие операции над ними возвращают именно `int`, который нужно явно приводить обратно.

Метод `elf.parser.ElfParser.parse(byte[])` собирает из переданных данных elf-файл. Производится проверка, что это elf (четыре первых байта), определяется кодировка и архитектура. Затем создаётся `BytearrParser` с учётом этих данных. Разбирается метаинформация (заголовок), затем заголовки секций. Создаётся таблица имён, для каждого заголовка секции находится имя, затем по имени производится поиск таблиц .strtab, .symtab и .text.

За дизассемблирование отвечает метод
`disassembly.RiscvDisassembler.disassemble (Section, boolean, Architecture, Symbol[])`.
Здесь (для удобства) создаётся ещё один `BytearrParser` для работы с содержимым `.text`. Секция бьётся на куски по 4 байта, каждые 4 байта собираются в `int` и отдаются в `disassembly.Command.of(int)`, возвращающую собственно команду (инструкцию). Внутри этого метода производится проверка `opcode`, соответсвтенно ему определяется тип команды (для каждого типа отдельный класс, ещё есть отдельный класс `Named` для `ecall`, `ebreak` и неизвестных инструкций). Затем из переданного массива символов ищутся символы типа `FUNC`, метки записываются в `Map<Integer, String> labels`. Затем для каждой команды определяется, не порождает ли она какую-либо ещё метку, и создаются метки `L0`, `L1`, `L2` …

Создаются команды-«заглушки», содержащие только адрес и метку, затем всё вместе сортируется по адресу и выводится.

Вывод программы для тестового файла

```
.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, 0x100000
   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, 0x11000
   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, a1, a4
   100f8:  00e60633   add        a2, a4, a2
   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	
[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

Приложение 1. Перечень команд

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred | succ		00000	000	00000	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

RV32M Standard Extension

0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Приложение 2. Листинг кода

bytearrparser/BytearrParser.java

```
package bytearrparser;

import elf.enums.Architecture;

public class BytearrParser {
    private final byte[] data;
    private final boolean littleEndian;
    private final Architecture arch;
    private int currentPos = 0;

    public BytearrParser(byte[] data, boolean littleEndian, Architecture
arch) {
        this.data = data;
        this.littleEndian = littleEndian;
        this.arch = arch;
    }

    public void moveTo(int newPos) {
        if (newPos < 0) throw new IllegalArgumentException();
        checkOpen();
        currentPos = newPos;
    }

    private int byteFrom(int pos) {
        return ((int) data[pos]) & 0xff;
    }

    public int shortFrom(int pos) {
        return (byteFrom(pos) << (littleEndian ? 0 : 8)) | (byteFrom(pos +
1) << (littleEndian ? 8 : 0));
    }

    public int intFrom(int pos) {
        return (shortFrom(pos) << (littleEndian ? 0 : 16)) |
(shortFrom(pos + 2) << (littleEndian ? 16 : 0));
    }

    public long longFrom(int pos) {
        return ((long) intFrom(pos) & 0xffff_ffffL) << (littleEndian ? 0 :
32) |
                ((long) intFrom(pos + 4) & 0xffff_ffffL) << (littleEndian
? 32 : 0);
    }

    public int nextByte() {
        checkOpen();
        return byteFrom(currentPos++);
    }

    public int nextShort() {
        checkOpen();
        return shortFrom((currentPos += 2) - 2);
    }
```

```
    public int nextInt() {
        checkOpen();
        return intFrom((currentPos += 4) - 4);
    }

    public long nextLong() {
        checkOpen();
        return longFrom((currentPos += 8) - 8);
    }

    public long nextWord() {
        checkOpen();
        return switch (arch) {
            case x32 -> nextInt();
            case x64 -> nextLong();
            default -> throw new AssertionError();
        };
    }

    public long wordFrom(int pos) {
        return switch (arch) {
            case x32 -> intFrom(pos);
            case x64 -> longFrom(pos);
            default -> throw new AssertionError();
        };
    }

    public String nullTerminatedStringFrom(int pos) {
        StringBuilder sb = new StringBuilder();
        char c = (char) byteFrom(pos++);
        while (c != '\u0000') {
            sb.append(c);
            c = (char) byteFrom(pos++);
        }
        return sb.toString();
    }

    public void close() {
        currentPos = -1;
    }

    private void checkOpen() {
        if (currentPos == -1) {
            throw new IllegalStateException();
        }
    }

    public byte[] slice(int offset, int size) {
        byte[] res = new byte[size];
        System.arraycopy(data, offset, res, 0, size);
        return res;
    }
}
```

disassembly/RiscvDisassembler.java

```
package disassembly;

import bytearrparser.BytearrParser;
import elf.enums.Architecture;
import elf.enums.SymbolType;
import elf.objects.Section;
import elf.objects.Symbol;
import util.Utilities;

import java.util.*;

public class RiscvDisassembler {
    public static String disassemble(Section progbits, boolean
littleEndian, Architecture arch, Symbol[] symbols) {
        byte[] bytes = progbits.rawContent();
        BytearrParser parser = new BytearrParser(bytes, littleEndian,
arch);
        int commandsNum = bytes.length / 4; //Currently only 32-bit
commands are supported
        List<Command> commands = new ArrayList<>(commandsNum);
        for (int i = 0; i < commandsNum; i++) {
            int code = parser.nextInt();
            if ((code & 3) != 3) throw new IllegalArgumentException("16-
bit instructions are not supported");
            if (((code >> 2) & 7) == 7)
                throw new IllegalArgumentException("more-than-32-bit
instructions are not supported");
            commands.add(Command.of(code,
Utilities.assertIsInt(progbits.virtualAddress + i * 4)));
        }
        Map<Integer, String> labels = new HashMap<>();
        for (Symbol sym : symbols) {
            if (sym.type() != SymbolType.List.FUNC) continue;
            labels.put(Utilities.assertIsInt(sym.value()), sym.name());
        }
        int iter = 0;
        for (Command com : commands) {
            if (!com.links()) continue;
            int to = com.link();
            if (labels.containsKey(to)) continue;
            labels.put(to, "L" + (iter++));
        }
        labels.forEach((i, s) -> commands.add(new LabelCommandStub(0, i,
s)));
        commands.sort(
                Comparator.comparingInt(c -> ((Command) c).address)
                        .thenComparing((o1, o2) -> o1 instanceof
LabelCommandStub ? -1 : o2 instanceof LabelCommandStub ? 1 : 0)
        );
        StringBuilder res = new StringBuilder();
        for (Command com : commands) {
            res.append(com.toString(com.links() ? labels.get(com.link()) :
null));
        }
        return res.toString();
    }
}
```

disassembly/Command.java

```
package disassembly;

