Лабораторная работа №4

Сайфуллин Искандар M3136

Версия Python: 3.11.6
Репозиторий: https://github.com/skkv-itmo2/itmo-comp-arch-2023-riscv-1sSay

1. Вывод программы

```
.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
```

2. Техническое описание

Сначала выполнялась проверка на количество аргуметном программы и существование ELF-файла (если аргументов не два или файл не существует, то программа завершается с ненулевым кодом).

Выполнение программы состоит из трёх частей:

- Парсинг заголовка ELF-файла
- Парсинг секций ELF-файла
- Дизассемблирование секции `.text`

Структура ELF-файла

> ELF (Executable and Linkable Format) - стандарт исполняемых файлов, использующийся в UNIX-подобных системах.

1. **Заголовок ELF-файла**. Находится в самом начале файла и занимает 52 или 64 байта. В нём последовательно хранится информация об общей структуре файла.
2. **Заголовки программы**. В них находятся описания частей программы.
3. **Секции**. В них последовательно хранятся секции (например, `.text`, `.symtab` и `.strtab`)
4. **Заголовки секций**. Здесь хранится таблица с атрибутами секций.

2.1 Парсинг заголовка ELF-файла

Прежде всего необходимо распарсить заголовок файла, который занимает 52 байта (или 64 байта в зависимости от формата). Он находится в самом начале файла. Необходима следующие поля:

- `e_type` - тип файла (1 - 32-битный, 2 - 64-битный)
- `e_shoff` - сдвиг до заголовков секций
- `e_shentsize` - размер одного заголовка секции
- `e_shnum` - количество секций
- `e_shstrndx` - индекс секции `.shstrtab`

Значения этих атрибутов находятся в соотвествии со стандартом.

2.2 Парсинг заголовков секций

В атрибуте `e_shoff` находится индекс байта, с которого начинаются последовательно описываться заголовки секций. Их количество находится в `e_shnum`, размер каждого заголовка лежит в `e_shentsize`. Необходимо найти 4 секции:

- `.text`
- `.symtab`
- `.strtab`
- `.shstrtab`

2.2.1 Структура заголовка секции

Данные в заголовке секции хранятся в таком же стиле, как и у заголовка программы, то есть последовательно перечисляются атрибуты секции. Необходимы следующие атрибуты:

- `sh_name` - имя секции. Точнее индекс первого байта в таблице `shstrtab`, где хранится строка, которая и является именем секции.
- `sh_type` - тип секции. (Например, `SECTION` или `STRTAB`)
- `sh_offset` - сдвиг секции относительно начала файла.
- `sh_size` - размер секции в байтах.

То есть нам нужно сверить имя секции и тип секции. Так, нужно найти все 3 секции, кроме .shstrtab. Индекс её находится в заголовке файла в поле `e_shstrndx`.

2.2.2 Структура `shstrtab` и `strtab`

На эти секции ссылается `.symtab`. Вообще, там лежат строки: названия секций и полей соответственно.

Строка - последовательсность char'ов, оканчивающаяся нулевым байтом.

2.2.3 Структура `.symtab`

По `sh_offset` переходим на начало секции. Далее берём описание каджого поля:

- `st_name` - сдвиг в таблице `strtab`, где хранится строка с именем поля (или нулевое значение, если имени нет)
- `st_value` - значение поля
- `st_size` - размер поля
- `st_info` - информация о поле. Далее будет подробное описание
- `st_other` - сейчас используется для обозначение видимости поля
- `st_shndx` - индекс типа, которым является поле

Парсер складывает всю информацию о поле в словарь:

```
field["Value"] = entry["st_value"]
field["Size"] = entry["st_size"]
field["Type"] = self.entry_types[entry["st_info"] & 0xF]
field["Bind"] = self.entry_binds[entry["st_info"] >> 4]
field["Vis"] = self.entry_visibility[entry["st_other"] & 0x3]
```

`Type` - тип поля (Например, секция или функция)
`Bind` - привязка поля (Например, переменная локальна или глобальная)
`Vis` - видимость переменных (Например, поле может быть скрытым)

Также в словарь добавляем имя и индекс поля. Далее так проходимся по всем полям (их количество есть в заголовке секции `.symtab`). В конце выводим эту таблицу по заданному формату.

