Лабораторная работа №3	M3136	2022
ISA	Рязанова Екатерина Владимировна	

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

Инструментарий: Microsoft Visual C++ в составе Visual Studio Community 2022

Описание системы кодирования команд RISC-V

ISA - Instruction Set Architecture - Архитектура Набора Команд. Этот документ описывает архитектуру памяти, набор команд, доступных программисту, пишущему на машинном языке, количество регистров, доступные типы данных и тд. Есть несколько архитектур памяти, одна из которых – reg-reg (Регистрово-регистровая архитектура). Есть фиксированное число регистров R0, R1 ... RN, где цифры – не индексы, а часть названия, соответственно, итерировать по ним нельзя. Есть две вариации:

Reg-Reg 2 - операторы принимают два операнда	SUB R1 R2 → R1 -= R2
Reg-Reg 3 - операторы принимают три операнда	SUB R1 R2 R3 → R1 = R2 - R3 SUB R1 R1 R2 → R1 -= R2

Для кодирования команд есть два способа, один из которых – команды фиксированной длины. Команды фиксированной длины удобнее для декодирования – всегда знаем, что блок из 4 б = 1 команда), но туда не помещаются длинные хитрые команды.

RISC – один из подходов в формировании ISA и устройства процессора. С ростом количества транзисторов разных команд на процессоре становилось больше. Возникла идея убрать с кристалла сложные команды, которые нечасто используются, а оставшиеся транзисторы использовать для улучшения часто используемых команд, например, ускорить конвейер. При этом сложные команды будут выполняться дольше, но встречаются они редко. Но за счет уменьшения времени работы часто вызываемых команд в целом получилось ускорение. Эта идея получила название RISC - Сокращенный набор команд. Для RISC характерно описанное выше reg-reg, фиксированная длина команд (4б), а также большое количество регистров общего назначения, простое обращение к памяти - ([reg + const], const ~ 16bit). Инструкции занимают 1 такт, ISA содержит мало инструкций, а режимы адресации простые.

RISC-V – первая открытая и свободная система инструкций и процессорная архитектура с широкими перспективами коммерческого применения. У нее имеется 32-битный набор команд (инструкций), который называется RV32I. (для удобства тут не приведена вся таблица команд в зависимости от полей, она слишком длинная, поэтому блоки команд из нее сопоставлены соответствующим отрывкам кода).

31:25	24:20	19:15	14:12	11:7	6:0	Тип
funct7	rs2	rs1	funct3	rd	op	R
$imm_{11:0}$		rs1	funct3	rd	op	I
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B
$imm_{31:12}$				rd	op	U
$imm_{20,10:1,11,19:12}$				rd	op	J

Таблица 1

Это структура (формат) команд RISC-V, где указаны, с какого по какой бит идут отдельные части.

Команда однозначно определяется набором op, funct3, funct7 и типом. В стандартном наборе команд имеются такие, как: сложение, вычитание, логические операции, побитовые сдвиги и тд. Нас интересует операция умножения (и деления – они просто идут парой), которая определяется следующим образом:

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply умножение	`rd = (rs1 * rs2)31:0`
0110011 (51)	001	0000001	R	`mulh rd, rs1, rs2`	multiply high signed signed умножение старших разрядов со знаком на число со знаком	`rd = (rs1 * rs2)63:32`
0110011 (51)	010	0000001	R	`mulhsu rd, rs1, rs2`	multiply high signed unsigned умножение старших разрядов со знаком на число без знака	`rd = (rs1 * rs2)63:32`
0110011 (51)	011	0000001	R	`mulhu rd, rs1, rs2`	multiply high unsigned unsigned умножение старших разрядов без знака на число без знака	`rd = (rs1 * rs2)63:32`
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed) деление (со знаком)	`rd = rs1 / rs2`
0110011 (51)	101	0000001	R	`divu rd, rs1, rs2`	divide unsigned деление (без знака)	`rd = rs1 / rs2`
0110011 (51)	110	0000001	R	`rem rd, rs1, rs2`	remainder (signed) остаток от деления (со знаком)	`rd = rs1 % rs2`
0110011 (51)	111	0000001	R	`remu rd, rs1, rs2`	remainder unsigned остаток от деления (без знака)	`rd = rs1 % rs2`

Таблица 2

Описание структуры файла ELF

ELF — формат исполняемых двоичных файлов. Каждый файл ELF состоит из одного заголовка ELF, за которым следуют данные файла. Данные могут включать:

Таблицу заголовков программы, описывающую ноль или более сегментов памяти

Таблицу заголовков разделов, описывающую ноль или более разделов

Данные, на которые ссылаются записи в таблице заголовков программы или таблице заголовков разделов

Формат заголовка (для 32-битной версии):

Заголовок ELF определяет, использовать 32-битные или 64-битные адреса. Заголовок содержит три поля, на которые влияет этот параметр, и смещает другие поля, следующие за ними. Заголовок ELF имеет длину 52 или 64 байта для 32-битных и 64-битных двоичных файлов соответственно. У нас 32-битные адреса.

Первые 4 бита (EI_MAG0 - EI_MAG3) должны быть соответственно 0x7f 0x45 0x4c 0x46. Это сигнатура файла.

Затем 1 бит (e_ident[EI_CLASS])определяет, 32 и 64-битный файл. Следующий бит (EI_DATA) обозначает метод кодирования данных: 1 – Little endian (наш случай), 2 – Big endian. Еще один бит (e_ident[EI_VERSION]) отвечает за версию elf заголовка - 1. Следующий бит (EI_OSABI) отвечает за особенности ОС и ABI. 8-ой бит (EI_ABIVERSION) – версия ABI. С 9 по 16-й биты зарезервированы. Затем поле e_type занимает 2 бита и определяет тип файла.

Нужное нам поле e_machine (2 бита) определяет целевую ISA. У нас это должно быть 0xF3 для RICS-V. Далее поле e_version определяет номер версии формата (4 бита). Затем еще идет 4-битное поле e_entry – адрес памяти точки входа.

Следующее 4-битное поле e_phoff указывает на начало таблицы заголовков программы. 4-битное e_phoff указывает на начало таблицы заголовков программы. За ним 4-битное e_shoff указывает на начало таблицы заголовков секций.

4-битное поле e_flags интерпретируется в зависимости от целевой архитектуры. E_ehsize (2 бита) содержит размер заголовка – 52 бита для 32-битного формата. e_phentsize (2 бита) содержит размер записи таблицы заголовков программы. Дальше e_phnum (2 бита) содержит количество

записей в таблице заголовков программы. Аналогично для заголовков секций: по 2 бита поля e_shentsize и e_shnum содержат соответственно размер записи таблицы заголовков секций и количество записей в таблице заголовков секций. И, наконец, 2-битное поле e_shstrndx содержит индекс записи таблицы заголовков секций, которая содержит имена секций.