@SuppressWarnings("unused")
public abstract sealed class Command permits BCommand, Command.Named,
ICommand, JCommand, RCommand, SCommand, UCommand, LabelCommandStub {
    protected static final String UNKNOWN_INSTRUCTION =
"unknown_instruction";
    public final int code;
    public final int address;

    public Command(int code, int address) {
        this.code = code;
        this.address = address;
    }

    public final String hexCode() {
        String res = Integer.toUnsignedString(code, 16);
        return "0".repeat(8 - res.length()) + res;
    }

    protected static int slice(int i, int from, int to) {
        return (i >>> to) & (1 << from - to + 1) - 1;
    }

    protected static final String[] REGISTER_MNEMONIC = {"zero", "ra",
"sp", "gp", "tp", "t0", "t1", "t2", "s0", "s1", "a0", "a1", "a2", "a3",
"a4", "a5", "a6", "a7", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9",
"s10", "s11", "t3", "t4", "t5", "t6"};

    protected static String toHex(int i) {
        return (i < 0 ? "-" : "") + "0x" +
Integer.toUnsignedString(Math.abs(i), 16);
    }


    protected static int extendHighest(int i, int highest) {
        return (i >> highest & 1) == 0 ? i : -1 >> highest << highest | i;
    }

    public abstract String name();

    public String args() {
        return "";
    }

    public boolean links() {
        return false;
    }

    public int link() {
        return 0;
    }

    private static String pad(String s, int toLen) {
```

```
        return s + (s.length() < toLen ? " ".repeat(toLen - s.length()) :
"");
    }

    public String toString(String withLabel) {
        return (withLabel == null ? "   %05x:\t%08x\t%s\t%s\n" : "
%05x\t%08x\t%s\t%s <%s>\n")
                .formatted(address, code, pad(name(), 7), args(),
withLabel);
    }

    static final class Named extends Command {
        private final String name;

        public Named(int code, int address, String name) {
            super(code, address);
            this.name = name;
        }

        @Override
        public String name() {
            return name;
        }
    }

    @Override
    public String toString() {
        return toString(null);
    }

    public static Command of(int code, int address) {
        return switch (slice(code, 6, 2)) {
            case 0b01101 -> new UCommand(code, address); //LUI
            case 0b00101 -> new UCommand(code, address); //AUIPC
            case 0b11011 -> new JCommand(code, address); //JAL
            case 0b11001 -> new ICommand(code, address); //JALR
            case 0b11000 -> new BCommand(code, address);
            case 0b00000 -> new ICommand(code, address);
            case 0b01000 -> new SCommand(code, address);
            case 0b00100 -> new ICommand(code, address);
            case 0b01100 -> new RCommand(code, address);
            case 0b11100 -> switch (code) {
                case 0x00000073 -> new Named(code, address, "ecall");
                case 0x00100073 -> new Named(code, address, "ebreak");
                default -> new Named(code, address, UNKNOWN_INSTRUCTION);
            };
            default -> new Named(code, address, UNKNOWN_INSTRUCTION);
        };
    }
}
```

disassembly/RCommand.java

```
package disassembly;

final class RCommand extends Command {
    public final int funct7;
```

```
    public final int rs2;
    public final int rs1;
    public final int funct3;
    public final int rd;
    public final int opcode;

    public RCommand(int code, int address) {
        super(code, address);
        funct7 = slice(code, 31, 25);
        rs2 = slice(code, 24, 20);
        rs1 = slice(code, 19, 15);
        funct3 = slice(code, 14, 12);
        rd = slice(code, 11, 7);
        opcode = slice(code, 6, 0);
    }

    public String name() {
        return switch (funct7 << 10 | funct3 << 7 | opcode) {
            case 0b00000000000110011 -> "add";
            case 0b01000000000110011 -> "sub";
            case 0b00000000010110011 -> "sll";
            case 0b00000000100110011 -> "slt";
            case 0b00000000110110011 -> "sltu";
            case 0b00000001000110011 -> "xor";
            case 0b00000001010110011 -> "srl";
            case 0b01000001010110011 -> "sra";
            case 0b00000001100110011 -> "or";
            case 0b00000001110110011 -> "and";
            case 0b00000010000110011 -> "mul";
            case 0b00000010010110011 -> "mulh";
            case 0b00000010100110011 -> "mulhsu";
            case 0b00000010110110011 -> "mulhu";
            case 0b00000011000110011 -> "div";
            case 0b00000011010110011 -> "divu";
            case 0b00000011100110011 -> "rem";
            case 0b00000011110110011 -> "remu";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public String args() {
        return "%s, %s, %s".formatted(REGISTER_MNEMONIC[rd],
REGISTER_MNEMONIC[rs1], REGISTER_MNEMONIC[rs2]);
    }


}
```

disassembly/ICommand.java

```
package disassembly;

final class ICommand extends Command {
    public final int imm;
    public final int rs1;
    public final int funct3;
```

```
    public final int rd;
    public final int opcode;

    public ICommand(int code, int address) {
        super(code, address);
        imm = extendHighest(slice(code, 31, 20), 11); //imm11_0
        rs1 = slice(code, 19, 15);
        funct3 = slice(code, 14, 12);
        rd = slice(code, 11, 7);
        opcode = slice(code, 6, 0);
    }

    public String name() {
        return switch (funct3 << 7 | opcode) {
            case 0b0001100111 -> "jalr";
            case 0b0000000011 -> "lb";
            case 0b0010000011 -> "lh";
            case 0b0100000011 -> "lw";
            case 0b1000000011 -> "lbu";
            case 0b1010000011 -> "lhu";
            case 0b0000010011 -> "addi";
            case 0b0100010011 -> "slti";
            case 0b0110010011 -> "sltiu";
            case 0b1000010011 -> "xori";
            case 0b1100010011 -> "ori";
            case 0b1110010011 -> "andi";
            case 0b0010010011 -> "slli";
            case 0b1010010011 -> slice(imm, 10, 10) == 1 ? "srai" :
"srli";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public String args() {
        String name = name();
        return switch (name) {
            case "jalr", "lb", "lh", "lw", "lbu", "lhu" -> "%s,
%d(%s)".formatted(REGISTER_MNEMONIC[rd], imm, REGISTER_MNEMONIC[rs1]);
            case "slli", "srai", "srli" -> "%s, %s,
0x%x".formatted(REGISTER_MNEMONIC[rd], REGISTER_MNEMONIC[rs1], slice(imm,
4, 0));
            default -> "%s, %s, %d".formatted(REGISTER_MNEMONIC[rd],
REGISTER_MNEMONIC[rs1], imm);
        };
    }
}
```

disassembly/SCommand.java

```
package disassembly;

final class SCommand extends Command {
    public final int imm;
    public final int rs2;
    public final int rs1;
    public final int funct3;
```