2.2.4 Структура `.text`

В этой секции хранятся команды по 4 байта каждая (Далее будет подробное описание). Размер секции хранится в заголовке, то есть количество команд равно `sh_size / 4`.

2.3 Дизассемблирование команд

RISC-V - это стандарт команд и процессорная архитектура, которая пастулирует фиксированную длину команд.
При одинаковой кодировке архитектур с 32-, 64- и 128-битными регистрами общего назначения и операциями. (RV32I, RV64I и RV128I, соответственно)
Архитектура использует только модель little-endian.
В базовой спецификации RV32I 39 команд и 32 регистра. Также есть расширение RV32M для целочисленного умножения и деления.

2.3.1 Описание регистров (в формате ABI)

В задании необходимы 32 целочисленный регистра:

- `zero` - always zero
- `ra` - return address
- `sp` - stack pointer
- `gp` - global pointer
- `tp` - thread pointer
- `t0, s3, s4` - temporary
- `s0, s1` - saved register
- `a0-7` - argument
- `s2-11` - saved register
- `t3-6` - temporary

2.3.2 Типы команд

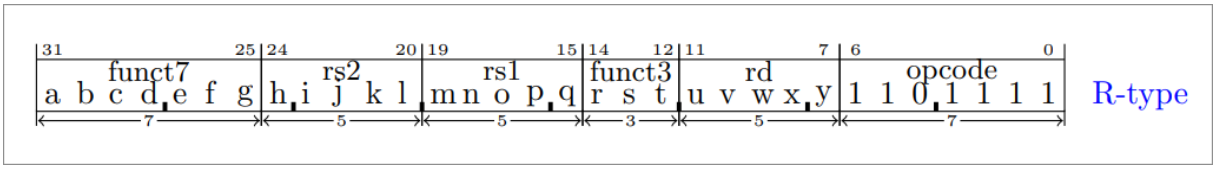
- `R` - register/register
- `I` - immediate
- `U` - upper immediate
- `B` - branch
- `S` - store
- `J` - jump

`rs1` — номер регистра в котором находится первый операнд
`rs2` — номер регистра в котором находится второй операнд
`rd` — номер регистра в который будет записан результат `imm` - некоторая константа (она имеет разный смысл и декодирование в зависимости от типа команды)

Декодирование команд типа J и B такие странные, так как это упрощает декодирование команд на аппаратном уровне (по крайней мере я так понял)

2.3.2.1 Register/register

Например, сложение или вычитание.
Устройство:



Сначала смотрим на `opcode`, `funct3` и `funct7`. Если они совпадают с одной из команд из набора RV32I или RV32M, то это известная операция и её можно дизассемблировать. Например:

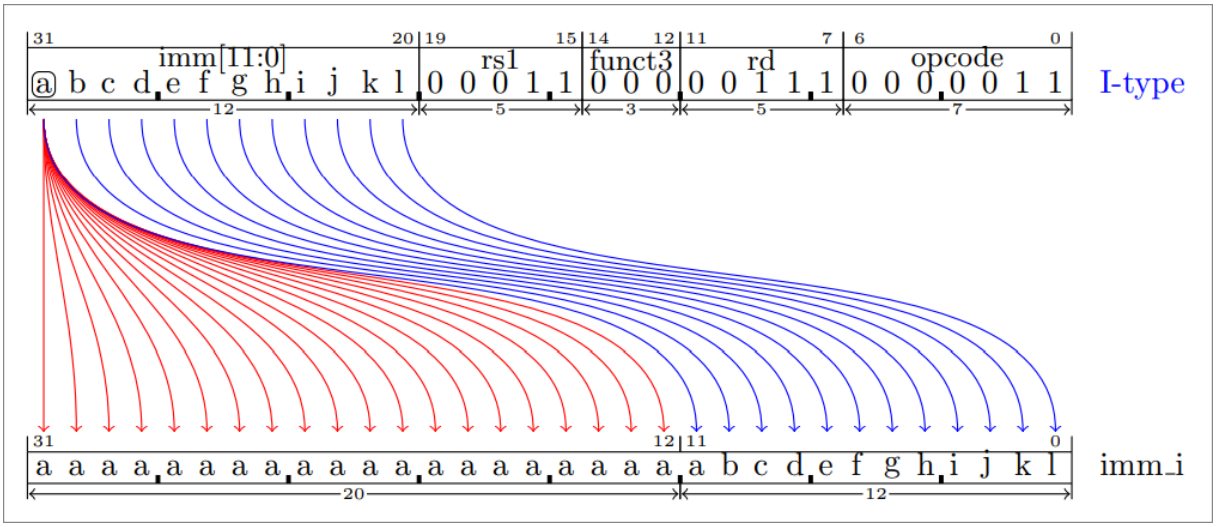
Команда: `00000000111001100000011000110011`

