Пакеты проектирования сверхбольших интегральных схем

Лекция 2

Рассказывает:

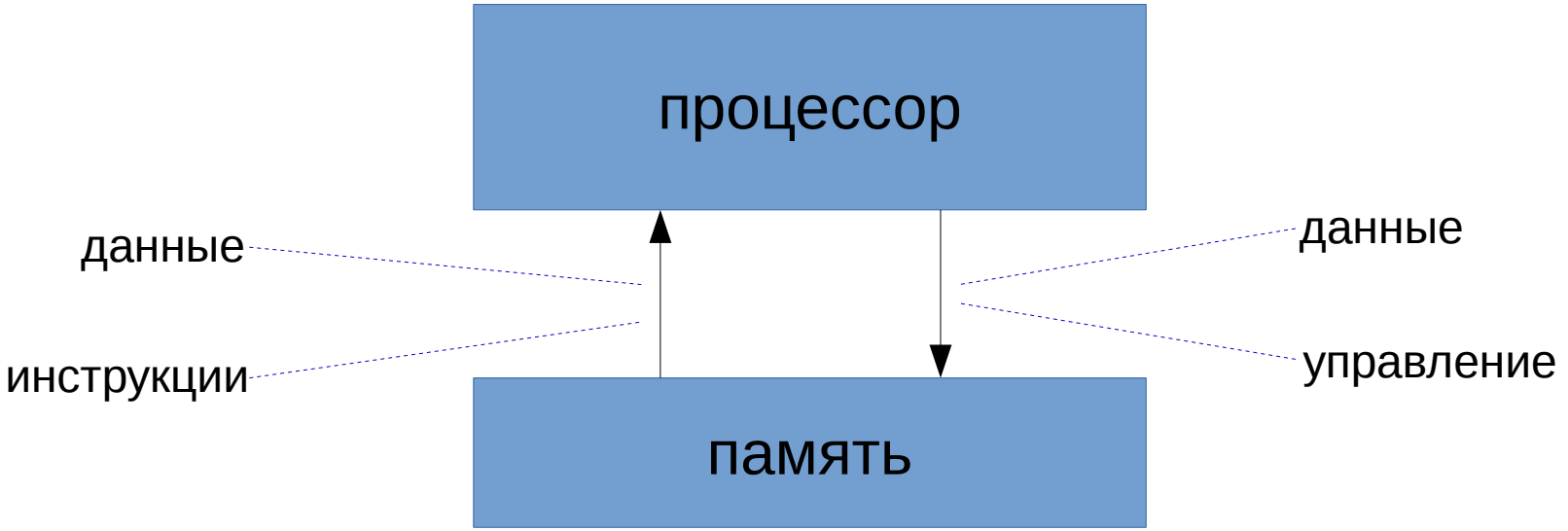
Подымов Владислав Васильевич

e-mail:
valdus@yandex.ru

Осень 2016

Чем будем заниматься в ближайшее время

Проектировать **процессор** (*надо всё-таки заставить вас понять, что было в курсе “Языки описания схем”*)

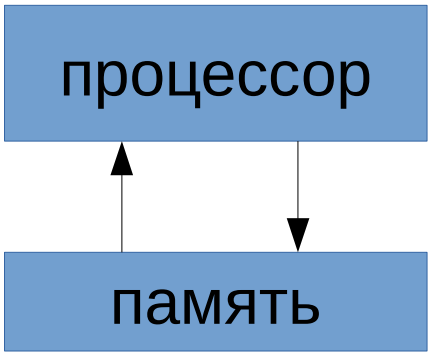


Как и в курсе “Языки описания схем”, во всех понятиях и концепциях будем держаться близко к тому, что происходит в **MIPS**

Устройство процессора

Процессор читает из памяти инструкции, одну за одной,

и исполняет их



Инструкция – это массив нолей и единиц фиксированной ширины, в котором сказано,

- откуда взять значения данных
- что сделать с этими данными
- куда положить результат

Договоримся о ширине инструкций: **32 бита**

[0000 0011 0101 0101 1101 1011 0000 1010] - это может быть инструкцией

(с отсылкой к тому, что будет дальше, будем называть это **машинным кодом** инструкции)

Устройство процессора

Внешняя память – это очень медленное устройство по сравнению с процессором

Если ничего с этим не делать, то процессор будет работать
настолько же медленно, насколько и память

Чтобы процессор мог работать “в полную мощность”,
обычно заводят несколько промежуточных уровней памяти

Чем ближе (*физически и логически*) уровень памяти к процессору, тем

- меньше объём памяти этого уровня
- быстрее работает эта память

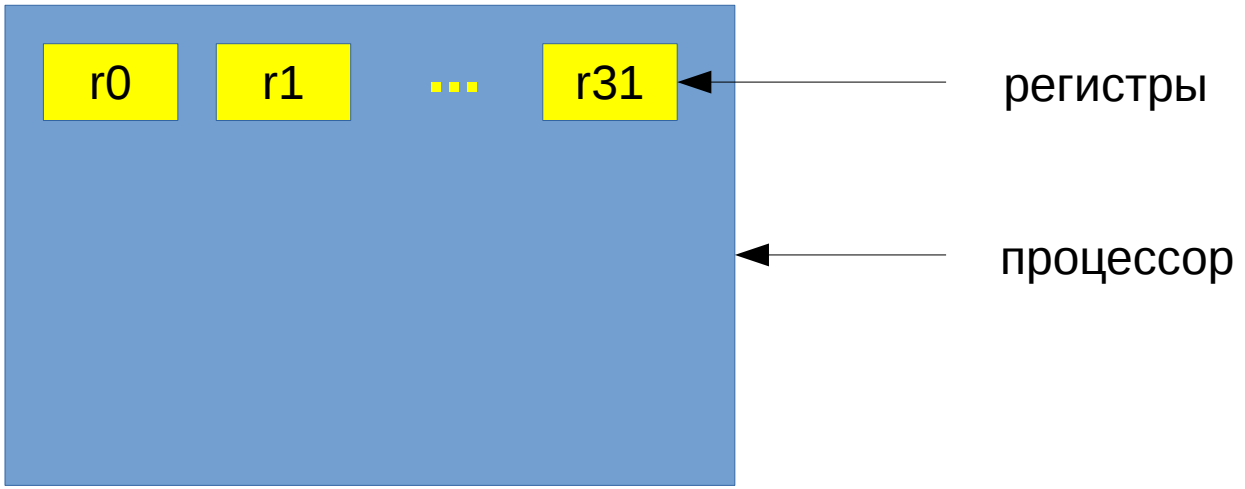
Самая близкая к процессору память – это **регистры** (*register*)

Не путать с регистрами Verilog (reg)!

Регистр хранит массив нолей и единиц фиксированной ширины

Чтобы жить было проще, будем (*пока что*) проектировать процессор,
в котором из всех уровней памяти есть **только регистры**

Устройство процессора



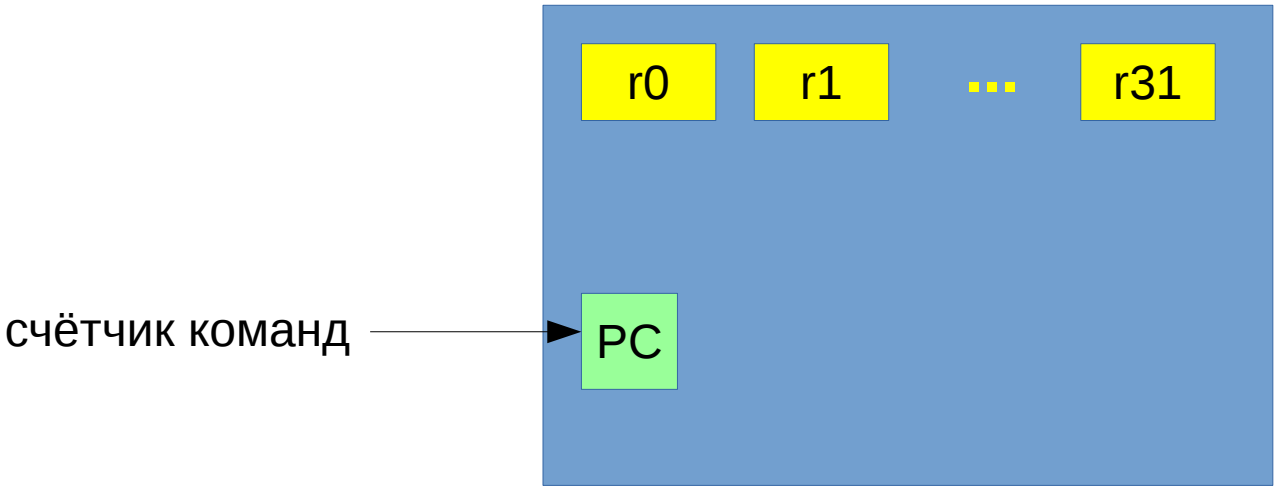
Договоримся о том, сколько регистров будет в процессоре и какова их ширина:

- **32 регистра** (*значит, для кодирования номера регистра нужно 5 бит*)
- ширина регистра – **32 бита** (*такая же, как и ширина инструкции*)

Чтобы не усложнять себе жизнь, *сейчас* будем считать, что

- у нас нет регистров специального назначения (*например, выдающего всегда 0*)
- назначение **всех** регистров – **только** хранение данных согласно инструкциям