В структуре ELF-файла присутствует таблица заголовков программ, но в данной лабораторной мы ее нигде не используем.

Таблица заголовков секций

Информация, которая хранится в ELF-файле, организована по секциям, каждая из которых имеет свое имя. У каждой секции своя функция, например, служебная информация, отладочная информация ,код или данные программы. Количество элементов массива задается полем e_shnum заголовка файла, смещение хранится в поле e_shoff.

Первое 4-битное поле sh_name хранит индекс имени секции (смещение в данных секции, ее индекс задается в поле e_shstrndx заголовка. По этому смещению находится строка, которая заканчивается нулевым байтом – имя секции). Затем поле sh_type (4 бита) определяет тип секции. Далее 4-битное поле sh_flags определяет атрибуты секции.

Все остальные поля занимают 4 бита. sh_addr - Виртуальный адрес раздела в памяти, для секций. sh_offset - смещение от начала файла, по которому размещаются данные секции. Sh_size обозначает размер секции в файле. Sh_link содержит индекс связанной секции, sh_info содежит дополнительную информацию. Sh_addralign хранит требование по выравниванию адреса начала секции в памяти. И, наконец, sh_entsize содержит размер каждой записи для секции.

Таблица символов

Таблица символов объектного файла содержит информацию, необходимую для поиска и перемещения символьных определений и ссылок программы. Индекс таблицы символов является нижним индексом в этом массиве.

st_name	Это поле содержит индекс в symtab, которая содержит символьные представления имен символов.

st_value	Это поле определяет значение связанного символа
st_size	Это поле хранит размер символа.
st_info	Это поле определяет тип символа и атрибуты привязки.
st_other	Это поле в настоящее время содержит 0 и не имеет определенного значения.
st_shndx	Каждая запись таблицы символов "определена" по отношению к некоторой секции; этот элемент содержит соответствующий индекс таблицы заголовка секций.

Таблица символов нужна для связывания. Программы часто состоят не из одного файла. Нередко используются чужие библиотеки, которые даже бывают скомпилированы (и исходников даже нет). Они поставляются просто в виде файла (в т.ч. может быть и ELF). И дается файл заголовка (например .h для C), там только названия процедур и параметры) и вызов процедуры идет как раз по символьному имени. То есть в таблице символов находится по названию символ и из него берется адрес процедуры в файле.

Описание кода

В самом начале перечислены названия регистров (всего их у процессора 32), они наименованы в соответствии с ABI.

```
const char regnames[32][5] = {
    "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"};
```

Затем идет формат хранения команд, их всего 6(подробно чуть далее):

```
enum  cmd_fmt {R, IL, I3, I0, S, B, U, J};
```

Далее перечислены константы-названия столбцов в symtab (bind, size, index) – расшифровка кодов. В symtab записи тоже состоят из нескольких полей.

```
const char* getbindstr(unsigned char inf) {
    switch (inf >> 4) {
    case 0: return "LOCAL";
    case 1: return "GLOBAL";
    case 2: return "WEAK";
    case 13: return "LOPROC";
    case 15: return "HIPROC";
```

```
    default: return "UNDEF";
    }
}

const char* gettypestr(unsigned char inf) {
    switch (inf & 0b1111) {
    case 0: return "NOTYPE";
    case 1: return "OBJECT";
    case 2: return "FUNC";
    case 3: return "SECTION";
    case 4: return "FILE";
    case 13: return "LOPROC";
    case 15: return "HIPROC";
    default: return "UNDEF";
    }
}

void getindexstr(unsigned short idx, char* rstr) {
//    char* str[];
    switch (idx) {
    case 0: {strcpy_s(rstr,10,"UNDEF"); break; }
    case 0xFF00: {strcpy_s(rstr, 10, "LOPROC"); break;}
    case 0xFF1F: {strcpy_s(rstr, 10, "HIPROC"); break;}
    case 0xFFF1: {strcpy_s(rstr, 10, "ABS"); break;}
    case 0xFFF2: {strcpy_s(rstr, 10, "COMMON"); break; }
    case 0xFFFF: {strcpy_s(rstr, 10, "HIRESERVE"); break; }
    default: {
        sprintf_s(rstr, 10, "%d", idx);
    }
    }
}
```

Затем начинаем доставать поля из 32-битной последовательности. (cmd -32битная команда). Младшие 7 бит – opcode команды. Аналогично берем часть rd – сдвигаем (=пропускаем) первые 7 бит, которые были рассмотрены в op, и берем следующие 5 и так далее.

```
unsigned int op_opcode(unsigned int cmd) {
    return (cmd & 0b1111111);
}

unsigned int op_rd(unsigned int cmd) {
    return ((cmd >> 7) & 0b11111);
}

unsigned int op_rs1(unsigned int cmd) {
    return ((cmd >> 15) & 0b11111);                //ссылки на регистры
}
```

```
unsigned int op_rs2(unsigned int cmd) {
    return ((cmd >> 20) & 0b11111);                 //ссылки на регистры
}

unsigned int op_funct3(unsigned int cmd) {
    return ((cmd >> 12) & 0b111);
}

unsigned int op_funct7(unsigned int cmd) {
    return (cmd >> 25);
}
```

Rs1, rs2, funct3, funct7 – стандартные расположения частей (берутся всегда по-одинаковому для R-типа). Но еще в разных форматах команд (когда, например, дается не ссылка на регистр, а либо адрес, либо значение – immediate), они в зависимости от форматов берутся по-разному. И далее берем часть immediate, которая зависит от типа команды – I, S, B, U, J. (Например, у типа I immediate занимает 12 бит согласно таблице 1).

```
int op_Iimm(unsigned int cmd) {
    return (((long long)cmd<<32) >> 52);
}

int op_Simm(unsigned int cmd) {
    return (((cmd >> 7) & 0b11111)|((((long long)cmd<<32) >> 57)<<5));
}

int op_Bimm(unsigned int cmd) {
    return (((cmd >> 7) & 0b11110) | ((cmd >> 20) & 0b11111100000) | ((cmd <<
4)&(1<<11)) | ((((long long)cmd<<32)>>63)<<12));
}

int op_Uimm(unsigned int cmd) {
    return ((((long long)cmd<<32)>>44)<<12);
}

int op_Jimm(unsigned int cmd) {
    return ((cmd & 0xFF000) | ((cmd >> 19) & 0x3FE) | ((cmd >>9) & (1 << 11))
| ((((long long)cmd<<32) >> 63)<<20));
}
```

