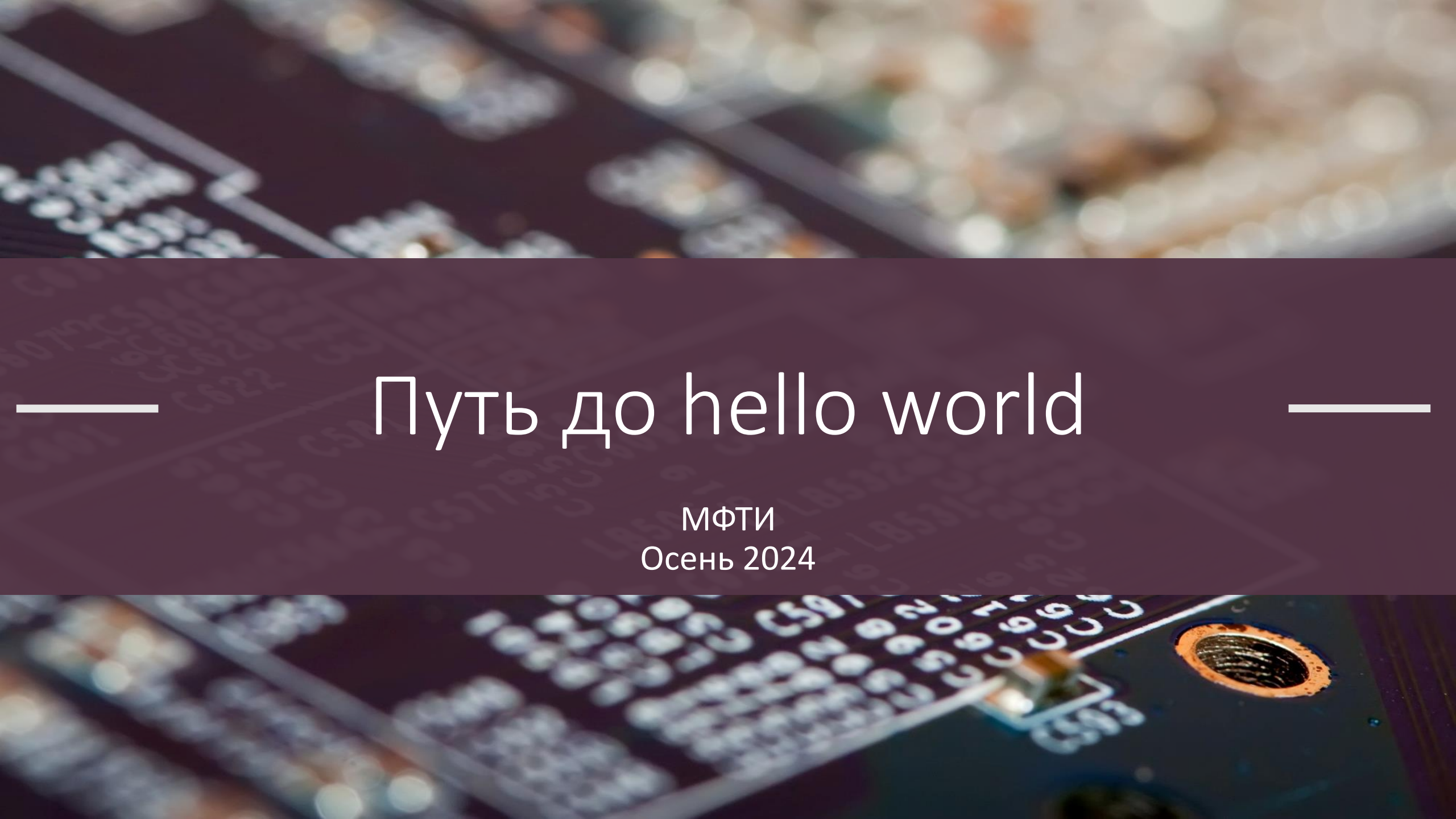


Путь до hello world

МФТИ
Осень 2024

Повторение

- Execution environment задает:
  - Как программа запускается (startup)
  - Как инициализируются статические данные
  - Как программа завершает выполнение

Hosted Execution Environment (Linux)

- Что происходит, когда вы запускаете программу в командной строке?

Hosted Execution Environment (Linux)

- Что происходит, когда вы запускаете программу в командной строке?
- Командная оболочка делает `fork` и в новом процессе запускает `exec`

Hosted Execution Environment (Linux)

- Что происходит, когда вы запускаете программу в командной строке?
- Командная оболочка делает `fork` и в новом процессе запускает `exec`
- `exec` запускает новый процесс и передает в него аргументы командной строки

Hosted Execution Environment (Linux)

- Что происходит, когда вы запускаете программу в командной строке?
- Командная оболочка делает `fork` и в новом процессе запускает `exec`
- `exec` запускает новый процесс и передает в него аргументы командной строк
- Что вы знаете о данном процессе?

