Трудно быть процессором: аппаратная реализация Java и история picoJava-II

Илья Гаврилин, Синтакор

mailto: ilya.gavrilin@syntacore.com

Syntacore™
RISC-V cores and tools

О себе

- Занимаюсь Managed-runtimes в Синтакор
- Обучаюсь в магистратуре МФТИ ФРКТ
- Соавтор порта компилятора Maglev(Chromium) на RISC-V
- Увлекаюсь разработкой аппаратуры

Syntacore™
RISC-V cores and tools

Проблема производительности

- Мы привыкли, что Java работает на десктопе или в облаке
- Что делать если железо слабое?

- JIT потребляет ресурсов больше, чем у нас есть
- Платформа может запрещать генерировать исполняемый код

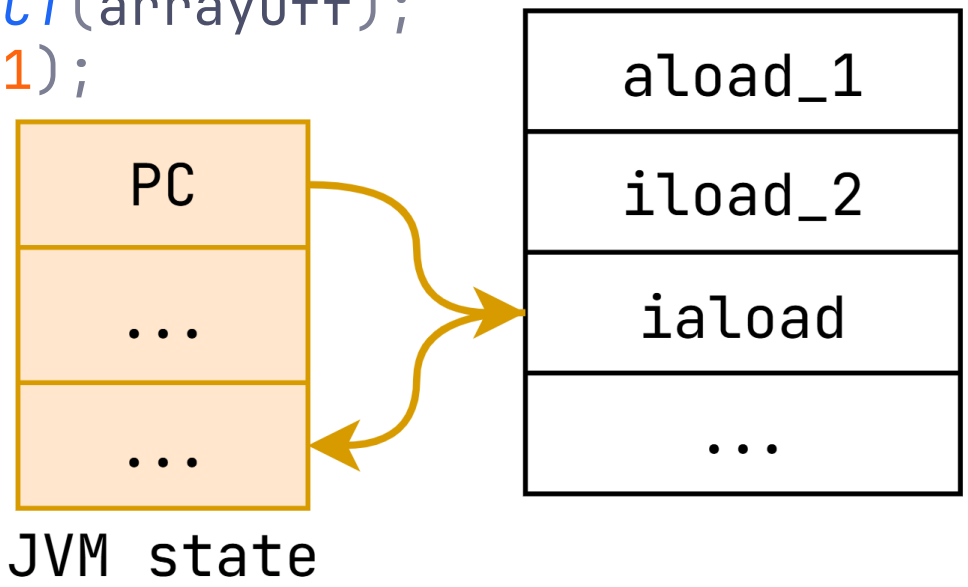
А помните, как было?

```
void Interpreter::main_loop(interpreterState istate) {
  while (!stop) {
  switch (opcode)
  case opc_iaload:
    arrayOop arrObj = (arrayOop)STACK_OBJECT(arrayOff);
    jint     index = STACK_INT(arrayOff + 1);
    ARRAY_INDEX_CHECK(arrObj, index);
    SET_STACK_INT(
      GET_HEAP_INT(arrObj, index)
    );
    UPDATE_PC_AND_TOS_AND_CONTINUE();
  ...
  }
}
```

- Для исполнения одного байткода в худшем случае придётся пройти весь switch
- Реализации интерпретатора обновляют состояние JVM каждый шаг

А помните, как было?

```
void Interpreter::main_loop(interpreterState istate) {
  while (!stop) {
  switch (opcode)
  case opc_iaload:
    arrayOop arrObj = (arrayOop)STACK_OBJECT(arrayOff);
    jint     index = STACK_INT(arrayOff + 1);
    ARRAY_INDEX_CHECK(arrObj, index);
    SET_STACK_INT(
      GET_HEAP_INT(arrObj, index)
    );
    UPDATE_PC_AND_TOS_AND_CONTINUE();
  ...
  }
}
```



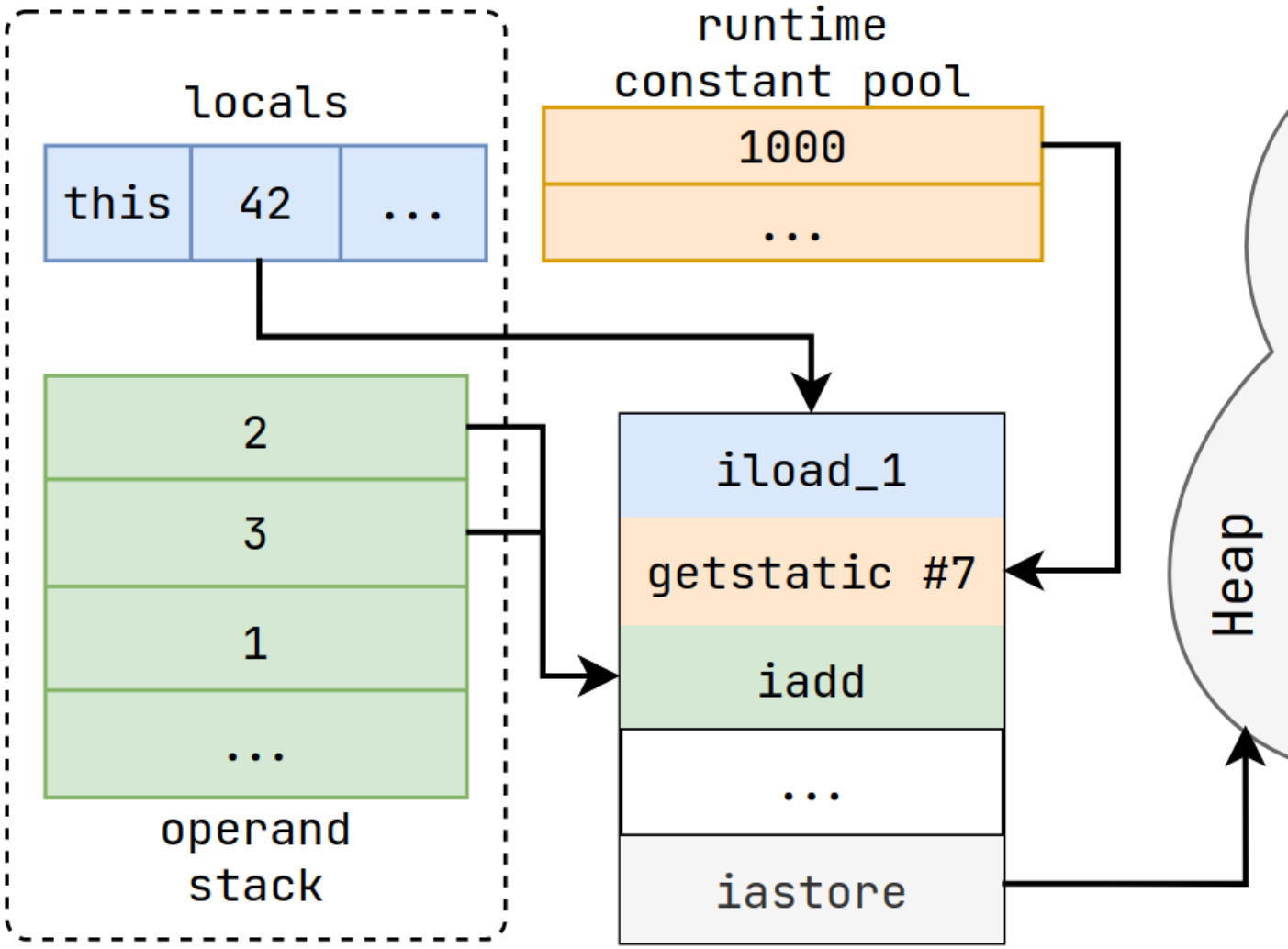
- Для исполнения одного байткода в худшем случае придётся пройти весь switch
- Реализации интерпретатора обновляют состояние JVM каждый шаг

И память не забудьте

- Java байткод не так прост

Run-Time Data Areas:

- PC Register
- JVM Stacks
  - Locals
  - Operand Stack
- Heap
- Method Area
  - Run-Time Constant Pool
- Native Stack (optional)



Java is a platform

Java isn't platform independent; Java is a platform.
- Bjarne Stroustrup

Link: https://www.stroustrup.com/bs_faq.html

Java is a platform

Java isn't platform independent; Java is a platform.
- Bjarne Stroustrup

- JVM bytecode чем-то напоминает ассемблер некой архитектуры
- Описание этого набора команд – JVMS §6
- Но этот ассемблер включает runtime-контракт:
  - linking
  - exceptions
  - monitors

Link: https://www.stroustrup.com/bs_faq.html

6. The Java Virtual Machine Instruction Set

6.1. Assumptions: The Meaning of "Must"
6.2. Reserved Opcodes
6.3. Virtual Machine Errors
6.4. Format of Instruction Descriptions
6.5. Instructions

JVM Specification 25

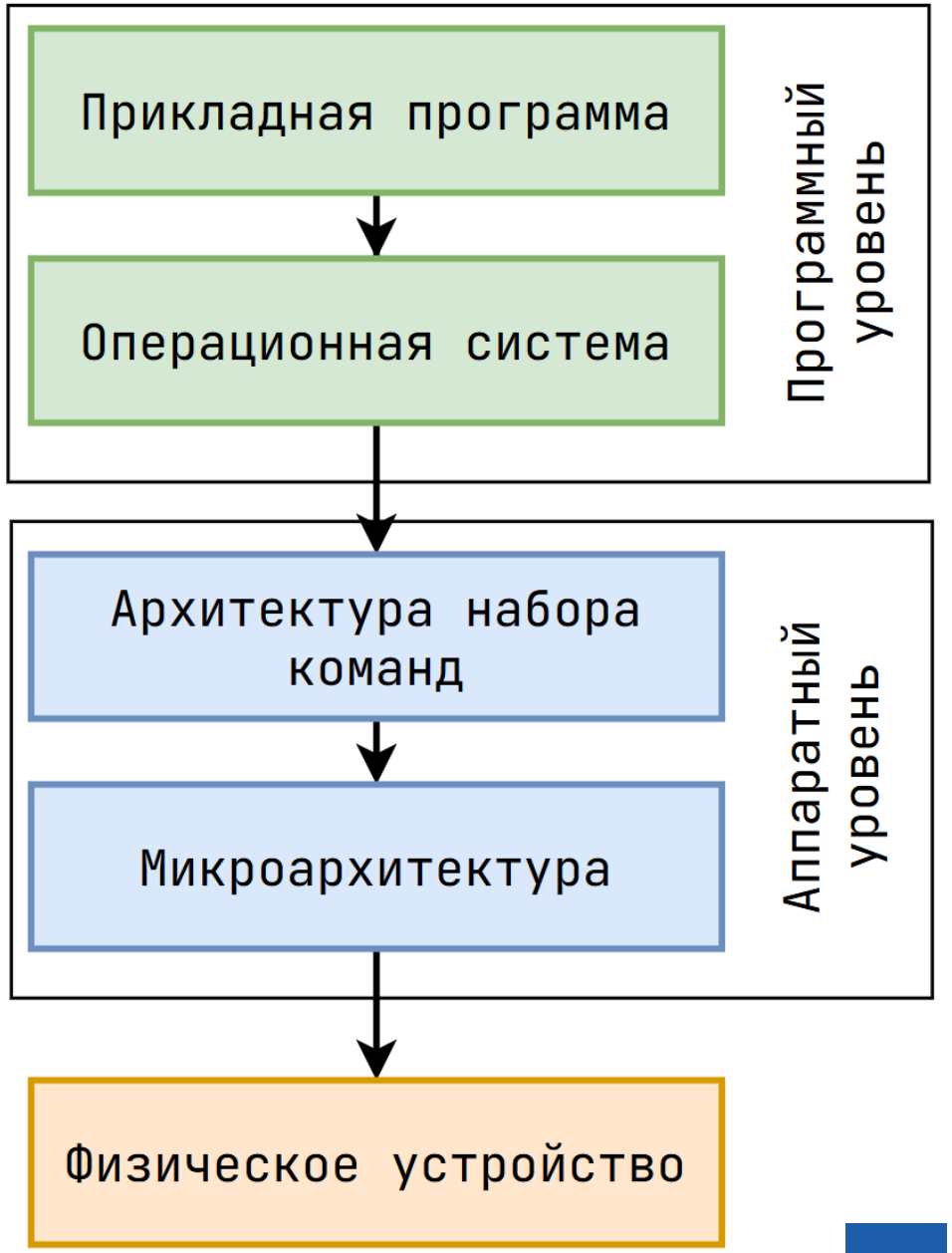
Что такое ISA?

- Архитектура набора команд (ISA) - абстрактная модель, определяющая интерфейс взаимодействия с аппаратурой

ISA определяет:

- Семантику инструкций и исключений
- Набор наблюдаемых состояний
- Правила перехода управления

[JVM Specification 25]



Сравним архитектуры

Попробуем сравнить популярные действия:

- Работа с переменными
- Доступ к массивам
- Вызов функций

```
class Example {
  int foo(int i) { return i;}
  int bar(int[] a, int i) {
    return foo(a[i]);
  }
}
```

Сравним архитектуры: RISC-V

Попробуем сравнить популярные действия:

- Работа с переменными
- Доступ к массивам
- Вызов функций

```
class Example {
  int foo(int i) { return i;}
  int bar(int[] a, int i) {
    return foo(a[i]);
  }
}
```

RISC-V:

- Регистры для хранения переменных
- Массив становится адресом в памяти
- Функция представляет собой метку

```
# a0 = int[] a, a1 = i
slli t0, a1, 2    # t0 = i * 4
add  t0, a0, t0   # t0 = &(a) + t0
lw   a0, 0(t0)    # a2 = &(t0)
call foo
```

Сравним архитектуры: JVM

Попробуем сравнить популярные действия:

- Работа с переменными
- Доступ к массивам
- Вызов функций

JVM:

- Стек + массив локальных переменных
- Массив - специальная структура
- Функция - вызов с разрешением

```
class Example {
  int foo(int i) { return i;}
  int bar(int[] a, int i) {
    return foo(a[i]);
  }
}
```

```
aload_0 # this
aload_1 # int[] a
iload_2 # i
iaload  # a[i]
invokevirtual #7 # foo:(I)I
```

Сравним архитектуры

RISC-V:

- Регистры для хранения переменных
- Массив становится адресом в памяти
- Функция представляет собой метку

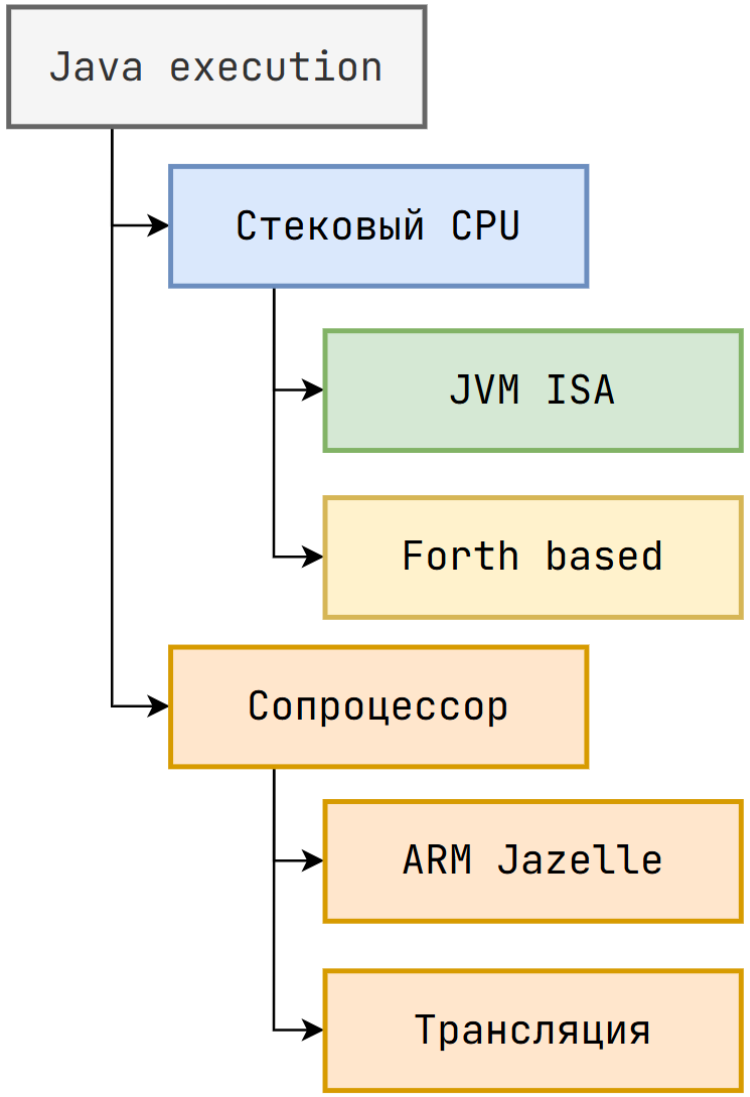
```
# a0 = int[] a, a1 = i
slli t0, a1, 2   # t0 = i * 4
add  t0, a0, t0  # t0 = &(a) + t0
lw   a0, 0(t0)   # a2 = &(t0)
call foo
```