```
    public final int opcode;

    public SCommand(int code, int address) {
        super(code, address);
        int imm11_5 = slice(code, 31, 25);
        rs2 = slice(code, 24, 20);
        rs1 = slice(code, 19, 15);
        funct3 = slice(code, 14, 12);
        int imm4_0 = slice(code, 11, 7);
        opcode = slice(code, 6, 0);
        imm = extendHighest(imm11_5 << 5 | imm4_0, 11);
    }

    public String name() {
        return switch (funct3 << 7 | opcode) {
            case 0b0000100011 -> "sb";
            case 0b0010100011 -> "sh";
            case 0b0100100011 -> "sw";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public String args() {
        return "%s, %d(%s)".formatted(REGISTER_MNEMONIC[rs2], imm,
REGISTER_MNEMONIC[rs1]);
    }
}
```

disassebly/UCommand.java

```
package disassembly;

final class UCommand extends Command {
    public final int imm;
    public final int rd;
    public final int opcode;

    public UCommand(int code, int address) {
        super(code, address);
        int imm31_12 = slice(code, 31, 12);
        rd = slice(code, 11, 7);
        opcode = slice(code, 6, 0);
        imm = imm31_12 << 12;
    }

    public String name() {
        return switch (opcode) {
            case 0b0110111 -> "lui";
            case 0b0010111 -> "auipc";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public String args() {
        return switch (opcode) {
```

```
            case 0b0110111 -> "%s, 0x%x".formatted(REGISTER_MNEMONIC[rd],
imm);
            case 0b0010111 -> "%s, %d".formatted(REGISTER_MNEMONIC[rd],
imm);
            default -> UNKNOWN_INSTRUCTION;
        };
    }
}
```

disassembly/BCommand.java

```
package disassembly;

final class BCommand extends Command {
    public final int imm;
    public final int rs2;
    public final int rs1;
    public final int funct3;
    public final int opcode;

    public BCommand(int code, int address) {
        super(code, address);
        int imm12 = slice(code, 31, 31);
        int imm10_5 = slice(code, 30, 25);
        rs2 = slice(code, 24, 20);
        rs1 = slice(code, 19, 15);
        funct3 = slice(code, 14, 12);
        int imm4_1 = slice(code, 11, 8);
        int imm11 = slice(code, 7, 7);
        opcode = slice(code, 6, 0);
        imm = Command.extendHighest(imm12 << 12 | imm11 << 11 | imm10_5 <<
5 | imm4_1 << 1, 12);
    }

    public String name() {
        return switch (funct3 << 7 | opcode) {
            case 0b0001100011 -> "beq";
            case 0b0011100011 -> "bne";
            case 0b1001100011 -> "blt";
            case 0b1011100011 -> "bge";
            case 0b1101100011 -> "bltu";
            case 0b1111100011 -> "bgeu";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public boolean links() {
        return !name().equals(UNKNOWN_INSTRUCTION);
    }

    @Override
    public int link() {
        return address + imm;
    }

    @Override
```

```
    public String args() {
        return (links() ? "%s, %s, %x" : "%s, %s,
%d").formatted(REGISTER_MNEMONIC[rs1], REGISTER_MNEMONIC[rs2], links() ?
link() : imm);
    }
}
```

disassembly/JCommand.java

```
package disassembly;

final class JCommand extends Command {
    public final int imm;
    public final int rd;
    public final int opcode;

    public JCommand(int code, int address) {
        super(code, address);
        int imm20 = slice(code, 31, 31);
        int imm10_1 = slice(code, 30, 21);
        int imm11 = slice(code, 20, 20);
        int imm19_12 = slice(code, 19, 12);
        rd = slice(code, 11, 7);
        opcode = slice(code, 6, 0);
        imm = Command.extendHighest(imm20 << 20 | imm19_12 << 12 | imm11
<< 11 | imm10_1 << 1, 20);
    }

    public String name() {
        return switch (opcode) {
            case 0b1101111 -> "jal";
            default -> UNKNOWN_INSTRUCTION;
        };
    }

    @Override
    public String args() {
        return (links() ? "%s, %x" : "%s,
%d").formatted(REGISTER_MNEMONIC[rd], links() ? link() : imm);
    }

    @Override
    public boolean links() {
        return opcode == 0b1101111;
    }

    @Override
    public int link() {
        return address + imm;
    }
}
```

disassembly/LabelCommandStub.java

```
package disassembly;

public final class LabelCommandStub extends Command {
    public final String label;
```

```
    public LabelCommandStub(int code, int address, String label) {
        super(code, address);
        this.label = label;
    }

    @Override
    public String name() {
        return "";
    }

    @Override
    public String toString(String withLabel) {
        return "%08x   <%s>:\n".formatted(address, label);
    }
}
```

elf/enums/Architecture.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

public enum Architecture {
    x32(1),
    x64(2),
    ;
    final int value;

    Architecture(int value) {
        this.value = value;
    }

    public int value() {
        return value;
    }

    public static Architecture of(int value) {
        for (Architecture it : Architecture.values()) {
            if (it.value() == value) {
                return it;
            }
        }
        throw new IncorrectFileSignature("Incorrect value " + value + "
for Architecture");
    }
}
```

elf/enums/EMachine.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
```

```
public interface EMachine {
    int value();

    enum List implements EMachine {
        NONE(0),
        M32(1),
        SPARC(2),
        IAMCU(6),
        MIPS(8),
        S370(9),
        MIPS_RS3_LE(10),
        PARISC(15),
        PPC(20),
        PPC64(21),
        S390(22),
        SPU(23),
        V800(36),
        FR20(37),
        RH32(38),
        MCORE(39),
        RCE(39),
        ARM(40),
        OLD_ALPHA(41),
        SH(42),
        SPARCV9(43),
        TRICORE(44),
        ARC(45),
        H8_300(46),
        H8_300H(47),
        H8S(48),
        H8_500(49),
        IA_64(50),
        MIPS_X(51),
        COLDFIRE(52),
        MMA(54),
        PCP(55),
        NCPU(56),
        NDR1(57),
        STARCORE(58),
        ME16(59),
        ST100(60),
        TINYJ(61),
        X86_64(62),
        MCST_ELBRUS(175),
        TI_C6000(140),
        AARCH64(183),
        RISCV(243),
        BPF(247),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
```

```
            return value;
        }
    }


    static EMachine of(int value) {

            for (EMachine it : EMachine.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for EMachine");