Далее начинаем разбирать команды. Дается 32-битная cmd, вызываем функцию opcode, которая берет 7 младших бит, и в зависимости от него происходит первый switch, то есть начинаем разбирать команды. Этот Opcode не дает однозначную команду, он делит их на группы. И дальше уже

внутри группы идет дальнейший разбор, что это за конкретно команда (case).

op	funct3	funct7	Тип	Инструкция	Описание	Операция
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte загрузка байта	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half загрузка половины слова	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word загрузка слова	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned загрузка байта без знака	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned загрузка половины слова без знака	rd = ZeroExt([Address]$_{15:0}$)

Например, команд с opcode 3 несколько штук, и чтобы различить их между собой, нужно смотреть на funct3. Это происходит в следующем отрывке кода:

```
switch (op_opcode(cmd)) {
    case 0x03: {
        switch (op_funct3(cmd)) {
        case 0b000: return "lb";
        case 0b001: return "lh";
        case 0b010: return "lw";
        case 0b100: return "lbu";
        case 0b101: return "lhu";
        }
        return "unknown_instruction";
    }
```

Аналогично рассматриваются другие части:

`case 0x0f: return "fence";` - в тз написано, что можно не обрабатывать

```
case 0x13: {
        switch (op_funct3(cmd)) {
        case 0b000: return "addi";
        case 0b001: return "slli";
        case 0b010: return "slti";
        case 0b011: return "sltiu";
        case 0b100: return "xori";
        case 0b101: return op_funct7(cmd) ? "srai" : "srli";
        case 0b110: return "ori";
        case 0b111: return "andi";
        }
        return "unknown_instruction";
    }
```

Соответствует командам

0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate сложение с константой	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000'	I	slli rd, rs1, uimm	shift left logical immediate логический сдвиг константы влево	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate установить регистр назначения в 1, если регистр-источник меньше, чем константа	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned установить регистр назначения в 1, если регистр-источник меньше, чем константа. Операция беззнаковая	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate ИСКЛЮЧАЮЩЕЕ ИЛИ с константой	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000'	I	srli rd, rs1, uimm	shift right logical immediate логический сдвиг константы вправо	rd = rs1 >> uimm
0010011 (19)	101	0100000'	I	srai rd, rs1, uimm	shift right arithmetic imm. арифметический сдвиг константы вправо	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate логическая операция ИЛИ с константой	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate логическая операция И с константой	rd = rs1 & SignExt(imm)

(здесь и далее в формате case 0xNN – то, что написано в коде – это число в 16ричной СС, а в поясняющих таблицах (число в скобочках) – в десятичной).

```
case 0x17: return "auipc";
```

соответствует

0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC прибавить старшую половину константы к счетчику команд	rd = {upimm, 12'b0} + PC

```
case 0x23: {
        switch (op_funct3(cmd)) {
        case 0b000: return "sb";
        case 0b001: return "sh";
        case 0b010: return "sw";
        }
        return "unknown_instruction";
    }
```

Соответствует

0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte сохранить байт в памяти	$[Address]_{7:0} = rs2_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half сохранить половину слова в памяти	$[Address]_{15:0} = rs2_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word сохранить слово в памяти	$[Address]_{31:0} = rs2$

```
case 0x37: return "lui";
```

соответствует

0110111 (55)	–	–	U	lui	rd, upimm	load upper immediate загрузить старшую половину константы в регистр	rd = {upimm, 12'b0}

```
case 0x63: {
        switch (op_funct3(cmd)) {
        case 0b000: return "beq";
        case 0b001: return "bne";
        case 0b100: return "blt";
        case 0b101: return "bge";
        case 0b110: return "bltu";
        case 0b111: return "bgeu";
        }
        return "unknown_instruction";
    }
```

Соответствует

1100011 (99)	000	–	B	beq	rs1, rs2, label	branch if = переход, если равно	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne	rs1, rs2, label	branch if ≠ переход, если не равно	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt	rs1, rs2, label	branch if < переход, если меньше	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge	rs1, rs2, label	branch if ≥ переход, если больше или равно	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu	rs1, rs2, label	branch if < unsigned переход, если меньше, без учета знака	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu	rs1, rs2, label	branch if ≥ unsigned переход, если больше, без учета знака	if (rs1 ≥ rs2) PC = BTA

```
case 0x67: return "jalr";
case 0x6F: return "jal";
```

соответствует

1100111 (103)	000	–	I	jalr	rd, rs1, imm	jump and link register переход с возвратом по адресу в регистре	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal	rd, label	jump and link переход с возвратом	PC = JTA, rd = PC + 4

Отдельное внимание стоит уделить блоку `case 0x33` – команды R – типа. У них opcode `0x33`, но такой же и у умножения, поэтому нужно разграничивать случаи. Когда funct7 = 1, то это умножение (и деление), соответствует (это из таблицы 2)

```
if (op_funct7(cmd) == 1)

        switch (op_funct3(cmd)) {
        case 0b000: return "mul";
```

```
        case 0b001: return "mulh";
        case 0b010: return "mulhsu";
        case 0b011: return "mulhu";
        case 0b100: return "div";
        case 0b101: return "divu";
        case 0b110: return "rem";
        case 0b111: return "remu";
        }
```

Иначе, смотрим на funct3 , это уже команды из RV32I. У них всех funct7=0.

```
else switch (op_funct3(cmd)) {
        case 0b000: return op_funct7(cmd) ? "sub" : "add";
        case 0b001: return "sll";
        case 0b010: return "slt";
        case 0b011: return "sltu";
        case 0b100: return "xor";
        case 0b101: return op_funct7(cmd) ? "sra" : "srl";
        case 0b110: return "or";
        case 0b111: return "and";
        }
```

соответствует

0110011 (51)	000	0000000	R	add	rd, rs1, rs2	add сложение	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub	rd, rs1, rs2	sub вычитание	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll	rd, rs1, rs2	shift left logical логический сдвиг влево	rd = rs1 << $rs2_{4:0}$
0110011 (51)	010	0000000	R	slt	rd, rs1, rs2	set less than установить rd в 1, если rs1 меньше, чем rs2	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu	rd, rs1, rs2	set less than unsigned установить rd в 1, если rs1 меньше, чем rs2, беззнаковая операция	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor	rd, rs1, rs2	xor логическая операция ИСКЛЮЧАЮЩЕЕ ИЛИ	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl	rd, rs1, rs2	shift right logical логический сдвиг вправо	rd = rs1 >> $rs2_{4:0}$
0110011 (51)	101	0100000	R	sra	rd, rs1, rs2	shift right arithmetic арифметический сдвиг вправо	rd = rs1 >>> $rs2_{4:0}$
0110011 (51)	110	0000000	R	or	rd, rs1, rs2	or логическая операция ИЛИ	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and	rd, rs1, rs2	and логическая операция И	rd = rs1 & rs2

