Лабораторная работа №4: ISA

Цыганов Пётр Игоревич M3139

December 2023

https://github.com/skkv-itmo2/itmo-comp-arch-2023-riscv-pit-tsy

1 Инструментарий:

Python 3.11.7

2 Результат работы

```
.text

00010074        <main>:
   10074:       ff010113            addi sp, sp, -16
   10078:       00112623              sw ra, 12(sp)
   1007c:       030000ef             jal ra, 0x100ac <mmul>
   10080:       00c12083              lw ra, 12(sp)
   10084:       00000513            addi a0, zero, 0
   10088:       01010113            addi sp, sp, 16
   1008c:       00008067            jalr zero, 0(ra)
   10090:       00000013            addi zero, zero, 0
   10094:       00100137             lui sp, 0x100
   10098:       fddff0ef             jal ra, 0x10074 <main>
   1009c:       00050593            addi a1, a0, 0
   100a0:       00a00893            addi a7, zero, 10
   100a4:       0ff0000f           fence iorw, iorw
   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        <L2>:
   100c8:       fec50e13            addi t3, a0, -20
   100cc:       000f0313            addi t1, t5, 0
   100d0:       000f8893            addi a7, t6, 0
```

```
   100d4:       00000813            addi a6, zero, 0

000100d8       <L1>:
   100d8:       00088693            addi a3, a7, 0
   100dc:       000e0793            addi a5, t3, 0
   100e0:       00000613            addi a2, zero, 0

000100e4       <L0>:
   100e4:       00078703              lb a4, 0(a5)
   100e8:       00069583              lh a1, 0(a3)
   100ec:       00178793            addi a5, a5, 1
   100f0:       02868693            addi a3, a3, 40
   100f4:       02b70733             mul a4, a4, a1
   100f8:       00e60633             add a2, a2, a4
   100fc:       fea794e3             bne a5, a0, 0x100e4, <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, 0x100d8, <L1>
   10114:       050f0f13            addi t5, t5, 80
   10118:       01478513            addi a0, a5, 20
   1011c:       fa5f16e3             bne t5, t0, 0x100c8, <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

3 Что было реализовано?

Была реализована поддержка RV32I, RV32M, symtab.

4 Открытие ELF-файла

ELF файл был открыт как бинарный файл в hex формате. Один байт равен числу из двух шестнадцатиричных циферок. Для удобства ELF записывается в массив байтов, представленных в виде шестнадцатиричных чисел.

```
def bytes_str_to_array(bytes_str):
    result = []
    for i in range(0, len(bytes_str), 2):
        result.append(bytes_str[i : i + 2])
    return result

with open(input_file, 'rb') as file:
    elf = bytes_str_to_array(file.read().hex())
```

4.1 to_uint(hexes)

Эта функция используется в дальнейшем в коде, чтобы переводить массив байтов в безнаковый целочисленный формат. Она переводит массив байт согласно кодированию little endian. То есть последний элемент массива байт будет иметь старший разряд.

```
def to_uint(hexes):
    result = 0
    for i in range(len(hexes) - 1, -1, -1):
        result = (result << 8) + int(hexes[i], 16)
    return result
```

5 Парсинг ELF-файла

5.1 Elf header

Структура elf header была взята с
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
Функция read_header(elf) парсит elf header и возвращает словарь:

1. e_ident - эти байты предоставляют машинно-независимые данные, с помощью которых можно декодировать и интерпретировать содержимое файла.
2. e_type - определяет тип объектного файла.
3. e_machine - указывает требуемую архитектуру для отдельного файла.
4. e_version - определяет версию объектного файла
5. e_entry - виртуальный адрес, на который система сначала передает управление, таким образом запуская процесс. Если файл не имеет связанной точки входа, этот элемент содержит ноль.
6. e_phoff - offset таблицы заголовков программы в байтах.

7. e_shoff - offset таблицы заголовков разделов в байтах.
8. e_flags - особые процессорные флаги.
9. e_ehsize - размер заголовка ELF-файла.
10. e_phentsize - размер в байтах одной записи в таблице заголовков программы файла.
11. e_phnum - количество записей в таблице заголовков программы
12. e_shentsize - размер заголовка раздела в байтах.
13. e_shnum - количество записей в таблице заголовков разделов.
14. e_shstrndx - индекс таблицы заголовков разделов записи, связанной с таблицей строк имен разделов .shstrtab.

В дальнейшем в словаре header будет содержаться header elf файла, который считала программа.

5.2 Table of section headers

5.2.1 Структура

Структура section header была взята с
https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.
Таблица заголовков секций начинается с адреса e_shoff в elf файле. В ней e_shnum записей, каждая имеет размер равный e_shentsize. Функция get_sections(elf, header) парсит эту табличку.