    }
}
```

elf/enums/EType.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface EType {
    int value();

    enum List implements EType {
        NONE(0),
        REL(1),
        EXEC(2),
        DYN(3),
        CORE(4),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }

    record ProcSpecific(int value) implements EType {
        @Override
        public String toString() {
            return "ProcSpecific(" + value + ")";
        }
    }

    record OSSpecific(int value) implements EType {
```

```
        @Override
        public String toString() {
            return "OSSpecific(" + value + ")";
        }
    }

    static EType of(int value) {
        if (65280 <= value && value <= 65535) {
            return new ProcSpecific(value);
        } else if (65024 <= value && value <= 65279) {
            return new OSSpecific(value);
        } else {
            for (EType it : EType.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for EType");
        }
    }
}
```

elf/enums/metainf.extenum

```
OS_ABI values {
   NONE : 0
   HPUX : 1
   NETBSD : 2
   GNU : 3
   SOLARIS : 6
   AIX : 7
   IRIX : 8
   FREEBSD : 9
   TRU64 : 10
   MODESTO : 11
   OPENBSD : 12
   OPENVMS : 13
   NSK : 14
   AROS : 15
   FENIXOS : 16
   CLOUDABI : 17
   OPENVOS : 18
   ProcSpecific : 64..255
}

EType values {
    NONE : 0
    REL : 1
    EXEC : 2
    DYN : 3
    CORE : 4
    ProcSpecific : 65280..65535
    OSSpecific : 65024..65279
}

EMachine values {
```

```
   NONE : 0x0
   M32 : 0x01
   SPARC : 0x02
   386 : 0x03
   68K : 0x04
   88K : 0x05
   IAMCU : 0x06
   860 : 0x07
   MIPS : 0x08
   S370 : 0x09
   MIPS_RS3_LE : 0x0A
   PARISC : 0x0F
   960 : 0x13
   PPC : 0x14
   PPC64 : 0x15
   S390 : 0x16
   SPU : 0x17
   V800 : 0x24
   FR20 : 0x25
   RH32 : 0x26
   MCORE : 0x27
   RCE : 0x27
   ARM : 0x28
   OLD_ALPHA : 0x29
   SH : 0x2A
   SPARCV9 : 0x2B
   TRICORE : 0x2C
   ARC : 0x2D
   H8_300 : 0x2E
   H8_300H : 0x2F
   H8S : 0x30
   H8_500 : 0x31
   IA_64 : 0x32
   MIPS_X : 0x33
   COLDFIRE : 0x34
   68HC12 : 0x35
   MMA : 0x36
   PCP : 0x37
   NCPU : 0x38
   NDR1 : 0x39
   STARCORE : 0x3A
   ME16 : 0x3B
   ST100 : 0x3C
   TINYJ : 0x3D
   X86_64 : 0x3E
   MCST_ELBRUS : 0xAF
   TI_C6000 : 0x8C
   AARCH64 : 0xB7
   RISCV : 0xF3
   BPF : 0xF7
   65816 : 0x101
}

PType values {
   NULL : 0
   LOAD : 1
   DYNAMIC : 2
```

```
   INTERP : 3
   NOTE : 4
   SHLIB : 5
   PHDR : 6
   TLS : 7
   OSSpecific : 0x60000000..0x6FFFFFFF
   ProcSpecific : 0x70000000..0x7FFFFFFF
}

SHType values {
   NULL : 0x0
   PROGBITS : 0x1
   SYMTAB : 0x2
   STRTAB : 0x3
   RELA : 0x4
   HASH : 0x5
   DYNAMIC : 0x6
   NOTE : 0x7
   NOBITS : 0x8
   REL : 0x9
   SHLIB : 0x0A
   DYNSYM : 0x0B
   INIT_ARRAY : 0x0E
   FINI_ARRAY : 0x0F
   PREINIT_ARRAY : 0x10
   GROUP : 0x11
   SYMTAB_SHNDX : 0x12
   NUM : 0x13
   OSSpecific : 0x60000000..0x7FFFFFFF
}
```

elf/enums/OS_ABI.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface OS_ABI {
    int value();

    enum List implements OS_ABI {
        NONE(0),
        HPUX(1),
        NETBSD(2),
        GNU(3),
        SOLARIS(6),
        AIX(7),
        IRIX(8),
        FREEBSD(9),
        TRU64(10),
        MODESTO(11),
        OPENBSD(12),
        OPENVMS(13),
```

```
        NSK(14),
        AROS(15),
        FENIXOS(16),
        CLOUDABI(17),
        OPENVOS(18),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }

    record ProcSpecific(int value) implements OS_ABI {
        @Override
        public String toString() {
            return "ProcSpecific(" + value + ")";
        }
    }

    static OS_ABI of(int value) {
        if (64 <= value && value <= 255) {
            return new ProcSpecific(value);
        } else {
            for (OS_ABI it : OS_ABI.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for OS_ABI");
        }
    }
}
```

elf/enums/PType.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface PType {
    int value();

    enum List implements PType {
        NULL(0),
        LOAD(1),
        DYNAMIC(2),
        INTERP(3),
```

```
        NOTE(4),
        SHLIB(5),
        PHDR(6),
        TLS(7),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }

    record OSSpecific(int value) implements PType {
        @Override
        public String toString() {
            return "OSSpecific(" + value + ")";
        }
    }

    record ProcSpecific(int value) implements PType {
        @Override
        public String toString() {
            return "ProcSpecific(" + value + ")";
        }
    }

    static PType of(int value) {
        if (1610612736 <= value && value <= 1879048191) {
            return new OSSpecific(value);
        } else if (1879048192 <= value && value <= 2147483647) {
            return new ProcSpecific(value);
        } else {
            for (PType it : PType.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for PType");
        }
    }
}
```

