

# Computer Articecture

---

## Computer Articecture

Organization stuff

22-09-05

Ohm's law

Logic gate symbols

22-09-12

Architecture Types

I/O -- input/output

Machine word /TODO

RISC vs CISC

Multiplexer

Clock

22-09-19

Half adder

Adder

Two's complement

Пример

Subtractor

Comparator blocks

Сокращения

Операторы <, >

ALU (arithmetic-logic unit)

Counter

Memory array

Addresses

One-ported array

Multi-ported array

дз момент

Memory types

DRAM & SRAM

22-09-26

Big endian / Little endian

MIPS

MIPS: register set /todo

Типы инструкций

R-type

I-type

J-type

Примеры

Shift operations /todo

if-statement

Function call

Call stack  
Calling convention  
Automatic variables

22-10-04  
Реализация Load Word  
Single-cycle problems

22-10-10  
FSM (main controller)  
Fetch

22-10-24  
System Verilog  
Example: the appropriate scheme for a code  
Формат числа  
Элементарные операции  
А что если проводочеков больше, чем один?  
halfadder  
Sequential logic  
    always\_ff  
    always\_latch  
    always\_comb  
enum  
Блокирующее и неблокирующее присваивание

22-11-03  
AT&T syntax  
    EXIT\_SUCCESS, но в асселблере  
    Пишем hello world  
    циклы

22-11-07 \todo  
API & ABI  
Совместимость API

22-11-21  
A typical memory hierarchy  
Temporal and spatial locality  
Как устроен кэш  
Типы устройства кэша  
    1. Direct mapped cache  
        Реализация  
        Какие минусы?  
    2. Multi-way set associative cache  
    3. Fully associative cache  
        Пример: 4-канальный кэш с двумя сетами  
LRU replacement  
Изменение данных по адресу и обновление кэша  
    1. Write-through  
    2. Write-back

## Organization stuff

---

[chaika.konstantin.v@gmail.com](mailto:chaika.konstantin.v@gmail.com)

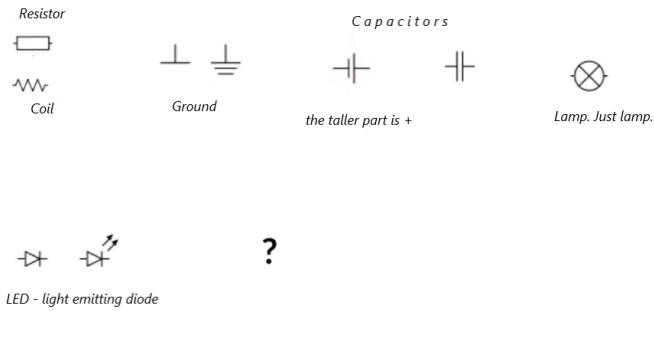
- start subject with [JUB-Arch] prefix
- in subject line: name, surname, theme

**22-09-05**

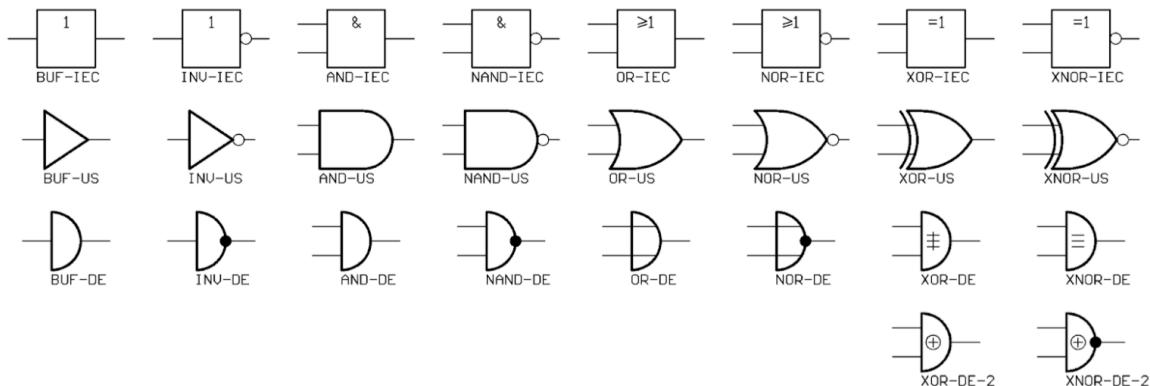
## Ohm's law

$R * I = U$ , where  $R$  - resistance (Om),  $I$  - current (A),  $U$  - Voltage (V)