```
def get_sections(elf, header):
    start_data = header['shoff']
    sections = []
    for _ in range(header['shnum']):
        section_info = get_section_info(elf, start_data);
        sections.append(section_info)

        start_data += header['shentsize']
    return sections
```

Функция get_section_info(elf, start) возвращает словарь, содержащий распарсенную информацию об записи в таблице.

1. sh_name - смещение к строке в разделе .shstrtab, представляющей название этого раздела.
2. sh_type - определяет тип этого заголовка.
3. sh_flags - определяет атрибуты этого раздела.
4. sh_addr - виртуальный адрес раздела в памяти для загружаемых разделов.
5. sh_offset - смещение секции в ELF файле.
6. sh_size - размер раздела в байтах.
7. sh_link - содержит индекс раздела соответствующего раздела. Используется в зависимости от типа раздела.
8. sh_info - содержит дополнительную информацию о разделе.

9. sh_addralign - содержит требуемое выравнивание раздела.

10. sh_entsize - Содержит размер в байтах каждой записи для разделов, содержащих записи фиксированного размера. В противном случае это поле содержит ноль.

В дальнейшем sections - это массив содержащий заголовки секций.

5.2.2 Таблица имён секций

Функция get_section_name(sh_name) возвращает имя соответствующей секции.

```
def get_section_name(sh_name):
    global elf, header, sections

    shstrndx = header['shstrndx']
    ptr = sections[shstrndx]['offset'] + sh_name

    name = ''
    while elf[ptr] != '00':
        name += chr(int(elf[ptr], 16))
        ptr += 1
    return name
```

В заголовке секции содежится поле sh_name, оно указывает на смещение в табличке .shstrtab, где записано имя секции.

e_shstrndx - индекс в табличке заголовков секций, соответствующий .shstrtab. Из записи в табличке заголовков секций мы получаем offset в elf файле. Сдвигаемся на него и sh_name и получаем имя секции, символ с кодом 0 сигнализирует, что это конец имени.

6 Парсинг .symtab

Информация про .symtab была взята отсюда:
https://docs.oracle.com/cd/E26502_01/html/E26507/chapter6-79797html#scrolltoc

6.1 Структура

1. st_name - смещение в таблице .strtab, где находится имя символа

2. st_value - значение связанного символа. В зависимости от контекста может быть как адресом, так и абсолютным значением.

3. st_size - размер соответствующего символа. Это значение может быть равно нулю, если у символа нет размера, или он неизвестен.

4. st_info - тип символа и атрибуты привязки.

```
#define ELF32_ST_BIND(info)          ((info) >> 4)
#define ELF32_ST_TYPE(info)          ((info) & 0xf)
#define ELF32_ST_INFO(bind, type)    (((bind)<<4)+((type)&0xf))
```

5. st_other - symbol's visibility.