Устройство процессора

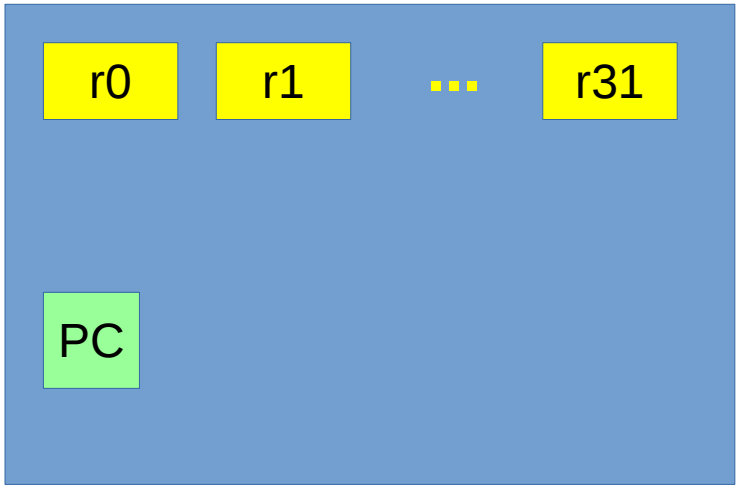


Чтобы знать, какую инструкцию обрабатывать следующей, процессор должен где-то хранить информацию о смещении этой инструкции в памяти

Эта информация хранится в **счётчике команд** (*program counter*)

Счётчик команд хранит массив нолей и единиц –
смещение **в байтах** следующей инструкции в памяти

Устройство процессора



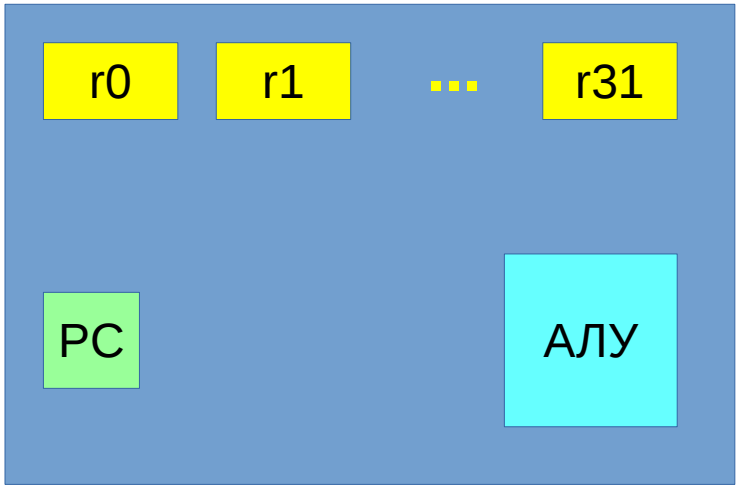
После исполнения инструкции счётчик увеличивается на **4** ◄—— (*откуда взялось это магическое число?*)

Если инструкция велит нарушить обычный порядок исполнения,
то счётчик может изменяться по-другому (*как велит инструкция*)

Договоримся о ширине счётчика команд:

32 бита (*такая же, как ширина регистра и ширина инструкции*)

Устройство процессора



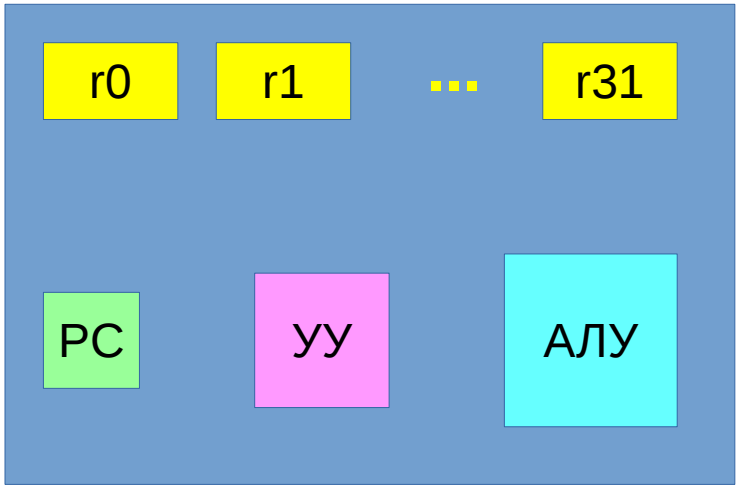
Часть инструкций производит арифметико-логические операции над числами:

- сложить два числа
- произвести циклический сдвиг числа
- вычесть одно число из другого
- ...

Такого рода операции производятся **арифметико-логическим устройством** (**АЛУ**)

АЛУ – это **комбинационная схема**:
результат операции немедленно получается на выходе

Устройство процессора



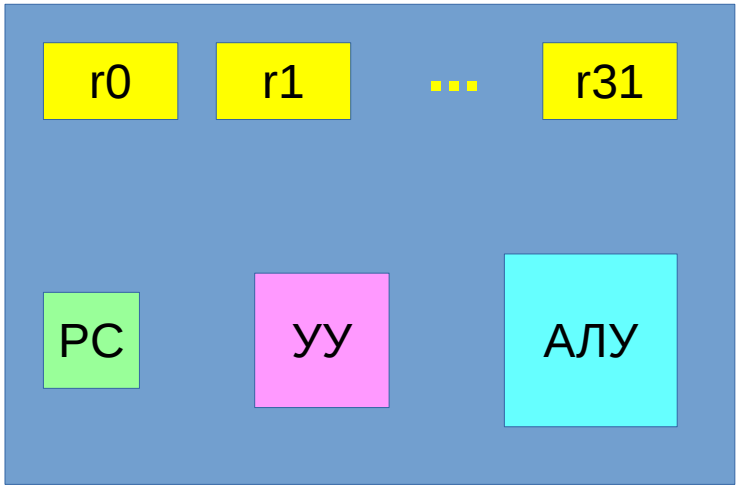
Согласно основному принципу проектирования схем, процессор можно разделить на две половины:

- **операционный автомат** (*схема, по которой курсируют данные*)
- **управляющий автомат** (*схема, нужным образом выставляющая управляющие сигналы, согласно которым данные преобразуются нужным образом*)

Регистры, счётчик команд и АЛУ содержат управляющие сигналы (например, “*загрузить значение в регистр*”)

Эти сигналы выставляет отдельный блок – **устройство управления** (**УУ**)

Устройство процессора



Вот такой процессор мы будем проектировать

(*по крайней мере поначалу*)

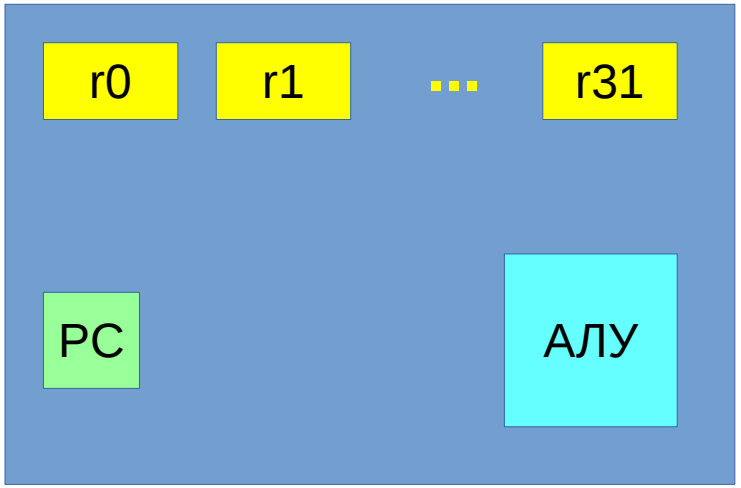
Теперь можно точно сказать, откуда процессор берёт данные и куда кладёт:

- берёт из памяти, регистров и счётчика команд
- кладёт в память, регистры и счётчик команд

А как процессор преобразует данные?

Это зависит от того, под исполнение какого **набора инструкций** проектируется процессор

Что вас ждёт в ближайшее время



На ближайшем семинаре мы будем

- строить операционный автомат совсем простого процессора
- учиться проверять, правильно ли написан этот автомат,

ещё до того, как написан управляющий автомат

(*так как в курсе “Языки описания схем” было мало практики,*
придётся здесь познавать всё заново)

Формат инструкций

Каждая инструкция имеет короткую (***ассемблерную***) запись, например

Эта запись содержит **название** и **аргументы** (*не более трёх аргументов*)

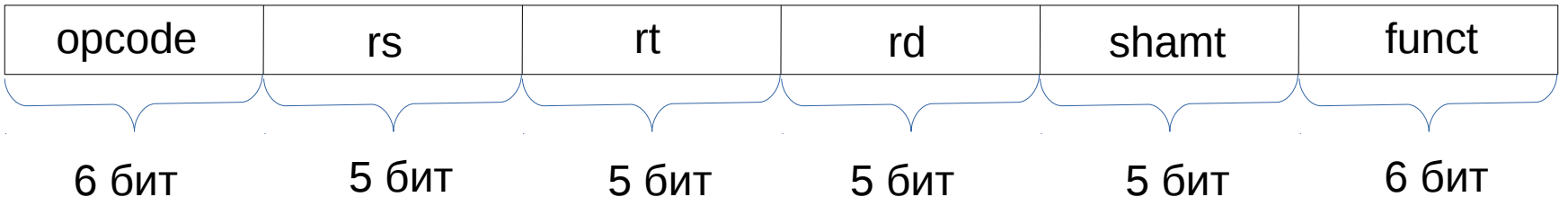
Ассемблерной записью определяется, как выглядят
и что значат подмассивы машинной записи

Инструкции делятся на группы по назначению и тому, как они исполняются:

- **R-тип** (*register*) работает только со значениями регистров
- **I-тип** (*immediate*) работает со значениями регистров
  и использует **константу**, явно заданную в инструкции
- **J-тип** (*jump*) инструкции безусловного перехода;
  используют только константу из самой инструкции
- ...