## BASIC ELECTRONIC COMPONENTS



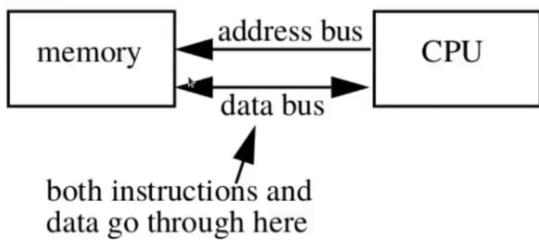
## Logic gate symbols



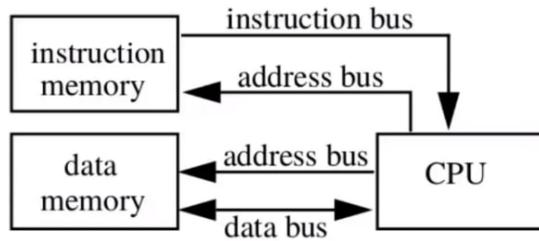
**22-09-12**

# Architecture Types

**Von Neumann Architecture**



**Harvard Architecture**



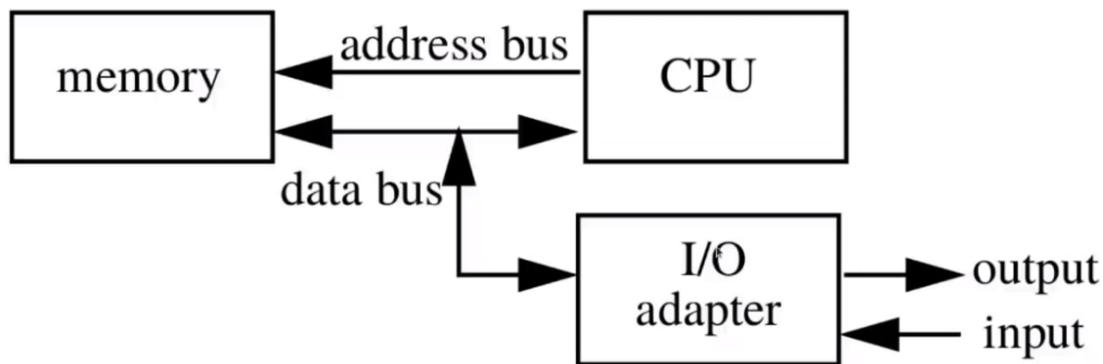
## ***Von Neumann Architecture:***

- Data and commands are stored in the same memory
- Commands and data are visually indistinguishable in memory

## ***Harvard Architecture:***

- Data is stored separately from commands
- You can handle both data and commands in parallel

## I/O -- input/output



## Machine word /TODO

CPU -- Central Processing Unit -- центральный процессор.

<https://habr.com/ru/company/selectel/blog/542074/>

## RISC vs CISC

### RISC vs CISC

Feature	RISC	CISC
Instruction size	1 word	1 to 54 bytes
Execution time	1 clock	1 to 100s of cycles
Addressing modes	small	large
Work done per instruction	small	varies
Instruction count/program	large	smaller

#### RISC:

- фиксированная длина инструкции
- инструкция целиком выполняется за один такт. Не нужно тратить энергию на сохранение промежуточных состояний

#### CISC:

- инструкции могут быть разной длины. Прежде чем выполнить команду, нужно за неограниченное время разобраться, что делает команда.
- 

## Multiplexer

Схема, у которой один/несколько гейтов на вход и только один на выход.

# Clock

Clock (CLK) -- защёлка. Какая-то штука, которая как-то генерируется и синхронизирует все гейты.

=0 => значение на выходе не меняется

=1 => значение меняется

**22-09-19**

---

## Half adder

Полусумматор, на входе не подается бит переноса

$$S = A \oplus B$$

## Adder

На вход подается бит переноса. Бит переносится, если в числе две единицы либо есть единица и старый бит переноса.

$$S = A \oplus B$$

$$C_{out} = A \& B + A \& C_{in} + B \& C_{in}$$

## Two's complement

aka обратный код

Фиксированное число битов. Первый бит показывает, какой знак у числа.

Bits	Unsigned value	Signed value (Two's complement)
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0111 1110	126	126
0111 1111	127	127
1000 0000	128	-128
1000 0001	129	-127
1000 0010	130	-126
1111 1110	254	-2
1111 1111	255	-1

*Как перевести положительное число в отрицательное?*

1. Инвертировать все биты

2. Добавить единицу

Путь такой, потому что иначе у нуля появился бы знак.

## Пример

$$n = 8$$

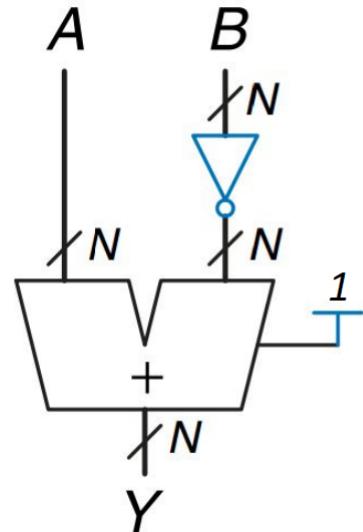
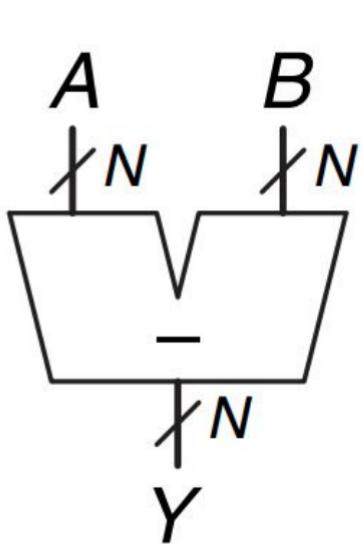
$$10_{10} = 00001010 \rightarrow 11110101 \rightarrow 11110110 = -10_{10}$$

$$-10 + 10 = +\begin{array}{r} 11110110 \\ 00001010 \end{array} = \underbrace{100000000}_8 \bmod (1 \ll n) = 0$$

## Subtractor

Вычитает из одного чиселка другое. Будем реализовывать вычитатель через сумматор: сначала найдем two's complement для B (инвертируем все биты + добавим 1), потом сложим.