JVM:

- Стек + массив локальных переменных
- Массив - специальная структура
- Функция - вызов с разрешением

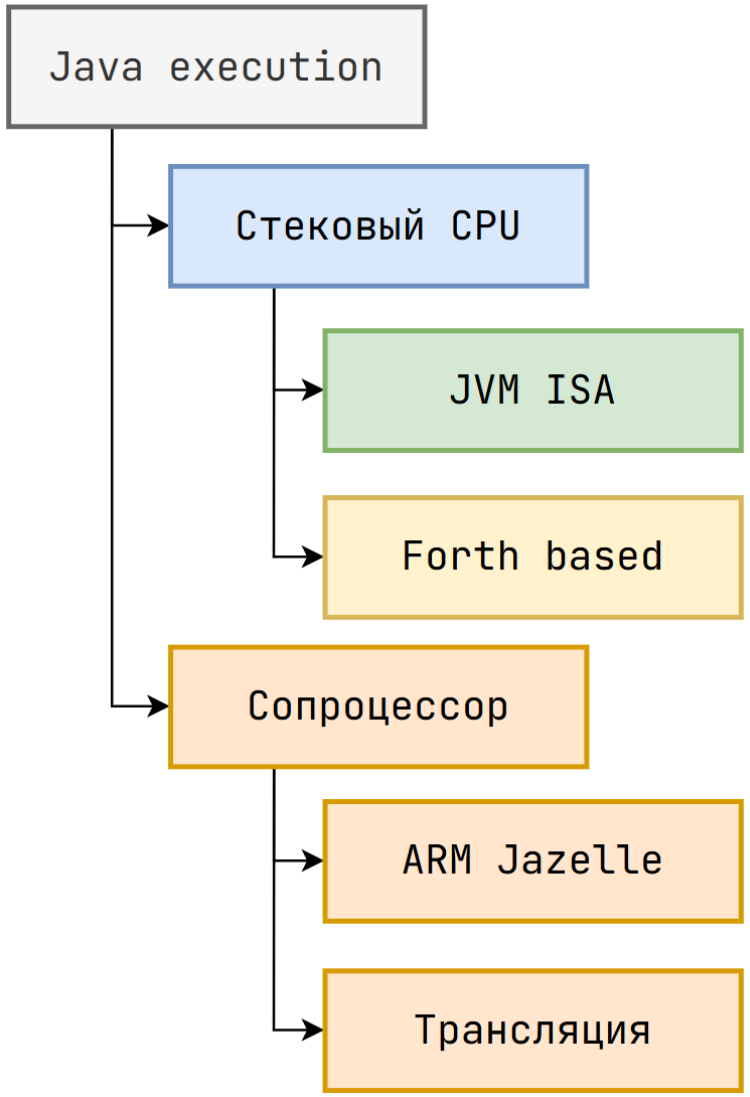
```
aload_0 # this
aload_1 # int[] a
iload_2 # i
iaload  # a[i]
invokevirtual #7 # foo:(I)I
```

Золотая реализация - Нет, "железная"



- Существует много попыток исполнять Java-байткод в железе
  - *Forth* – один из первых стековых языков программирования
  - *ARM Jazelle* –расширение ISA ARM, позволяющее передавать Java-байткод в соответствующий сопроцессор

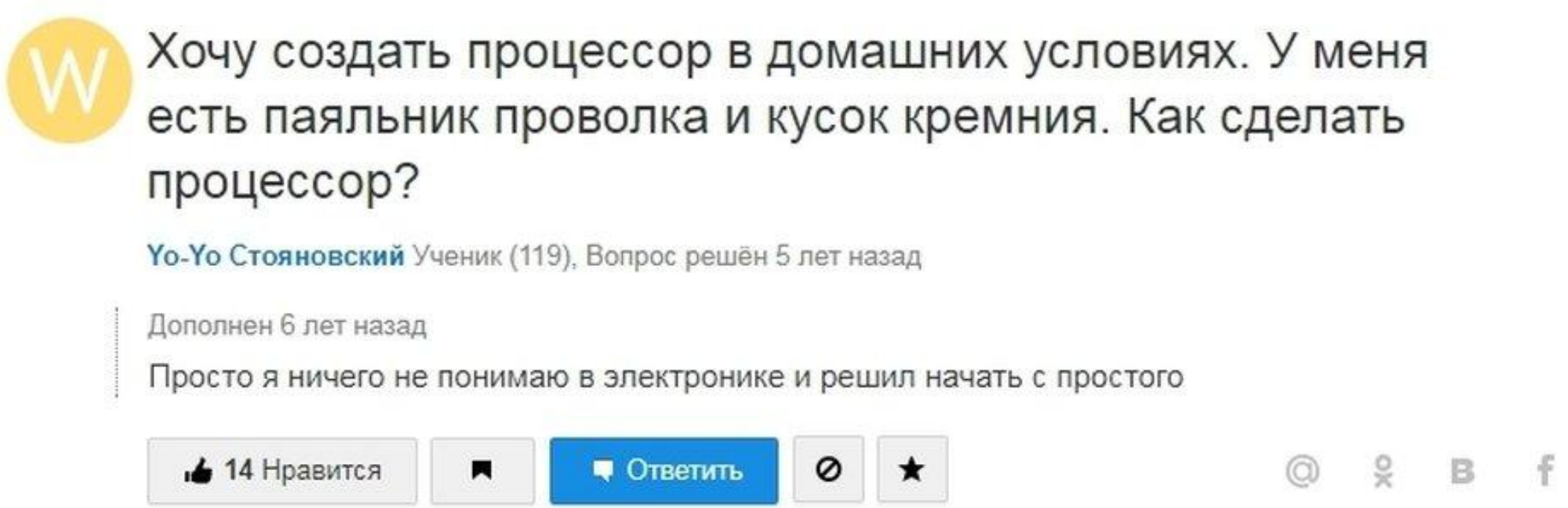
Золотая реализация - Нет, "железная"



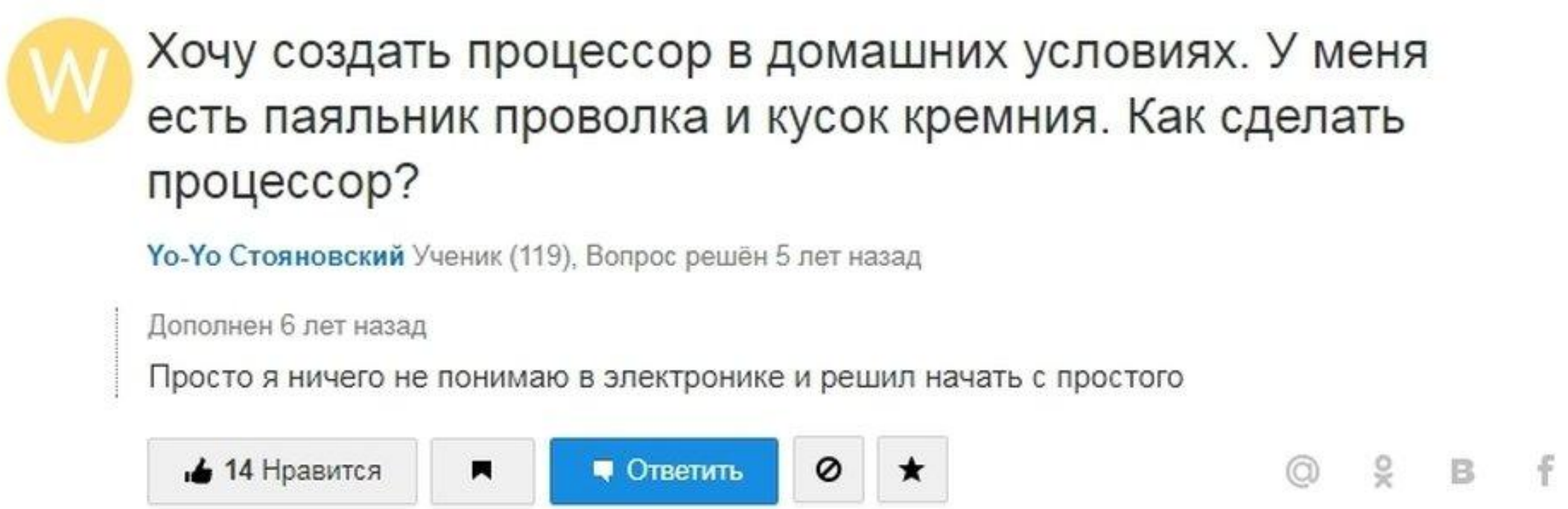
- Существует много попыток исполнять Java-байткод в железе
  - Forth – один из первых стековых языков программирования
  - ARM Jazelle –расширение ISA ARM, позволяющее передавать Java-байткод в соответствующий сопроцессор

- Сегодня посмотрим на вариант стекового процессора, который напрямую исполняет байткод
- **PicoJava-II**: https://github.com/ekiwi/picojava2-archive
- Исполняет **Java 1.1** (classfile ver. 45.3)

Как сделать процессор (в домашних условиях)

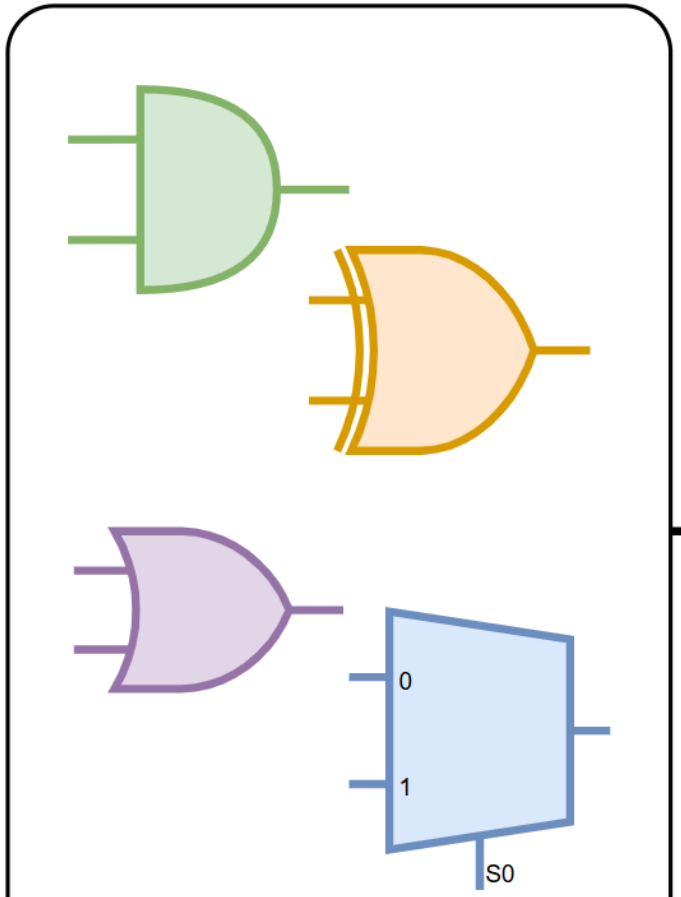


Как сделать процессор (в домашних условиях)



- Зачастую физические цифровые устройства описываются на специальных языках разработки аппаратуры - HDL
- Примеры популярных HDL: Verilog, SystemVerilog, VHDL

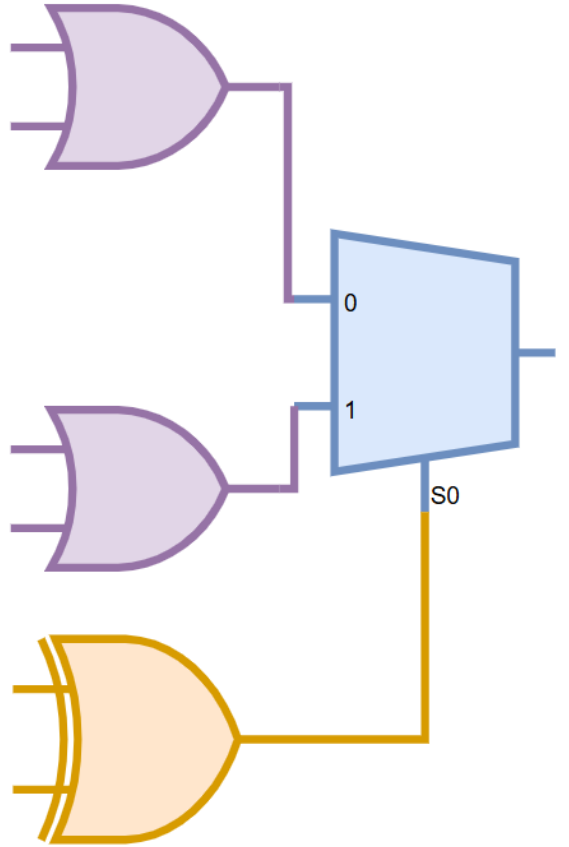
А что мы создаём?