elf/enums/SectionFlags.java

```
package elf.enums;

public class SectionFlags {
    private static final long SHF_WRITE = 0x1;
    private static final long SHF_ALLOC = 0x2;
    private static final long SHF_EXECINSTR = 0x4;
    private static final long SHF_MERGE = 0x10;
```

```
    private static final long SHF_STRINGS = 0x20;
    private static final long SHF_INFO_LINK = 0x40;
    private static final long SHF_LINK_ORDER = 0x80;
    private static final long SHF_OS_NONCONFORMING = 0x100;
    private static final long SHF_GROUP = 0x200;
    private static final long SHF_TLS = 0x400;
    private static final long SHF_MASKOS = 0x0FF00000;
    private static final long SHF_MASKPROC = 0xF0000000;
    private static final long SHF_ORDERED = 0x4000000;
    private static final long SHF_EXCLUDE = 0x8000000;

    public final long value;
    public final boolean writable; //Writable
    public final boolean allocates; //Occupies memory during execution
    public final boolean executable; //Executable
    public final boolean mergable; //Might be merged
    public final boolean strings; //Contains null-terminated strings
    public final boolean infoLinkPresent; //'sh_info' contains SHT index
    public final boolean preserveOrderAfterCombining; //Preserve order
after combining
    public final boolean osSpecificHandlingRequired; //Non-standard OS
specific handling required
    public final boolean memberOfGroup; //Section is member of a group
    public final boolean threadLocal; //Section hold thread-local data
    public final long osSpecific; //OS-specific
    public final long procSpecific; //Processor-specific
    public final boolean specialOrderingRequired; //Special ordering
requirement (Solaris)
    public final boolean excludedUnlessReferencedOrAllocated; //Section is
excluded unless referenced or allocated (Solaris)

    public SectionFlags(long value) {
        this.value = value;
        writable = (value & SHF_WRITE) != 0;
        allocates = (value & SHF_ALLOC) != 0;
        executable = (value & SHF_EXECINSTR) != 0;
        mergable = (value & SHF_MERGE) != 0;
        strings = (value & SHF_STRINGS) != 0;
        infoLinkPresent = (value & SHF_INFO_LINK) != 0;
        preserveOrderAfterCombining = (value & SHF_LINK_ORDER) != 0;
        osSpecificHandlingRequired = (value & SHF_OS_NONCONFORMING) != 0;
        memberOfGroup = (value & SHF_GROUP) != 0;
        threadLocal = (value & SHF_TLS) != 0;
        osSpecific = value & SHF_MASKOS;
        procSpecific = value & SHF_MASKPROC;
        specialOrderingRequired = (value & SHF_ORDERED) != 0;
        excludedUnlessReferencedOrAllocated = (value & SHF_EXCLUDE) != 0;
    }

    @Override
    public String toString() {
        return "SectionFlags {\n" +
                "value = " + value + "\n" +
                (writable ? "V" : "X") + " writable\n" +
                (allocates ? "V" : "X") + " allocates\n" +
                (executable ? "V" : "X") + " executable\n" +
                (mergable ? "V" : "X") + " mergable\n" +
```

```
                (strings ? "V" : "X") + " strings\n" +
                (infoLinkPresent ? "V" : "X") + " info link present\n" +
                (preserveOrderAfterCombining ? "V" : "X") + " preserve
order after combining\n" +
                (osSpecificHandlingRequired ? "V" : "X") + " os specific
handling required\n" +
                (memberOfGroup ? "V" : "X") + " member of group\n" +
                (threadLocal ? "V" : "X") + " thread local\n" +
                (osSpecific) + " os specific\n" +
                (procSpecific) + " proc specific\n" +
                (specialOrderingRequired ? "V" : "X") + " special ordering
required\n" +
                (excludedUnlessReferencedOrAllocated ? "V" : "X") + "
excluded unless referenced or allocated\n" +
                '}';
    }
}
```

elf/enums/SHType.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface SHType {
    int value();

    enum List implements SHType {
        NULL(0),
        PROGBITS(1),
        SYMTAB(2),
        STRTAB(3),
        RELA(4),
        HASH(5),
        DYNAMIC(6),
        NOTE(7),
        NOBITS(8),
        REL(9),
        SHLIB(10),
        DYNSYM(11),
        INIT_ARRAY(14),
        FINI_ARRAY(15),
        PREINIT_ARRAY(16),
        GROUP(17),
        SYMTAB_SHNDX(18),
        NUM(19),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
```

```
        public int value() {
            return value;
        }
    }

    record OSSpecific(int value) implements SHType {
        @Override
        public String toString() {
            return "OSSpecific(" + value + ")";
        }
    }

    static SHType of(int value) {
        if (1610612736 <= value && value <= 2147483647) {
            return new OSSpecific(value);
        } else {
            for (SHType it : SHType.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for SHType");
        }
    }
}
```

elf/enums/symbol.extenum

```
SymbolType values {
    NOTYPE : 0
    OBJECT : 1
    FUNC : 2
    SECTION : 3
    FILE : 4
    ProcSpecific : 13..15
}

SymbolBinding values {
   LOCAL : 0
   GLOBAL : 1
   WEAK : 2
   ProcSpecific : 13..15
}

SymbolVisibility values {
    DEFAULT : 0
    INTERNAL : 1
    HIDDEN : 2
    PROTECTED : 3
}
```

elf/enums/SymbolBinding.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;
```

```
/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface SymbolBinding {
    int value();

    enum List implements SymbolBinding {
        LOCAL(0),
        GLOBAL(1),
        WEAK(2),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }

    record ProcSpecific(int value) implements SymbolBinding {
        @Override
        public String toString() {
            return "ProcSpecific(" + value + ")";
        }
    }

    static SymbolBinding of(int value) {
        if (13 <= value && value <= 15) {
            return new ProcSpecific(value);
        } else {
            for (SymbolBinding it : SymbolBinding.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for SymbolBinding");
        }
    }
}
```

elf/enums/SymbolIndex.java

```
package elf.enums;

public interface SymbolIndex {
    int value();

    boolean isSpecific();

    enum List implements SymbolIndex {
        UNDEF(0),
        BEFORE(65280),
        AFTER(65281),
```

```
    ABS(65521),
    COMMON(65522),
    ;
    final int value;

    List(int value) {
        this.value = value;
    }

    @Override
    public int value() {
        return value;
    }

    @Override
    public boolean isSpecific() {
        return true;
    }
}

record ProcSpecific(int value) implements SymbolIndex {
    @Override
    public String toString() {
        return "ProcSpecific(" + value + ")";
    }

    @Override
    public boolean isSpecific() {
        return true;
    }
}

record Reserved1(int value) implements SymbolIndex {
    @Override
    public String toString() {
        return "Reserved1(" + value + ")";
    }

    @Override
    public boolean isSpecific() {
        return true;
    }
}

record Reserved2(int value) implements SymbolIndex {
    @Override
    public String toString() {
        return "Reserved2(" + value + ")";
    }

    @Override
    public boolean isSpecific() {
        return true;
    }
}

record Usual(int value) implements SymbolIndex {
```

```
        @Override
        public String toString() {
            return String.valueOf(value);
        }

        @Override
        public boolean isSpecific() {
            return false;
        }
    }

    static SymbolIndex of(int value) {
        if (65282 <= value && value <= 65311) {
            return new ProcSpecific(value);
        } else if (65280 <= value && value <= 65520) {
            return new Reserved1(value);
        } else if (65523 <= value && value <= 65535) {
            return new Reserved2(value);
        } else {
            for (SymbolIndex it : SymbolIndex.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            return new Usual(value);
        }
    }
}
```

elf/enums/SymbolType.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface SymbolType {
    int value();

    enum List implements SymbolType {
        NOTYPE(0),
        OBJECT(1),
        FUNC(2),
        SECTION(3),
        FILE(4),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
```