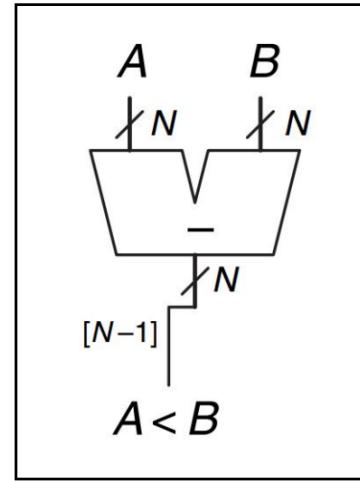
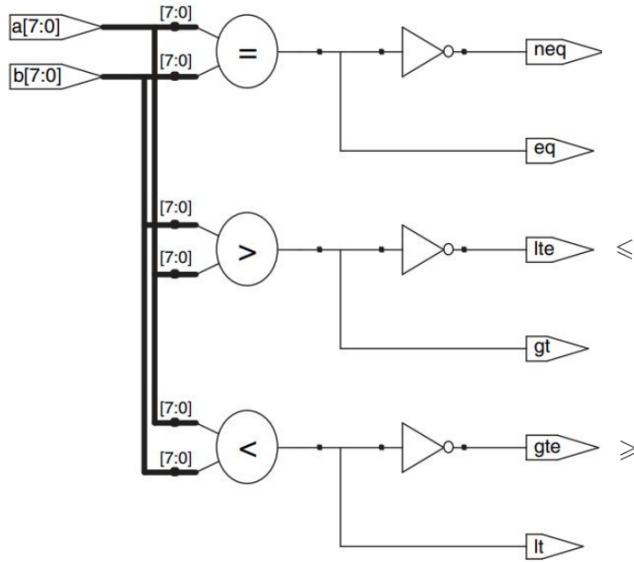
$$S = A + \overline{B} + 1$$



/ N значит, что на входе N-разрядное число

## Comparator blocks

### Сокращения

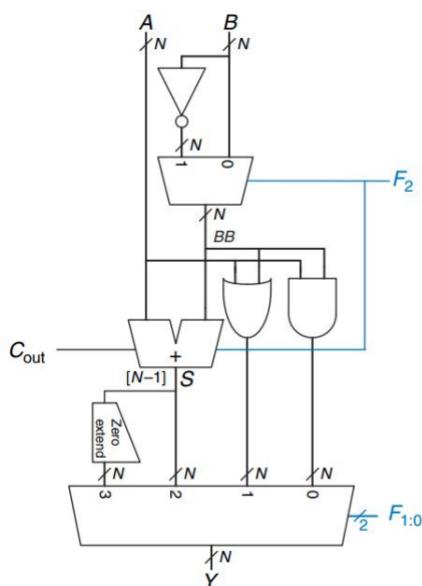


### Операторы $<$ , $>$

Можно вычесть из одного чиселка другое и посмотреть на старший бит -- знак результата

## ALU (arithmetic-logic unit)

Черные линии -- данные, синий -- управляющие сигналы:



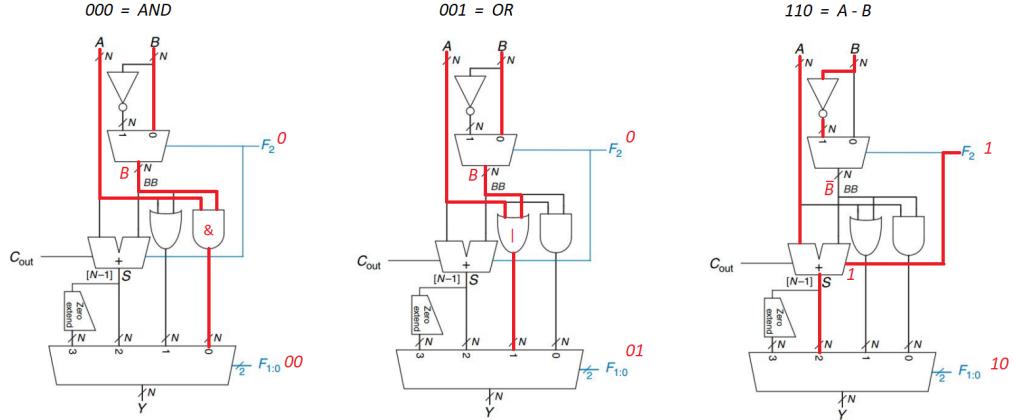
$F_{2:0}$	Function
000	$A \text{ AND } B$
001	$A \text{ OR } B$
010	$A + B$
011	not used
100	$A \text{ AND } \bar{B}$
101	$A \text{ OR } \bar{B}$
110	$A - B$
111	SLT (set less than)

Значение в верхней перевернутой трапеции:

- $F_2 = 0 \Rightarrow$  через трапецию проходит значение  $B$
- $F_2 = 1 \Rightarrow$  через трапецию проходит значение  $\bar{B}$