Библиотека

```
val io = IO(
new Bundle {
val a,b,c,d,e,f =
    Input(Bool())
val out =
    Output(Bool())
})
val in0 = i.a | i.b
val in1 = i.c | i.d
val sel = i.e ^ i.f
o.out :=
  Mux(sel, in1, in0)
```

Описание схемы



Логическая схема

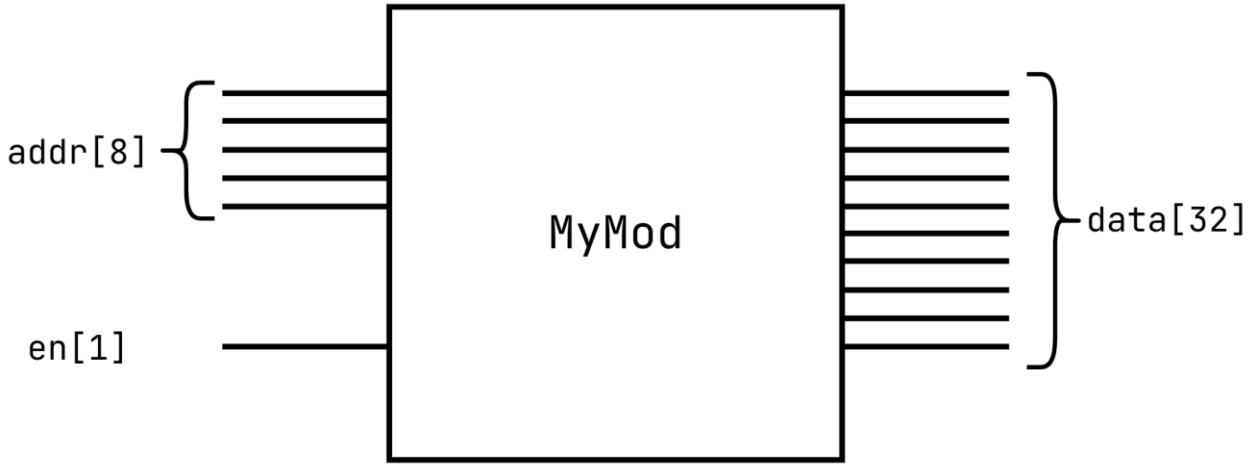
Привычный язык общения

- Сейчас существует большое количество языков и фреймворков для описания аппаратуры
- Chisel – библиотека + DSL на основе Scala

```
class IOPorts extends Bundle { // Интерфейс блока
  val addr  = Input(UInt(8.W))    // 8-битный вход (провода)
  val data  = Output(UInt(32.W))  // 32-битный выход
  val en    = Input(Bool())       // управляющий сигнал
}
class MyMod extends Module {   // Аппаратный блок
  val io = IO(new IOPorts)        // объявление портов блока
  io.data := io.addr              // связывание проводов
  ...
```

Железная абстракция

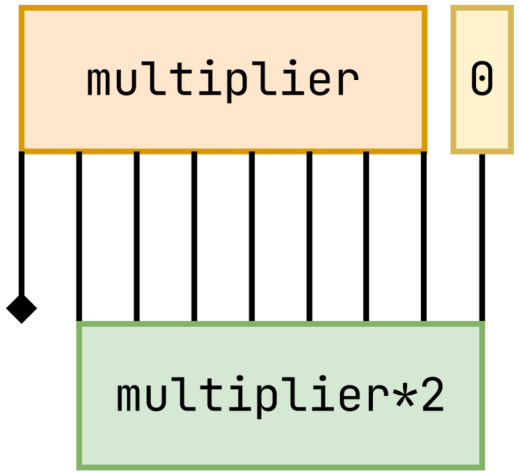
```
class IOPorts extends Bundle { // Интерфейс блока
  val addr  = Input(UInt(8.W))     // 8-битный вход (провода)
  val data  = Output(UInt(32.W))   // 32-битный выход
  val en    = Input(Bool())        // управляющий сигнал
}
class MyMod extends Module {    // Аппаратный блок
  val io = IO(new IOPorts)         // объявление портов блока
  io.data := io.addr               // связывание проводов
  ...
```

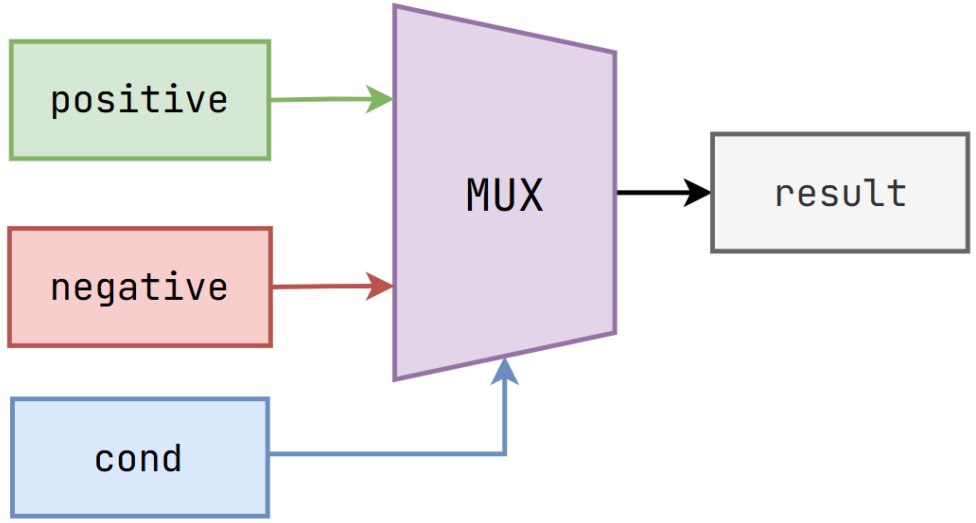


- Основная абстракция – Модуль
- Функции можно использовать для разделения логики
- В некоторых модулях можем описывать соединение модулей

Иная плоскость мысли

- Не все привычные операции легко реализовать в аппаратуре
- Умножение двух произвольных чисел – потребует много логических вентилей
- Некоторые операции, напротив, сделать крайне легко






```
multiplier * 2 = multiplier << 1
```

```
result = cond ? positive : negative
```

Попробуем что-то написать

Software way:

```
uint32_t ishl(uint32_t val,
              uint32_t shift) {
  assert(shift < 32u);
  // ishl(x, n) = x * 2^n
  uint32_t base = 2;
  while (shift > 0) {
    if (shift % 2 == 1) val *= base;
    base = base * base;
    shift = shift / 2;
  }
  return val;
}
```

- Реализуем операцию побитового левого сдвига для integer
- Сдвиг более чем на 31 не имеет смысла

Попробуем что-то написать

Software way:

```
uint32_t ishl(uint32_t val,
              uint32_t shift) {
  assert(shift < 32u);
  // ishl(x, n) = x * 2^n
  uint32_t base = 2;
  while (shift > 0) {
    if (shift % 2 == 1) val *= base;
    base = base * base;
    shift = shift / 2;
  }
  return val;
}
```

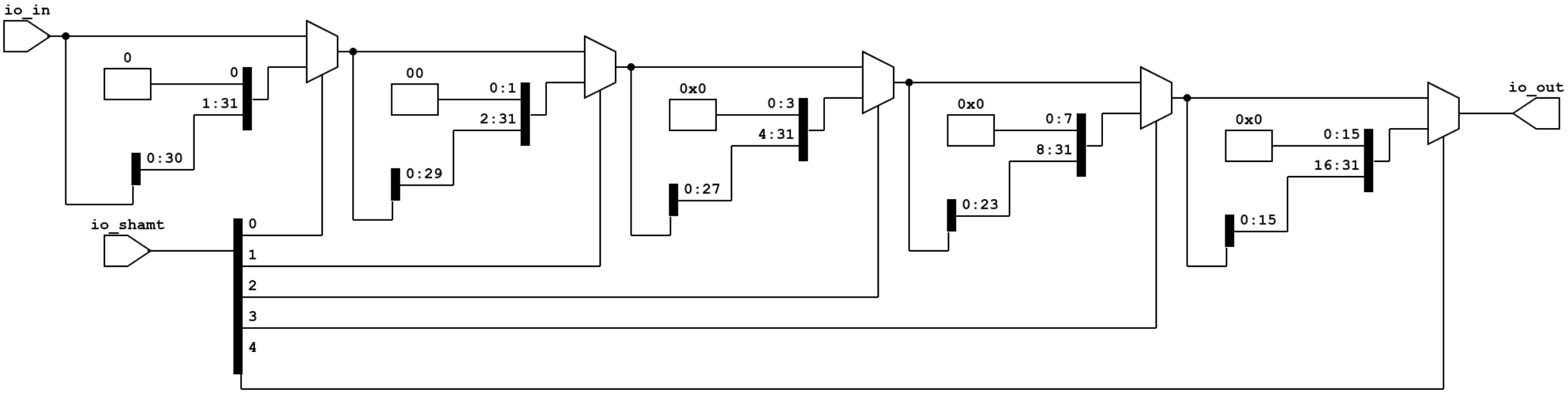
Hardware way*:

```
uint32_t ishl(uint32_t val,
              uint32_t shift) {
  assert(shift < 32u);
  // shift = {1 + 2 + 4 + 8 + 16}
  if (shift & 0b00001) val *= 2^1;
  if (shift & 0b00010) val *= 2^2;
  if (shift & 0b00100) val *= 2^4;
  if (shift & 0b01000) val *= 2^8;
  if (shift & 0b10000) val *= 2^16;
  return val;
}
```

*: тривиальные else блоки намеренно опущены

- Подходы при написании кода будут несколько различны
- Разработчику, через код, нужно уметь видеть логические элементы

Логическая схема

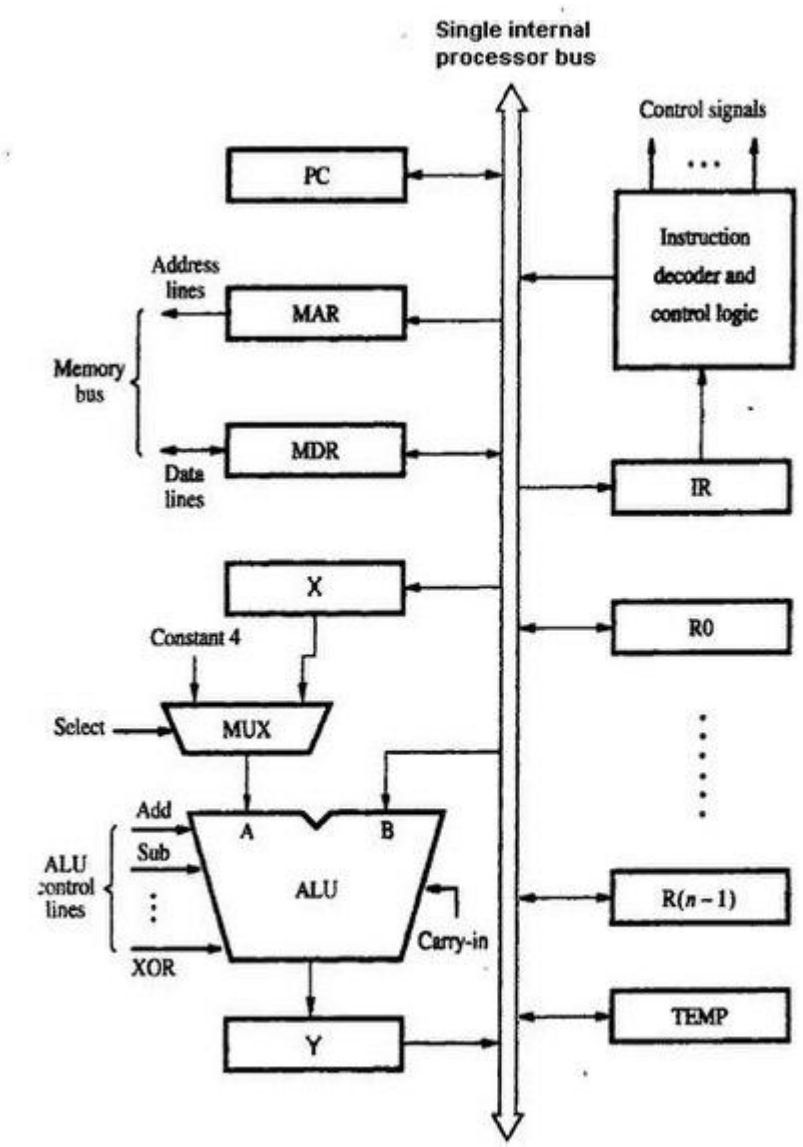


- Подходы при написании кода будут несколько различны
- Разработчику, через код, нужно уметь видеть логические элементы

Хорошего пути!

- Вычислители процессора образуют **Datapath**
- Вычислители оперируют данными из **регистров** или шин данных
- **Регистр** – поле заданной длины, быстро доступное при работе
- В русской литературе: **операционный автомат**

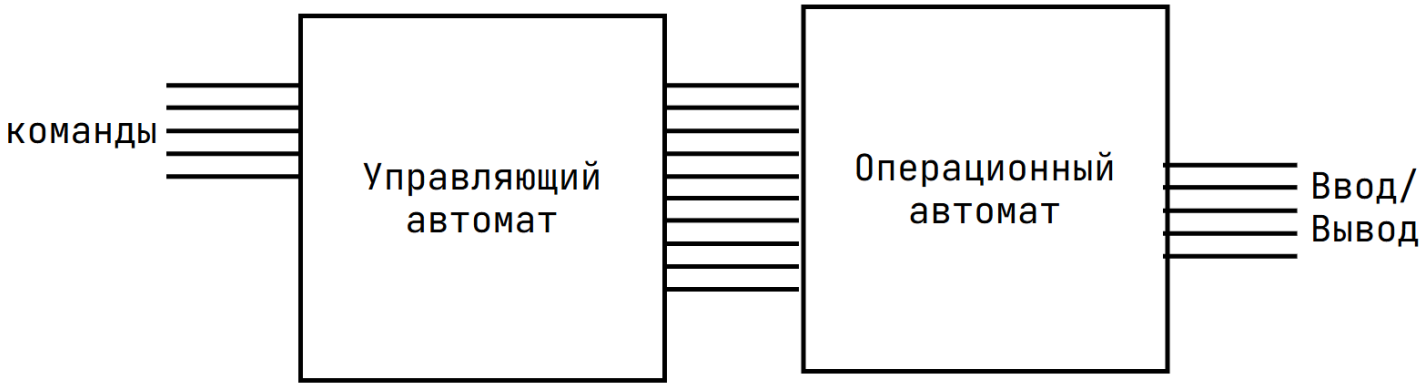
- Но операционный автомат бесполезен без управления

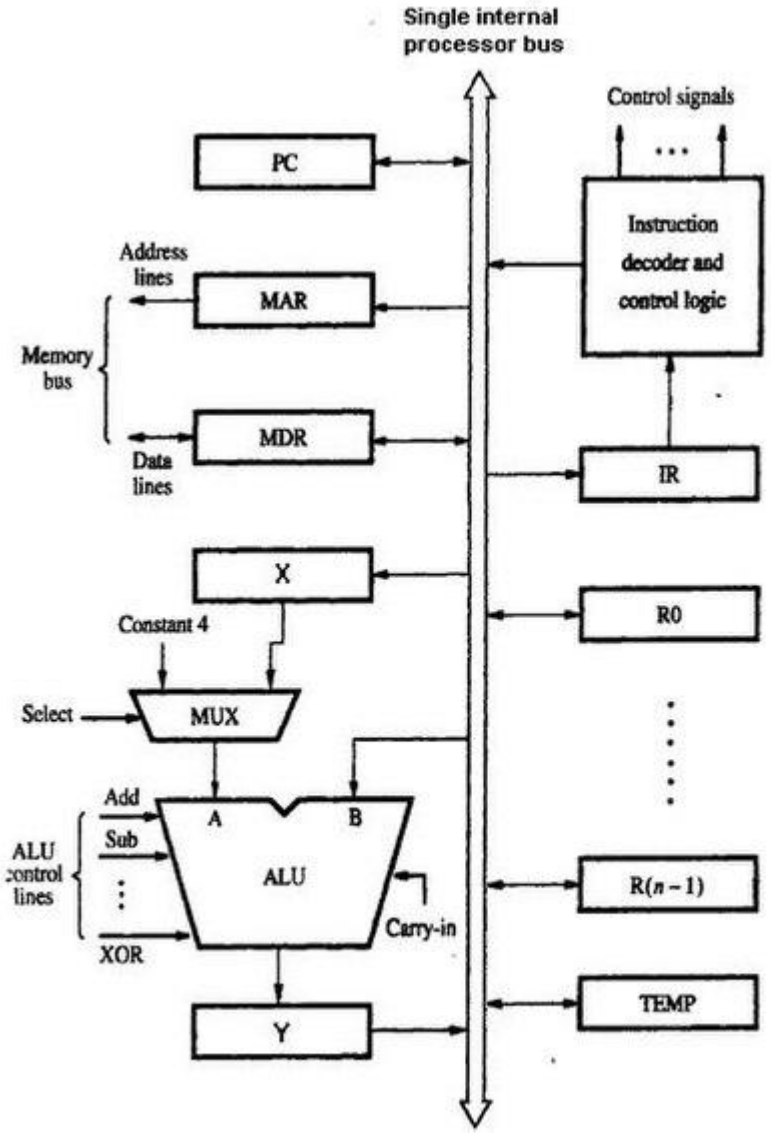


Single Bus Organization

Хорошего пути!