```
        }
    }

    record ProcSpecific(int value) implements SymbolType {
        @Override
        public String toString() {
            return "ProcSpecific(" + value + ")";
        }
    }

    static SymbolType of(int value) {
        if (13 <= value && value <= 15) {
            return new ProcSpecific(value);
        } else {
            for (SymbolType it : SymbolType.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for SymbolType");
        }
    }
}
```

elf/enums/SymbolVisibility.java

```
package elf.enums;

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface SymbolVisibility {
    int value();

    enum List implements SymbolVisibility {
        DEFAULT(0),
        INTERNAL(1),
        HIDDEN(2),
        PROTECTED(3),
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }


    static SymbolVisibility of(int value) {
```

```
            for (SymbolVisibility it : SymbolVisibility.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for SymbolVisibility");

    }
}
```

elf/objects/Elf.java

```
package elf.objects;

import elf.parser.MetaInf;

public class Elf {
    public final MetaInf metaInf;
    public final StringTable strtab;
    public final StringTable names;
    public final SymbolTable symtab;
    public final Section text;

    public Elf(MetaInf metaInf, StringTable strtab, StringTable names,
SymbolTable symtab, Section text) {
        this.metaInf = metaInf;
        this.strtab = strtab;
        this.names = names;
        this.symtab = symtab;
        this.text = text;
    }
}
```

elf/objects/Section.java

```
package elf.objects;

import bytearrparser.BytearrParser;
import elf.enums.SHType;
import elf.enums.SectionFlags;
import util.Utilities;

public class Section {
    public final String name;
    public final SHType type;
    public final long virtualAddress;
    public final int link;
    public final int info;
    public final SectionFlags flags;
    public final long addressAlignment;
    public final long entrySize;
    protected final BytearrParser parser;
    public final long offset;
    public final long size;

    public Section(String name, SectionHeader header, BytearrParser
```

```
parser) {
        this.name = name;
        this.type = header.shType();
        this.virtualAddress = header.shAddr();
        this.link = header.shLink();
        this.info = header.shInfo();
        this.flags = header.shFlags();
        this.addressAlignment = header.shAddrAlign();
        this.entrySize = header.shEntSize();
        this.parser = parser;
        this.offset = header.shOffset();
        this.size = header.shSize();
    }

    public byte[] rawContent() {
        return parser.slice(Utilities.assertIsInt(offset),
Utilities.assertIsInt(size));
    }
}
```

elf/objects/SectionHeader.java

```
package elf.objects;


import bytearrparser.BytearrParser;
import elf.enums.SHType;
import elf.enums.SectionFlags;

public record SectionHeader(
        int shName,
        SHType shType,
        SectionFlags shFlags,
        long shAddr,
        long shOffset,
        long shSize,
        int shLink,
        int shInfo,
        long shAddrAlign,
        long shEntSize
) {
    public static SectionHeader read(BytearrParser parser){
        int shName = parser.nextInt();
        SHType shType = SHType.of(parser.nextInt());
        SectionFlags shFlags = new SectionFlags(parser.nextWord());
        long shAddr = parser.nextWord();
        long shOffset = parser.nextWord();
        long shSize = parser.nextWord();
        int shLink = parser.nextInt();
        int shInfo = parser.nextInt();
        long shAddrAlign = parser.nextWord();
        long shEntSize = parser.nextWord();
        return new SectionHeader(
                shName, shType, shFlags, shAddr, shOffset,
                shSize, shLink, shInfo, shAddrAlign, shEntSize
        );
    }
```

```
    @Override
    public String toString() {
        return "SectionHeader (" +
                "name = " + shName +
                ", type = " + shType +
                ", addr = " + shAddr +
                ", offset = " + shOffset +
                ", size = " + shSize +
                ", link = " + shLink +
                ", info = " + shInfo +
                ", addrAlign = " + shAddrAlign +
                ", entSize = " + shEntSize +
                ')';
    }
}
```

elf/objects/StringTable.java

```
package elf.objects;

import bytearrparser.BytearrParser;
import elf.enums.SHType;
import util.Utilities;

public class StringTable extends Section {
    public StringTable(String name, SectionHeader header, BytearrParser
parser) {
        super(name, header, parser);
        if (type != SHType.List.STRTAB) throw new
IllegalArgumentException(header + " doesn't specify StringTable");
    }

    public String readString(int offset) {
        return
parser.nullTerminatedStringFrom(Utilities.assertIsInt(this.offset +
offset));
    }

    @Override
    public String toString() {
        return "index " + offset + ": " + new String(rawContent());
    }
}
```

elf/objects/Symbol.java

```
package elf.objects;

import bytearrparser.BytearrParser;
import elf.enums.*;

public record Symbol(String name,
                     long value,
                     long size,
                     SymbolBinding bind,
                     SymbolType type,
                     int other,
```

```
                    SymbolVisibility visibility,
                    SymbolIndex shndx) {
    public static Symbol read(BytearrParser parser, Architecture arch,
StringTable names) {
        return switch (arch) {
            case x64 -> {
                int name = parser.nextInt();
                int info = parser.nextByte();
                int other = parser.nextByte();
                int shndx = parser.nextShort();
                long value = parser.nextLong();
                long size = parser.nextLong();
                yield new Symbol(
                        names.readString(name),
                        value, size,
                        SymbolBinding.of(info >> 4),
                        SymbolType.of(info & 0xf),
                        other,
                        SymbolVisibility.of(other & 0x3),
                       SymbolIndex.of(shndx)
                );
            }
            case x32 -> {
                int name = parser.nextInt();
                int value = parser.nextInt();
                int size = parser.nextInt();
                int info = parser.nextByte();
                int other = parser.nextByte();
                int shndx = parser.nextShort();
                yield new Symbol(
                        names.readString(name),
                        value, size,
                        SymbolBinding.of(info >> 4),
                        SymbolType.of(info & 0xf),
                        other,
                        SymbolVisibility.of(other & 0x3),
                        SymbolIndex.of(shndx)
                );
            }
            default -> throw new AssertionError();
        };
    }

    @Override
    public String toString() {
        return "0x%-15X %5d %-8s %-8s %-8s %6s %s".formatted(value, size,
type, bind, visibility, shndx, name);
        }
}
```

elf/objects/SymbolTable.java

```
package elf.objects;

import bytearrparser.BytearrParser;
import elf.enums.Architecture;
import elf.enums.SHType;
```