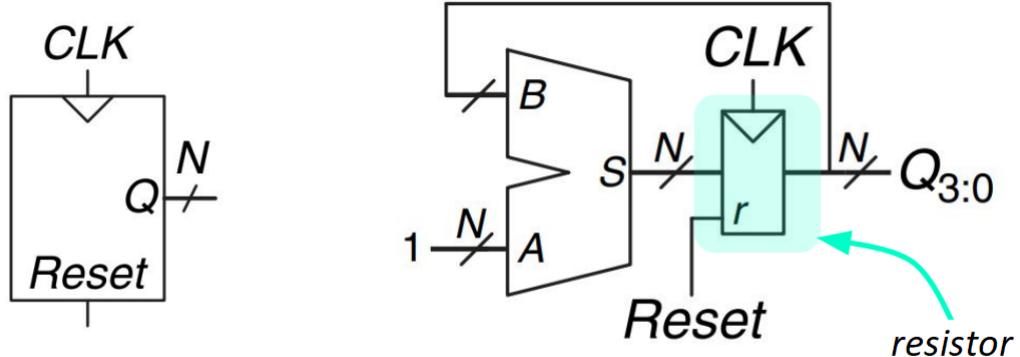
*Zero extend*: на вход подается  $(n - 1)$ -ый бит, добавляем его нулями

*SLT*: вычитаем  $B$  из  $A$ , отправляем старший бит в *zero extend*.



## Counter

Счетчик



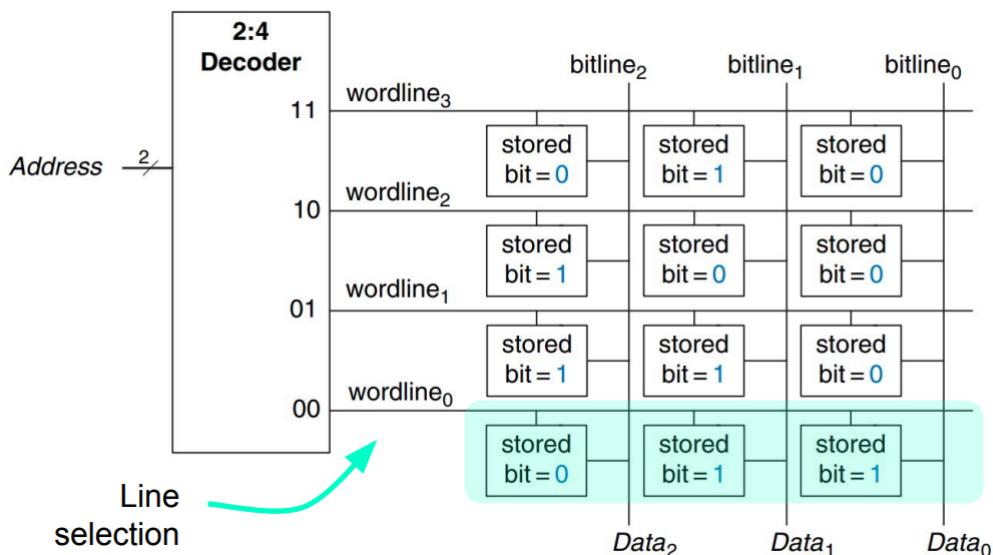
# Memory array

## Addresses

### One-ported array

Wordline -- линия машинных слов.

Есть шина адреса (2 проводочки). С помощью декодера выбираем, на какой wordline попадем. Сами wordlines не пересекаются.



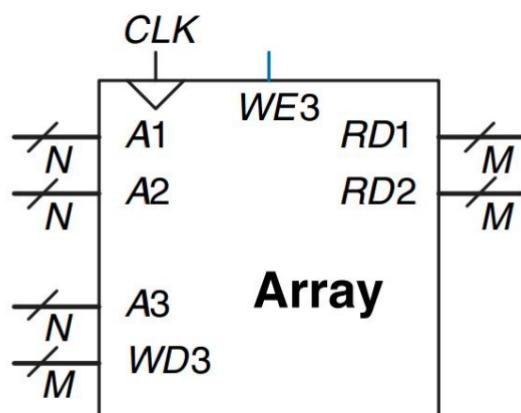
$$\text{Data}_2 = 0, \text{Data}_1 = 1, \text{Data}_0 = 1$$

На картинке линия однопортовая, потому что есть только один вход для адреса.

### Multi-ported array

RD == read data, WD == write data

WE3 -- Write Enable -- управляющий сигнал



В RD1 будет выстроено слово по адресу A1, в RD2 -- по адресу A2.

В WD3 будет что-то записано по адресу A3, если WE3 == 1.

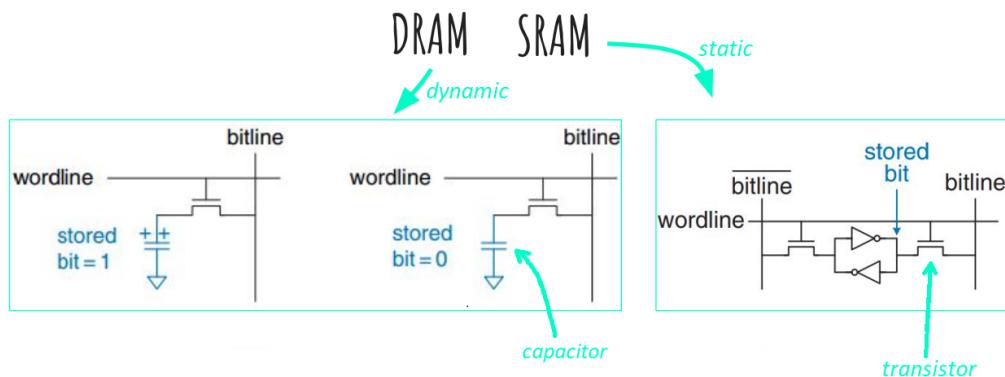
## ДЗ МОМЕНТ

- A memory can have more than one address bus and more than one data bus at the same time
- The maximum number of words stored in memory is determined by the bit depth of the address bus
- The address bus capacity and the data bus capacity can be different