Opcode (младшие 7 битов): `0110011`
fuct3: `000`
fuct7: `0000000` То есть это команда `add`, регистры:
`rd`: 12 -> a2
`rs1`: 12 -> a2
`rs2`: 14 -> a4 Значит команда вывод программы будет таким:

```
<addr>: 00e60633   add a2 a2 a4
```

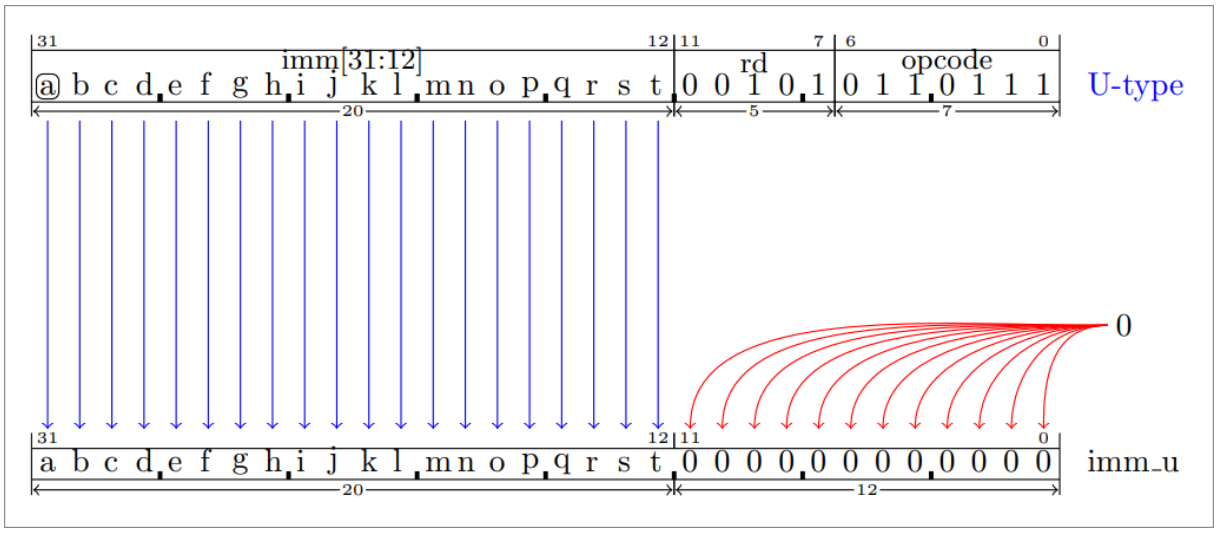
2.3.2.2 Immediate

Почти тоже самое, что и register/register, однако вместо `rs2` - некоторая константа. (Младшие 12 бит)