- Логику управление определяет **control unit**
- Дешифровывает команду и на основе **микрокода** даёт команды на исполнение
- **Микрокод** – программа реализующая инструкции процессора
- В русской литературе: **управляющий автомат**





Single Bus Organization

Код, но очень маленький

```
RS1 ← stack.pop() // arrayref
RS2 ← stack.pop() // index
if RS1 == 0 raise NullPointer
AREG0 ← RS1 + 4   // arr size
```

```
if RS2 < 0
  raise OutOfBounds
```

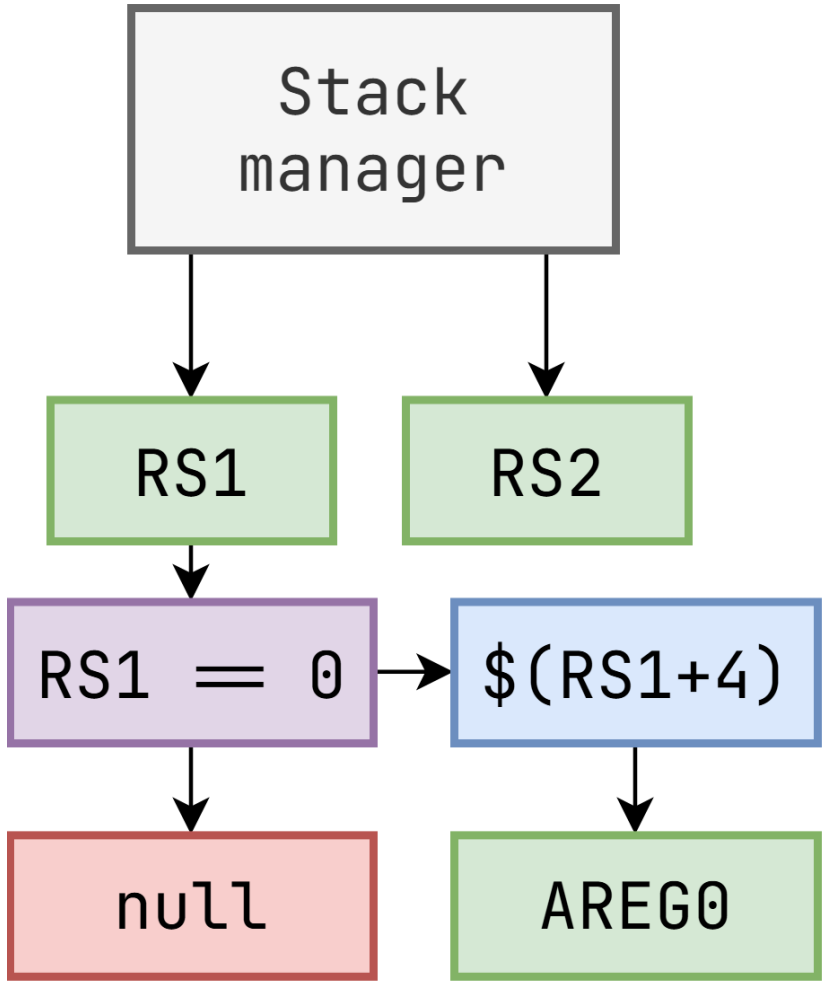
```
LEN ← MEM[AREG0]
```

```
OFFSET ← RS2 << 2
REG2   ← AREG0 + OFFSET
```

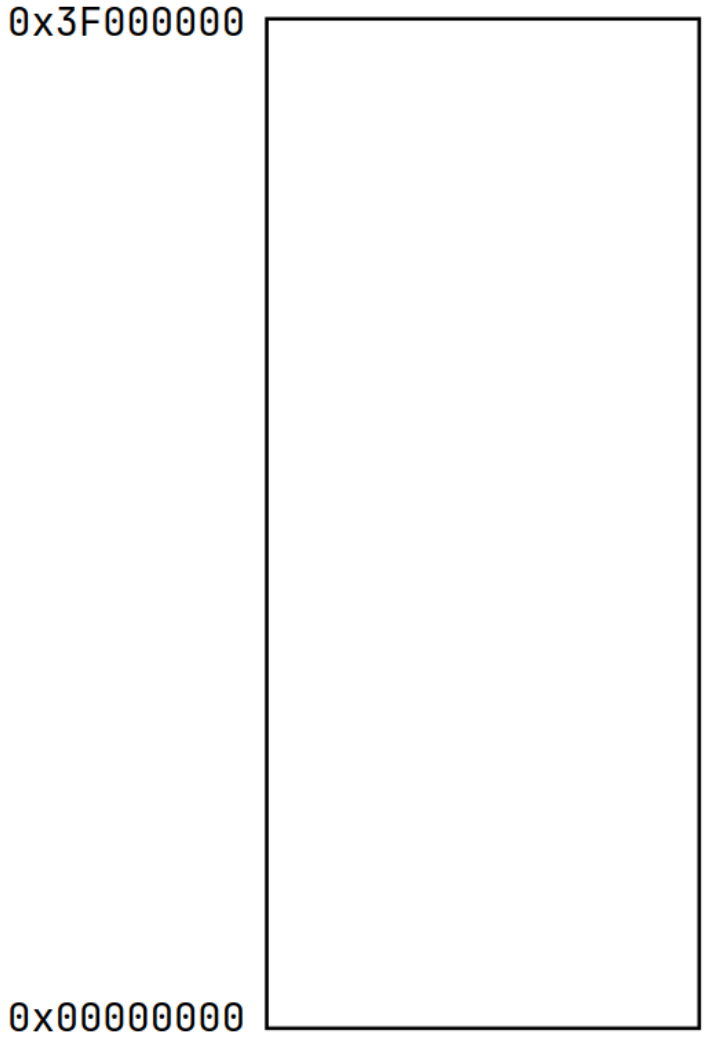
```
if RS2 ≥ LEN
    raise OutOfBounds
```

```
VALUE ← MEM[REG2]
stack.push(VALUE)
```

`iaload [arrayref, index] => [value]`

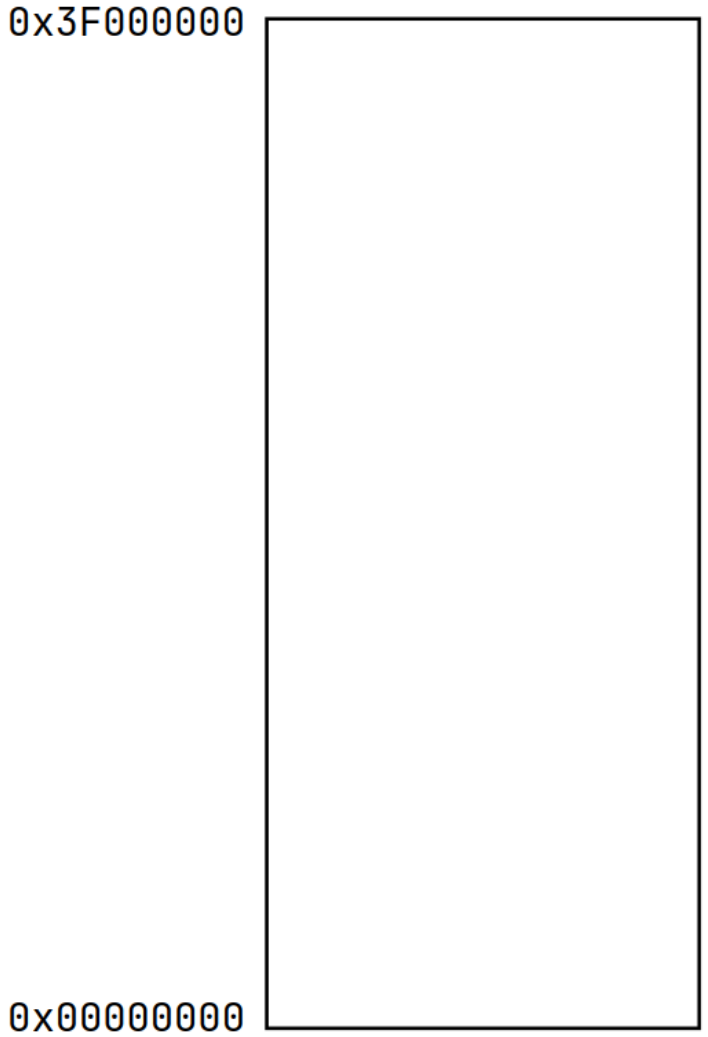


Сферический конь в вакууме



- PicoJava-II предоставляет нам лишь адресное пространство, адресуемое 30-битными числами (1GB)
  - Нет виртуализации
  - Непрерывное адресное пространство

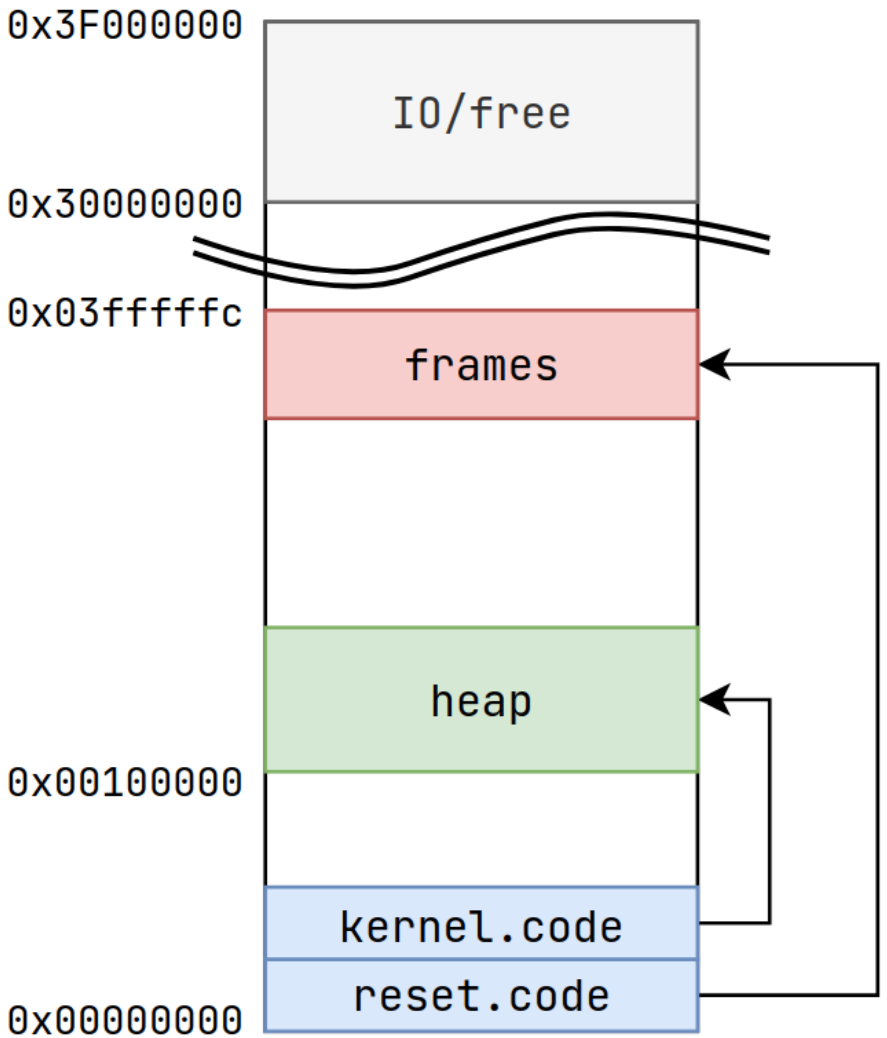
Сферический конь в вакууме



- PicoJava-II предоставляет нам лишь адресное пространство, адресуемое 30-битными числами (1GB)
  - Нет виртуализации
  - Непрерывное адресное пространство

- Аппаратура полагается на память:
  - Стек должен быть расположен по конкретному адресу
  - Объекты должны иметь в памяти определённую структуру
  - Необходима программа, которую будем исполнять