Hosted Execution Environment (Linux)

- Что происходит, когда вы запускаете программу в командной строке?
- Командная оболочка делает `fork` и в новом процессе запускает `exec`
- `exec` запускает новый процесс и передает в него аргументы командной строки
- Что вы знаете о данном этапе?

Hosted Execution Environment (Linux): exec

- Что делает exec?

Hosted Execution Environment (Linux): exec

- Что делает `exec`?
  - Подготавливает память
  - Подготавливает регистры
  - Открывает `stdin`, `stdout`, `stderr`
  - *По сути подготавливает среду исполнения*

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Что именно?
  - Подготавливает регистры
    - Что именно?

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
  - Подготавливает регистры
    - Что именно?

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
  - Подготавливает регистры
    - Что именно?

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
    - Подготавливает стек и кучу
  - Подготавливает регистры
    - Что именно?

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
    - Подготавливает стек и кучу
  - Подготавливает регистры
    - Инициализирует указатель на стек

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
    - Подготавливает стек и кучу
  - Подготавливает регистры
    - Инициализирует указатель на стек
    - Почему он не подготавливает другие регистры (например указатель на глобальные данные)?

Hosted Execution Environment (Linux): exec

- exec подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
    - Подготавливает стек и кучу
  - Подготавливает регистры
    - Инициализирует указатель на стек
    - Почему он не подготавливает другие регистры (например указатель на глобальные данные)?
    - Потому что остальное может изменяться от бинарника к бинарнику

Hosted Execution Environment (Linux): exec

- `exec` подготавливает среду исполнения:
  - Подготавливает память
    - Загружает секции из исполняемого файла
    - Подгружает динамические библиотеки (программа, которая это делает, называется динамический линкер)
    - Подготавливает стек и кучу
  - Подготавливает регистры
    - Инициализирует указатель на стек
    - Почему он не подготавливает другие регистры (например указатель на глобальные данные)?
    - Потому что остальное может изменяться от бинарника к бинарнику
  - В конце происходит прыжок по адресу метки `_start`

На `_start`, внимание, `main`!

- Посмотрим, что происходит в `_start`
- Компиляция:

```
> riscv64-unknown-elf-gcc -march=rv32i hello.c -o hello.elf
```

- Дизассемблирование:

```
> riscv64-unknown-elf-objdump -S hello.elf > hello.linux.dis
```

На _start, внимание, main!

```
000100d8 <_start>:
   100d8:       00004197                auipc   gp,0x4
   100dc:       73818193                addi    gp,gp,1848 # 14810 <__global_pointer$>
   100e0:       d4c18513                addi    a0,gp,-692 # 1455c <__stdio_exit_handler>
   100e4:       09418613                addi    a2,gp,148 # 148a4 <__BSS_END__>
   100e8:       40a60633                sub     a2,a2,a0
   100ec:       00000593                li      a1,0
   100f0:       421000ef                jal     10d10 <memset>
   100f4:       00001517                auipc   a0,0x1
   100f8:       b9050513                addi    a0,a0,-1136 # 10c84 <__libc_fini_array>
   100fc:       0b8000ef                jal     101b4 <atexit>
   10100:       2f1000ef                jal     10bf0 <__libc_init_array>
   10104:       00012503                lw      a0,0(sp)
   10108:       00410593                addi    a1,sp,4
   1010c:       00000613                li      a2,0
   10110:       070000ef                jal     10180 <main>
   10114:       f81ff06f                j       10094 <exit>
```

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692
        addi    a2,gp,148
        sub     a2,a2,a0
        li      a1,0
        jal     10d10 <memset>
        auipc   a0,0x1
        addi    a0,a0,-1136
        jal     101b4 <atexit>
        jal     10bf0 <__libc_init_array>
        lw      a0,0(sp)
        addi    a1,sp,4
        li      a2,0
        jal     10180 <main>
        j       10094 <exit>
```

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692 # <--- Подготовка первого аргумента для memset (указатель)
        addi    a2,gp,148 # <------ Подготовка третьего аргумента
        sub     a2,a2,a0  # <------ для memset (размер)
        li      a1,0 # <----- Подготовка второго аргумента для memset (значение)
        jal     10d10 <memset> # <----- Вызов memset, зануляющий BSS
        auipc   a0,0x1
        addi    a0,a0,-1136
        jal     101b4 <atexit>
        jal     10bf0 <__libc_init_array>
        lw      a0,0(sp)
        addi    a1,sp,4
        li      a2,0
        jal     10180 <main>
        j       10094 <exit>
```

BSS – сегмент (неразрывная область памяти), в которой хранятся глобальные объекты не инициализированные в исходном коде. Они инициализируются нулевым значением.