Далее снова пробегаемся по кодам, это возвращает формат команды (ранее был объявлен enum – типы команд, стандартных их 6 штук. Но есть 3 разных команды I – IL, I0, I3. IL – команда загрузки, I0 – команда без операндов, I3 – команда с 3 операндами).

```
cmd_fmt cmdtype(unsigned int cmd) {
    switch (op_opcode(cmd)) {
    case 0x03: return IL;
    case 0x0f: return I0;
    case 0x13: return I3;
    case 0x17: return U;
```

```
    case 0x23: return S;
    case 0x33: return R;
    case 0x37: return U;
    case 0x63: return B;
    case 0x67: return I3;
    case 0x6F: return J;
    case 0x73: return I0;
    }
    return I0;
}
```

```
Далее начинается блок объявления структур ELF-файла.
typedef struct elf32_header{
    unsigned char  e_magic[4];
    unsigned char  e_class;
    unsigned char  e_data;
    unsigned char  e_version;
    unsigned char  e_osabi;
    unsigned char  e_abiversion;
    unsigned char  e_pad[7];
    unsigned short e_type;
    unsigned short e_machine;
    unsigned int   e_version2;
    unsigned int   e_entry;
    unsigned int   e_phoff;
    unsigned int   e_shoff;
    unsigned int   e_flags;
    unsigned short e_ehsize;
    unsigned short e_phentsize;
    unsigned short e_phnum;
    unsigned short e_shentsize;
    unsigned short e_shnum;
    unsigned short e_shstrndx;
} Elf32_Header;
```

----Структура заголовка файла ELF

```
typedef struct elf32_pheader {
    unsigned int    p_type;
    unsigned int    p_offset;
    unsigned int    p_vaddr;
    unsigned int    p_paddr;
    unsigned int    p_filesz;
    unsigned int    p_memsz;
    unsigned int    p_flags;
    unsigned int    p_align;
} Elf32_ProgHeader;
```

---Заголовки программ

```
typedef struct elf32_sheader {
```

```
    unsigned int    sh_name;
    unsigned int    sh_type;
    unsigned int    sh_flags;
    unsigned int    sh_addr;
    unsigned int    sh_offset;
    unsigned int    sh_size;
    unsigned int    sh_link;
    unsigned int    sh_info;
    unsigned int    sh_addralign;
    unsigned int    sh_entsize;
} Elf32_SecHeader;
```

-----Заголовки секций

Мы работаем с секциями, поэтому нас интересует именно sheader.

Строка unsigned int e_shoff; указывает адрес в файле, с которого начнется таблица заголовков.
В строке unsigned short e_shentsize; описываются сами заголовки, то есть размер заголовка и количество строк в этой таблице – строка unsigned short e_shnum;. В самом заголовке все данные уже есть, нам остается лишь пойти в файле по этому адресу e_shoff и считать e_shnum строчек. А сами элементы – header, то есть это элемент таблицы заголовков (название секции, тип секции, адреса, длина, флаги и тд).

```
typedef struct elf32_symbol {
    unsigned int    st_name;
    unsigned int    st_value;
    unsigned int    st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    unsigned short  st_shndx;
} Elf32_Symbol;
```

---Как раз описание одного символа.

Дальше в коде идет объявление переменных:
int i,flab,msymtab,symcount,mtext;
unsigned int tc, txtoff, txtlen, tcmd, pc;
где I – счетчик, flab - при выводе в эту переменную задаем номер метки, если мы нашли ее. В файле вывода это строки 00010074 <main> ; 000100ac <mmul> (стандартное из symtab), а также <L0>, <L1>, <L2> - созданные нами, их в symtab нет. Msymtab – номер секции файла, в которой лежит таблица symtab. Mtext – номер записи в таблице секций, в котором лежит секция text. Symcount – количество символов в таблице symtab.
Дальше беззнаковые числа: tc – текущий адрес, начиная с 0 (в массиве, куда считывали секцию text), txtoff – начальный адрес программы, txtlen – длина сегмента текст(кода программы), сразу поделенная на 4 – сколько всего

команд в коде, pc – программный счетчик (текущий адрес с учетом смещения 10074 – адрес начала программы).

Затем идет стандартная процедура сортировки, она нужна, чтобы отсортировать адреса меток:

```
void qs(unsigned int* arr, int first, int last)
{
    if (first < last)
    {
        int left = first, right = last, middle = arr[(left + right) / 2];
        do
        {
            while (arr[left] < middle) left++;
            while (arr[right] > middle) right--;
            if (left <= right)
            {
                unsigned int tmp = arr[left];
                arr[left] = arr[right];
                arr[right] = tmp;
                left++;
                right--;
            }
        } while (left <= right);
        qs(arr, first, right);
        qs(arr, left, last);
    }
}
```

До этого были служебные процедуры, функции, объявления, а теперь начинается main – основная процедура, которая выполняется при запуске.

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

сначала тоже небольшое объявление переменных:

```
FILE* ef,*af; - входной и выходной файлы
    int i; - счетчик
    size_t readed;
    Elf32_Header ElfHeader;
```

Дальше передаем названия файлов и пытаемся их открыть: их должно быть 2.

```
if (argc < 2) {
        printf("Too few paramaters.\n");
        exit(-1);
    }
    if ((fopen_s(&ef, argv[1], "rb")) != 0) {
        printf("Unable to open input file\n");
        exit(-1);
    }
```

```
    if ((fopen_s(&af, argv[2], "wt")) != 0) {
        printf("Unable to open output file\n");
        exit(-1);
    }

readed = fread(&ElfHeader, 1, sizeof(Elf32_Header), ef);
    if (readed < sizeof(Elf32_Header)) {
        printf("Unexpected length of file\n");
        exit(-1);
    }
```
---Читаем заголовок

```
if ((ElfHeader.e_magic[0] != 0x7F) || (ElfHeader.e_magic[1] != 0x45) ||
(ElfHeader.e_magic[2] != 0x4C) || (ElfHeader.e_magic[3] != 0x46) ||
(ElfHeader.e_class != 1)) {
        printf("Unsupported file type\n");
        exit(-1);
    }
```
---Проверка, что присутствует сигнатура ELF, то есть первые четыре символа те, что надо.

```
if (ElfHeader.e_machine != 0xF3) {
        printf("Unsupported processor type\n");
        exit(-1);
    }
```
---Проверка, что программа для архитектуры RICS-V.

Далее начинаем чтение.

```
Elf32_ProgHeader* ProgHeads = new Elf32_ProgHeader[ElfHeader.e_phnum];
    fseek(ef, ElfHeader.e_phoff, SEEK_SET);
    fread(ProgHeads, ElfHeader.e_phentsize, ElfHeader.e_phnum, ef);
```
Это прочитали таблицу заголовков программ (но мы ее не используем).