## Memory types

### DRAM & SRAM

Capacitor -- конденсатор



Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

# Big endian / Little endian

Это про прямой и обратный порядок байтов

**Big-endian:** в начале младший байт, в конце старший

**Little endian:** в начале старший байт, в конце младший

# MIPS

*MIPS* (32-битная структура) -- система команд и микропроцессорных архитектур, разработанных в соответствии с концепцией проектирования процессоров RISC (то есть для процессоров с упрощенным набором команд). Позднее появились его 64-битные версии.

## MIPS: register set /todo

Регистр -- это, по сути, адрес в памяти.

Есть 32 регистра. Один из них всегда используется для нуля, остальные для любых нужных инструкций, но есть договоренность, для чего какой регистр используется.

Name	Number	Use
\$0	0	the constant value 0
\$at	1	assembler temporary
\$v0-\$v1	2–3	function return value
\$a0-\$a3	4–7	function arguments
\$t0-\$t7	8–15	temporary variables
\$s0-\$s7	16–23	saved variables
\$t8-\$t9	24–25	temporary variables
\$k0-\$k1	26–27	operating system (OS) temporaries
\$gp	28	global pointer
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	function return address

Обозначения:

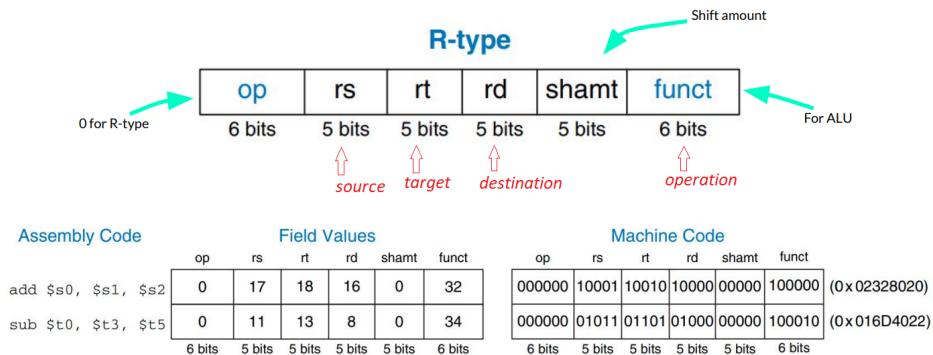
- `$r2` - регистр
- `(73)` - чиселко
- `-5($r2)` -- смещаемся влево (-) или вправо (+) от адреса, в котором хранится регистр `$r2`

## Типы инструкций

*Immediates* -- constants which values are immediately available from the instruction and do not require a register or memory access

### R-type

инструкция для триады регистров ( R : \$x, \$y, \$h ). Имеет размер 32 бита.

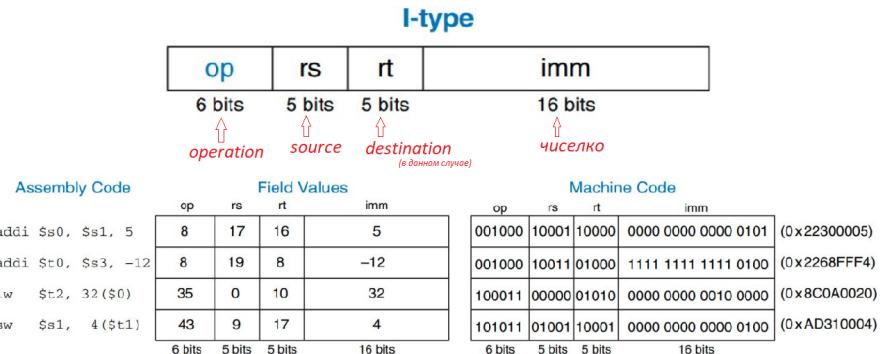


\$s0 = 16 -- destination, \$s1 = 17, \$s2 = 18 (см. таблицу выше)

### I-type

инструкция для пары регистров ( I: \$x, \$y ). I == immediate

Значение доступно сразу после выполнения команды (поэтому immediate). Имеет размер 16 битов.



В rt записывается результат

Если op = 8, под капотом операция записывается на ALU.

i в конце операции означает, что это команда и что она выполняется над каким-то адресом.

числа 32-битные, но для I-type у чисел 16 битов. immediate знаковое, поэтому есть механизм знакового расширения: если старший разряд = 1 -- добавляем единицами в начале; если старший разряд = 0 -- добавляем нулями.