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692 # <--- Подготовка первого аргумента для memset (указатель)
        addi    a2,gp,148 # <------ Подготовка третьего аргумента
        sub     a2,a2,a0  # <------ для memset (размер)
        li      a1,0 # <----- Подготовка второго аргумента для memset (значение)
        jal     10d10 <memset> # <----- Вызов memset, зануляющий BSS
        auipc   a0,0x1 # <------- Подготовка аргумента для вызова atexit, адрес
        addi    a0,a0,-1136 # <-- __libc_fini_array – деструктора глобальных объектов
        jal     101b4 <atexit> # <--- Регистрация __libc_fini_array в atexit
        jal     10bf0 <__libc_init_array>
        lw      a0,0(sp)
        addi    a1,sp,4
        li      a2,0
        jal     10180 <main>
        j       10094 <exit>
```

atexit – функция, регистрирующая функции, вызываемые при выходе из программы

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692 # <--- Подготовка первого аргумента для memset (указатель)
        addi    a2,gp,148 # <------ Подготовка третьего аргумента
        sub     a2,a2,a0  # <------ для memset (размер)
        li      a1,0 # <----- Подготовка второго аргумента для memset (значение)
        jal     10d10 <memset> # <----- Вызов memset, зануляющий BSS
        auipc   a0,0x1 # <------- Подготовка аргумента для вызова atexit, адрес
        addi    a0,a0,-1136 # <-- __libc_fini_array – деструктора глобальных объектов
        jal     101b4 <atexit> # <--- Регистрация __libc_fini_array в atexit
        jal     10bf0 <__libc_init_array> # <--- Вызов конструктора глобальных объектов
        lw      a0,0(sp)
        addi    a1,sp,4
        li      a2,0
        jal     10180 <main>
        j       10094 <exit>
```

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692 # <--- Подготовка первого аргумента для memset (указатель)
        addi    a2,gp,148 # <------ Подготовка третьего аргумента
        sub     a2,a2,a0  # <------ для memset (размер)
        li      a1,0 # <----- Подготовка второго аргумента для memset (значение)
        jal     10d10 <memset> # <----- Вызов memset, зануляющий BSS
        auipc   a0,0x1 # <------- Подготовка аргумента для вызова atexit, адрес
        addi    a0,a0,-1136 # <-- __libc_fini_array – деструктора глобальных объектов
        jal     101b4 <atexit> # <--- Регистрация __libc_fini_array в atexit
        jal     10bf0 <__libc_init_array> # <--- Вызов конструктора глобальных объектов
        lw      a0,0(sp) # <--- argc
        addi    a1,sp,4  # <--- argv
        li      a2,0     # <--- envp
        jal     10180 <main> # <--- Вызов main
        j       10094 <exit>
```

Стек – по сути единственная вещь, которую нужно подготовить, за исключением загрузки секций из elf-файла

На _start, внимание, main!

```
_start:
        auipc   gp,0x4     # <-----  Инициализация
        addi    gp,gp,1848 # <----- global pointer (gp)
        addi    a0,gp,-692 # <--- Подготовка первого аргумента для memset (указатель)
        addi    a2,gp,148 # <------ Подготовка третьего аргумента
        sub     a2,a2,a0  # <------ для memset (размер)
        li      a1,0 # <----- Подготовка второго аргумента для memset (значение)
        jal     10d10 <memset> # <----- Вызов memset, зануляющий BSS
        auipc   a0,0x1 # <------- Подготовка аргумента для вызова atexit, адрес
        addi    a0,a0,-1136 # <-- __libc_fini_array – деструктора глобальных объектов
        jal     101b4 <atexit> # <--- Регистрация __libc_fini_array в atexit
        jal     10bf0 <__libc_init_array> # <--- Вызов конструктора глобальных объектов
        lw      a0,0(sp) # <--- argc
        addi    a1,sp,4  # <--- argv
        li      a2,0     # <--- envp
        jal     10180 <main> # <--- Вызов main
        j       10094 <exit> # <- Просто прыжок в exit, потому что оттуда не вернемся
```

Freestanding Environment

- Примером Freestanding Environment является *Baremetal окружение*
- **Baremetal** (*Bare metal – голый металл*) – окружение наиболее близкое к CPU, без дополнительных программных абстраций в виде ОС
- Для работы в baremetal окружении используют `bsp` (Board Support Package) или `hal` (Hardware Abstraction Layer) для настройки аппаратуры перед попаданием в `main`

Собираем baremetal

- Сборка:
  - <пример сборки sample app’a с hello world’ом>
- Дизассемблирование:

```
> riscv64-unknown-elf-objdump -S hello.bsp.elf > hello.bsp.dis
```

- Перед тем как смотреть дизассемблер:
  - Кто загрузит elf в память и прыгнет на _start?

Собираем baremetal

- Сборка:
  - <пример сборки sample app’a с hello world’ом>
- Дизассемблирование:

```
> riscv64-unknown-elf-objdump -S hello.bsp.elf > hello.bsp.dis
```

- Перед тем как смотреть дизассемблер:
  - Кто загрузит elf в память и прыгнет на `_start`?
  - elf-файл загружает внешний загрузчик в память (например, запись во флэш память на плате)
  - Бинарник собирается так, чтобы точка входа находилась по специальному адресу, с которого процессор начинает исполнение программы сразу после включения (иногда это не `0x0`)

`_start` нам больше не нужен

```
Disassembly of section .text.startup:f0000000 <_bsp_start>:

f0000000:    30001073            csrw   mstatus,zero
f0000004:    00000093            li     ra,0
f0000008:    00000113            li     sp,0
f000000c:    00000193            li     gp,0
f0000010:    00000213            li     tp,0
f0000014:    00000293            li     t0,0
f0000018:    00000313            li     t1,0
f000001c:    00000393            li     t2,0
f0000020:    00000413            li     s0,0
f0000024:    00000493            li     s1,0
f0000028:    00000513            li     a0,0
f000002c:    00000593            li     a1,0
f0000030:    00000613            li     a2,0
```

Так как загрузчик теперь ориентируется не на название символа, а на адрес начала исполнения, то метку можно называть как угодно, главное – расположить по нужному адресу

В данном случае такая метка – `_bsp_start`

Новые инструкции в rv32i?!

```
Disassembly of section .text.startup:f0000000 <_bsp_start>:

f0000000:    30001073                csrw   mstatus,zero # Unknown instruction
f0000004:    00000093                li     ra,0
f0000008:    00000113                li     sp,0
f000000c:    00000193                li     gp,0
f0000010:    00000213                li     tp,0
f0000014:    00000293                li     t0,0
f0000018:    00000313                li     t1,0
f000001c:    00000393                li     t2,0
f0000020:    00000413                li     s0,0
f0000024:    00000493                li     s1,0
f0000028:    00000513                li     a0,0
f000002c:    00000593                li     a1,0
f0000030:    00000613                li     a2,0
```

Базовый набор инструкций RV32I

Jumps & Calls
JAL
JALR
BEQ
BNE
BLT
BGE
BLTU
BGEU

Loads & Stores
LB
LH
LW
LBU
LHU
SB
SH
SW

Arithmetics	
ADD	ADDI
SUB	
OR	ORI
XOR	XORI
AND	ANDI
SRL	SRLI
SLL	SLLI
SRA	SRAI

Special
FENCE
ECALL
EBREAK

Upper immediate
LUI
AUIPC

Data flow			
SLT	SLTU	SLTI	SLTIU

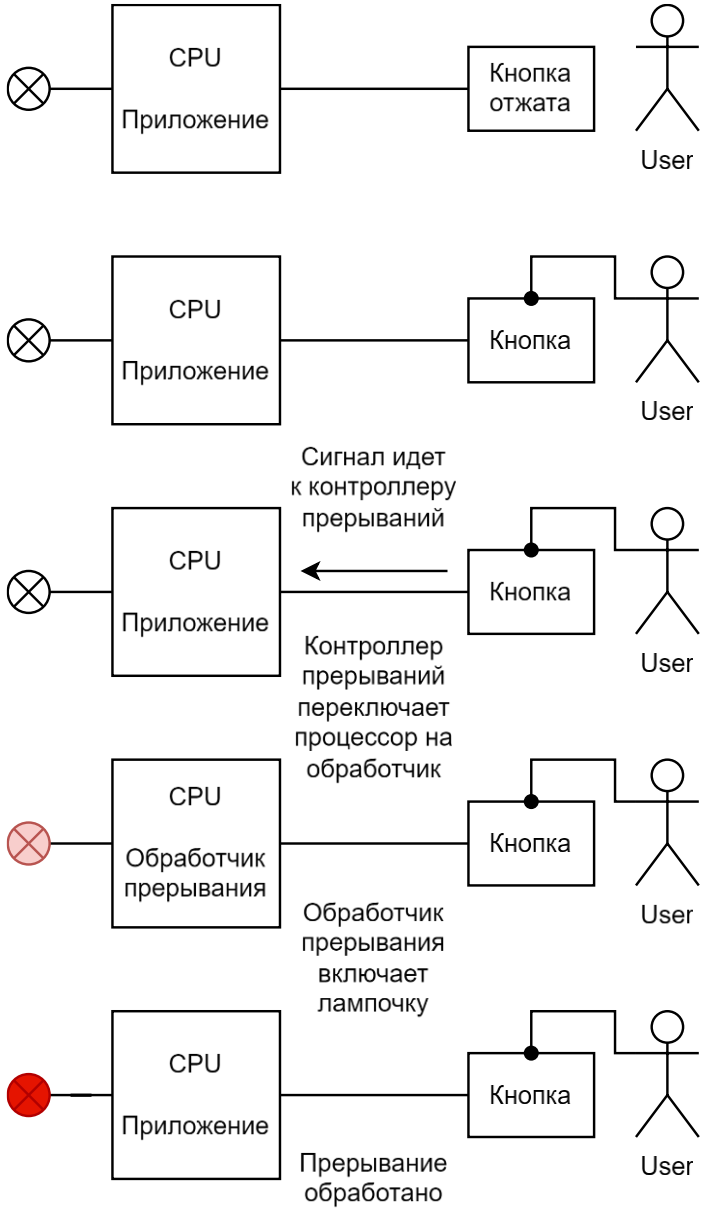
Должен ли RV32I процессор понимать еще какие-либо команды?

Прерывания

- **Прерывание** – запрос процессору на *прерывание* исполнения текущего потока инструкций для обработки произошедшего события
- Причина прерывания может быть как внешней (от периферии), так и внутренней (событие в процессоре или принудительно вызвано программой)
- При возникновении прерывания контроллер прерываний передает управление соответствующему *обработчику прерывания*, который обязан вернуть управление после окончания обработки

Прерывания: пример

- **Обработчик прерывания** – функция, вызываемая в результате прерывания
- «Ждет» и делает переход к нужному обработчику модуль процессора, называемый **обработчиком прерываний**
- По ABI после возврата из обработчика значение регистров должно быть тем же, что и до возникновения прерывания



Прерывания в RISC-V

- В RISC-V управление контроллером прерываний происходит с помощью **Control and Status Registers** (**CSR**)
- Операционная система работает с прерываниями за вас
- Если же вы собираете baremetal приложение, то скорее всего вы самостоятельно работаете с прерываниями (и другими низкоуровневыми механизмами)
- Работа с CSR регистрами описана в расширении `Zicsr`
- Подробнее с CSR мы будем знакомиться, когда будем говорить об ОС в RISC-V

Реальный базовый набор инструкций RV32I + Zicsr

Jumps & Calls
JAL
JALR
BEQ
BNE
BLT
BGE
BLTU
BGEU

Loads & Stores
LB
LH
LW
LBU
LHU
SB
SH
SW

Arithmetics	
ADD	ADDI
SUB	
OR	ORI
XOR	XORI
AND	ANDI
SRL	SRLI
SLL	SLLI
SRA	SRAI

Special
FENCE
ECALL
EBREAK

Upper immediate
LUI
AUIPC

Data flow			
SLT	SLTU	SLTI	SLTIU

Control and status register					
CSRRW	CSRRS	CSRRC	CSRRWI	CSRRSI	CSRRCI

Executable and Linkable Format

- «Эльфийский» формат исполняемых бинарных файлов (ELF) состоит из двух основных частей:
  - Заголовок
  - Данные
- Заголовок ELF является обязательным и нужен для того, чтобы данные корректно интерпретировались при линковке и исполнении
- Данные в ELF файле состоят из:
  - Таблицы заголовков сегментов
  - Таблицы заголовков секций
  - Содержимого сегментов и секций

Задание: пишем симулятор

- Реализуйте загрузчик elf файлов в вашем симуляторе
- Для этого необходимо:
  1. Инициализировать память, для этого необходимо загрузить в память все сегменты
  2. Инициализировать стек и указатель на него
  3. Начать исполнение с адреса символа `_start`

To be continued …

На следующем занятии

- Узнаем как устроен стек загрузки Linux в RISC-V
- Узнаем что такое привилегированное исполнение и как оно помогает
- Узнаем про разные уровни привилегированности в RISC-V