Аналогично с заголовками секций, которые нам уже нужны:
```
Elf32_SecHeader* SecHeads = new Elf32_SecHeader[ElfHeader.e_shnum];
    fseek(ef, ElfHeader.e_shoff, SEEK_SET);
    fread(SecHeads, ElfHeader.e_shentsize, ElfHeader.e_shnum, ef);
```

Затем читаем таблицу строк с названиями секций (запросили массив, размер прочитали из заголовка и читаем):
```
char* SecNames = new char[SecHeads[ElfHeader.e_shstrndx].sh_size];
    fseek(ef, SecHeads[ElfHeader.e_shstrndx].sh_offset, SEEK_SET);
    fread(SecNames, SecHeads[ElfHeader.e_shstrndx].sh_size, 1, ef);
```

Дальше пробегаем по всем секциям, нашли секцию symtab по признаку, что ее тип 2, запомнили номер и нашли секцию text по названию:
```
for (i = 0; i < ElfHeader.e_shnum; i++) {
        if (SecHeads[i].sh_type == 2) msymtab = i;
```

```
        if (strcmp(&SecNames[SecHeads[i].sh_name], ".text") == 0) mtext = i;
    }
```

Потом создаем массив строк с названиями символов (это работаем с symtab – там хранятся только данные, а названия не хранятся, но в symtab есть поле shlink, это как раз номер секции, где хранятся строки с названиями символов, и мы по этому номеру читаем секцию и называем ее symnames, то есть там строки с названиями секция хранятся в этом массиве):

```
char* SymNames = new char[SecHeads[SecHeads[msymtab].sh_link].sh_size];
    fseek(ef, SecHeads[SecHeads[msymtab].sh_link].sh_offset, SEEK_SET);
    fread(SymNames, SecHeads[SecHeads[msymtab].sh_link].sh_size, 1, ef);
```

После этого читаем сам symtab:

```
symcount = SecHeads[msymtab].sh_size / SecHeads[msymtab].sh_entsize;
    Elf32_Symbol* Symbols = new Elf32_Symbol[symcount];
    fseek(ef, SecHeads[msymtab].sh_offset, SEEK_SET);
    fread(Symbols, 1, SecHeads[msymtab].sh_size, ef);
```

Затем читаем секцию text:

```
txtlen = SecHeads[mtext].sh_size >> 2;
    txtoff = SecHeads[mtext].sh_addr;
    unsigned int* txt = new unsigned int[txtlen]; -создаем массив, где
хранятся именно команды txt
    unsigned int* mlabels = new unsigned int[txtlen]; - такого же размера
создается массив, где будут храниться адреса меток L0, L1, L2
    fseek(ef, SecHeads[mtext].sh_offset, SEEK_SET);
    fread(txt, 4, txtlen, ef); - чтение самой секции
```

`txtoff = SecHeads[mtext].sh_addr;` ------Адрес в памяти, по которому должна располагаться эта секция. Нулевой байт в массиве, куда мы считали эту секцию, должен иметь такой адрес, поэтому и происходит такое смещение. То есть в заголовке указано, по какому адресу в памяти нужно загрузить код и откуда его запустить. Другими словами, это «адрес» нулевого массива txt.

```
int mlabcnt = 0; - счетчик, сколько мы нашли меток L0, L1, L2
    int fmlab = -1; метка L0, L1 или L2 (ее номер)
    int tc = 0; - объявлено ранее
```

Дальше цикл – проходимся по командам.
Первый проход – ищем метки. Метки возникают в командах перехода – branch и jump (B, J). Сами тела одинаковые, просто ссылки на данные у B это Bimm, у J – Jimm, вызываются разные функции.
Цикл проходит по всему коду, по всей секции text и смотрит каждую команду: если это командаперехода, он начинает считать ,по какому адресу

делается переход, и смотрим, есть такая метка или нет. Первый цикл ищет в символах, есть этот адрес там или нет. Но ищет он не любые, а только те, которые относятся к text. Если нашли, то ничего не делаем. А вот если нет такого, то тогда добавляем адрес метки и увеличиваем счетчик.

```
while (tc < txtlen) {
        pc = txtoff + tc * 4;
        switch (cmdtype(txt[tc])) {
        case B: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Bimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            if (flab >= 0) break;
            mlabels[mlabcnt] = op_Bimm(txt[tc]) + pc;
            mlabcnt++;
            break;
        }
        case J: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Jimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            if (flab >= 0) break;
            mlabels[mlabcnt] = op_Jimm(txt[tc]) + pc;
            mlabcnt++;
            break;
        }
        };
        tc++;
    }
```

Если снашли таких меток больше, чем одну, то вызываем сортировку этих меток: `if (mlabcnt>1) qs(mlabels, 0, mlabcnt - 1);`

После того, как нашли все метки, начинаем само дизассемблирование.

Снова проходим весь массив команд. Pc – абсолютный адрес, а txtoff – адрес начала программы (в нашем случае это 10074), потом tc умножаем на 4, тк каждая команда по 4 байта:

```
pc = txtoff + tc * 4;
```

Дальше снова в цикле ищем метку. Если нашли в символах такую метку, то выводим ее название из символов:

```
for (i = 0; i < symcount; i++) {
            if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx ==
mtext) && (pc == Symbols[i].st_value)) {
                fprintf(af, "%08x   <%s>:\n", pc,
&SymNames[Symbols[i].st_name]);
            }
        }
```

Дальше пробегаемся по массиву наших меток, которые мы нашли. Если в этом массиве есть такой адрес, то выводим эту метку:

```
for (i = 0; i < mlabcnt; i++) {
            if (pc == mlabels[i]) {
                fprintf(af, "%08x   <L%d>:\n", pc, i);
            }
        }
```

Потом вызывается функция, определяющая тип команды, и в зависимости от этого (case) выводится то, что необходимо:

```
switch (cmdtype(txt[tc])) {
        case R: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %s\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])]); break;
        case IL: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %d(%s)\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Iimm(txt[tc]),
regnames[op_rs1(txt[tc])]); break;
        case I3: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %d\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], regnames[op_rs1(txt[tc])],
op_Iimm(txt[tc])); break;
        case I0: fprintf(af, "   %05x:\t%08x\t%7s\n", pc, txt[tc],
cmdname(txt[tc])); break;
        case S: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %d(%s)\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rs2(txt[tc])], op_Simm(txt[tc]),
regnames[op_rs1(txt[tc])]); break;
        case U: fprintf(af, "   %05x:\t%08x\t%7s\t%s, 0x%x\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Uimm(txt[tc])>>12); break;
```