`lw` == `load word` -- загружает слово из памяти

`sw` == `save word` -- сохраняет слово в памяти

- хотим взять значение в `$s1 = 17` и записать в адрес `4($t1)` -- адрес регистра `$t1` + смещение на 4 байта

### J-type

Имеет размер 26 битов. J == (jump)

### J-type

op	addr
6 bits	26 bits

## Примеры

```
addi $s0, $0, 4      # $s0 = 0 + 4 = 4
addi $s1, $0, 1      # $s1 = 0 + 1 = 1
sll  $s1, $s1, 2      # $s1 = 1 << 2 = 4
beq $s0, $s1, target # $s0 == $s1, so branch is taken
addi $s1, $s1, 1      # not executed
sub  $s1, $s1, $s0      # not executed

target:
add  $s1, $s1, $s0      # $s1 = 4 + 4 = 8
```

## Shift operations /todo

**sll** -- shift left logical -- сдвигает все биты на один влево

**srl** -- shift right logical -- сдвигает все биты на один вправо. Старший бит становится равным нулю

**sra** -- shift right arithmetic

## if-statement

### High-Level Code

```
if (i == j)
    f = g + h;
f = f - i;
```

### MIPS Assembly Code

```
# $s0 = f, $s1 = g, $s2 = h, $s3 = i, $s4 = j
bne $s3, $s4, L1      # if i != j, skip if block
add $s0, $s1, $s2      # if block: f = g + h
L1:
sub $s0, $s0, $s3      # f = f - i
```

## Function call

Когда вызывается функция, на стек кладется указатель на начало её исполнения. Формируется stack frame.

## Call stack

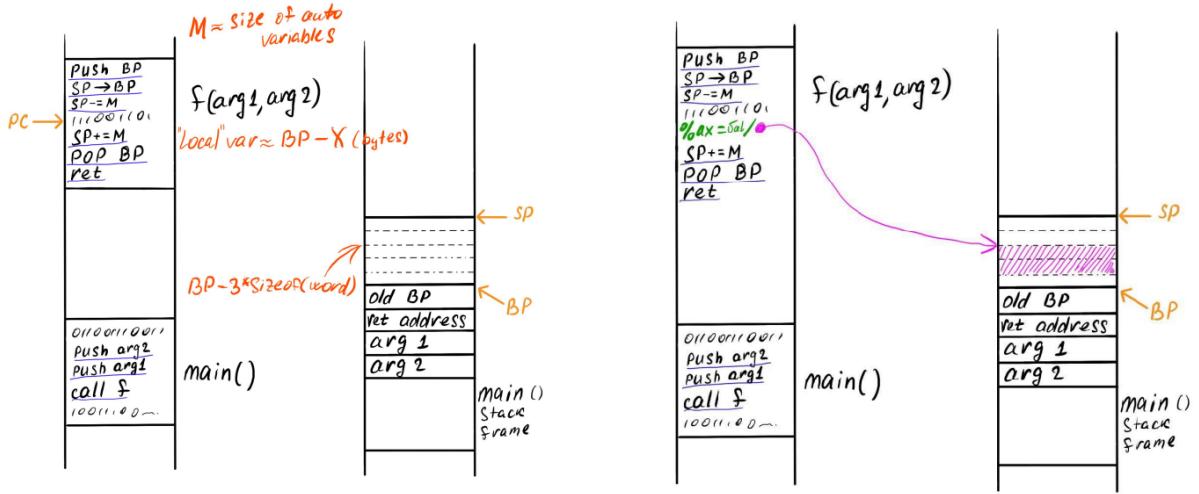
### Calling convention

Соглашение о порядке, в котором переменные/стек-фреймы кладутся на стек.

**Пролог функции** -- стек-фрейм появляется на стеке; **эпилог функции** -- стек-фрейм убирается со стека.

## Automatic variables

**Automatic variable** -- a local variable which is allocated and deallocated automatically when a program flow enters and leaves the variable's scope.



Можно вручную резервировать место на стеке, сдвигая указатель на начало стека. Так гарантируем, что на стеке будет место для новых переменных, например.

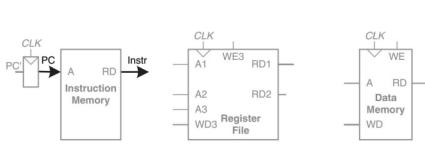
Сначала делаем `-=M`, потом `+M`, потому что стек заполняется от наибольшего адреса к наименьшему.

## 22-10-04

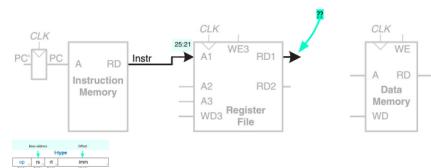
### Реализация Load Word

1. Fetch LW instruction from memory
2. Read source operand from register file
3. Sign-extend the immediate
4. Compute memory address
5. Write data back to register file

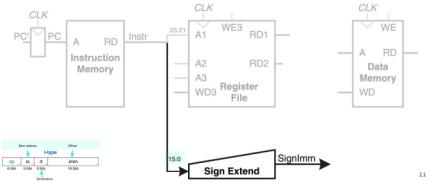
### 1. FETCH LW INSTRUCTION FROM MEMORY



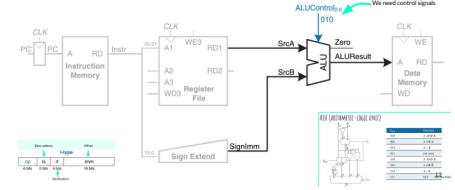
### 2. READ SOURCE OPERAND FROM REGISTER FILE