```
#define ELF32_ST_VISIBILITY(o)       ((o)&0x3)
```

6. st_shndx - содержит соответствующий индекс таблицы заголовков разделов.

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOOS	10
STB_HIOS	12
STB_LOPROC	13
STB_HIPROC	15

Рис. 1: Таблица декодирование bind

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_COMMON	5
STT_TLS	6
STT_LOOS	10
STT_HIOS	12
STT_LOPROC	13
STT_SPARC_REGISTER	13
STT_HIPROC	15

Рис. 2: Таблица декодирования type

Name	Value
STV_DEFAULT	0
STV_INTERNAL	1
STV_HIDDEN	2
STV_PROTECTED	3
STV_EXPORTED	4
STV_SINGLETON	5
STV_ELIMINATE	6

Рис. 3: Таблица декодирования symbol Visibility

6.2 Имя символа

В таблице заголовков секций находим информацию про таблицу .strtab. Смещаемся на sh_offset, а потом на st_offset. Парсим имя символа аналогично тому, как парсили имя секции ранее.

7 Парсинг меток

В таблице .symtab можно найти именованные метки. Они имеют тип "FUNC". Поэтому мы проходимся по всему .symtab и запоминаем метки. В данном случае st_value будет представлять адрес/offset метки.

Также в RISC-V есть инструкции с метками. Если в инструкции есть метка, но её нет в .symtab, то мы создаём метку L%i, где %i - первый не использованный номер метки.

Функция get_labels(text, symtab) возвращает словарь меток.

```
def get_labels(text, symtab):
    labels = dict()
    for entry in symtab:
        if get_st_type(entry) == 'FUNC':
            labels[entry['value']] = entry['name']

    cnt = 0
    for cmd in text:
        if cmd[2] == 'J' or cmd[2] == 'B':
```

```
            value = cmd[0] + cmd[5]
            if value not in labels:
                labels[value] = 'L' + str(cnt)
                cnt += 1
    return labels
```

8 Парсинг .text

unpriv-isa-asciidoc.pdf - документация RISC-V. Из неё берём как парсить команды(все таблички, которые взял приложу). Каждая команда это 32 бита, то есть 4 байта.

То есть в табличке заголовков секций мы находим секцию '.text'. Смещаемся на sh_offset и начинаем парсить секцию. Она имеет размер sh_size, то есть вмещает в себя $\frac{sh\_size}{4}$ комманд. Мы просто идём по elf файлу, начиная с sh_offset и парсим столько комманд по 4 байта.

Каждая команда имеет свой opcode по нему мы можем определить тип команды. Смотря на тип команды мы определяем как дальше парсить. Для каждого типа биты команды парсятся по разному.

31 27	26 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

Рис. 4: Типы команд

Как видим opcode - это первые 7 битов команды. Смотря на них, мы определяем тип. В файле, ссылку на который я указал ранее, можно найти таблицу на странице 142-143 и на странице 144-145, которая показывает набор всех команд RV32I и RV32M и как их парсить. Она огромная, поэтому полностью её вставлять я не буду, но опишу по группам.

8.1 Регистры

rs1, rs2 и rd - регистры. Функция get_register_name(register_number) возвращает имя регистра по его номеру. Имена регистров взяты по ссылке:
https://en.wikichip.org/wiki/risc-v/registers

```
def get_register_name(register_number):
    register_names = [
        "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"
    ]