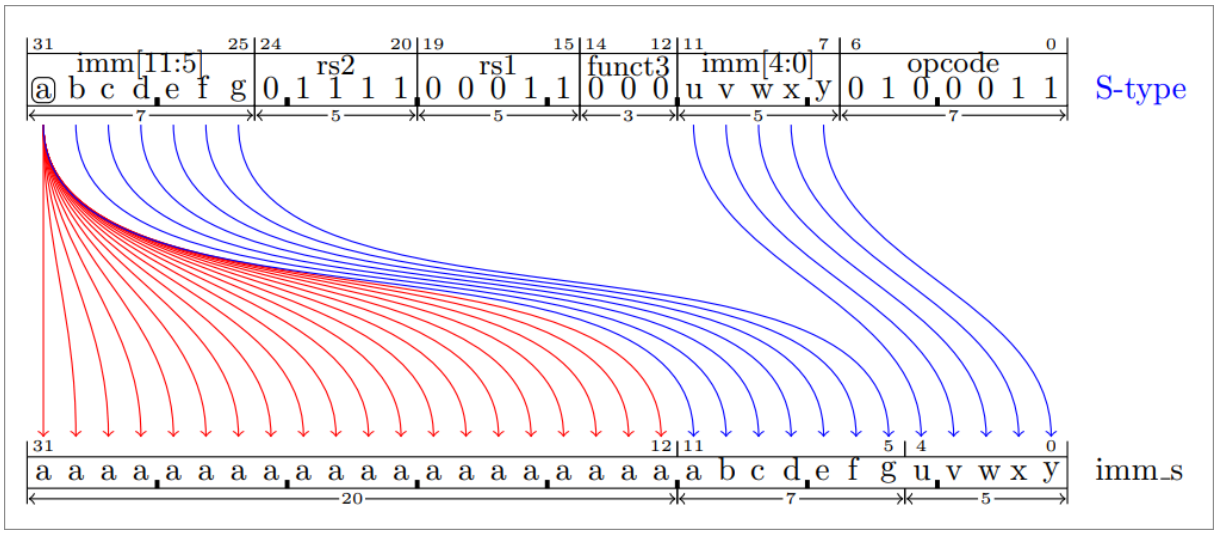
2.3.2.3 Upper immediate

Старшие 20 бит числа (в RISC-V приходится разделять такие оперции на две, так как одной команды не хватает, так как 32-битное число очевидно не поместится :D)



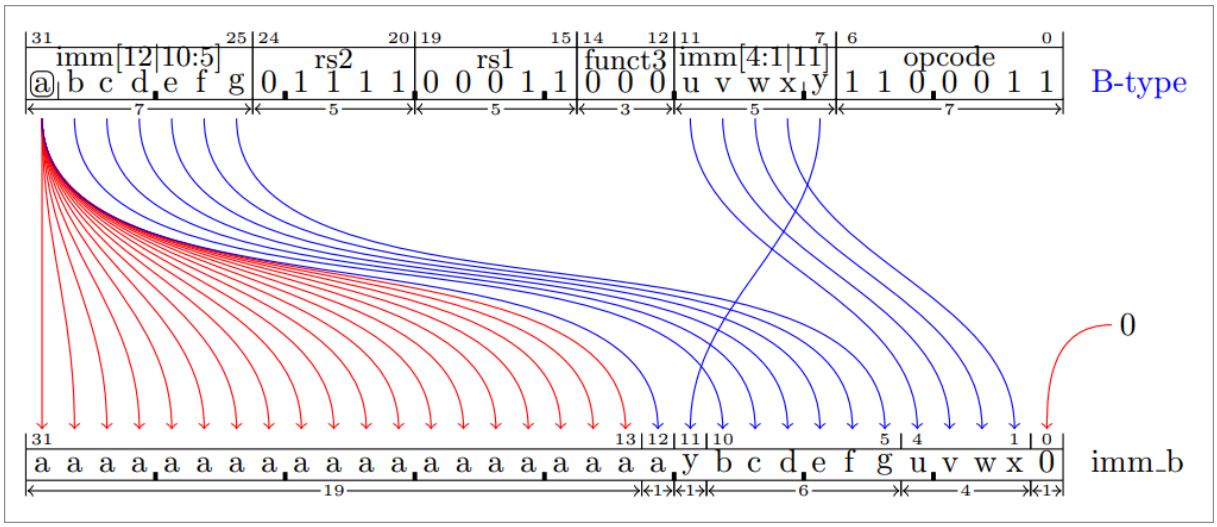
2.3.2.4 Store

Инструкция для операций с 3 регистрами, а также константой.