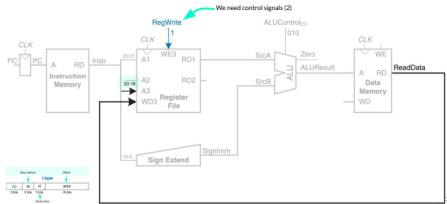
### 3. SIGN-EXTEND THE IMMEDIATE



### 4. COMPUTE MEMORY ADDRESS



### 5. WRITE DATA BACK TO REGISTER FILE



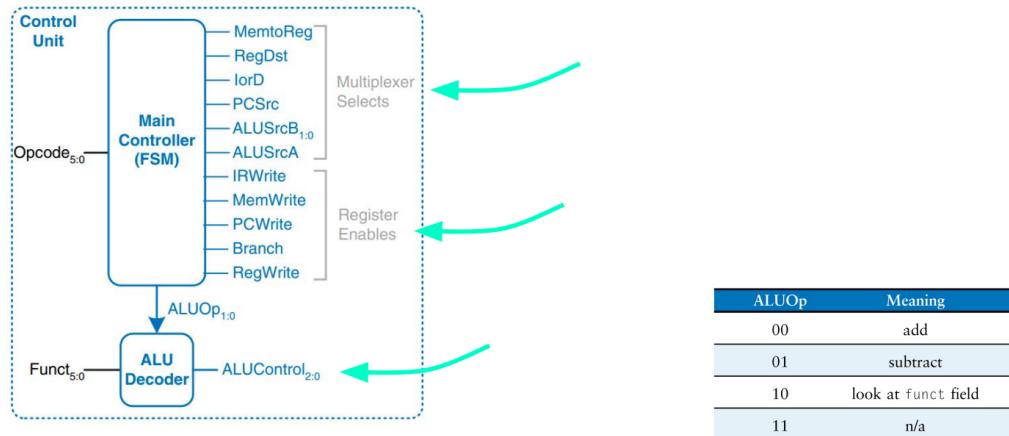
## Control unit

Первые 6 битов -- код операции.

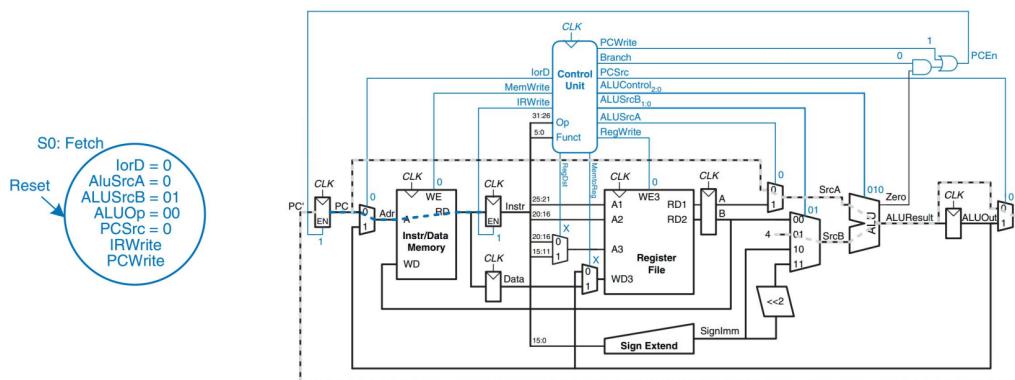
## Single-cycle problems

- The single-cycle MIPS processor uses separate command and data memory.
- The clock signal period must be long enough for the slowest command to execute. The slowest command is `lw`.
- Several adders (especially fast) make the circuit more expensive.

## FSM (main controller)



## Fetch



22-10-24

## System Verilog

Язык, используемый для описания и моделирования электронных систем. Расширение `.sv`

```

1 module f(input logic a, b, c, output logic y);
2     assign y = a & b & c | ~a & b;
3 endmodule

```

`module` -- ключевое слово, обозначающее компоненту.

```
1 | iverilog -g2012 aboba.sv
2 | ./a.out
```

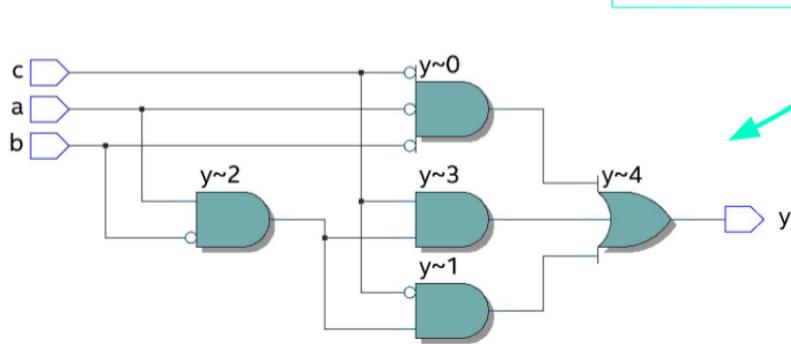
Скомпилировать и запустить файлы

## Example: the appropriate scheme for a code

```
module sillyfunction(input logic a, b, c,
                      output logic y);

    assign y = ~a & ~b & ~c |
              a & ~b & ~c |
              a & ~b & c;

endmodule
```



## Формат числа

[width]'[base][value]

[base] может быть b, h, d

Запись [n:m] означает взять биты с m по n включительно.