    if 0 <= register_number < len(register_names):
```

```
        return register_names[register_number]
    else:
        return register_number
```

8.2 funct3 и funct7

Эти биты помогают нам отличить одну команду от другой. По этим битам мы определеям, что за команда перед нами.

8.3 Immediate

Immediate - может быть как константой, так и смещением в зависимости от инструкции. Также стоит заметить, что зачастую это знаковый тип. Покажем как вычислять его для каждого типа команд. Как парсить для отдельного типа было взято из документации(начиная со страницы 49):
https://github.com/johnwinans/rvalp/releases/download/v0.17/rvalp.pdf

8.3.1 U-type

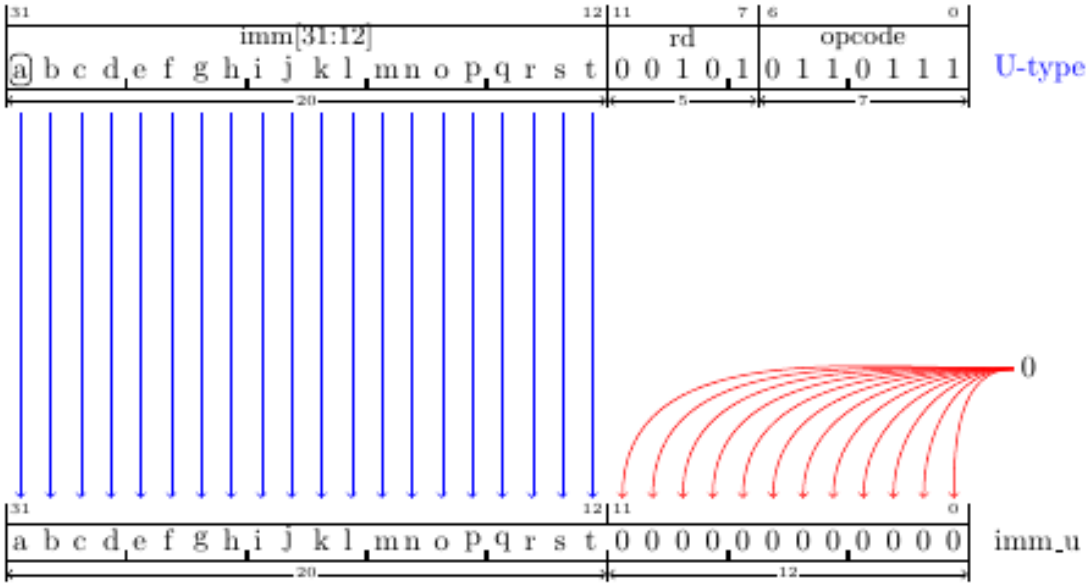

Рис. 5: U-type

Примечание: выводится U-type без ноликов. Об этом упомянуто в документации.