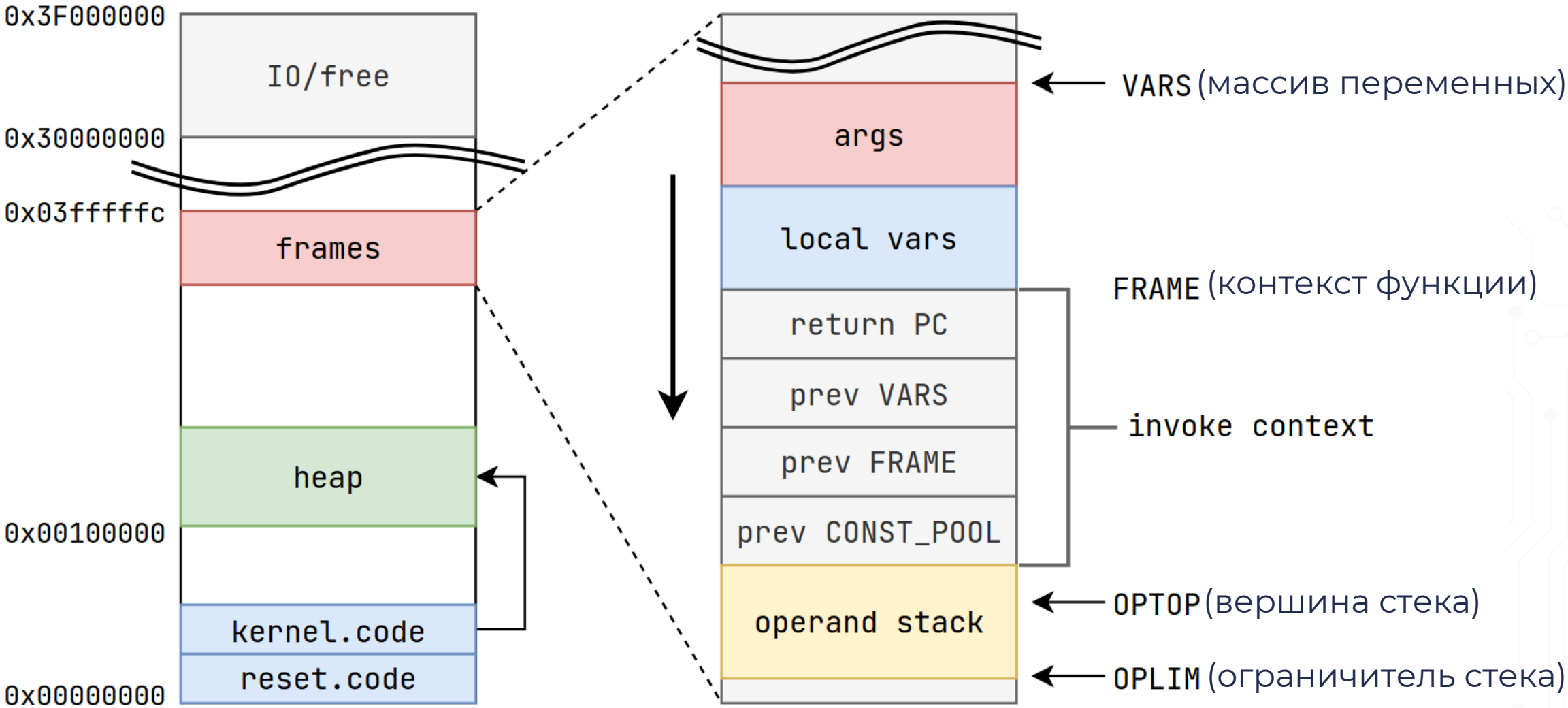
Начало начал



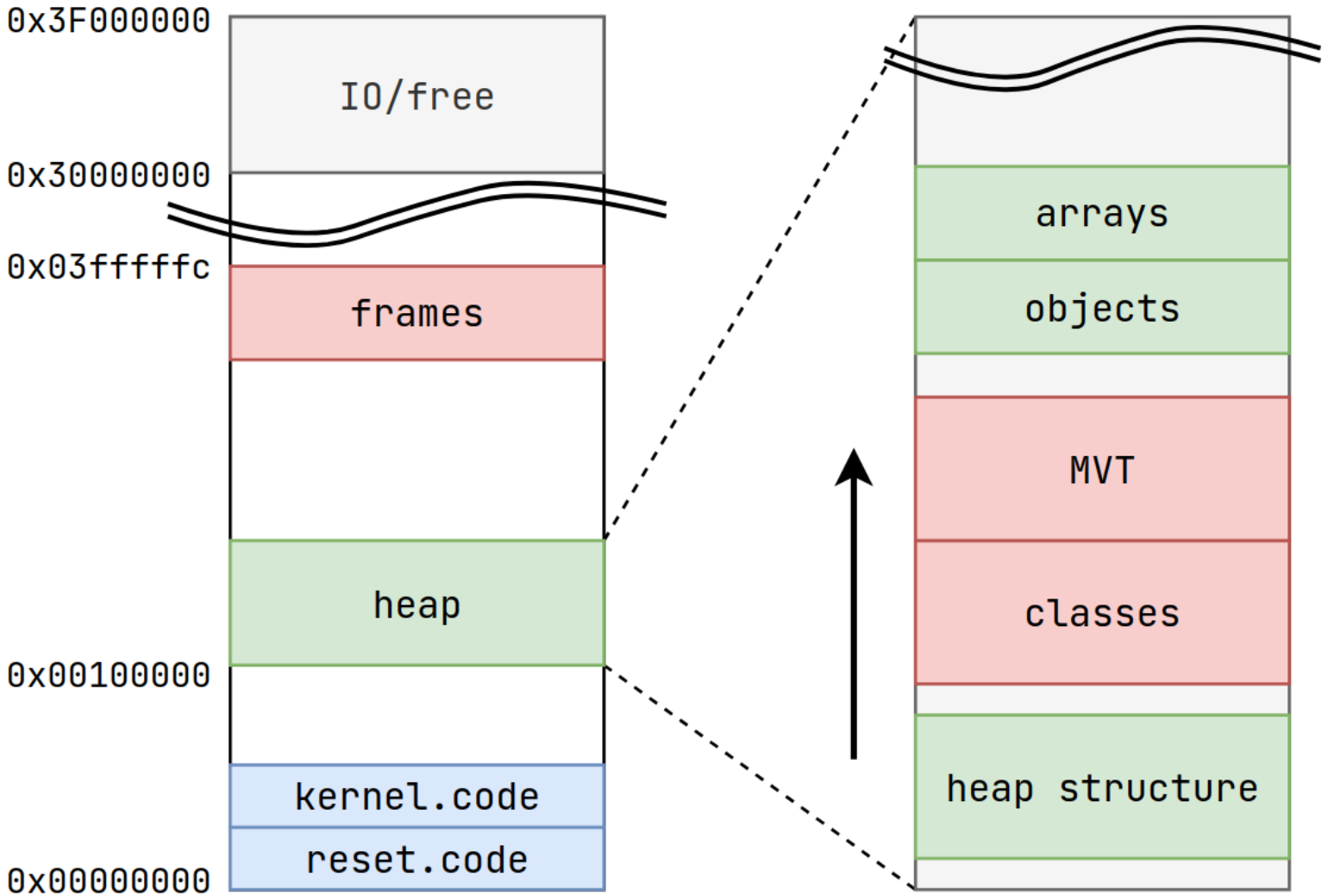
- Нужны две базовые программы, написанные на байткоде:
- reset.code
  - Приводит процессор в архитектурно-определённое состояние
  - Отвечает за инициализацию стека
- kernel.code
  - Отвечает за кучу и аллокации там
  - Прочие функции при необходимости

- Для взаимодействия с внешними устройствами резервируем верхнюю часть адресного пространства

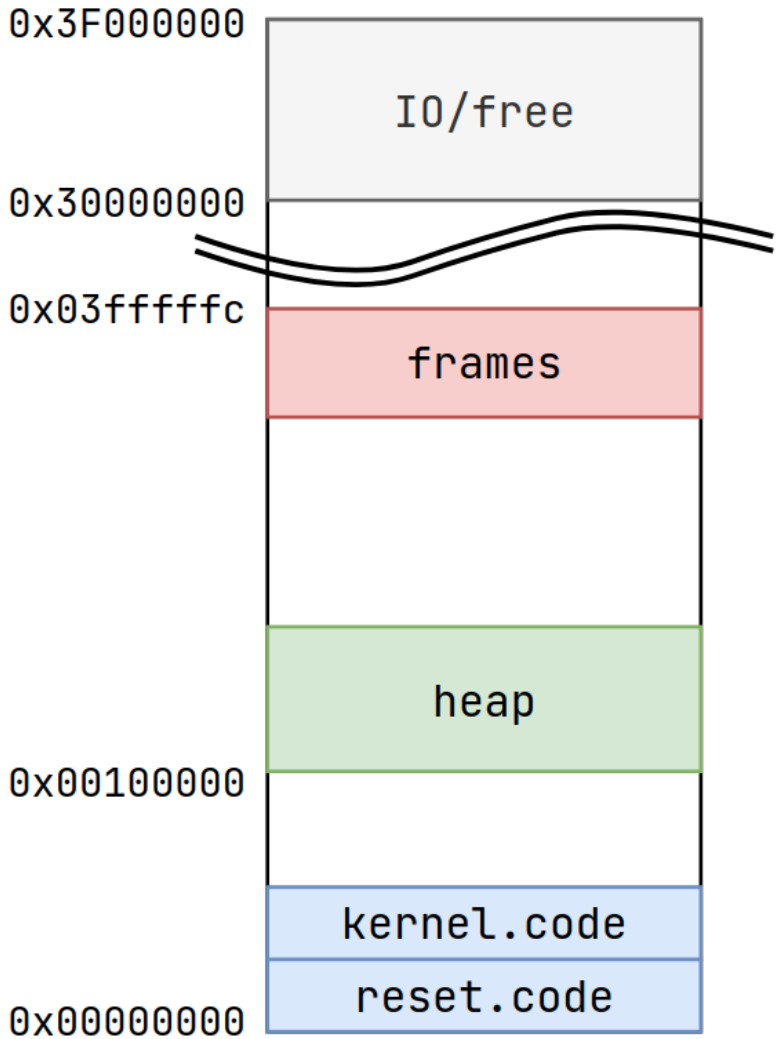
Пробуем вызвать функцию



А что там с heap



Ловушка Джокера

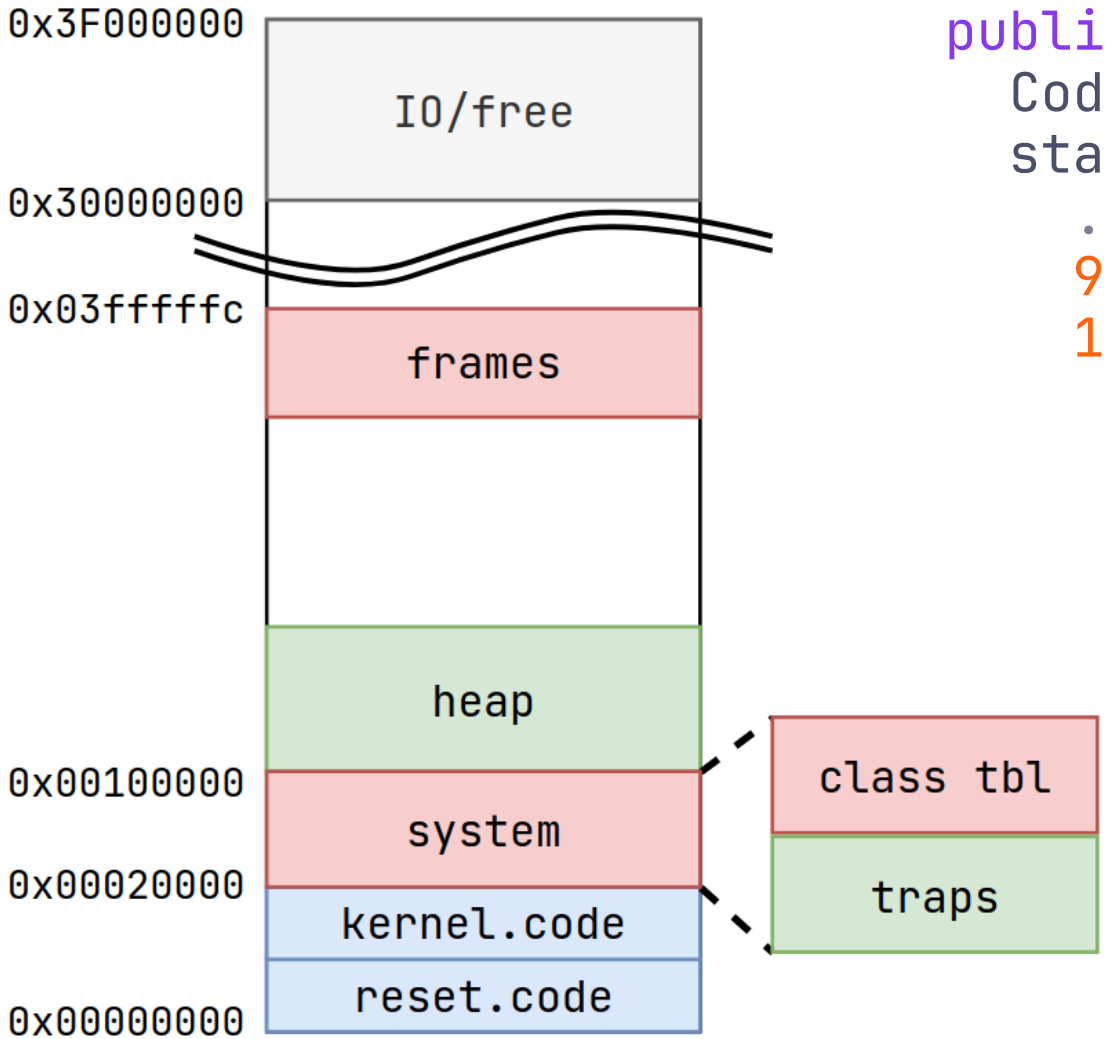


invokevirtual
indexbyte1
indexbyte2

```
Constant pool:
   #1 = Class              #2
   #2 = Utf8               Main
   #6 = Utf8               ()V
  #14 = Utf8               sayHello
  #32 = Methodref          #1.#33  // Main.sayHello:()V
  #33 = NameAndType        #14:#6  // sayHello:()V

public static int main();
  Code:
  stack=2, locals=2, args_size=1
     ...
     9:  invokevirtual #32  // Method sayHello:()V
     12: return
```

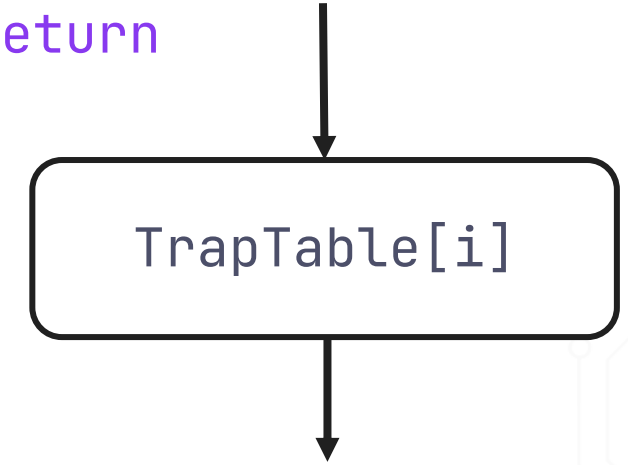
- Как аппаратуре понять, что вызывать?

Ловушка Джокера




```
public static int main();
  Code:
  stack=2, locals=2, args_size=1
     ...
     9:  invokevirtual #32
     12: return
```

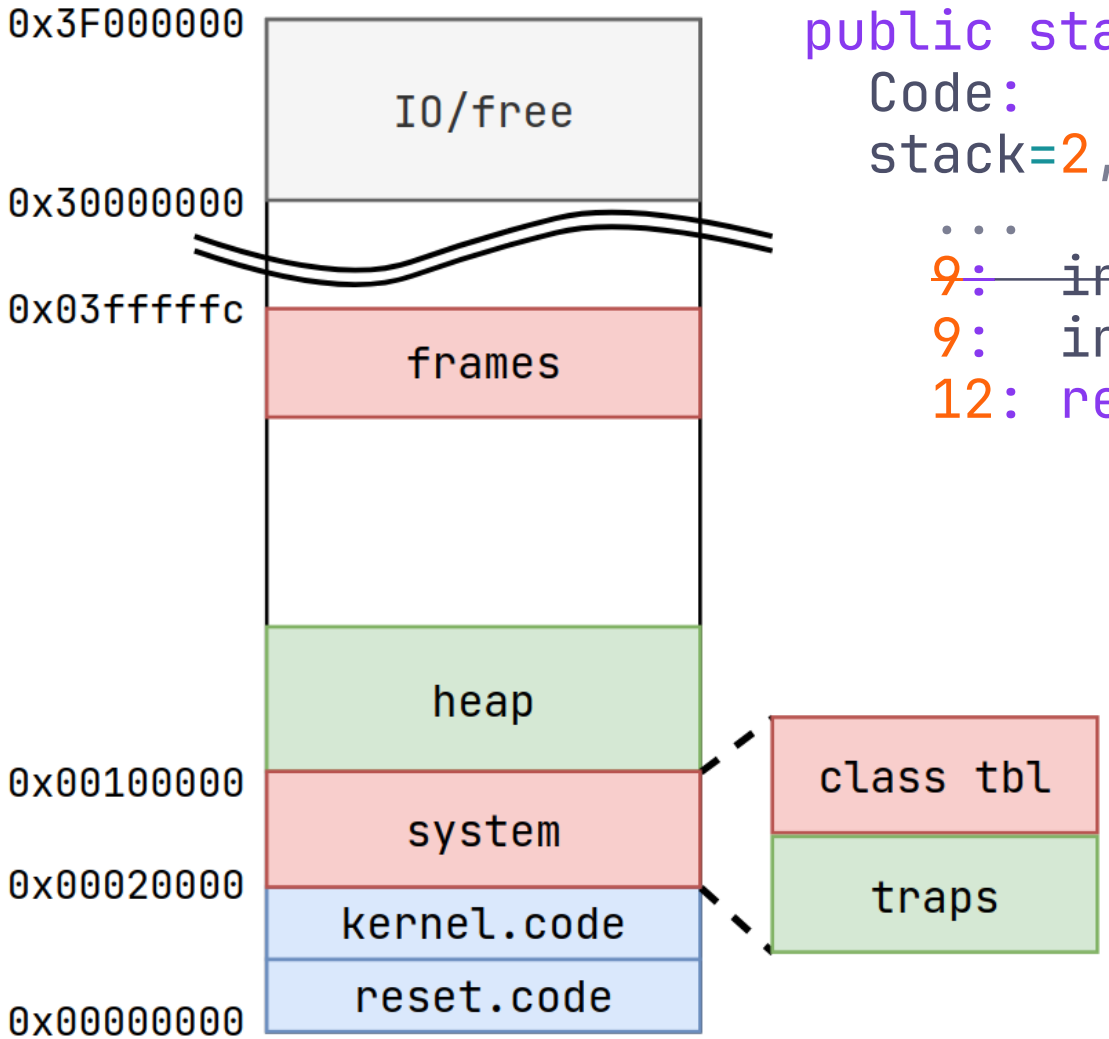
invokevirtual
indexbyte1
indexbyte2




```
public class invokevirtual {
public static Method
invokevirtual:"()v" { ... }
```

Ловушка Джокера

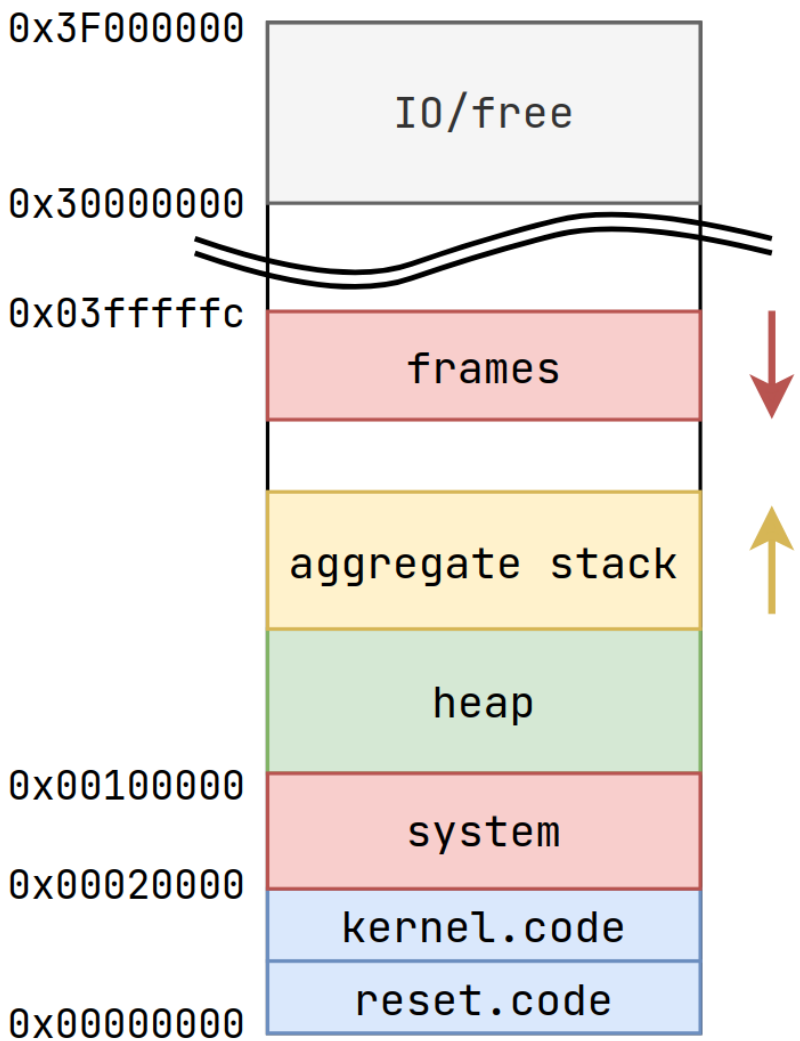
invokevirtual_quick
index
nargs




```
public static int main();
  Code:
  stack=2, locals=2, args_size=1
     ...
     9:  invokevirtual #32
     9:  invokevirtual_quick 5, 1 // index = 5
     12: return
```