Инструкции R-типа

Все эти инструкции имеют такой формат:



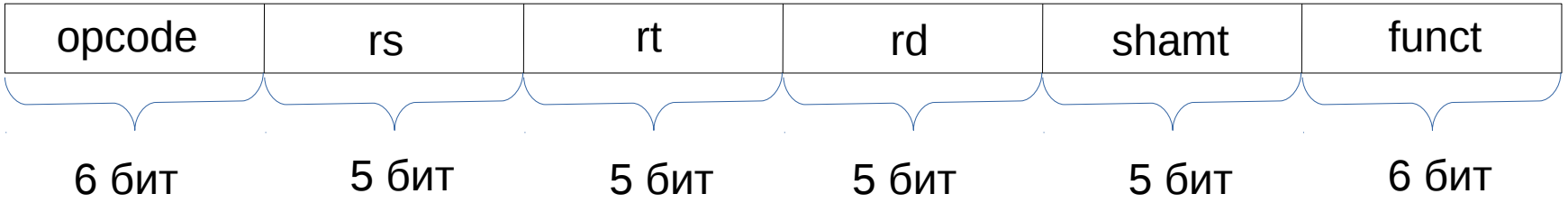
- **opcode** – код операции; в нашем простом случае это массив [000000] и означает “это операция R-типа”
- **rs**, **rt** – номера регистров, значения которых используются
- **rd** – номер регистра, в который пишется значение
- **shamt** – если это инструкция, согласно которой нужно сдвинуть массив бит регистра, то здесь записана константа: на сколько позиций сдвинуть
- **funct** – код конкретной операции R-типа

Небольшая ремарка:

- регистр отождествляем с его номером, если ясно, что мы говорим про регистр
- если **r** – регистр, то **$r** – значение, которое в нём хранится

(*как в ассемблерной записи*)

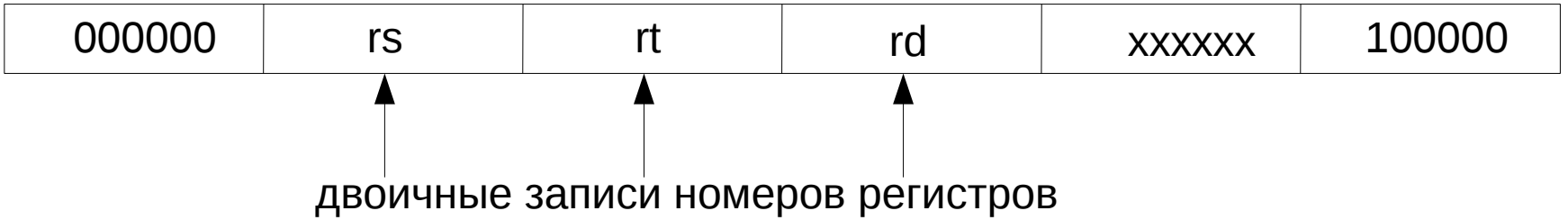
Инструкции R-типа



Арифметические инструкции (*знаковая арифметика с переполнением*)

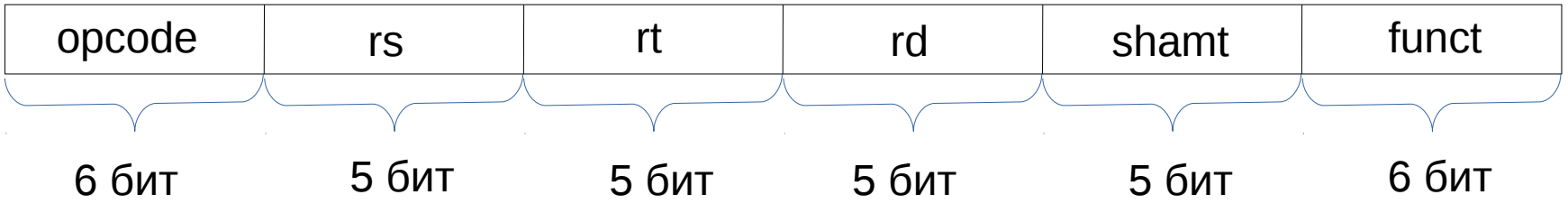
add $rd, $rs, $rt Записать в **rd** значение **$rs + $rt**

Как это выглядит в машинном коде:



sub $rd, $rs, $rt Записать в **rd** значение **$rs - $rt**

Инструкции R-типа



Сдвиговые инструкции

sll $rd, $rs, shamt
funct = [000000]

Взять значение **$rs**; сдвинуть его влево на **shamt** бит, заполняя правые биты нолями; результат записать в **rd**

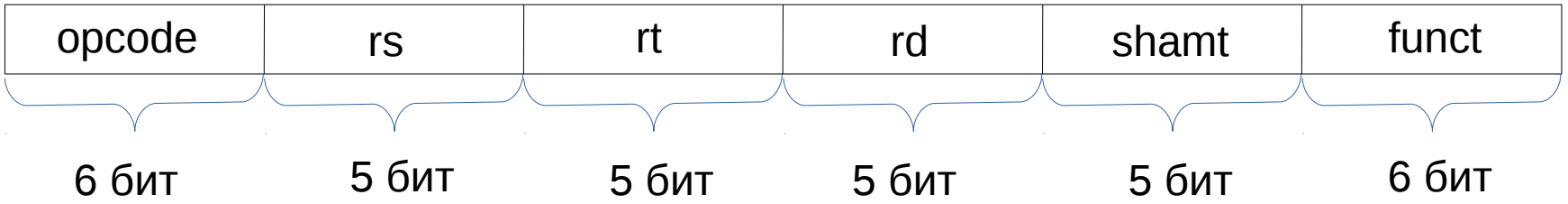
srl $rd, $rs, shamt
funct = [000010]

Взять значение **$rs**; сдвинуть его вправо на **shamt** бит, заполняя левые биты нолями; результат записать в **rd**

sra $rd, $rs, shamt
funct = [000011]

Взять значение **$rs**; сдвинуть его вправо на **shamt** бит, заполняя левые биты **знаком $rs** (его *левым битом*); результат записать в **rd**

Инструкции R-типа

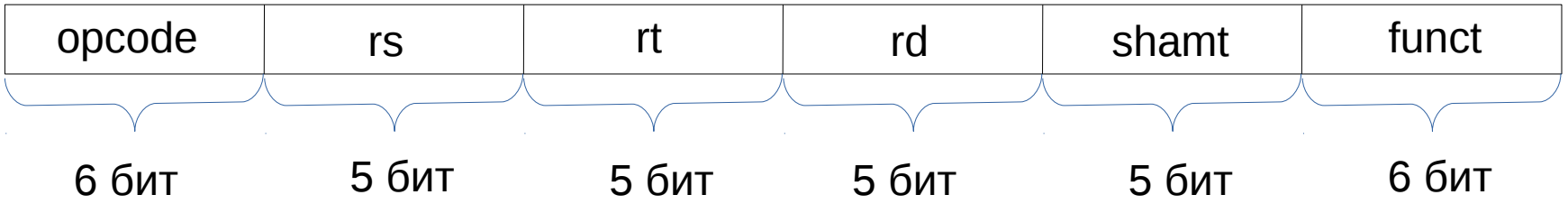


Побитовые логические инструкции

and $rd, $rs, $rt	**funct** = [10100]
nor $rd, $rs, $rt	**funct** = [10111]
xor $rd, $rs, $rt	**funct** = [10110]
or $rd, $rs, $rt	**funct** = [10101]

Записать в **rd** результат побитового применения операции (**and**, **nor**, **xor**, **or**) к значениям **$rs**, **$rt**

Инструкции R-типа



Инструкция сравнения знаковых чисел

slt $rd, $rs, $rt — если **$rs < $rt**, то записать в **rd** значение **[00...001]**, иначе записать в **rd** значение **[00..000]**

funct = [101010]

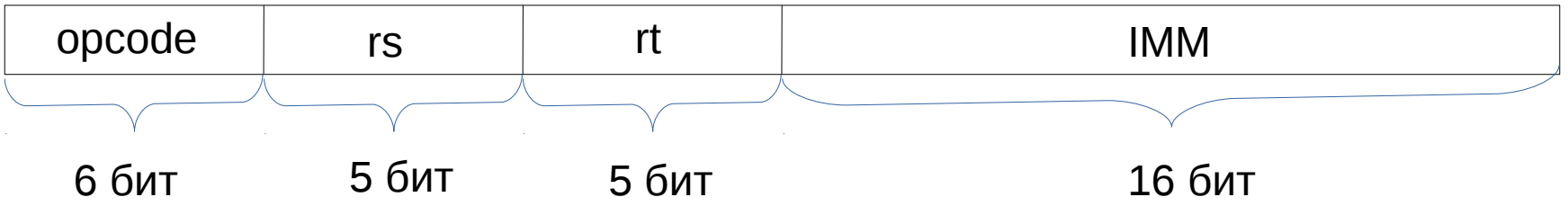
Есть и другие инструкции R-типа, например:

- беззнаковые аналоги знаковых операций
  - в ассемблерной записи в конец названия добавляется “**u**” (*например, **sltu***)
  - в **funct** правый бит становится “**1**” вместо “**0**”
- операции, которые пока затрагиваться не будут

(*надеюсь, что и потом не будут*)

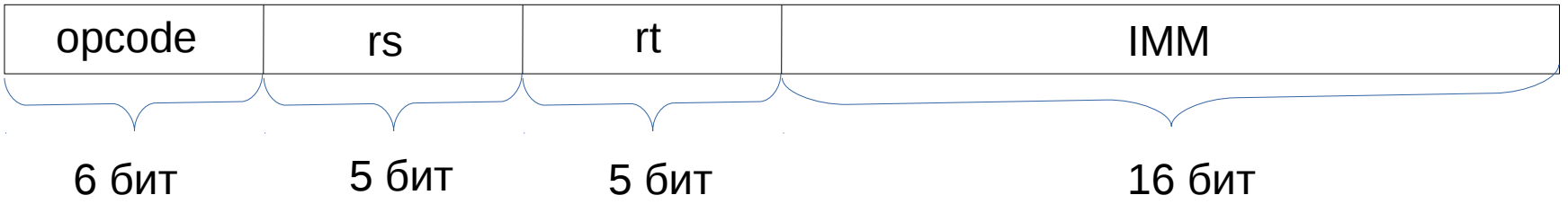
Инструкции I-типа

Все эти инструкции имеют такой формат:



- **opcode** – код операции
- **rs** – номер регистра, значение которого используется
- **rt** – номер регистра, в который пишется значение
- **IMM** – константа, используемая в операции

Инструкции I-типа



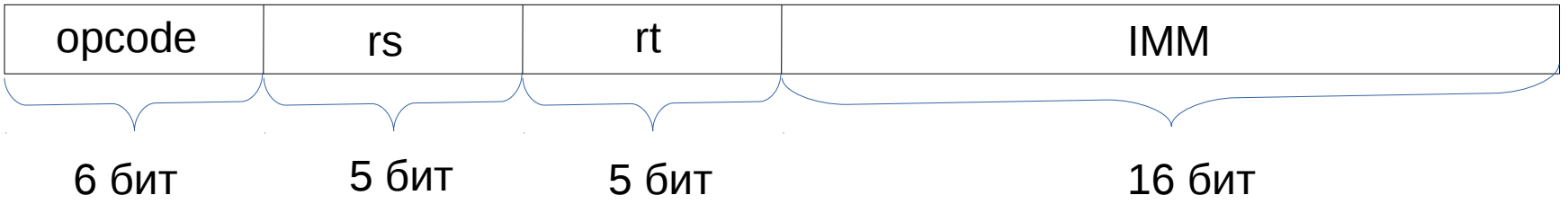
Арифметико-логические инструкции

addi $rt, $rs, IMM	**opcode** = [001000]
addiu $rt, $rs, IMM	**opcode** = [001001]
andi $rt, $rs, IMM	**opcode** = [001100]
ori $rt, $rs, IMM	**opcode** = [001101]
slti $rt, $rs, IMM	**opcode** = [001010]
sltiu $rt, $rs, IMM	**opcode** = [001011]

Это аналоги R-операций, из названия которых удалено “**i**”,
но вместо значения, записанного в регистр **rt**, берётся значение **IMM**

В беззнаковом случае **IMM** расширяется влево до 32 бит **нолями**, в знаковом - **знаком**

Инструкции I-типа



Инструкции ветвления

beq $rt, $rs, IMM Если **$rt == $rs**, то вместо **PC += 4** сделай **PC += (IMM << 2) + 4**

opcode = [000100]

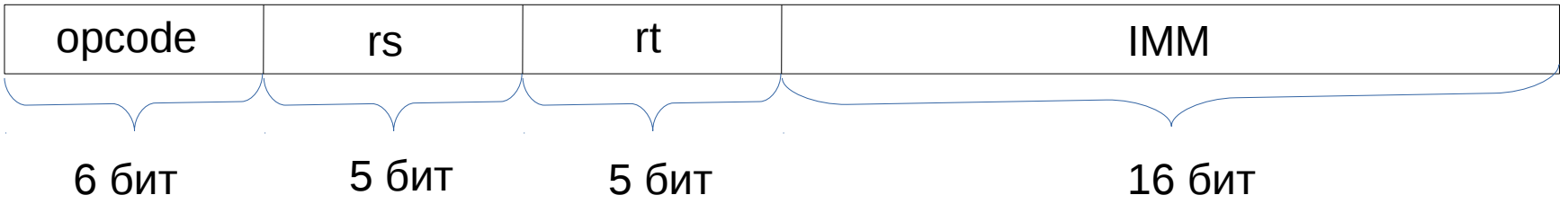
(то есть **IMM** – это то, на сколько **инструкций** следует продвинуться сверх положенного, если значения в регистрах совпадают)

bne $rt, $rs, IMM Если **$rt != $rs**, то вместо **PC += 4** сделай **PC += (IMM << 2) + 4**

opcode = [000101]

Здесь есть небольшие расхождения с тем, как всё работает в MIPS

Инструкции I-типа



Инструкции работы с памятью

lw $rt, IMM($rs) Загрузить в **rt** значение из памяти по смещению

opcode = [100011] **$rs + IMM байт**

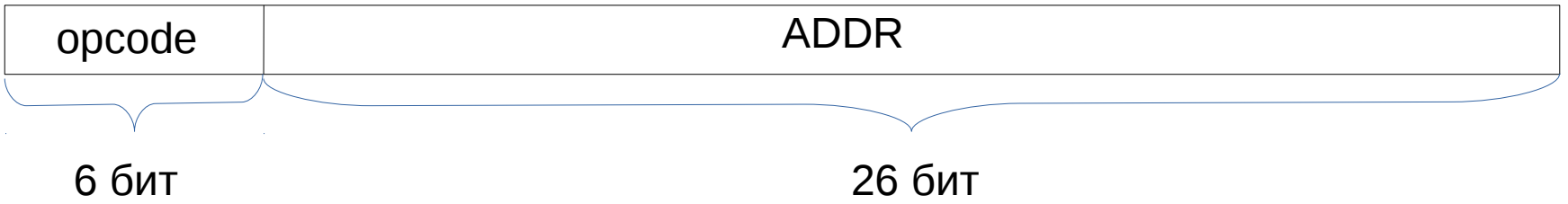
sw $rt, IMM($rs) Сохранить значение **$rt** в памяти по смещению

opcode = [101011] **$rs + IMM байт**

Как и раньше, есть ещё ряд инструкций, которые *пока* не затрагиваются

Инструкции J-типа

Все эти инструкции имеют такой формат:



Сейчас нам нужна одна такая инструкция:

j ADDR Записать в **PC** значение **[0000 ADDR 00]** – инструкция с этим смещением будет выполнена следующей

opcode = [000010]

Здесь есть небольшие расхождения с тем, как всё работает в MIPS

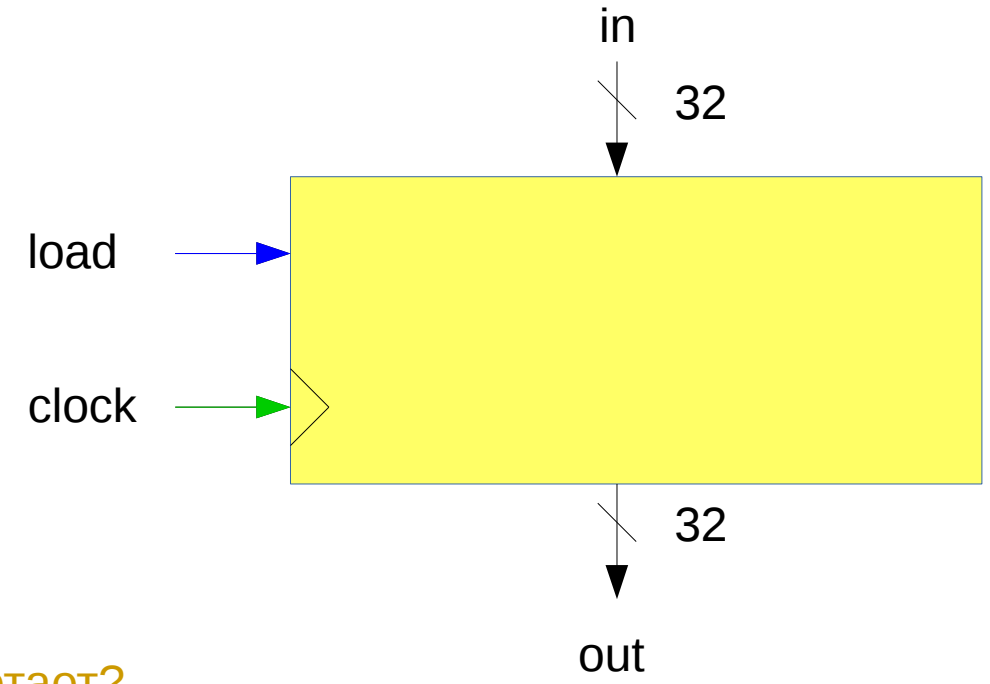
Операционный автомат процессора

И как же для этого всего построить операционный автомат?

Начать нужно с того, что

- внимательно посмотреть на все блоки процессора (*и памяти*),
  участвующие в пересылке и обработке данных
- понять, какие входные/выходные сигналы нужны в этих блоках
- разделить эти сигналы на управление и данные
- соединить проводами и дополнительными блоками эти сигналы так,
  чтобы можно было, посылая нужные управляющие сигналы,
  заставить данные преобразовываться так, как требуется инструкциями

Блоки: регистр



И как это работает?

Можно считать, что так:

```
always @(posedge clock)
  if(load) out <= in;
```

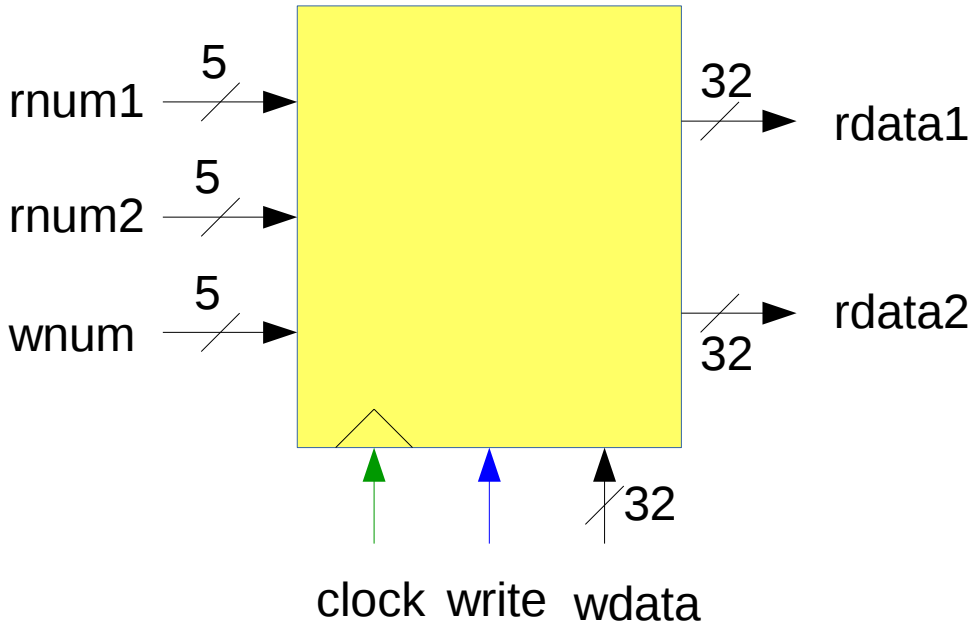
А как определить, где здесь управление и где данные?