(Все форматы взяты из тз)

Оличаются case J и B, так как это команды перехода, у них в конце нужно дописывать метки (если найдется), а у остальных – просто адреса, операнды.

```
case J: {
            flab = -1;
```

```
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Jimm(txt[tc])+pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            fmlab = -1;
            for (i = 0; i < mlabcnt; i++) {
                if ((op_Jimm(txt[tc]) + pc) == mlabels[i]) {
                    fmlab = i;
                    break;
                }
            }
            if (flab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x <%s>\n",
pc, txt[tc], cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Jimm(txt[tc]) +
pc, &SymNames[Symbols[flab].st_name]);
            else if (fmlab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x
<L%d>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rd(txt[tc])],
op_Jimm(txt[tc]) + pc, fmlab);
            else fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Jimm(txt[tc])+pc);
            break;
        }
        case B: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Bimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            fmlab = -1;
            for (i = 0; i < mlabcnt; i++) {
                if ((op_Bimm(txt[tc]) + pc) == mlabels[i]) {
                    fmlab = i;
                    break;
                }
            }
            if (flab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %08x
<%s>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc,
&SymNames[Symbols[flab].st_name]);
            else if (fmlab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s,
%08x <L%d>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc, fmlab);
```

```
            else fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %08x\n", pc,
txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc);
            break;
        }
```

Дальше просто выводим все в файл. Команды вывелись в зависимости от формата

```
default:fprintf(af, "\t%08x: %s\n", pc, cmdname(txt[tc]));
```

и после этого еще вывелась symtab

```
fprintf(af, "\n.symtab\nSymbol Value           Size
Type   Bind    Vis     Index Name\n");
    char indexstr[10];
    for (i = 0; i < symcount; i++) {
        getindexstr(Symbols[i].st_shndx,indexstr);
        fprintf(af, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n", i,
Symbols[i].st_value, Symbols[i].st_size, gettypestr(Symbols[i].st_info),
getbindstr(Symbols[i].st_info),"DEFAULT", indexstr,
&SymNames[Symbols[i].st_name]);
```

осталось закрыть все файлы

```
fclose(ef);
fclose(af);
```

Результат работы написанной программы на приложенном к заданию файле (дизассемблер и таблицу символов)

```
.text
00010074   <main>:
   10074:    ff010113        addi     sp, sp, -16
   10078:    00112623          sw     ra, 12(sp)
   1007c:    030000ef         jal     ra, 000100ac <mmul>
   10080:    00c12083          lw     ra, 12(sp)
   10084:    00000513        addi     a0, zero, 0
   10088:    01010113        addi     sp, sp, 16
   1008c:    00008067        jalr     zero, ra, 0
   10090:    00000013        addi     zero, zero, 0
   10094:    00100137         lui     sp, 0x100
   10098:    fddff0ef         jal     ra, 00010074 <main>
   1009c:    00050593        addi     a1, a0, 0
   100a0:    00a00893        addi     a7, zero, 10
   100a4:    0ff0000f       fence
   100a8:    00000073       ecall
000100ac   <mmul>:
   100ac:    00011f37         lui     t5, 0x11
```

```
   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   <L0>:
   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   <L2>:
   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, 000100e4 <L2>
   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, 000100d8 <L1>
   10114:    050f0f13        addi     t5, t5, 80
   10118:    01478513        addi     a0, a5, 20
   1011c:    fa5f16e3         bne     t5, t0, 000100c8 <L0>
   10120:    00008067        jalr     zero, ra, 0

.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
```

Список литературы

Сара Л. Харрис, Дэвид Харрис «ЦИФРОВАЯ СХЕМОТЕХНИКА И АРХИТЕКТУРА КОМПЬЮТЕРА: RISC-V»

Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

https://ejudge.ru/study/3sem/elf.html

https://ru.manpages.org/elf/5

Листинг кода

```cpp
#include <iostream>

const char regnames[32][5] = {
    "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"};

enum  cmd_fmt {R, IL, I3, I0, S, B, U, J};

const char* getbindstr(unsigned char inf) {
    switch (inf >> 4) {
    case 0: return "LOCAL";
    case 1: return "GLOBAL";
    case 2: return "WEAK";
    case 13: return "LOPROC";
    case 15: return "HIPROC";
    default: return "UNDEF";
    }
}

const char* gettypestr(unsigned char inf) {
    switch (inf & 0b1111) {
    case 0: return "NOTYPE";
    case 1: return "OBJECT";
    case 2: return "FUNC";
    case 3: return "SECTION";
    case 4: return "FILE";
    case 13: return "LOPROC";
    case 15: return "HIPROC";
    default: return "UNDEF";
    }
```

```
}

void getindexstr(unsigned short idx, char* rstr) {
    switch (idx) {
    case 0: {strcpy_s(rstr,10,"UNDEF"); break; }
    case 0xFF00: {strcpy_s(rstr, 10, "LOPROC"); break;}
    case 0xFF1F: {strcpy_s(rstr, 10, "HIPROC"); break;}
    case 0xFFF1: {strcpy_s(rstr, 10, "ABS"); break;}
    case 0xFFF2: {strcpy_s(rstr, 10, "COMMON"); break; }
    case 0xFFFF: {strcpy_s(rstr, 10, "HIRESERVE"); break; }
    default: {
        sprintf_s(rstr, 10, "%d", idx);
    }
    }
}

unsigned int op_opcode(unsigned int cmd) {
    return (cmd & 0b1111111);
}

unsigned int op_rd(unsigned int cmd) {
    return ((cmd >> 7) & 0b11111);
}

unsigned int op_rs1(unsigned int cmd) {
    return ((cmd >> 15) & 0b11111);
}

unsigned int op_rs2(unsigned int cmd) {
    return ((cmd >> 20) & 0b11111);
}

unsigned int op_funct3(unsigned int cmd) {
    return ((cmd >> 12) & 0b111);
}

unsigned int op_funct7(unsigned int cmd) {
    return (cmd >> 25);
}

int op_Iimm(unsigned int cmd) {
    return (((long long)cmd<<32) >> 52);
}

int op_Simm(unsigned int cmd) {
    return (((cmd >> 7) & 0b11111)|((((long long)cmd<<32) >> 57)<<5));
}
```

```
int op_Bimm(unsigned int cmd) {
    return (((cmd >> 7) & 0b11110) | ((cmd >> 20) & 0b11111100000) | ((cmd <<
4)&(1<<11)) | ((((long long)cmd<<32)>>63)<<12));
}

int op_Uimm(unsigned int cmd) {
    return ((((long long)cmd<<32)>>44)<<12);
}

int op_Jimm(unsigned int cmd) {
    return ((cmd & 0xFF000) | ((cmd >> 20) & 0x7FE) | ((cmd >>9) & (1 << 11))
| ((((long long)cmd<<32) >> 63)<<20));
}

const char *cmdname(unsigned int cmd) {
    switch (op_opcode(cmd)) {
    case 0x03: {
        switch (op_funct3(cmd)) {
        case 0b000: return "lb";
        case 0b001: return "lh";
        case 0b010: return "lw";
        case 0b100: return "lbu";
        case 0b101: return "lhu";
        }
        return "unknown_instruction";
    }
    case 0x0f: return "fence";
    case 0x13: {
        switch (op_funct3(cmd)) {
        case 0b000: return "addi";
        case 0b001: return "slli";
        case 0b010: return "slti";
        case 0b011: return "sltiu";
        case 0b100: return "xori";
        case 0b101: return op_funct7(cmd) ? "srai" : "srli";
        case 0b110: return "ori";
        case 0b111: return "andi";
        }
        return "unknown_instruction";
    }
    case 0x17: return "auipc";
    case 0x23: {
        switch (op_funct3(cmd)) {
        case 0b000: return "sb";
        case 0b001: return "sh";
        case 0b010: return "sw";
        }
        return "unknown_instruction";
```

```
    }
    case 0x33: {
        if (op_funct7(cmd) == 1)
            switch (op_funct3(cmd)) {
            case 0b000: return "mul";
            case 0b001: return "mulh";
            case 0b010: return "mulhsu";
            case 0b011: return "mulhu";
            case 0b100: return "div";
            case 0b101: return "divu";
            case 0b110: return "rem";
            case 0b111: return "remu";
            }
        else switch (op_funct3(cmd)) {
        case 0b000: return op_funct7(cmd) ? "sub" : "add";
        case 0b001: return "sll";
        case 0b010: return "slt";
        case 0b011: return "sltu";
        case 0b100: return "xor";
        case 0b101: return op_funct7(cmd) ? "sra" : "srl";
        case 0b110: return "or";
        case 0b111: return "and";
        }
        return "unknown_instruction";
    }
    case 0x37: return "lui";
    case 0x63: {
        switch (op_funct3(cmd)) {
        case 0b000: return "beq";
        case 0b001: return "bne";
        case 0b100: return "blt";
        case 0b101: return "bge";
        case 0b110: return "bltu";
        case 0b111: return "bgeu";
        }
        return "unknown_instruction";
    }
    case 0x67: return "jalr";
    case 0x6F: return "jal";
    case 0x73: return op_funct7(cmd) ? "ebreak" : "ecall";
    }
    return "unknown_instruction";
}

cmd_fmt cmdtype(unsigned int cmd) {
    switch (op_opcode(cmd)) {
    case 0x03: return IL;
    case 0x0f: return I0;
```

```
    case 0x13: return I3;
    case 0x17: return U;
    case 0x23: return S;
    case 0x33: return R;
    case 0x37: return U;
    case 0x63: return B;
    case 0x67: return I3;
    case 0x6F: return J;
    case 0x73: return I0;
    }
    return I0;
}


typedef struct elf32_header{
    unsigned char   e_magic[4];
    unsigned char   e_class;
    unsigned char   e_data;
    unsigned char   e_version;
    unsigned char   e_osabi;
    unsigned char   e_abiversion;
    unsigned char   e_pad[7];
    unsigned short  e_type;
    unsigned short  e_machine;
    unsigned int    e_version2;
    unsigned int    e_entry;
    unsigned int    e_phoff;
    unsigned int    e_shoff;
    unsigned int    e_flags;
    unsigned short  e_ehsize;
    unsigned short  e_phentsize;
    unsigned short  e_phnum;
    unsigned short  e_shentsize;
    unsigned short  e_shnum;
    unsigned short  e_shstrndx;
} Elf32_Header;

typedef struct elf32_pheader {
    unsigned int    p_type;
    unsigned int    p_offset;
    unsigned int    p_vaddr;
    unsigned int    p_paddr;
    unsigned int    p_filesz;
    unsigned int    p_memsz;
    unsigned int    p_flags;
    unsigned int    p_align;
} Elf32_ProgHeader;

typedef struct elf32_sheader {
```

```
    unsigned int    sh_name;
    unsigned int    sh_type;
    unsigned int    sh_flags;
    unsigned int    sh_addr;
    unsigned int    sh_offset;
    unsigned int    sh_size;
    unsigned int    sh_link;
    unsigned int    sh_info;
    unsigned int    sh_addralign;
    unsigned int    sh_entsize;
} Elf32_SecHeader;

typedef struct elf32_symbol {
    unsigned int    st_name;
    unsigned int    st_value;
    unsigned int    st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    unsigned short  st_shndx;
} Elf32_Symbol;


int  i,flab,msymtab,symcount,mtext;
unsigned int tc, txtoff, txtlen, tcmd, pc;

void qs(unsigned int* arr, int first, int last)
{
    if (first < last)
    {
        int left = first, right = last, middle = arr[(left + right) / 2];
        do
        {
            while (arr[left] < middle) left++;
            while (arr[right] > middle) right--;
            if (left <= right)
            {
                unsigned int tmp = arr[left];
                arr[left] = arr[right];
                arr[right] = tmp;
                left++;
                right--;
            }
        } while (left <= right);
        qs(arr, first, right);
        qs(arr, left, last);
    }
}

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

```
{
    FILE* ef,*af;
    int i;
    size_t readed;
    Elf32_Header ElfHeader;

    if (argc < 2) {
        printf("Too few paramaters.\n");
        exit(-1);
    }
    if ((fopen_s(&ef, argv[1], "rb")) != 0) {
        printf("Unable to open input file\n");
        exit(-1);
    }

    if ((fopen_s(&af, argv[2], "wt")) != 0) {
        printf("Unable to open output file\n");
        exit(-1);
    }

    readed = fread(&ElfHeader, 1, sizeof(Elf32_Header), ef);
    if (readed < sizeof(Elf32_Header)) {
        printf("Unexpected length of file\n");
        exit(-1);
    }
    if ((ElfHeader.e_magic[0] != 0x7F) || (ElfHeader.e_magic[1] != 0x45) ||
(ElfHeader.e_magic[2] != 0x4C) || (ElfHeader.e_magic[3] != 0x46) ||
(ElfHeader.e_class != 1)) {
        printf("Unsupported file type\n");
        exit(-1);
    }
    if (ElfHeader.e_machine != 0xF3) {
        printf("Unsupported processor type\n");
        exit(-1);
    }

    Elf32_ProgHeader* ProgHeads = new Elf32_ProgHeader[ElfHeader.e_phnum];
    fseek(ef, ElfHeader.e_phoff, SEEK_SET);
    fread(ProgHeads, ElfHeader.e_phentsize, ElfHeader.e_phnum, ef);