- _quick инструкции не приводят к вызову обработчика исключения
- Долгое время эти методы существовали в JVMS как зарезервированные для VM

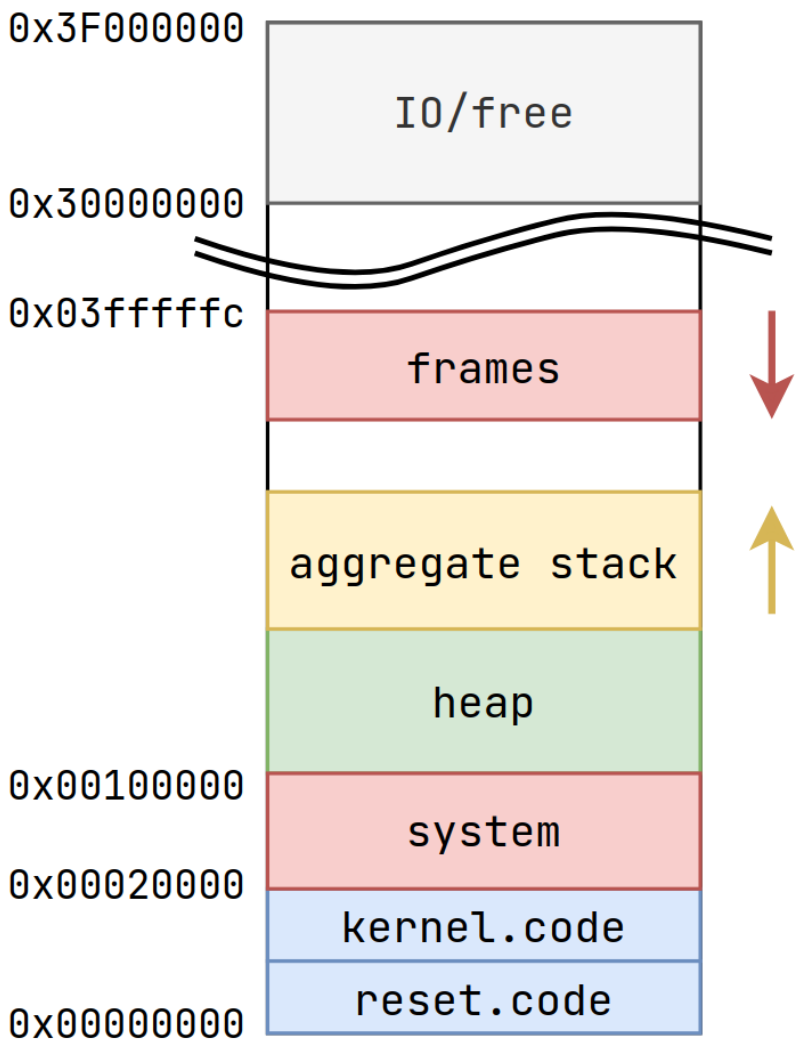
А что с Си?




```c
struct s { int a, b; };

int foo(struct s s1, int x, int y) {
    int local = x + y;
    int arr[2] = { s1.a, local };
    int *p = &x;

    return *p + s1.b;
}

void bar() {
    struct s s2 = { 3, 4 };
    foo(s2, 1, 2);
}
```

- Код на Си можно компилировать в байткод и использовать либо как метки, либо как JNI методы

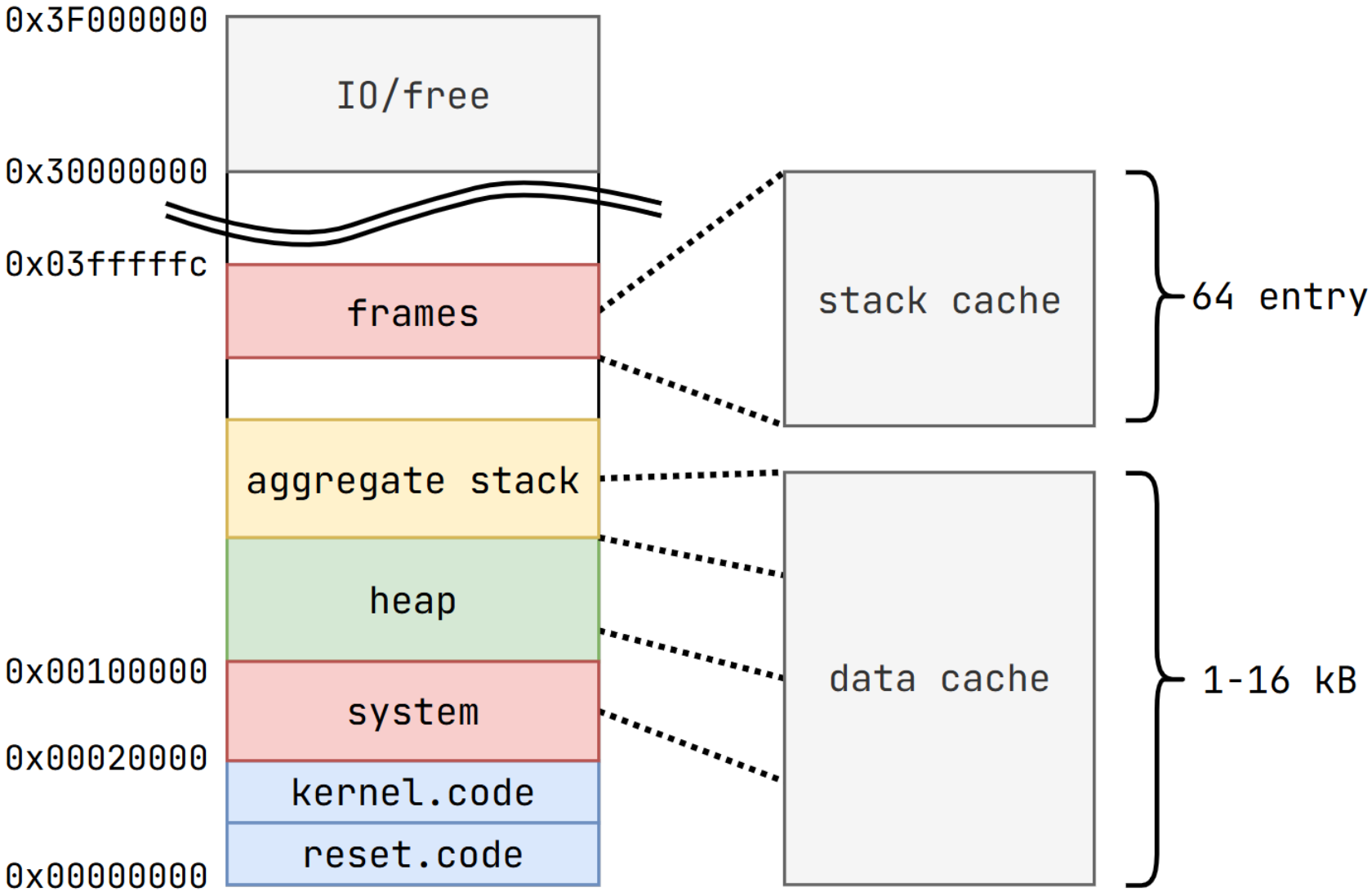
А что с Си?




```
struct s { int a, b; };

int foo(struct s s1, int x, int y) {
    int local = x + y;
    int arr[2] = { s1.a, local };
    int *p = &x;

    return *p + s1.b;
}

void bar() {
    struct s s2 = { 3, 4 };
    foo(s2, 1, 2);
}
```

- Не аггрегатные значения можно хранить на Java стеке

Ускоряем наш процессор!



Сегодня без процессоров

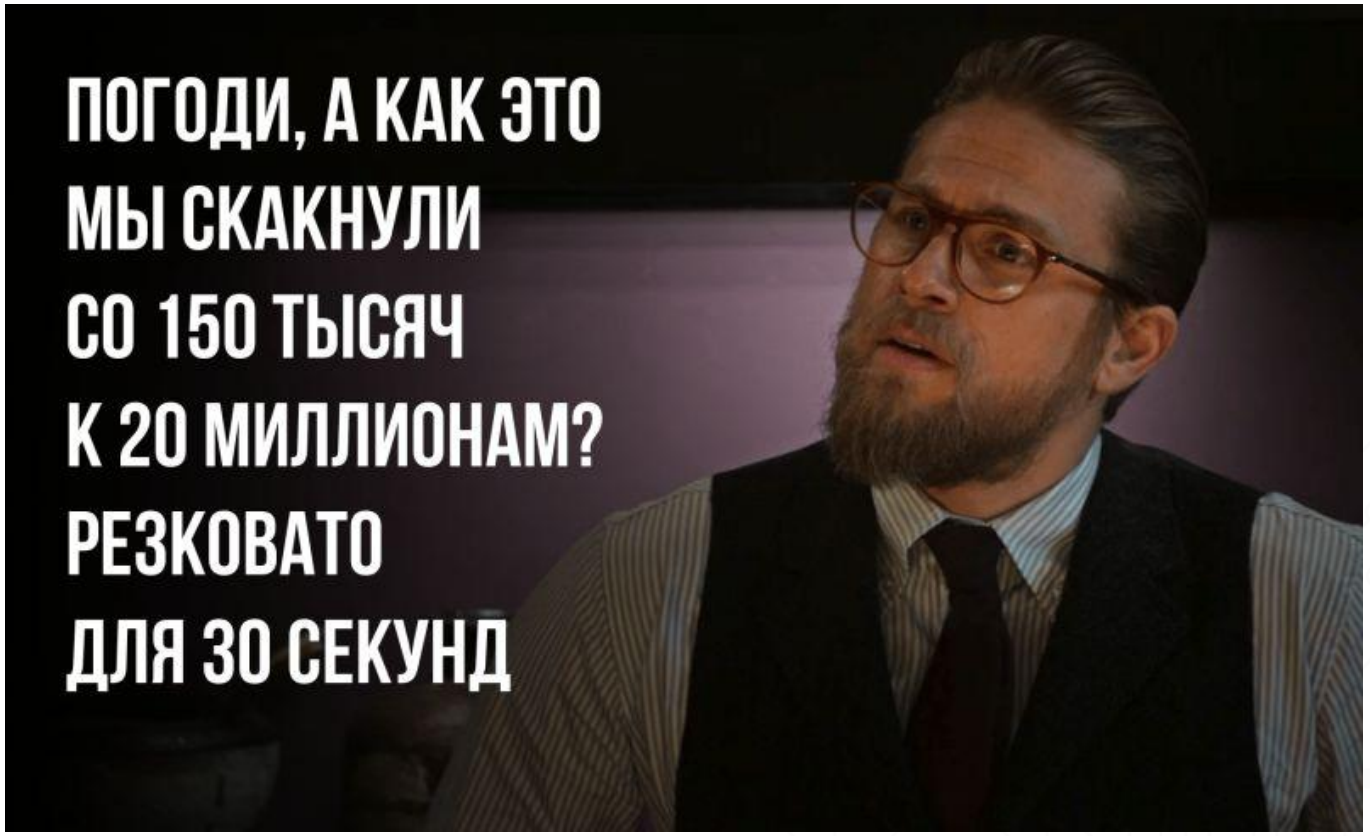
- Разработка железа – долгий процесс

Сегодня без процессоров

- Разработка железа – долгий процесс
- Высокие требования к инженерам

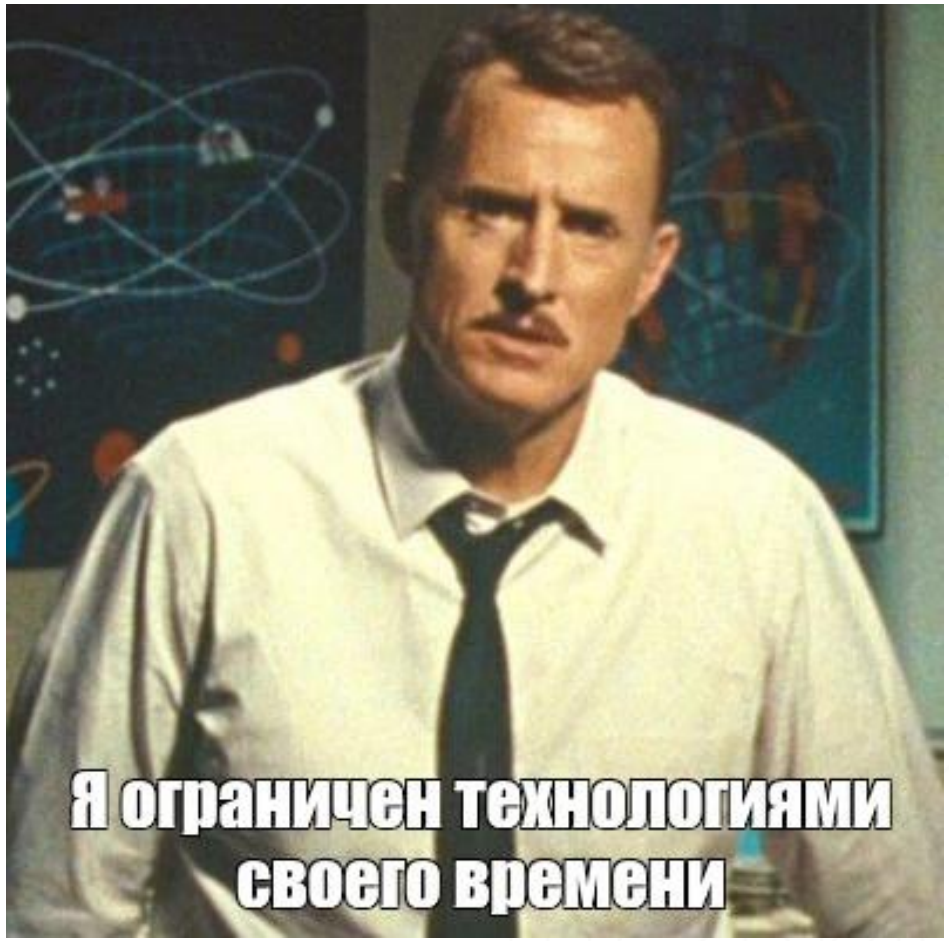
Сегодня без процессоров

- Разработка железа – долгий процесс
- Высокие требования к инженерам
- Стоимость ошибки в железе – огромная



Сегодня без процессоров

- Разработка железа – долгий процесс
- Высокие требования к инженерам
- Стоимость ошибки в железе – огромная
- Сложные операции приходится эмулировать



Сегодня без процессоров

- Разработка железа – долгий процесс
- Высокие требования к инженерам
- Стоимость ошибки в железе – огромная
- Сложные операции приходится эмулировать

Попробуем снова вернуться к программной реализации, но уже с накопленными знаниями