Здесь и дальше **чёрные** стрелки – данные, а **синие** стрелки – управление

(*зелёная стрелка – clock – не управление, к ней подводится тактовый генератор*)

Блоки: набор регистров

Можно из регистров соорудить более объёмный блок:

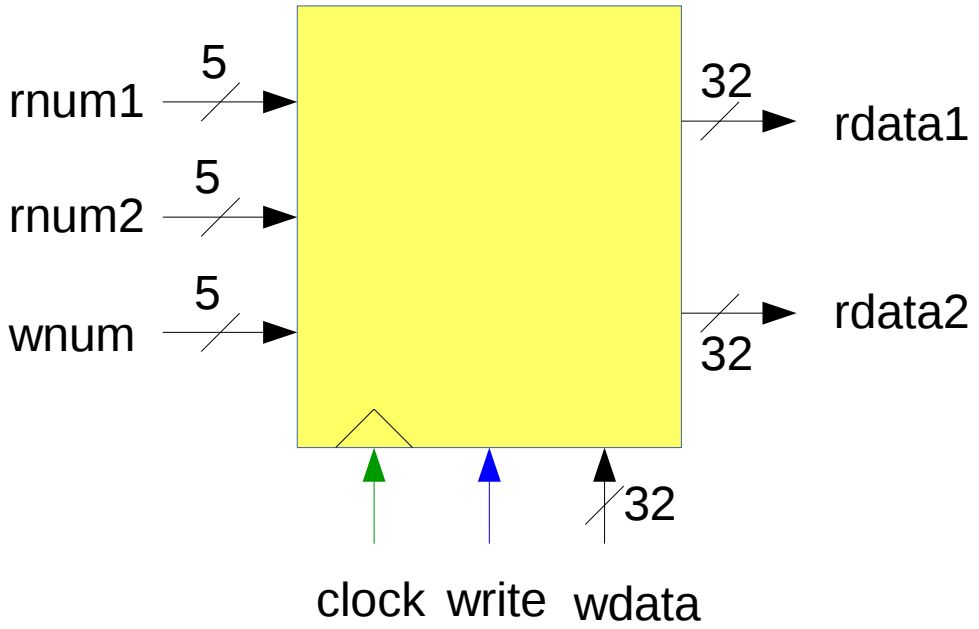


Шины данных:

- **rnum1**, **rnum2** – номера регистров, значения которых используются инструкцией
- **wnum** – номер регистра, в который производится запись по инструкции

Блоки: набор регистров

Можно из регистров соорудить более объёмный блок:

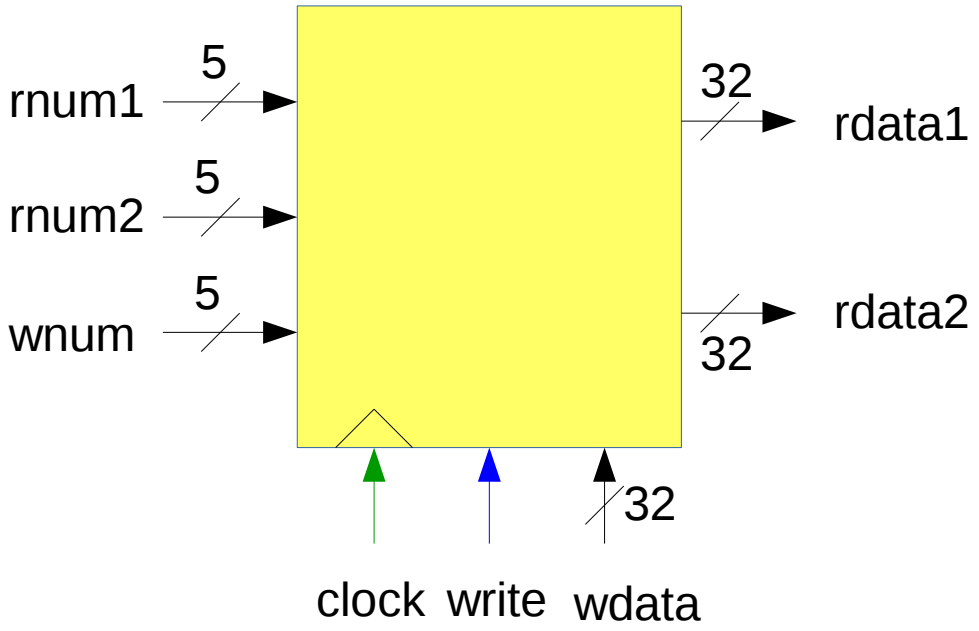


Шины данных:

- **rdata1**, **rdata2** – значения, хранящиеся в регистрах с номерами **rnum1**, **rnum2**
- **wdata** – значение, которое нужно записать в регистр с номером **wnum**

Блоки: набор регистров

Можно из регистров соорудить более объёмный блок:



Сигналы управления:

- **write** – нужно ли произвести запись

А как это устроено внутри?

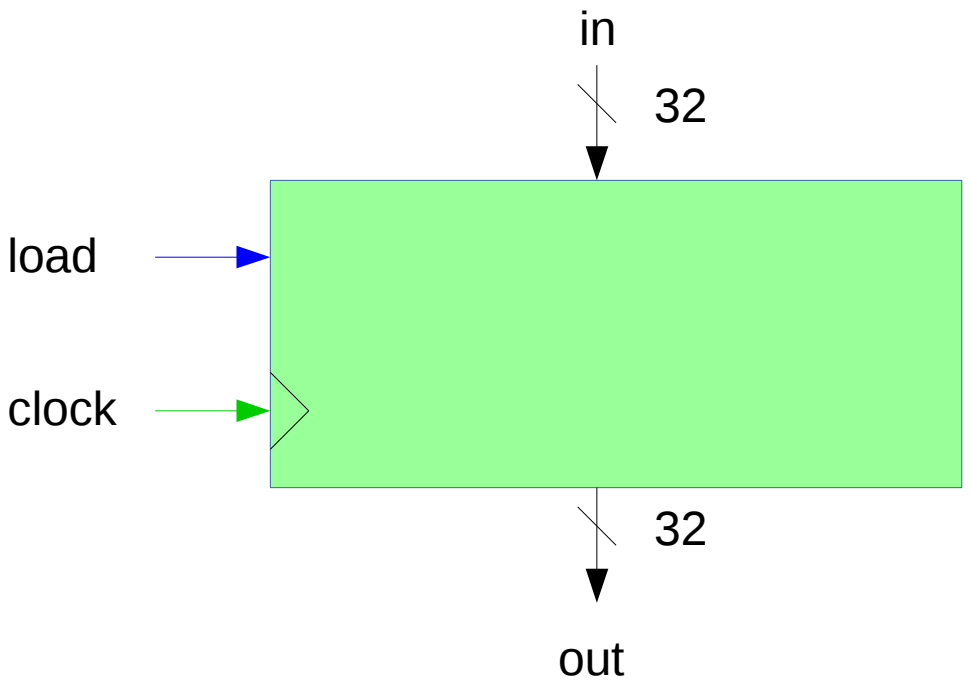
Так же, как дальше будет в рисунке про память

Блоки: счётчик команд

“Вшивать” ли в него “+4”?

Нет: тогда будет проблема с инструкциями ветвления

Проще сделать счётчик команд регистром,
а логику изменения его значений реализовать отдельно




```
always @(posedge clock)
  if(load) out <= in;
```

Блоки: память

В реальности память имеет довольно нетривиальное устройство

Но так как мы не хотим заниматься ей отдельно,
пока что можно её промоделировать **как можно более простым** блоком

Нужно определиться, какую функциональность мы хотим от памяти:

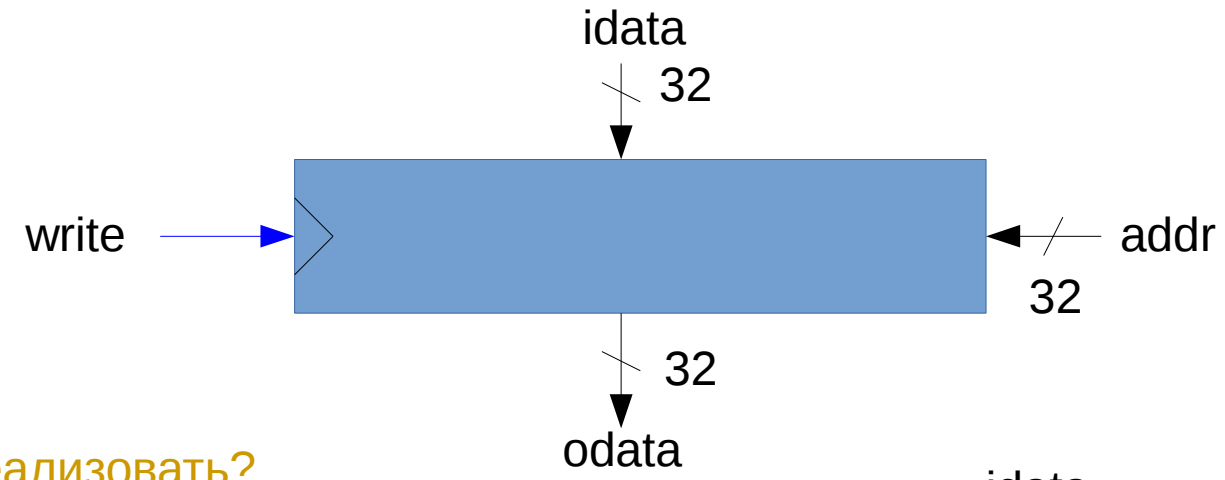
- уметь **читать** из себя **32-битовые слова**
- уметь **писать** в себя **32-битовые слова**
- уметь ничего не писать в себя