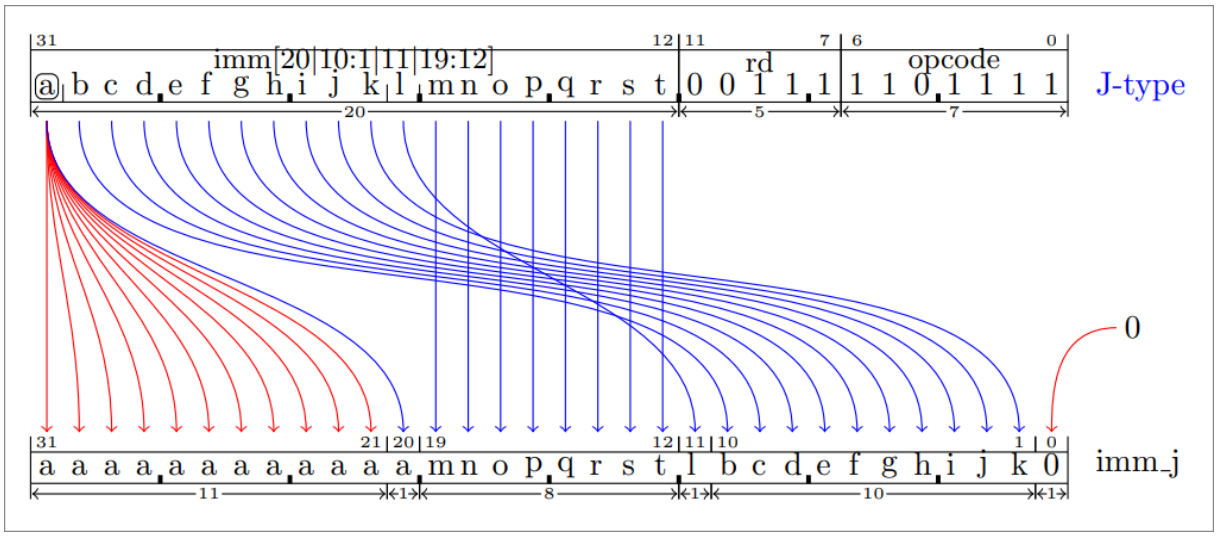
2.3.2.5 Branch

Операции ветвления. Например, `bne` переходит в другую команду, если значения в переданных регистрах не равны (branch if not equal)



2.3.2.6 Jump

Переход в другую команду по заданному в immediate адресу.



2.3.4 RV32M

Это расширение только для команд типа R. Различаются в `funct7` (на 2 позиции слева стоит единица) и `funct3` (все 8 команд имеют разные значения). Пример может таким же, какой приведён выше (только будет другое название команды, очевидно). Декодирование тоже такое же, как у обычной команды R-типа.

2.3.5 Метки

Сначала берём все поля из `.symtab`. Далее идёт последовательно по командам. Если встречается команда типа J или B, то добавляем в словать новые адреса, которых до этого не было и присваиваем им имена, начиная с L0. Далее при выводе команд (после дизассемблирования), если очередной адрес есть в словаре, то перед командой пишем её имя. Например:

Команда - `jal`
Адрес - `0x100ac`
Имя в словаре (оно было ещё в `.symtab`) - `[    9] 0x100AC ... mmul`

В итоге получаем:

```
1007c:  030000ef      jal ra, 0x100ac <mmul>

...

000100ac  <mmul>:
   100ac: 00011f37      lui t5, 0x11
```

Таким образом дизассемблируем все команды (кроме неизвестных, конечно же)

2.4 Ссылки

Про ELF:

https://refspecs.linuxbase.org/elf/gabi4+/contents.html

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

Про RISC-V:

RISC-V. Assembly Language Programming (John Winans, October 19, 2022), стр. 82-84

https://en.wikipedia.org/wiki/RISC-V#Design

https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view

https://github.com/riscv/riscv-isa-manual/releases/download/riscv-isa-release-056b6ff-2023-10-02/unpriv-isa-asciidoc.pdf