А помните, как было?

```
void Interpreter::main_loop(interpreterState istate) {
  while (!stop) {
  switch (opcode)
  case opc_iaload:
    arrayOop arrObj = (arrayOop)STACK_OBJECT(arrayOff);
    jint     index = STACK_INT(arrayOff + 1);
    ARRAY_INDEX_CHECK(arrObj, index);
    SET_STACK_INT(
      GET_HEAP_INT(arrObj, index)
    );
    UPDATE_PC_AND_TOS_AND_CONTINUE();
  ...
  }
}
```

- Для исполнения одного байткода в худшем случае придётся пройти весь switch
- Реализации интерпретатора обновляют состояние JVM каждый шаг

Интерпретируем по шаблону

- Сделаем для каждого байткода – обработчик на асссемблере
- Сделаем таблицу, в которой сохраним адреса всех обработчиков
- Каждый обработчик может вызвать следующий: через таблицу

```
void TemplateTable::iadd() {
  pop_i(x10);
  pop_i(x11);
  addi(x10, x10, x11);
  push_i(x10);

  dispatch_next();
}
```

```
class DispatchTable {
public:
  enum { length = 1 << BitsPerByte };
private:
  address _table[length];
public:
  address* table_for()
                { return _table; }
};
```

Интерпретируем по шаблону

- Сделаем для каждого байткода – обработчик на асссемблере
- Сделаем таблицу, в которой сохраним адреса всех обработчиков
- Каждый обработчик может вызвать следующий: через таблицу

```
void TemplateTable::iadd() {
  pop_i(x10);
  pop_i(x11);
  addi(x10, x10, x11);
  push_i(x10);

  dispatch_next();
}
```

```
class DispatchTable {
public:
  enum { length = 1 << BitsPerByte };
private:
  address _table[length];
public:
  address* table_for()
                { return _table; }
};
```

```
call(table_for()[++pc]);
```

```
void TemplateTable::imul()
```

Интерпретируем по шаблону

- Сделаем для каждого байткода – обработчик на асссемблере
- Сделаем таблицу, в которой сохраним адреса всех обработчиков
- Каждый обработчик может вызвать следующий: через таблицу

```
void TemplateTable::iadd() {
  pop_i(x10);
  pop_i(x11);
  addi(x10, x10, x11);
  push_i(x10);

  dispatch_next();
}
```

```
class DispatchTable {
public:
  enum { length = 1 << BitsPerByte };
private:
  address _table[length];
public:
  address* table_for()
                { return _table; }
};
```

```
call(table_for()[++pc]);
```

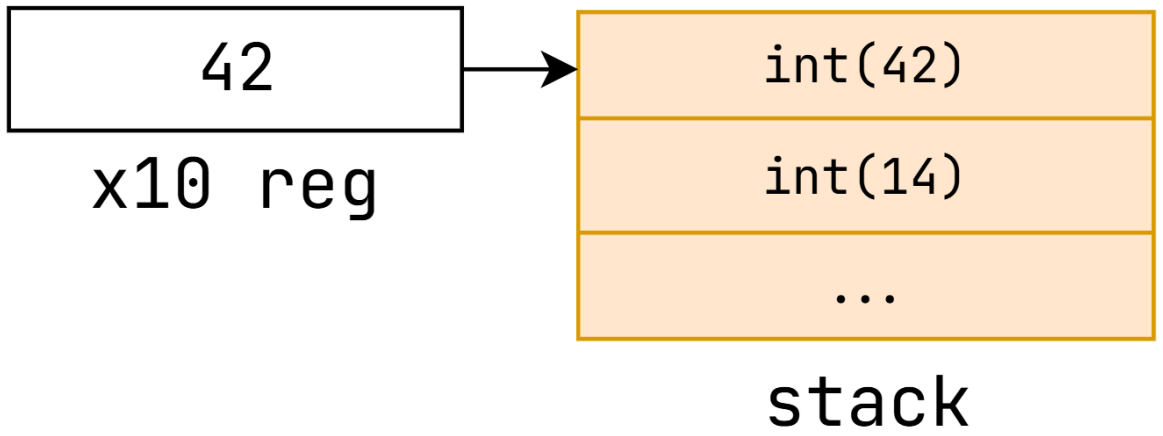
```
void TemplateTable::imul()
```

Интерпретируем по шаблону

- Сделаем для каждого байткода – обработчик на асссемблере
- Сделаем таблицу, в которой сохраним адреса всех обработчиков
- Каждый обработчик может вызвать следующий: через таблицу

```
void TemplateTable::iadd() {
  pop_i(x10);
  pop_i(x11);
  addi(x10, x10, x11);
  push_i(x10);

  dispatch_next();
}
```

```
class DispatchTable {
public:
  enum { length = 1 << BitsPerByte };
private:
  address _table[states][length];
public:
  address* table_for(TosState state)
                { return _table[state]; }
};
```



Интерпретируем по шаблону

- Сделаем для каждого байткода – обработчик на асссемблере
- Сделаем таблицу, в которой сохраним адреса всех обработчиков
- Каждый обработчик может вызвать следующий: через таблицу

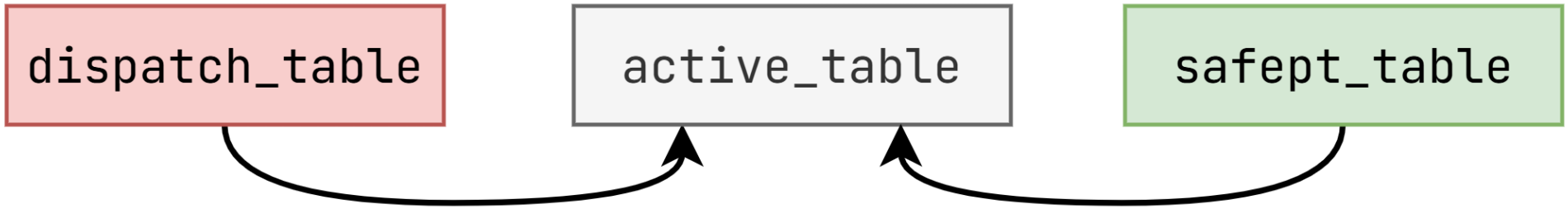
```
void TemplateTable::iadd() {
  // x10 - holds int
  pop_i(x11);
  addi(x10, x10, x11);

  update_tos(itos, itos);
  dispatch_next();
}
```

```
class DispatchTable {
public:
  enum { length = 1 << BitsPerByte };
private:
  address _table[states][length];
public:
  address* table_for(TosState state)
                  { return _table[state]; }
};
```

```
enum TosState { // describes the tos cache
btos = 0,ztos = 1     // byte, bool tos cached
ctos = 2,             // char tos cached
...
number_of_states,
};
```

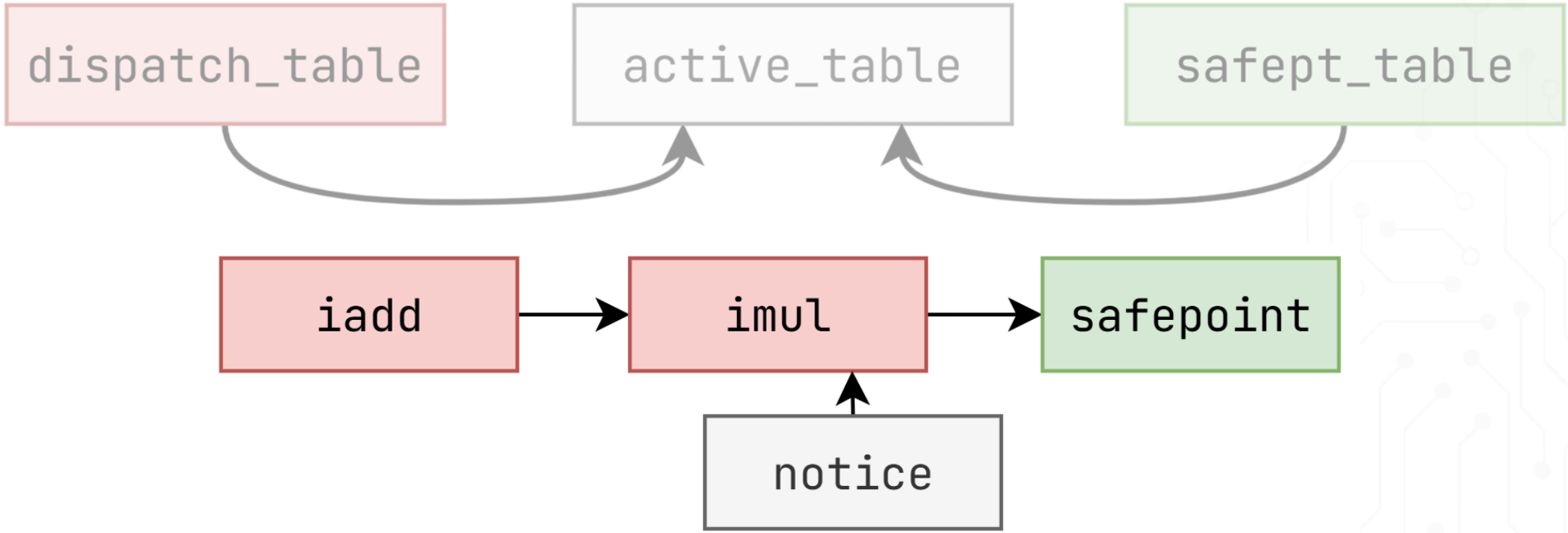
А что с безопасностью?

```
void TemplateInterpreter::notice_safepoints() {
  if (!_notice_safepoints) {
    _notice_safepoints = true;
    copy_table((address*)&_safept_table, (address*)&_active_table,
               sizeof(_active_table) / sizeof(address));
  }
}
```



А что с безопасностью?

```
void TemplateInterpreter::notice_safepoints() {
  if (!_notice_safepoints) {
    _notice_safepoints = true;
    copy_table((address*)&_safept_table, (address*)&_active_table,
               sizeof(_active_table) / sizeof(address));
  }
}
```



Выводы: производительность

- Шаблонный интерпретатор хоть и требует время для генерации обработчиков, работает значительно быстрее zero
- Затраты по памяти не очень велики:

```
$ java --XX:+UnlockDiagnosticVMOptions  -XX:+PrintInterpreter

Interpreter

code size          =     98K bytes
total space        =     98K bytes
wasted space       =      0K bytes

# of codelets     =    280
avg codelet size =    359 bytes
```

Volker Simonis: Template- vs. C++-Interpreter shootout

Выводы

- Сегодня посмотрели на аппаратную реализацию Java и смогли уйти живыми
- Узнали, как разрабатывают железо
- Поняли, как полученный опыт помогает улучшать интерпретатор

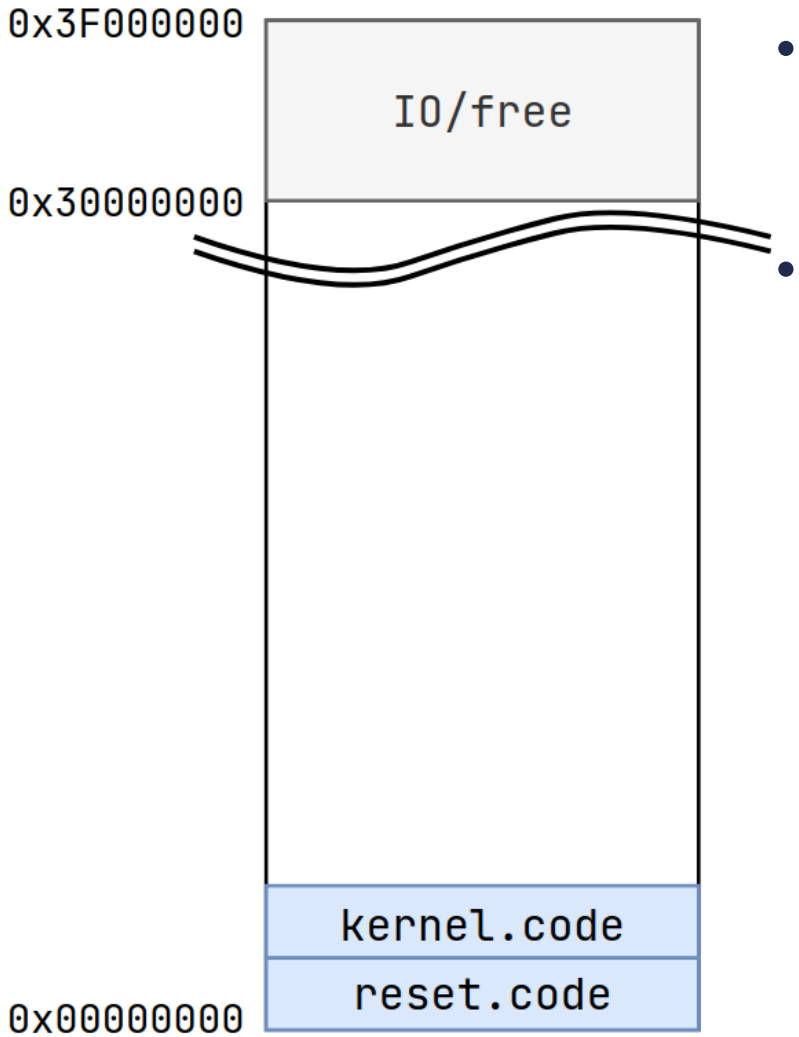
JVM Specification 25

PicoJava-II (archive)

Спасибо за внимание!

Backup

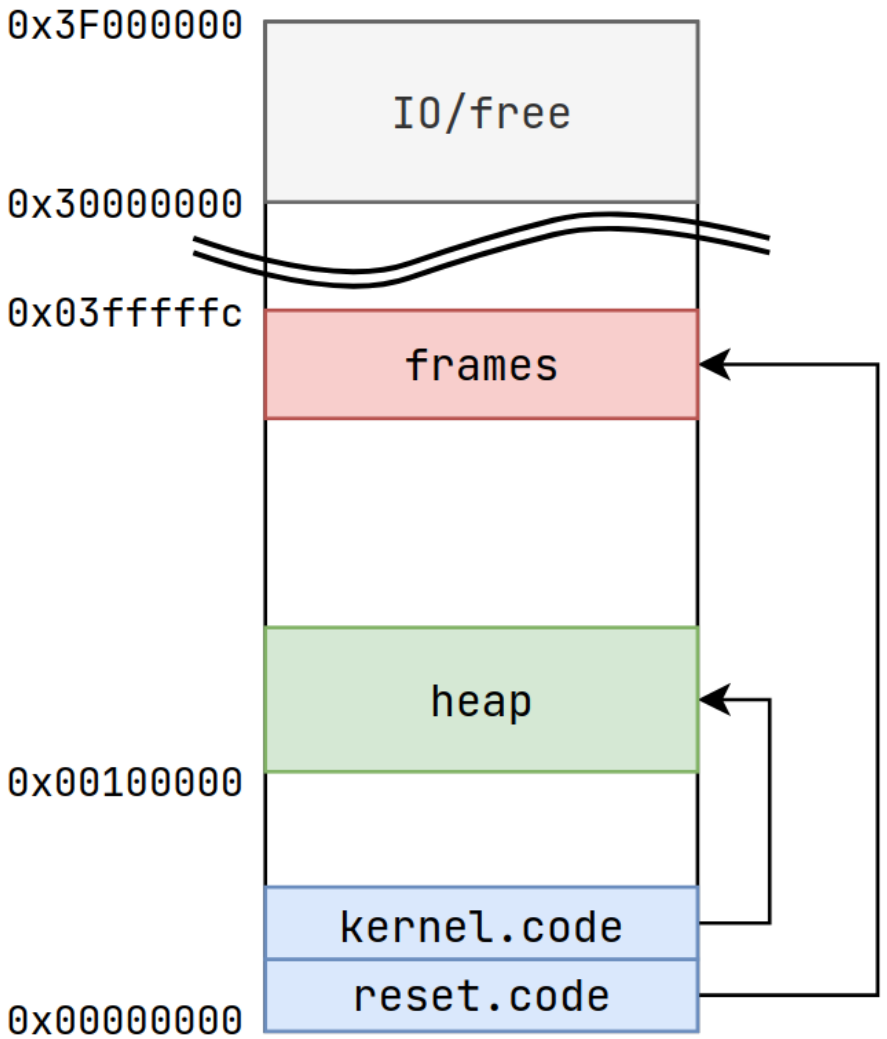
Создаём вселенную



- Нам бы хотелось иметь связь с устройствами вне процессора
  - Выделим для этого участок памяти
- Сообщим процессору, где у нас будет стек
  - Нам необходимы регистры

```
File reset.code:
public final class reset
extends java/lang/Object {
public static Method reset:"()I" {
// Write 0x003FFFFC to OPTOP register
  sipush 0xFFFC;
  sethi 0x003F;
  write_optop;
  ...
}}
```