Тогда нужны такие сигналы:

- input [31:0] **addr** – двоичная запись смещения,
  информацию по которому хотим прочитать или записать
- output [31:0] **odata** – сюда непрерывно выдаются данные по смещению raddr
- input [31:0] **idata** – данные, которые мы хотим записать в память
- input **write** – аналог сигнала **clock** в регистре: по его переднему фронту
  записываем данные **idata** по смещению **addr**

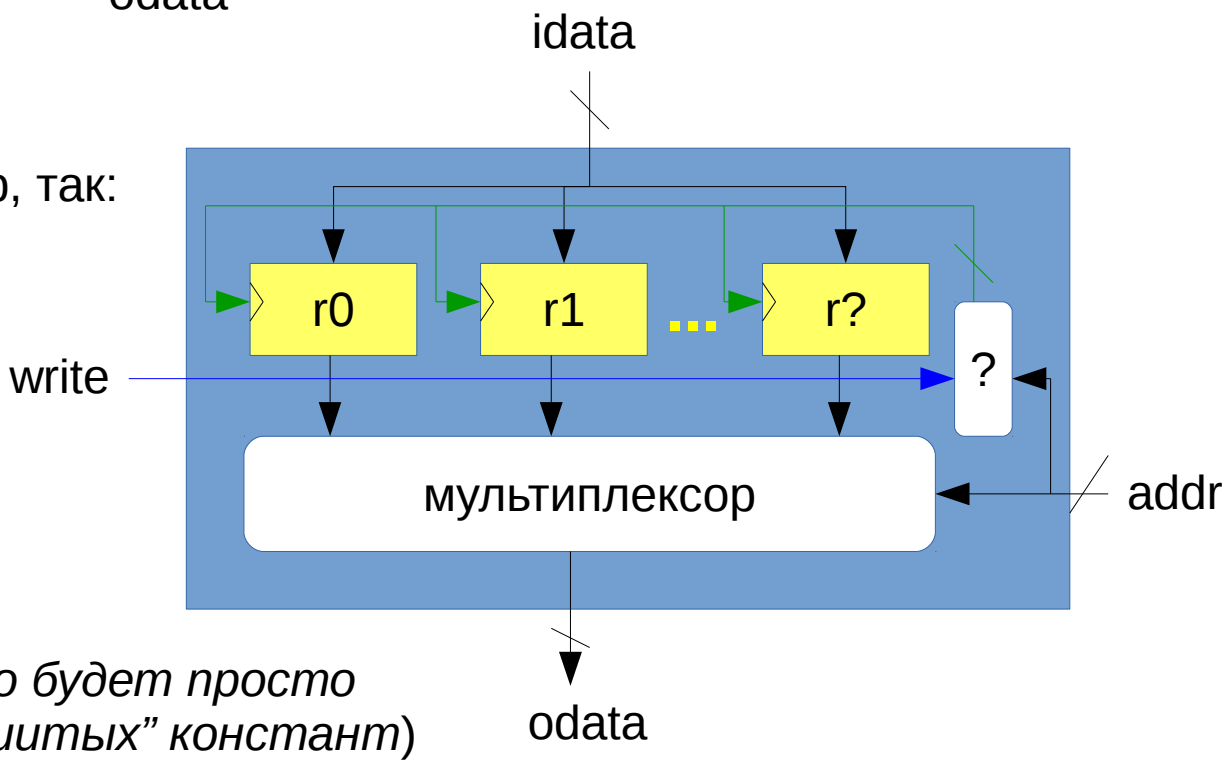
И как это разделить на данные и управление?

Блоки: память



А как это реализовать?

Например, так:

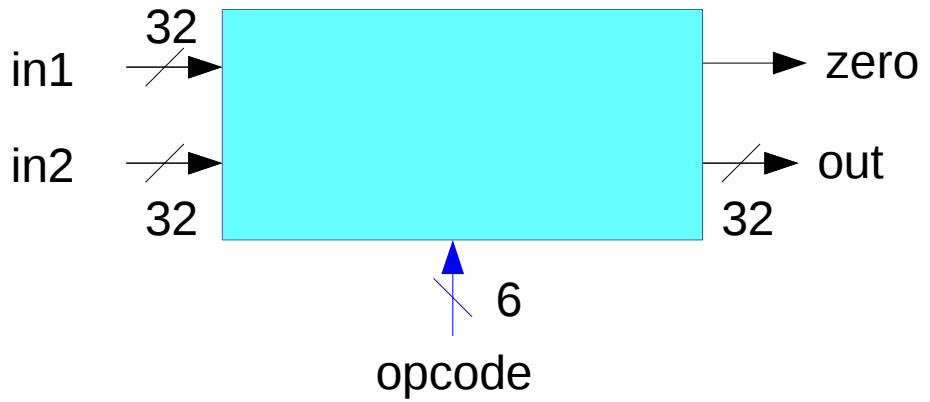


А если хотим ещё упростить себе жизнь, то можно сделать **две** памяти:

- одна – для данных
- другая – для инструкций

(*в ней нет write и idata, это будет просто мультиплексор с кучей “вшитых” констант*)

Блоки: АЛУ



А это как реализовать?

Если не гнаться за эффективностью, то, например, вот так:

```
always @(in1, in2, opcode)
  case(opcode)
  6'100000: out = in1 + in2;
  6'100010: out = in1 - in2;
  ...
  endcase
assign zero = out == 0;
```

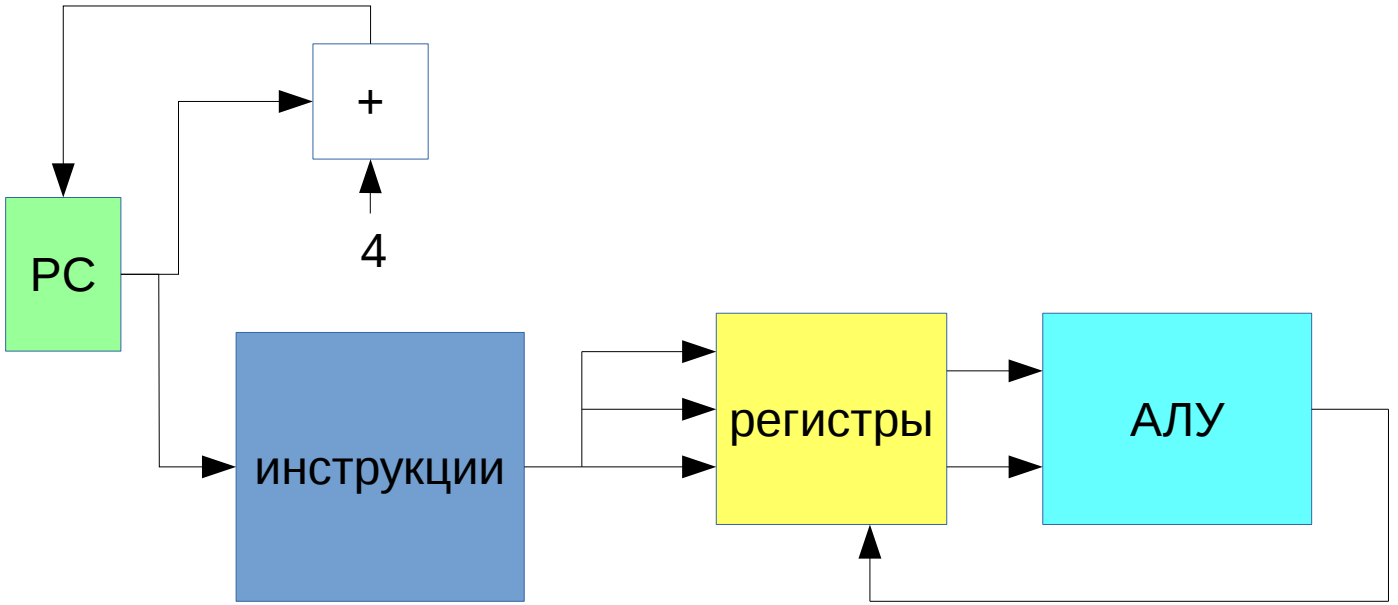
Операционный автомат

И как теперь соединить эти блоки вместе?