```
import util.Utilities;

import java.util.Arrays;

public class SymbolTable extends Section {
    private final Symbol[] symbols;

    public SymbolTable(String name, SectionHeader header, BytearrParser
parser, Architecture arch, StringTable symNames) {
        super(name, header, parser);
        if (type != SHType.List.SYMTAB)
            throw new IllegalArgumentException(header + " doesn't
represent SymbolTable");
        parser.moveTo(Utilities.assertIsInt(offset));
        symbols = new Symbol[Utilities.assertIsInt(size / entrySize)];
        for (int i = 0; i < symbols.length; i++) {
            symbols[i] = Symbol.read(parser, arch, symNames);
        }
    }

    public Symbol[] getSymbols() {
        return Arrays.copyOf(symbols, symbols.length);
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("Symbol Value
Size Type     Bind     Vis       Index Name\n");
        for (int i = 0; i < symbols.length; i++) {
            Symbol symbol = symbols[i];
            sb.append("[%4d] %s\n".formatted(i, symbol));
        }
        return sb.toString();
    }
}
```

elf/parser/ElfParser.java

```
package elf.parser;

import bytearrparser.BytearrParser;
import elf.enums.Architecture;
import elf.enums.EMachine;
import elf.enums.EType;
import elf.enums.OS_ABI;
import elf.objects.*;
import util.Utilities;

import java.util.HashMap;
import java.util.Map;

public class ElfParser {
    private static void checkMagic(byte[] data) {
        if (data[0] != 0x7f) throw new IncorrectFileSignature("Illegal
magic for Elf file (char 0)");
        if (data[1] != 0x45) throw new IncorrectFileSignature("Illegal
magic for Elf file (char 1)");
```

```
        if (data[2] != 0x4c) throw new IncorrectFileSignature("Illegal
magic for Elf file (char 2)");
        if (data[3] != 0x46) throw new IncorrectFileSignature("Illegal
magic for Elf file (char 3)");
    }

    public static Elf parse(byte[] data) {
        checkMagic(data);
        Architecture arch = Architecture.of(data[4]);
        boolean littleEndian = (data[5] == 1);
        BytearrParser parser = new BytearrParser(data, littleEndian,
arch);
        parser.moveTo(6);
        if (parser.nextByte() != 0x01) throw new IncorrectFileSignature();
        OS_ABI osabi = OS_ABI.of(parser.nextByte());
        int abiVersion = parser.nextByte();
        parser.moveTo(16);
        EType eType = EType.of(parser.nextShort());
        EMachine eMachine = EMachine.of(parser.nextShort());
        int eVersion = parser.nextInt();
        if (eVersion != 0x01) throw new IncorrectFileSignature();
        long eEntry = parser.nextWord();
        long ePhOff = parser.nextWord();
        long eShOff = parser.nextWord();
        int eFlags = parser.nextInt();
        int eEhSize = parser.nextShort();
        int ePhEntSize = parser.nextShort();
        int ePhNum = parser.nextShort();
        int eShEntSize = parser.nextShort();
        int eShNum = parser.nextShort();
        int eShStrNdx = parser.nextShort();
        MetaInf metaInf = new MetaInf(
                arch, littleEndian, osabi, abiVersion, eType, eMachine,
eVersion, eEntry,
                ePhOff, eShOff, eFlags, eEhSize, ePhEntSize, ePhNum,
eShEntSize, eShNum, eShStrNdx
        );
        SectionHeader[] headers = new SectionHeader[eShNum];
        for (int i = 0; i < eShNum; i++) {
            parser.moveTo(Utilities.assertIsInt(eShOff + (long) i *
eShEntSize));
            headers[i] = SectionHeader.read(parser);
        }
        StringTable names = new StringTable(".shstrtab",
headers[eShStrNdx], parser);
        Map<String, SectionHeader> named = new HashMap<>();
        for (SectionHeader header : headers) {
            named.put(names.readString(header.shName()), header);
        }
        // I won't implement all possible kinds of sections

        StringTable strtab = new StringTable(".strtab",
named.get(".strtab"), parser);
        SymbolTable symtab = new SymbolTable(".symtab",
named.get(".symtab"), parser, arch, strtab);
        Section text = new Section(".text", named.get(".text"), parser);
        return new Elf(metaInf, strtab, names, symtab, text);
```

```
    }
}
```

elf/parser/IncorrectFileSignature.java

```
package elf.parser;

public final class IncorrectFileSignature extends IllegalArgumentException
{
    public IncorrectFileSignature() {
    }

    public IncorrectFileSignature(String s) {
        super(s);
    }

    public IncorrectFileSignature(String message, Throwable cause) {
        super(message, cause);
    }

    public IncorrectFileSignature(Throwable cause) {
        super(cause);
    }
}
```

elf/parser/Metainf.java

```
package elf.parser;

import elf.enums.*;

public record MetaInf(
        Architecture arch,
        boolean littleEndian,
        OS_ABI osabi,
        int abiVersion,
        EType eType,
        EMachine eMachine,
        int eVersion,
        long eEntry,
        long ePhOff,
        long eShOff,
        int eFlags,
        int eEhSize,
        int ePhEntSize,
        int ePhNum,
        int eShEntSize,
        int eShNum,
        int eShStrNdx
) {

}
```

elf/parser/ProgramHeader.java

```
package elf.parser;

import elf.enums.PType;
```

```
public record ProgramHeader(
      PType pType,
      int pFlags,
      long pOffset,
      long pVaddr,
      long pPaddr,
      long pFilesz,
      long pMemsz,
      long pAlign
) {

}
```

main/Main.java

```
package main;

import disassembly.RiscvDisassembler;
import elf.objects.Elf;
import elf.parser.ElfParser;

import java.io.*;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class Main {
    private static String convert(byte[] data) {
        StringBuilder res = new StringBuilder(".text\n");
        Elf elf = ElfParser.parse(data);
        String text = RiscvDisassembler.disassemble(
                elf.text,
                elf.metaInf.littleEndian(),
                elf.metaInf.arch(),
                elf.symtab.getSymbols()
        );
        res.append(text);
        res.append("\n.symtab\n");
        res.append(elf.symtab);
        res.append("\n");
        return res.toString();
    }

    public static void main(String[] args) {
        if (args.length < 2) error("Usage: java -jar jarname.jar inputfile
outputfile");
        String inp = args[0], out = args[1];
        var inputFile = new File(inp);
        if (!inputFile.exists()) error("File " + inp + " doesn't exist");
        byte[] input = null;
        try (FileInputStream fis = new FileInputStream(inputFile)) {
            input = fis.readAllBytes();
        } catch (FileNotFoundException e) {
            error("File disappeared right out of our hands");
        } catch (IOException e) {
            error("Unknown IO Exception", e);
        }
        String result = convert(input);
        File outputFile = new File(out);
```