Создаём во вселенной heap



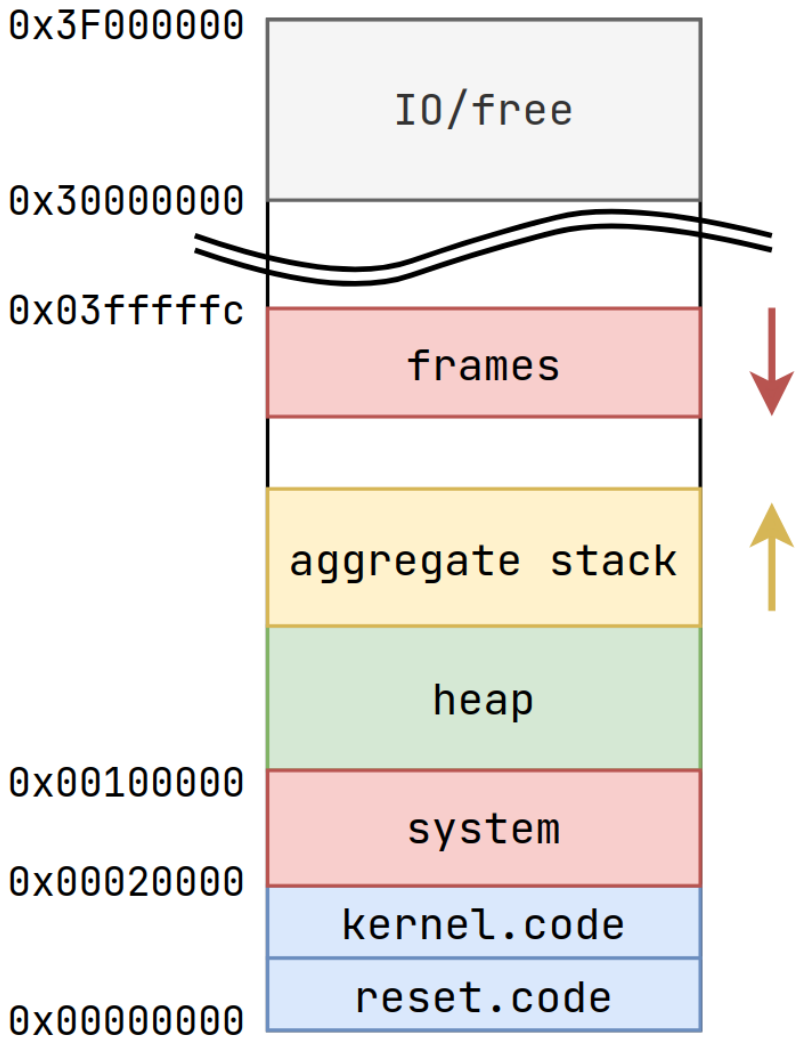
- Код ядра должен предоставлять системные функции
  - Ядро не должно быть доступно напрямую из Java

```
bipush 42;
jsr Kernel;
```

↓

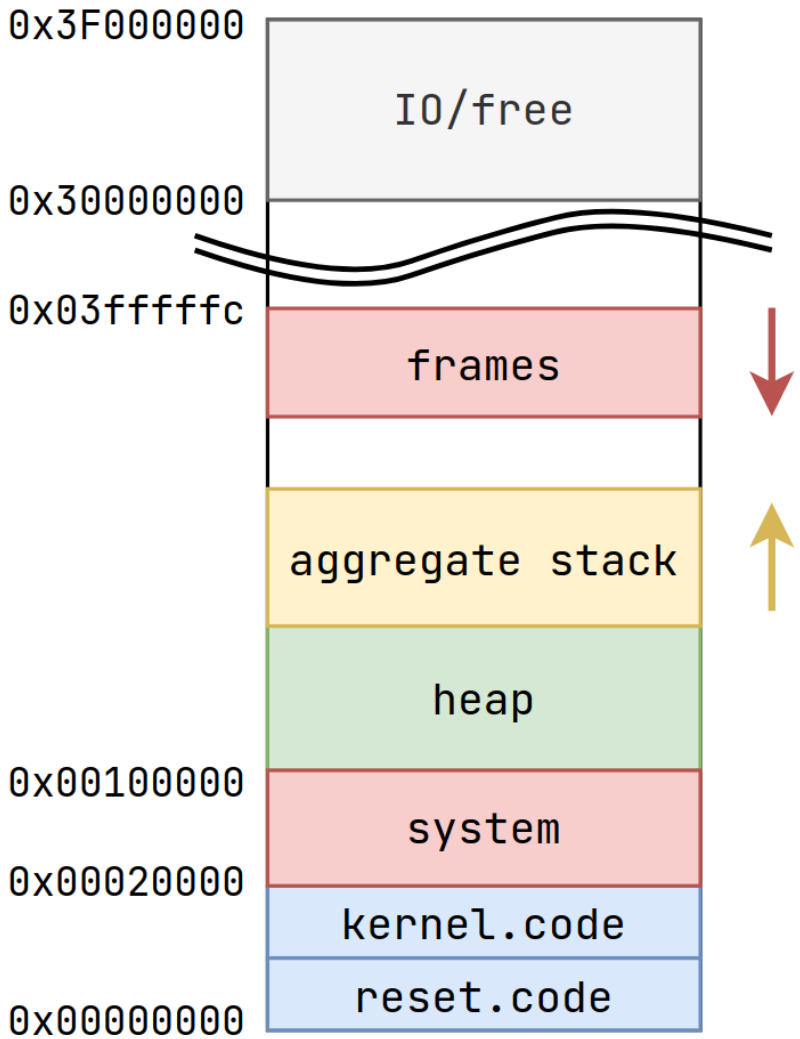
```
Kernel:

  iload_1;      // load request number
    bipush 42; // #42: static storage alloc
  if_icmpeq AllocateStaticWord;

  iload_1;      // load request number
    bipush 43; // #43: Resolve the class
  if_icmpeq ResolveClass;
    ...
  goto Unimplemented_Request;
```

Ты не пройдёшь!

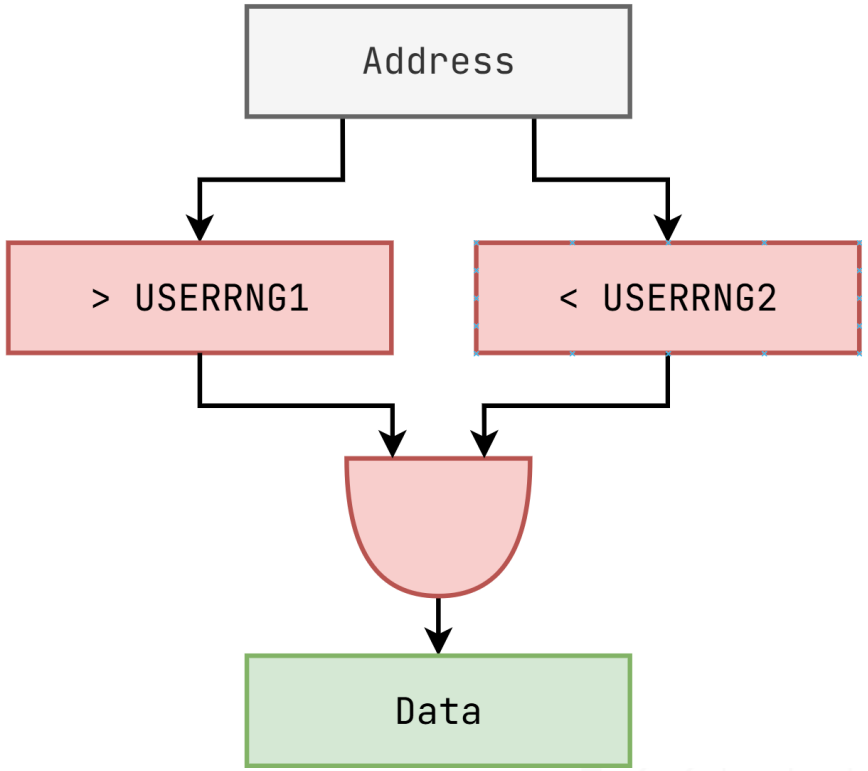


- При написании кода переменные изолированы в стеке аггрегатов
- Но кто мешает нам получить доступ к стеку по адресу?

Ты не пройдёшь!



- При написании кода переменные изолированы в стеке аггрегатов
- Но кто мешает нам получить доступ к стеку по адресу?



Почувствуем себя разработчиками железа

Попробуем разработать код, который будет производить побитовый сдвиг влево (ishl): `uint32_t ishl(uint32_t val, uint32_t shift)`

Почувствуем себя разработчиками железа

Попробуем разработать код, который будет производить побитовый сдвиг влево (ishl): `uint32_t ishl(uint32_t val, uint32_t shift)`

- Опишем наш модуль
- Входы:
  - Входное значение – 32 бита
  - Сдвиг – 32 бита
- Выход:
  - Сдвинутое значение – 32 бита

```
val io = IO(new Bundle {
     val (in, out)    = (Input(UInt(32.W)), Output(UInt(32.W)))
     val shift        =  Input(UInt(32.W))
})
```

Почувствуем себя разработчиками железа

Попробуем разработать код, который будет производить побитовый сдвиг влево (ishl): `uint32_t ishl(uint32_t val, uint32_t shift)`

- Опишем наш модуль
- Входы:
  - Входное значение – 32 бита
  - **Сдвиг – 5 бит**
- Выход:
  - Сдвинутое значение – 32 бита

```
val io = IO(new Bundle {
     val (in, out)    = (Input(UInt(32.W)), Output(UInt(32.W)))
     val shift        =  Input(UInt(log2Ceil(32).W))
})
```

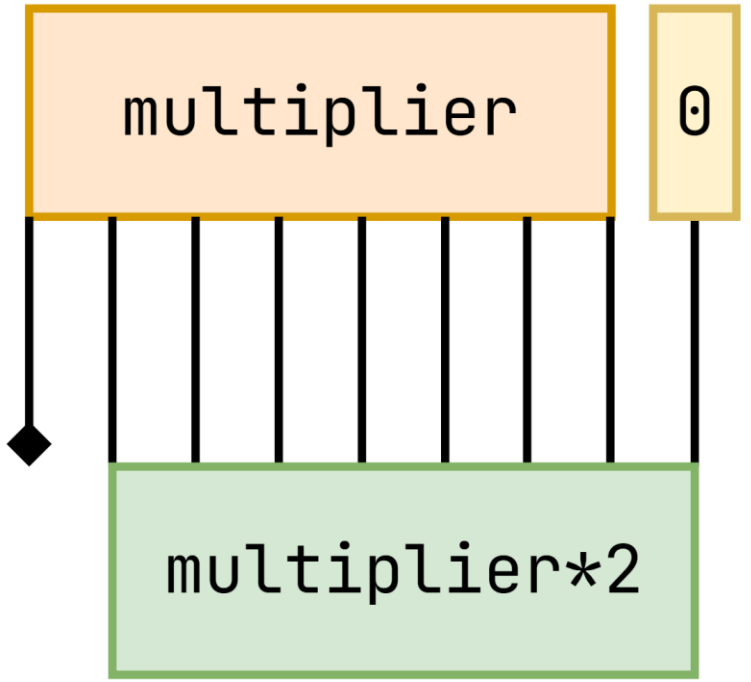
Почувствуем себя разработчиками железа

```
uint32_t ishl(uint32_t val, uint32_t shift)
{
  assert(shift < 32u);
  // ishl(x, n) = x * 2^n
  unsigned int multiplier = 1;

  for (unsigned i = 0; i < shift; i++) {
    multiplier += multiplier;
  }
  return val * multiplier;
}
```

- Подумаем, как перенести этот код в аппаратную реализацию

Почувствуем себя разработчиками железа

```
uint32_t ishl(uint32_t val, uint32_t shift)
{
  assert(shift < 32u);
  // ishl(x, n) = x * 2^n
  unsigned int multiplier = 1;

  for (unsigned i = 0; i < shift; i++) {
    multiplier += multiplier;
  }
  return val * multiplier;
}
```

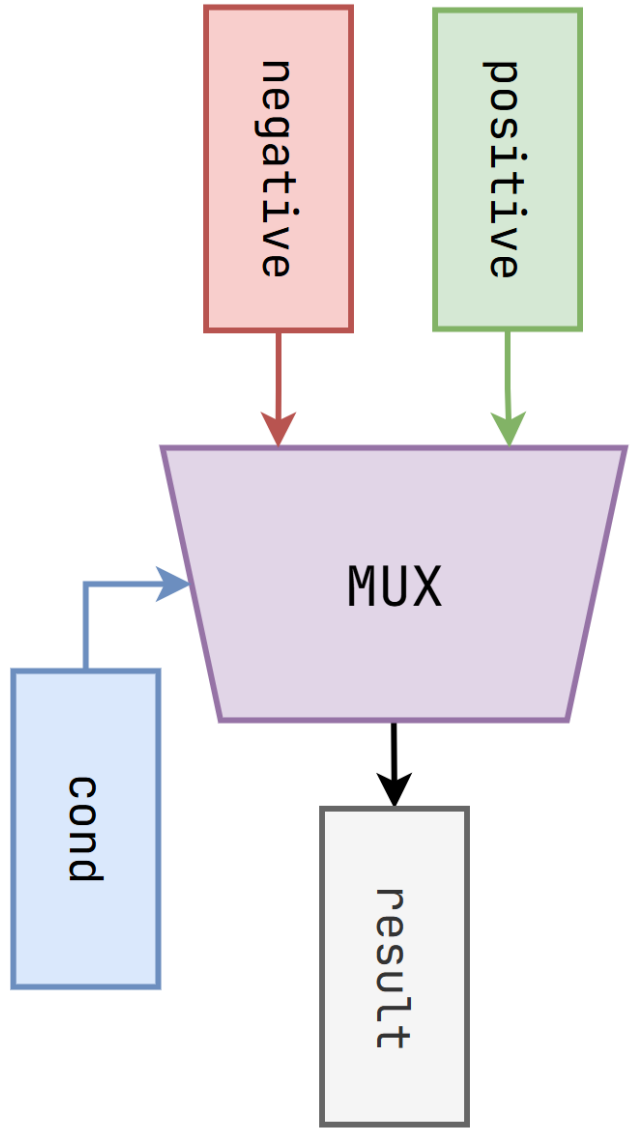


- Подумаем, как перенести этот код в аппаратную реализацию
- Не все операции в нашем коде эффективны
- Попробуем переписать код иначе

Почувствуем себя разработчиками железа

```
uint32_t ishl(uint32_t val, uint32_t shift)
{
  assert(shift < 32u);
  // shift = {1 + 2 + 4 + 8 + 16}

  if (shift & 0b00001) val *= 2^1;
  if (shift & 0b00010) val *= 2^2;
  if (shift & 0b00100) val *= 2^4;
  if (shift & 0b01000) val *= 2^8;
  if (shift & 0b10000) val *= 2^16;

  return val;
}
```

- MUX(cond, pos, neg) = cond ? pos : neg
- Теперь мы готовы сгенерировать код



Генерируем схему

```
class BarrelShifter(val w: Int) extends Module {
  val io = IO(new Bundle {
    val (in, out)    = (Input(UInt(w.W)), Output(UInt(w.W)))
    val shift        =  Input(UInt(log2Ceil(w).W))
  })
  var x = io.in
  for (i <- 0 until log2Ceil(w)) {
    x = Mux(io.shift(i), x << (1 << i), x)
  }
  io.out := x
}
```

- Делаем сдвиг универсальным, используя генерирующий цикл
- Специфицируем значения по всем путям мультиплексора
- Готовы посмотреть на получившийся результат?