Это зависит от того, какой набор операций мы хотим, чтобы процессор поддерживал

В любом случае нужно соединить шинами **все** пары мест,
где сигнал с данными генерируется и где он используется

Например, если у нас есть только инструкция **add**:



(*как именно шины разделяются на подшины, я тут не пишу – додумайте сами*)

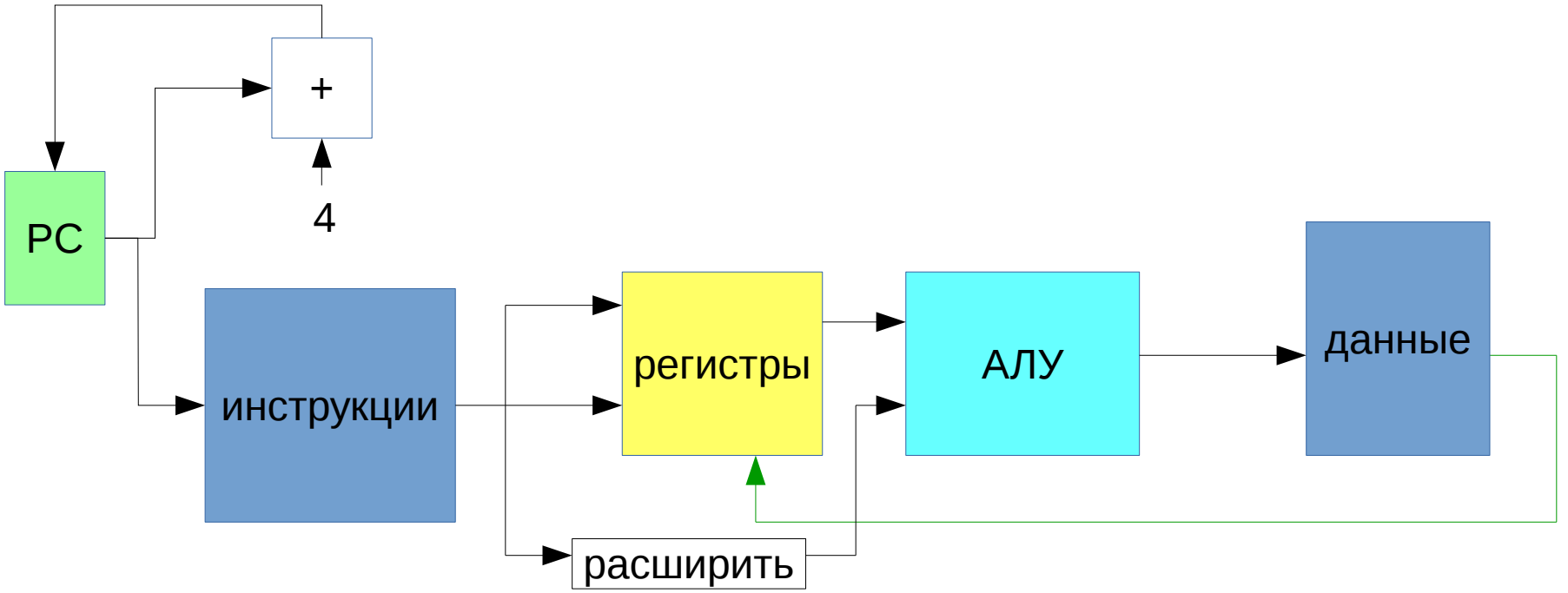
Операционный автомат

И как теперь соединить эти блоки вместе?

Это зависит от того, какой набор операций мы хотим, чтобы процессор поддерживал

В любом случае нужно соединить шинами **все** пары мест,
где сигнал с данными генерируется и где он используется

Немного сложнее, если у нас есть только инструкция **lw**:



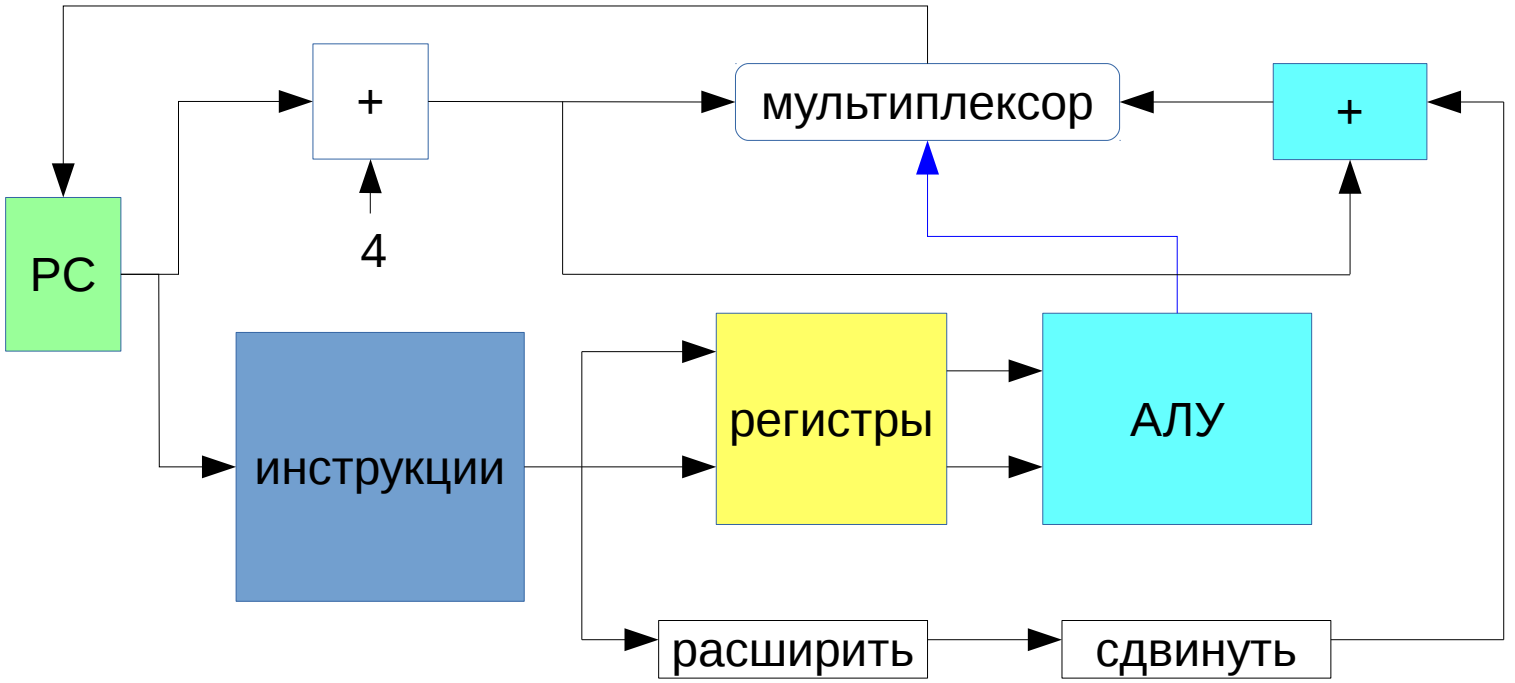
Операционный автомат

И как теперь соединить эти блоки вместе?

Это зависит от того, какой набор операций мы хотим, чтобы процессор поддерживал

В любом случае нужно соединить шинами **все** пары мест,
где сигнал с данными генерируется и где он используется

Ещё немного сложнее, если у нас есть только инструкция **beq**:

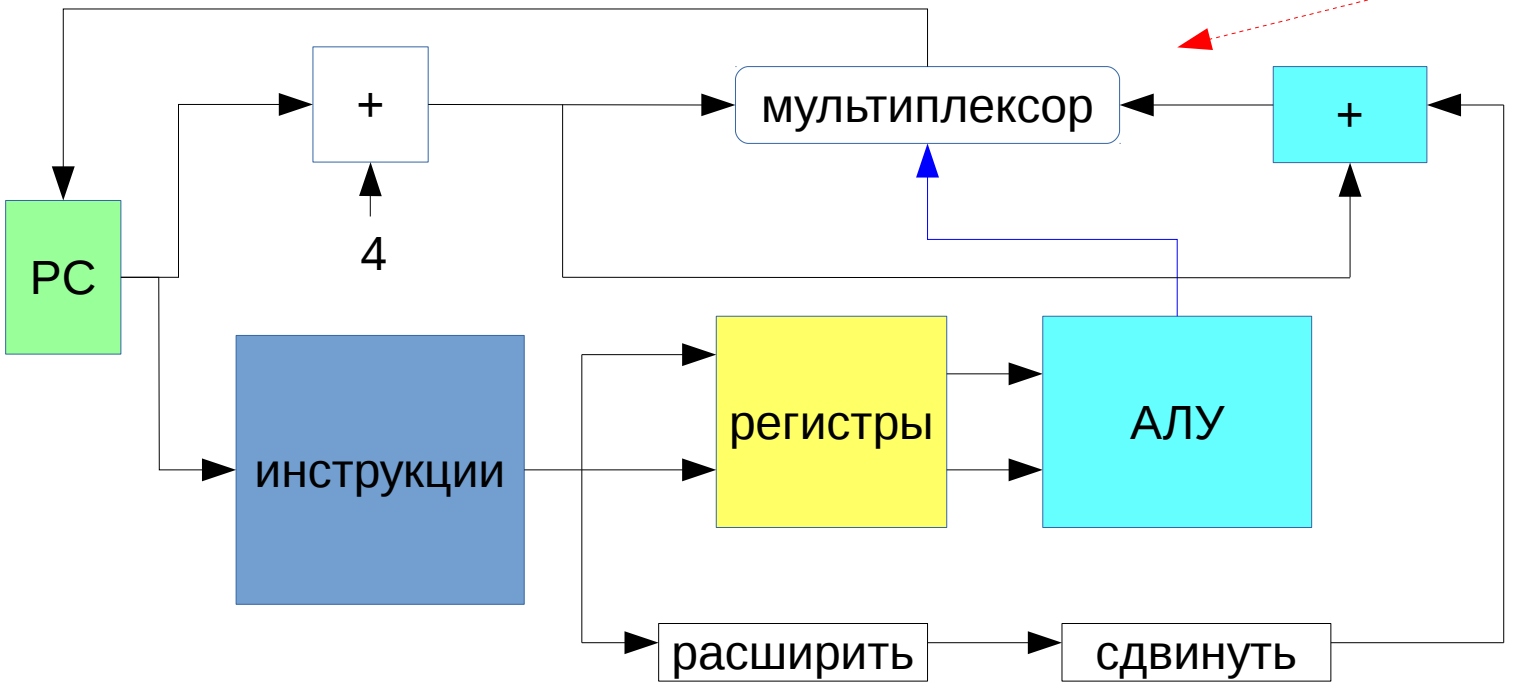


Операционный автомат

А как быстро и не задумываясь построить операционный автомат для нескольких инструкций, если имеется автомат для каждой отдельно?