    Elf32_SecHeader* SecHeads = new Elf32_SecHeader[ElfHeader.e_shnum];
    fseek(ef, ElfHeader.e_shoff, SEEK_SET);
    fread(SecHeads, ElfHeader.e_shentsize, ElfHeader.e_shnum, ef);

    char* SecNames = new char[SecHeads[ElfHeader.e_shstrndx].sh_size];
    fseek(ef, SecHeads[ElfHeader.e_shstrndx].sh_offset, SEEK_SET);
    fread(SecNames, SecHeads[ElfHeader.e_shstrndx].sh_size, 1, ef);
```

```
    for (i = 0; i < ElfHeader.e_shnum; i++) {
        if (SecHeads[i].sh_type == 2) msymtab = i;
        if (strcmp(&SecNames[SecHeads[i].sh_name], ".text") == 0) mtext = i;
    }

    char* SymNames = new char[SecHeads[SecHeads[msymtab].sh_link].sh_size];
    fseek(ef, SecHeads[SecHeads[msymtab].sh_link].sh_offset, SEEK_SET);
    fread(SymNames, SecHeads[SecHeads[msymtab].sh_link].sh_size, 1, ef);

    symcount = SecHeads[msymtab].sh_size / SecHeads[msymtab].sh_entsize;
    Elf32_Symbol* Symbols = new Elf32_Symbol[symcount];
    fseek(ef, SecHeads[msymtab].sh_offset, SEEK_SET);
    fread(Symbols, 1, SecHeads[msymtab].sh_size, ef);

    txtlen = SecHeads[mtext].sh_size >> 2;
    txtoff = SecHeads[mtext].sh_addr;
    unsigned int* txt = new unsigned int[txtlen];
    unsigned int* mlabels = new unsigned int[txtlen];
    fseek(ef, SecHeads[mtext].sh_offset, SEEK_SET);
    fread(txt, 4, txtlen, ef);

    int mlabcnt = 0;
    int fmlab = -1;
    int tc = 0;
    while (tc < txtlen) {
        pc = txtoff + tc * 4;
        switch (cmdtype(txt[tc])) {
        case B: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Bimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            if (flab >= 0) break;
            mlabels[mlabcnt] = op_Bimm(txt[tc]) + pc;
            mlabcnt++;
            break;
        }
        case J: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Jimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
```

```
                }
            }
            if (flab >= 0) break;
            mlabels[mlabcnt] = op_Jimm(txt[tc]) + pc;
            mlabcnt++;
            break;
        }
        };
        tc++;
    }

    if (mlabcnt>1) qs(mlabels, 0, mlabcnt - 1);

    fprintf(af,".text\n");
    tc = 0;
    while (tc < txtlen) {
        pc = txtoff + tc * 4;
        for (i = 0; i < symcount; i++) {
            if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx ==
mtext) && (pc == Symbols[i].st_value)) {
                fprintf(af, "%08x   <%s>:\n", pc,
&SymNames[Symbols[i].st_name]);
            }
        }
        for (i = 0; i < mlabcnt; i++) {
            if (pc == mlabels[i]) {
                fprintf(af, "%08x   <L%d>:\n", pc, i);
            }
        }
        switch (cmdtype(txt[tc])) {
        case R: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %s\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])]); break;
        case IL: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %d(%s)\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Iimm(txt[tc]),
regnames[op_rs1(txt[tc])]); break;
        case I3: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %d\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], regnames[op_rs1(txt[tc])],
op_Iimm(txt[tc])); break;
        case I0: fprintf(af, "   %05x:\t%08x\t%7s\n", pc, txt[tc],
cmdname(txt[tc])); break;
        case S: fprintf(af, "   %05x:\t%08x\t%7s\t%s, %d(%s)\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rs2(txt[tc])], op_Simm(txt[tc]),
regnames[op_rs1(txt[tc])]); break;
        case U: fprintf(af, "   %05x:\t%08x\t%7s\t%s, 0x%x\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Uimm(txt[tc])>>12); break;
        case J: {
            flab = -1;
```

```
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Jimm(txt[tc])+pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            fmlab = -1;
            for (i = 0; i < mlabcnt; i++) {
                if ((op_Jimm(txt[tc]) + pc) == mlabels[i]) {
                    fmlab = i;
                    break;
                }
            }
            if (flab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x <%s>\n",
pc, txt[tc], cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Jimm(txt[tc]) +
pc, &SymNames[Symbols[flab].st_name]);
            else if (fmlab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x
<L%d>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rd(txt[tc])],
op_Jimm(txt[tc]) + pc, fmlab);
            else fprintf(af, "   %05x:\t%08x\t%7s\t%s, %08x\n", pc, txt[tc],
cmdname(txt[tc]), regnames[op_rd(txt[tc])], op_Jimm(txt[tc])+pc);
            break;
        }
        case B: {
            flab = -1;
            for (i = 0; i < symcount; i++) {
                if (((Symbols[i].st_info & 0x0f) != 3) && (Symbols[i].st_shndx
== mtext) && ((op_Bimm(txt[tc]) + pc) == Symbols[i].st_value)) {
                    flab = i;
                    break;
                }
            }
            fmlab = -1;
            for (i = 0; i < mlabcnt; i++) {
                if ((op_Bimm(txt[tc]) + pc) == mlabels[i]) {
                    fmlab = i;
                    break;
                }
            }
            if (flab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %08x
<%s>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc,
&SymNames[Symbols[flab].st_name]);
            else if (fmlab >= 0) fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s,
%08x <L%d>\n", pc, txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc, fmlab);
```

```
            else fprintf(af, "   %05x:\t%08x\t%7s\t%s, %s, %08x\n", pc,
txt[tc], cmdname(txt[tc]), regnames[op_rs1(txt[tc])],
regnames[op_rs2(txt[tc])], op_Bimm(txt[tc]) + pc);
            break;
        }
        default:fprintf(af, "\t%08x: %s\n", pc, cmdname(txt[tc]));
        };

        tc++;
    }
    fprintf(af, "\n.symtab\nSymbol Value            Size
Type   Bind    Vis     Index Name\n");
    char indexstr[10];
    for (i = 0; i < symcount; i++) {
        getindexstr(Symbols[i].st_shndx,indexstr);
        fprintf(af, "[%4i] 0x%-15X %5i %-8s %-8s %-8s %6s %s\n", i,
Symbols[i].st_value, Symbols[i].st_size, gettypestr(Symbols[i].st_info),
getbindstr(Symbols[i].st_info),"DEFAULT", indexstr,
&SymNames[Symbols[i].st_name]);
    }
    fclose(ef);
    fclose(af);
}
```