```
        if (outputFile.exists()) {
            Scanner sc = new Scanner(System.in);
            boolean clear = false;
            boolean overwrite = false;
            System.out.println("File " + out + " already exists.
Overwrite? y/n");
            while (!clear) {
                String line = sc.nextLine();
                if (!line.isEmpty()) {
                    char first = line.charAt(0);
                    if (first == 'Y' || first == 'y') {
                        clear = true;
                        overwrite = true;
                    } else if (first == 'N' || first == 'n') {
                        clear = true;
                        overwrite = false;
                    }
                }
                if (!clear) System.out.println("The who, the what, the
why, the when, the where? (c) Try again.");
            }
            if (overwrite) {
                boolean success = outputFile.delete();
                if (!success) error("Oh God, this file is undeletable!
Turn and burn!");
            }
        }
        if (outputFile.exists()) {
            error("Err... I already deleted it! Where did it come from
again?");
        }
        try {
            outputFile.createNewFile();
        } catch (IOException e) {
            if (!"The system cannot find the path
specified".equals(e.getMessage())) error("Unknown IO exception", e);
            int lastSeparator = outputFile.getPath().lastIndexOf('\\');
            String dirs = outputFile.getPath().substring(0,
lastSeparator);

            boolean successfully = new File(dirs).mkdirs();
            if (!successfully) error("Can't create the path specified");

            try {
                outputFile.createNewFile();
            } catch (IOException e1) {
                if (!"The system cannot find the path
specified".equals(e1.getMessage()))
                    error("Unknown IO exception", e);
                error("Can't create the path specified");
            }
        }
        try (FileOutputStream fos = new FileOutputStream(outputFile)) {
            fos.write(result.getBytes(StandardCharsets.UTF_8));
        } catch (FileNotFoundException e) {
            error("What's going all with your files? It's been created
about Plank-time ago, but now it's nowhere.");
```

```
        } catch (IOException e) {
            error("Unknown IO exception", e);
        }
    }

    private static void error(String str) {
        error(str, null);
    }

    private static void error(String str, Exception cause) {
        System.err.println(str);
        if (cause != null) cause.printStackTrace();
        System.exit(0);
    }
}
```

preproc/ExtenumPreprocessor.kt

```
package preproc

import java.io.File

fun main() {
   val w =

"[\u0009\u000a\u000b\u000c\u000d\u001c\u001d\u001e\u001f\u0020\u1680\u2000
\u2001\u2002\u2003\u2004\u2005\u2006\u2008\u2009\u200a\u2028\u2029\u205f\u
3000]"
   val nameRegex = "[A-Za-z][A-Za-z_0-9]*"
   val number = "-?[0-9]*|-?0x[A-Fa-f0-9]*|-?0b[01]*"
   val beginPattern = Regex("$w*($nameRegex)$w*values$w*\\{$w*")
   val valuePattern = Regex("$w*($nameRegex)$w*:$w*($number)$w*")
   val rangePattern =
Regex("$w*($nameRegex)$w*:$w*($number)$w*\\.\\.\\.?($number)$w*")
   val endPattern = Regex("$w*}$w*")

   val files = File("src")
      .allSubfiles()
      .filter { it.extension == "extenum" }

   for (file in files) {
      val location =
file.invariantSeparatorsPath.split("/").drop(1).dropLast(1)
      val content = file.readLines()
      var name = ""
      var values = mutableListOf<Pair<String, Int>>()
      var ranges = mutableListOf<Triple<String, Int, Int>>()
      var state = 0 //0 - '}' passed, 1 - name passed, content expected
      val structures =
         mutableListOf<Triple<String, List<Pair<String, Int>>,
List<Triple<String, Int, Int>>>>()
      for (line in content) {
         if (line.matches(Regex("$w*"))) continue
         if (state == 0) {
            name =
               beginPattern.matchEntire(line)?.groups?.get(1)?.value ?:
throw AssertionError()
```

```
            state = 1
         } else {
            if (endPattern.matches(line)) {
               state = 0
               structures.add(Triple(name, values, ranges))
               values = mutableListOf()
               ranges = mutableListOf()
            } else if (valuePattern.matches(line)) {
               val (full, a, b) =
valuePattern.matchEntire(line)!!.groups.map { it!!.value }
               values.add(a to b.toInt0())
            } else if (rangePattern.matches(line)) {
               val (full, a, b, c) =
rangePattern.matchEntire(line)!!.groups.map { it!!.value }
               ranges.add(Triple(a, b.toInt0(), c.toInt0()))
            }
         }
      }

      for (structure in structures) {
         val content = constuct(location, structure.first,
structure.second, structure.third)
         val f = File("src/" + location.joinToString("/") + "/" +
structure.first + ".java")
         if (f.exists()) f.delete()
         f.createNewFile()
         f.writeText(content)
      }
   }
}

fun String.toInt0() = when {
   startsWith("0x") -> substring(2).toUInt(16).toInt()
   startsWith("0b") -> substring(2).toUInt(2).toInt()
   else             -> toInt()
}


fun constuct(
   location: List<String>,
   name: String,
   values: List<Pair<String, Int>>,
   ranges: List<Triple<String, Int, Int>>
): String {
   return """
package ${location.joinToString(".")};

import elf.parser.IncorrectFileSignature;

/**
 * Code automatically generated using Extended Enum Preprocessor
 */
public interface $name {
    int value();

    enum List implements $name {
${values.joinToString("\n") { "        ${it.first}(${it.second})," }}
```

```
        ;
        final int value;

        List(int value) {
            this.value = value;
        }

        @Override
        public int value() {
            return value;
        }
    }
${
      ranges.joinToString("\n") {
         """
    record ${it.first}(int value) implements $name {
        @Override
        public String toString() {
            return "${it.first}(" + value + ")";
        }
    }"""
      }
   }

    static $name of(int value) {
        ${
      ranges.joinToString(" else ") {
         """if (${it.second} <= value && value <= ${it.third}) {
            return new ${it.first}(value);
        }"""
      }
   }${if(ranges.isEmpty()) "" else " else {"}
            for ($name it : $name.List.values()) {
                if (it.value() == value) {
                    return it;
                }
            }
            throw new IncorrectFileSignature("Incorrect value " + value +
" for $name");
        ${if(ranges.isEmpty()) "" else "}"}
    }
}
   """.trimIndent()
}

private fun File.allSubfiles() =
   setOf(this).whileChanges {
      it.flatMapTo(mutableSetOf()) {
         if (it.isDirectory) it.listFiles()!!.toList() else listOf(it)
      }
   }

private fun <T> T.whileChanges(func: (T) -> T): T {
   var prev = this
   var res = func(this)
   while (res != prev) {
      prev = res
```

```
      res = func(res)
   }
   return res
}
```

util/Utilities.java

```
package util;

import java.io.File;
import java.io.FileInputStream;

public class Utilities {
    public static int assertIsInt(long it) {
        if ((long) ((int) it) == it) {
            return (int) it;
        } else {
            throw new AssertionError("Value " + it + " of Long is out of
Int range");
        }
    }

}
```