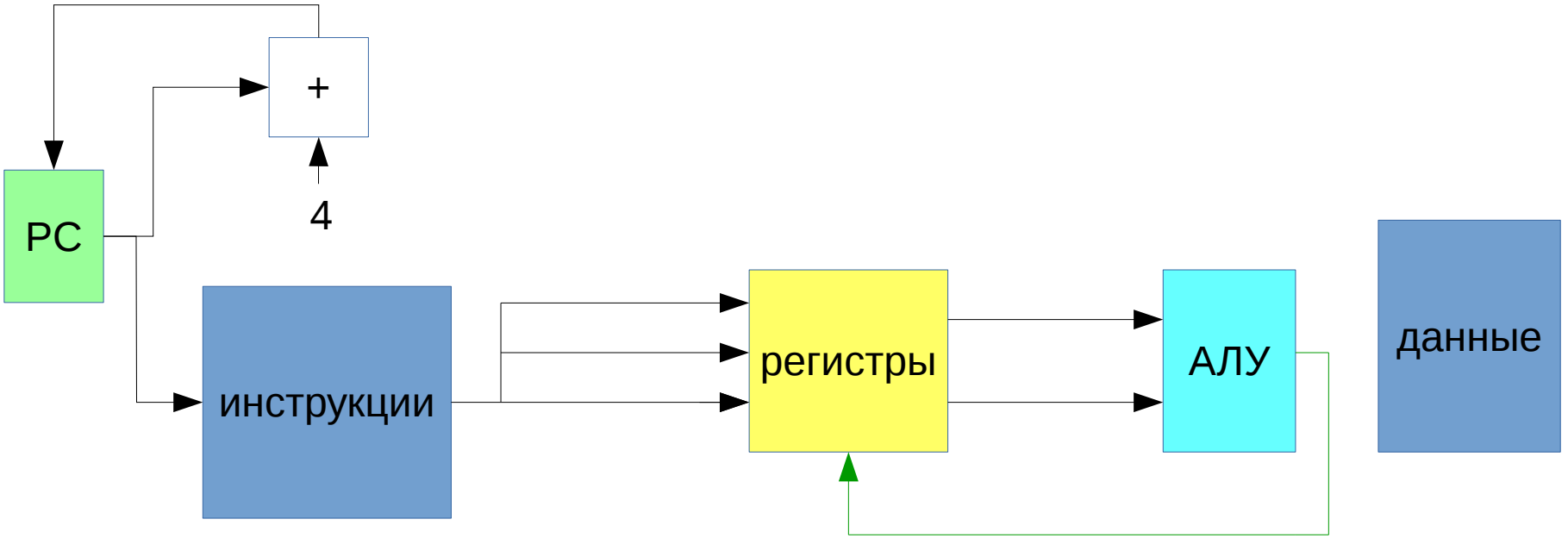
Если потоки данных не конфликтуют (*одинаковые или не пересекаются*), то просто рисуем всё, что есть

А если конфликтуют, то в точках конфликта можно поставить **мультиплексоры**, и пусть операционный автомат решает, кого когда выбирать



Операционный автомат

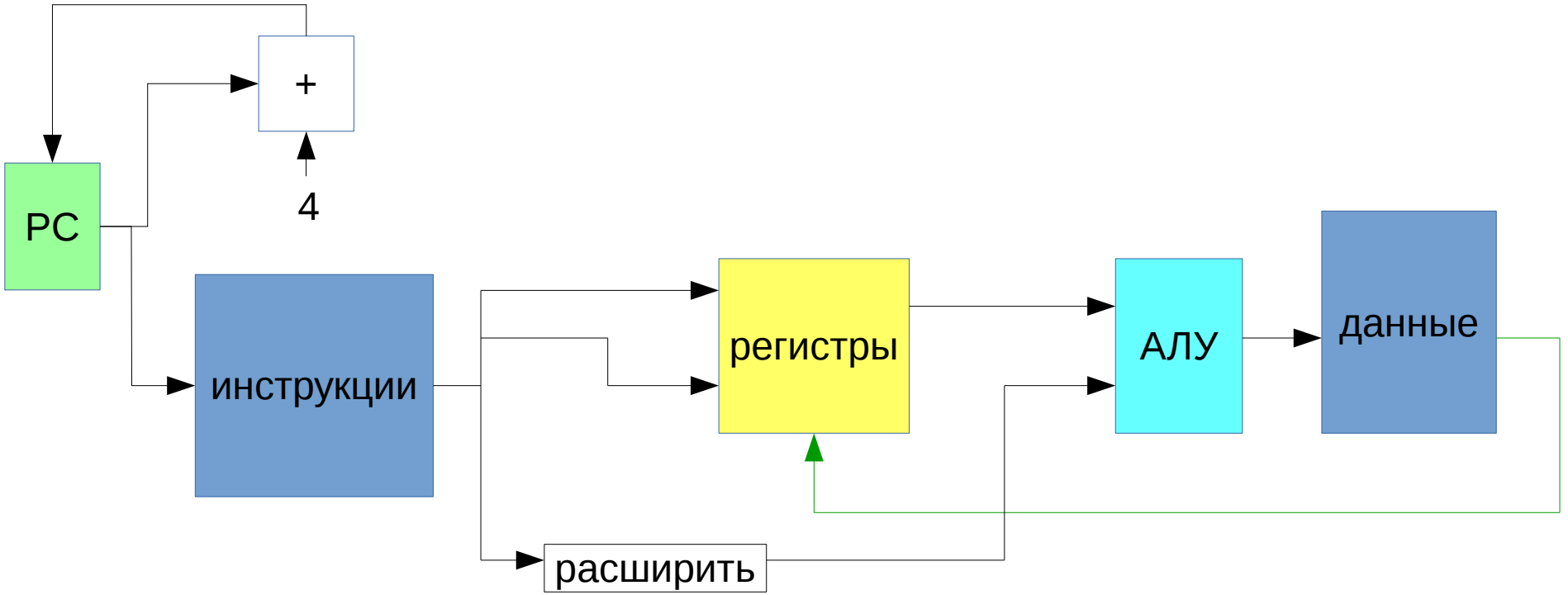
Например, если хочется поддержать одновременно **add** и **lw**:



Это add

Операционный автомат

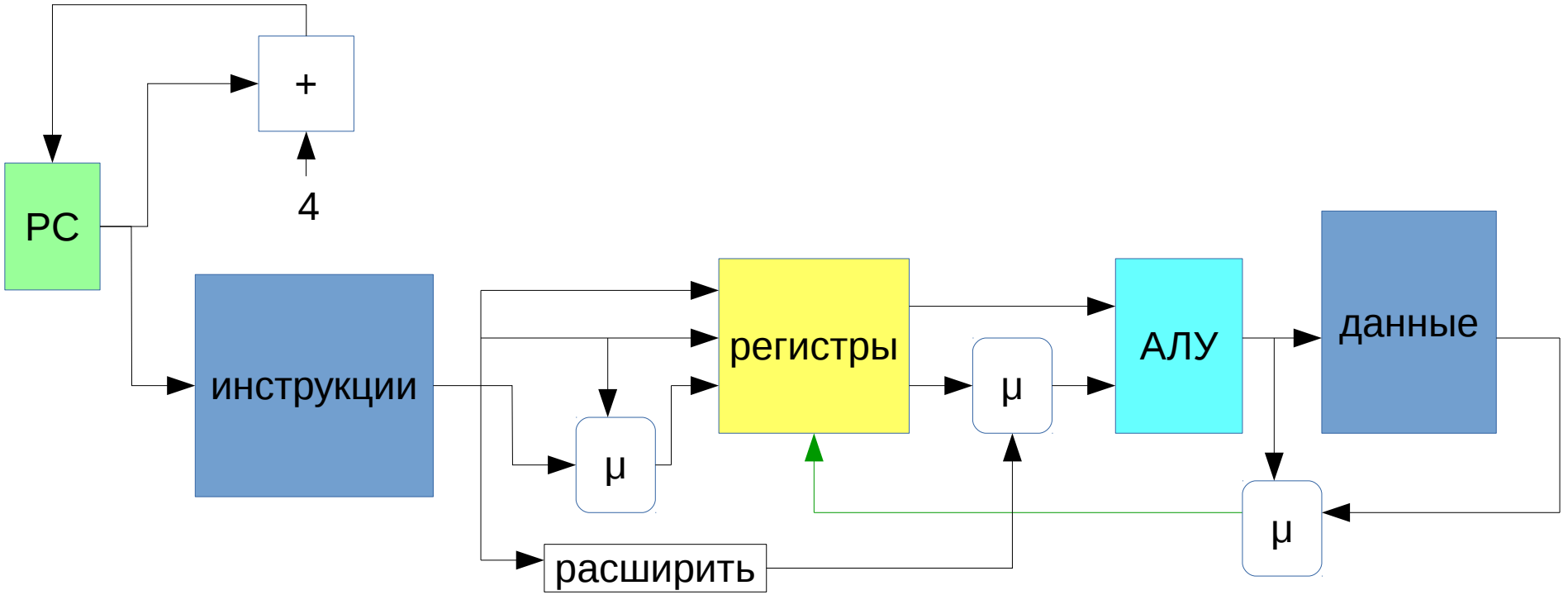
Например, если хочется поддержать одновременно **add** и **lw**:



Это lw

Операционный автомат

Например, если хочется поддержать одновременно **add** и **lw**:



Это add и lw