8.3.2 J-type

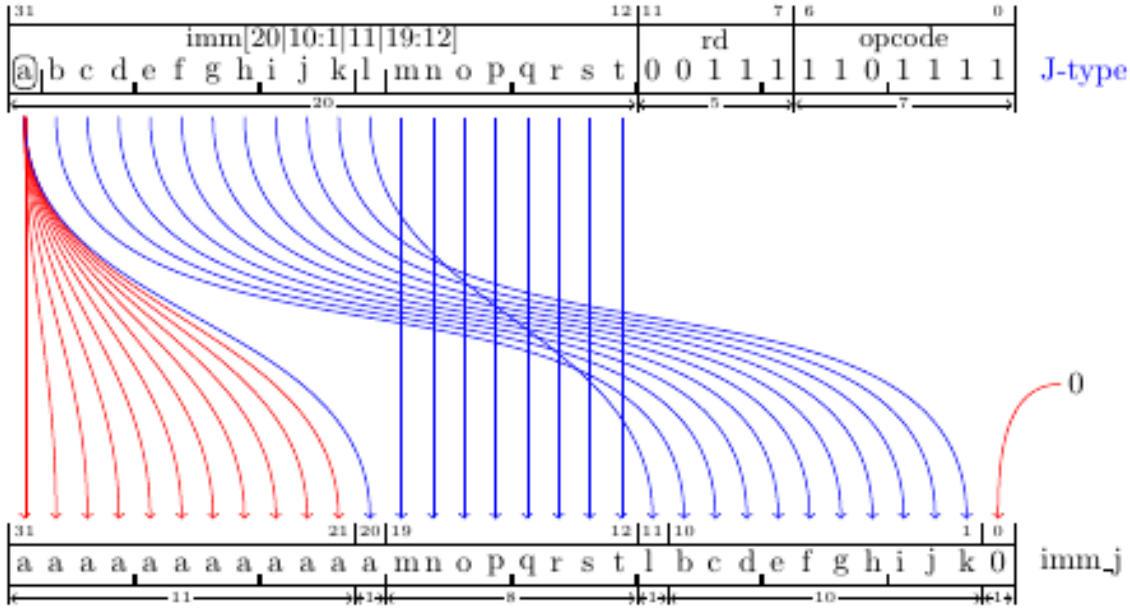

Рис. 6: J-type

8.3.3 I-type

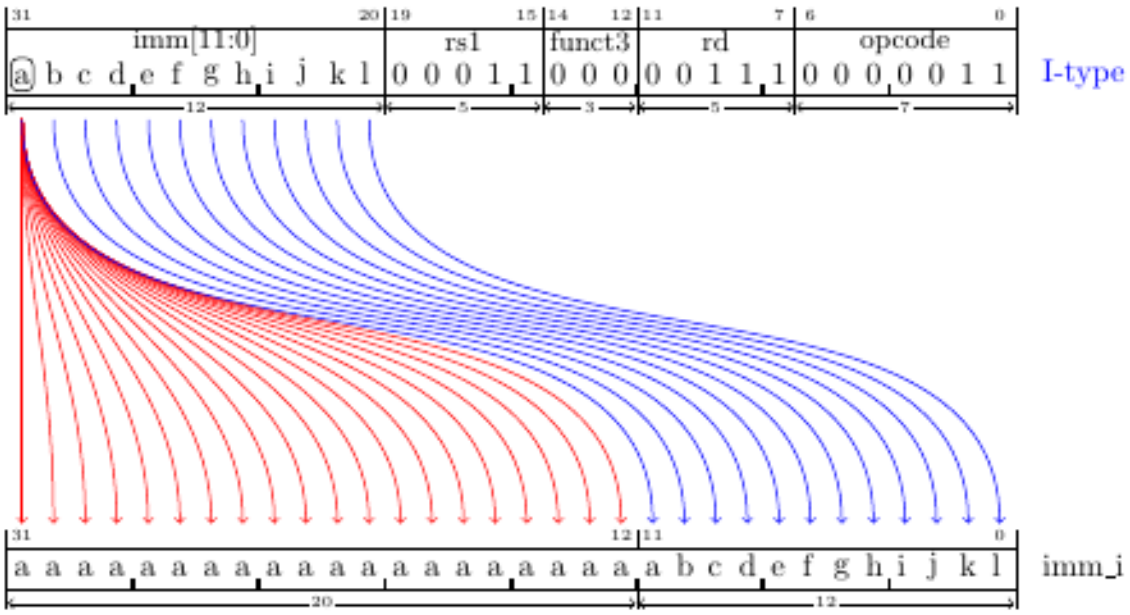

Рис. 7: I-type

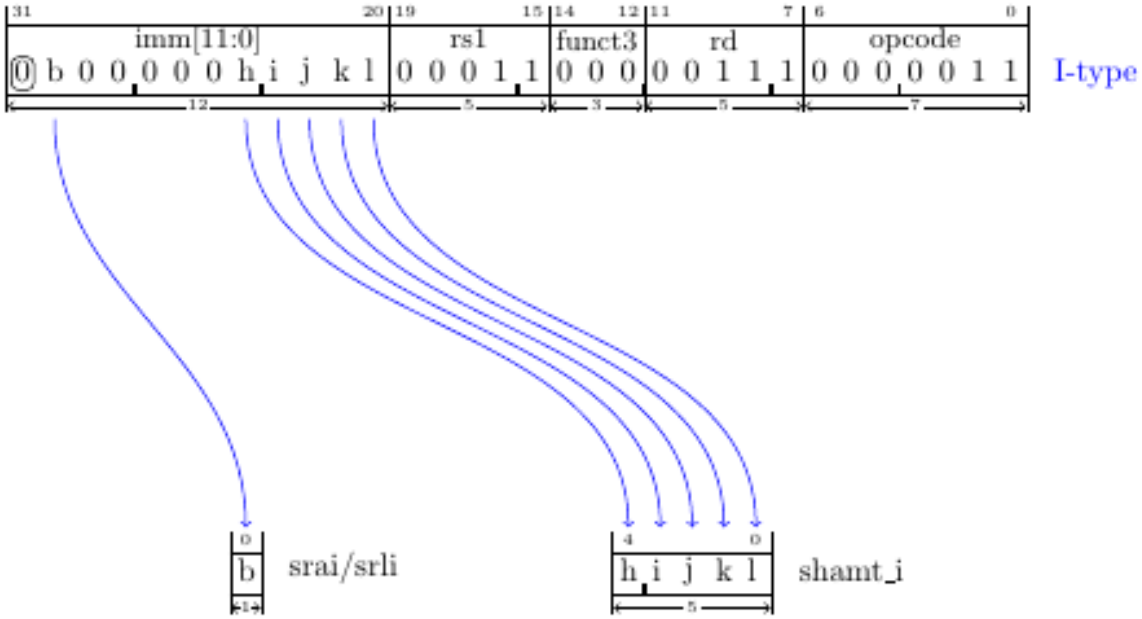

Рис. 8: I-type with shamt

8.3.4 S-type

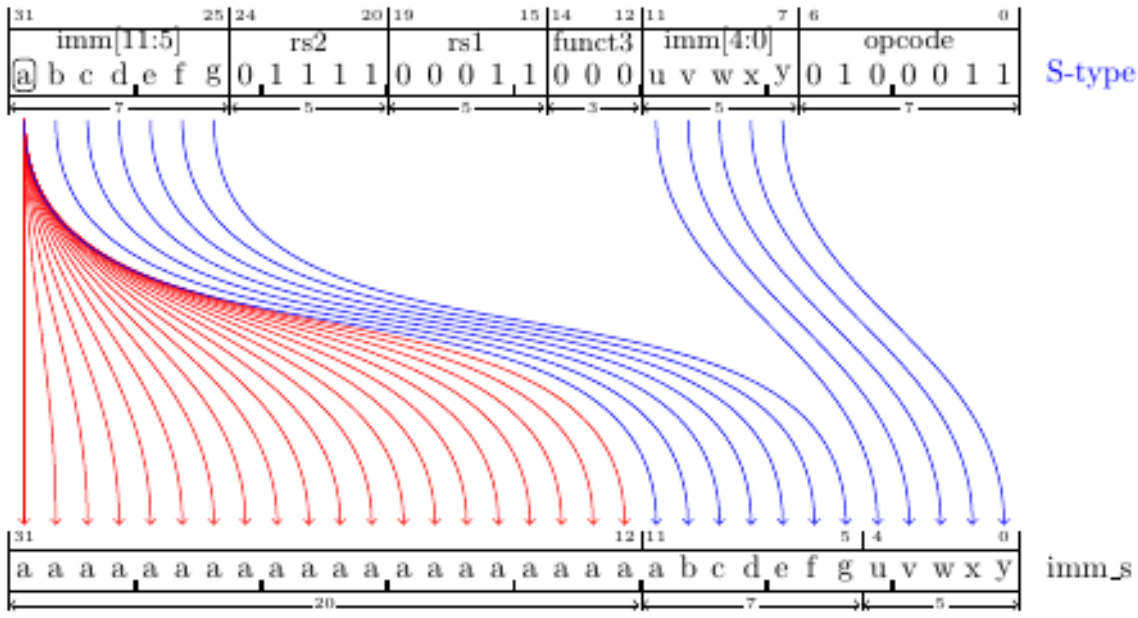

Рис. 9: S-type

8.3.5 B-type

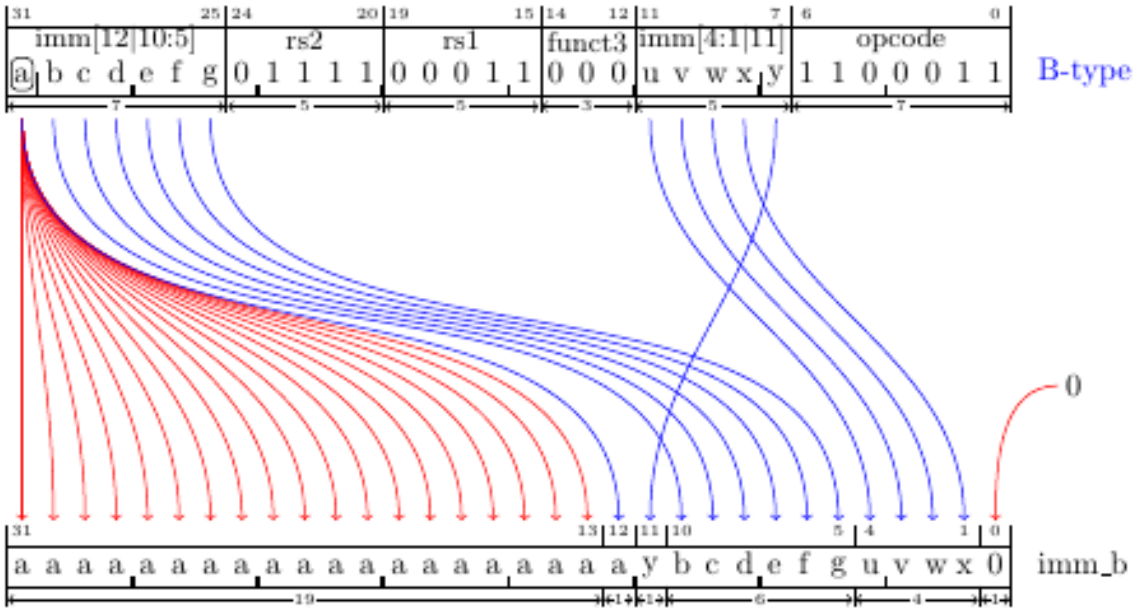

Рис. 10: B-type

8.4 Как парсить команды?

Из табличек, про которые писал ранее, мы понимаем какая команда перед нами. Как парсить отдельные параметры будет написано в предыдущей секции.

Мы смотрим на opcode и тем самым определяем тип команды, потом детализируем команду в зависимости от типа.

8.4.1 U-type

imm[31:12]	rd	0110111	LUI
imm[31:12]	rd	0010111	AUIPC

Как видим, у U-type соответствующий уникальный opcode по нему мы понимаем, какая перед нами команда. Immediate в данном случае представляет из себя offset.

8.4.2 J-type

imm[20|10:1|11|19:12]	rd	1101111	JAL

Команды J-type имеют уникальный opcode. В данном случае opcode = 0b1101111 соответсвует команда JAL. Immediate - это offset, а также метка.

8.4.3 B-type

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

Мы смотрим на opcode и на funct3. В сумме они дают уникальный код, который соответсвует уникальной команде. Immediate - это offset, а также метка.

8.4.4 load

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

opcode = 0b0000011. funct3 и funct7 - индефикатор уникальной команды.

8.4.5 I-type

парситься аналогично предыдущему, просто опкоды другие.

8.4.6 S-type

Пример операций(opcode соответствующий и уник funct3):

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

8.4.7 R-type

Соотвествующий opcode, уникальный funct7 и funct3, если объединить:

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

8.5 FENCE

fm	pred	succ	rs1	000	rd	0001111	FENCE

Биты в pred и succ проставляются в соответсвие с картинкой(взята из документации, которую указывал выше):

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
0	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
0	predecessor				successor				0	FENCE	0	MISC-MEM

8.5.1 Инструкции без аргументов

1000	0011	0011	00000	000	00000	0001111	FENCE.TSO
0000	0001	0000	00000	000	00000	0001111	PAUSE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

Есть инструкции, у которых определены все биты. Такие инструкции не имеют аргументов. Определяются они полным сравнинием побитно.