## Элементарные операции

```

1 module gates(input logic [3:0] a, b,
2               output logic [3:0] y1, y2, y3, y4, y5);
3   assign y1 = a & b; // AND
4   assign y2 = a | b; // OR
5   assign y3 = a ^ b; // XOR
6   assign y4 = ~(a & b); // NAND
7   assign y5 = ~(a | b); // NOR
8 endmodule

```

## А что если проводочков больше, чем один?

```

1 module invert(input logic [3:0]a, output logic [3:0]y);
2   assign y = ~a;
3 endmodule

```

Если размерности входа/выхода не сойдутся, будет ошибка на уровне компиляции

```

1 module and8(input logic [7:0]a, output logic [7:0]y);
2   assign y = &a; // <=> a1 & a2 & ... & a8
3 endmodule

```

## halfadder

Полусумматор

```

1 // halfadder.sv
2
3 module halfadder(input logic a, b,
4                   output logic s, cout);
5   assign s = a ^ b; // XOR
6   assign cout = a & b;
7 endmodule

```

```

1 // test.sv
2
3 module test;
4
5   // объявление переменных
6   reg a;
7   reg b;

```

```

8   wire s;
9   wire cout;
10  integer i;
11
12 // создание функции и привязка переменных к ее аргументам
13 // `ha0` -- имя функции
14 halfadder ha0(.a (a), .b (b), .s(s), .cout (cout));
15
16 initial begin
17   a <= 0;
18   b <= 0;
19   $monitor ("%0h", a) // <=> print
20   for (i = 0; i < 50; i = i+1) begin
21     #10 a <= $random;
22     b <= $random;
23   end
24 end
25 endmodule

```

#10 означает "подожди 10 мс". Если убрать, то значения не успевают перезаписываться.

## Sequential logic

### **always\_ff**

```

1 always_ff @(posedge clk)
2   q <= d;

```

ff от flip-flop, posedge от positive edge.

Значение q поменяется, только когда clk меняется с нуля на единицу.

Есть ключевое слово negedge, работает наоборот.

### **always\_latch**

```

1 always_latch
2   if (clk) q <= d;

```

блок выполняется всегда, если защелка закрыта.

## always\_comb

Можно задать таблицу истинности

```
1 module test(input logic [3:0] a, output logic [3:0] y);
2     always_comb
3         casez(a)
4             4'b01???: y <= 4'b1000;
5             4'b1010: y <= 4'b1001;
6             default: y <= 4'b1111;
7         endcase
8     endmodule
```

## enum

```
1 typedef enum logic [2:0] {s0, s1, s2, s3 = {111}} statetype;
2 statetype aboba;
```

`s0 = 000, s1 = 001, s2 = 010, s3 = 111.`

`statetype` -- имя enum-класса.

`aboba` -- число из трёх битов, принимающее значение из `statetype`: `s0`, `s1`, `s2` или `s3`.

## Блокирующее и неблокирующее присваивание

`=` --- блокирующее

```
1 q = a ^ b
2 p = q | b
```

Операции выполняются последовательно. Сначала изменится `q`, потом оно используется в строке 2.

`<=` --- неблокирующее

```
1 q <= a ^ b
2 p <= q | b
```

Операции выполняются одновременно. Сначала выполняются выражения справа, потом произойдет переписывание значения.

В строке 2 используется старое значение `q`.

## 22-11-03

### AT&T syntax

`$label` -- подстановление значения переменной `label` (?)

`$1` -- чиселко 1

`%eax` -- название регистра, начинается с `%`

`mov, call, int, jmp` -- команды

`mov $5, %eax` <=> в регистр `eax` запишется значение `5`

[https://faculty.nps.edu/cseagle/assembly/sys\\_call.html](https://faculty.nps.edu/cseagle/assembly/sys_call.html)

### EXIT\_SUCCESS, но в ассемблере

```
1 mov $1, %eax
2 xor %ebx, %ebx ; чистим регистр
3 int $0x80
4 ret
```

### Пишем hello world

```
1 .data
2     msg1:
3         .ascii "Computer Architecture Course\n/Autumn semester\\"
4
5     msg1len:
6         .long . - msg1
7
8 .text
9
10 .globl main
```

```

11 main:
12     mov $4, %eax
13     mov $1, %ebx
14     mov $msg1, %ecx
15     mov msg1len, %edx
16     int $0x80
17
18     mov $1, %eax
19     xor %ebx, %ebx
20     int $0x80
21     ret
22
23
24 helloprinter: ; функция
25     mov $4, %eax
26     mov $1, %ebx
27     mov $aboba, %ecx
28     mov len, %edx
29     int $0x80
30     ret

```

.data и .text -- отдельные блоки с данными

aboba: -- метка; используется, чтобы обратиться по адресу, где сохранена строка

int \$0x80 -- запускает записанную функцию

Вместо строк 12-16 можно написать call helloprinter

```

1 .data
2     msg1:
3         .ascii "Computer Architecture Course\n/Autumn semester\\n"
4     len:
5         .long . - msg1
6
7 .text
8
9
10
11
12 .globl main
13 main:
14     call printer
15     mov $1, %eax
16     xor %ebx, %ebx
17     int $0x80
18     ret
19
20 printer:
21     mov $4, %eax
22     mov $1, %ebx
23     mov $msg1, %ecx
24     mov len, %edx
25     int $0x80

```

## ЦИКЛЫ

```
1 .globl main
2 main:
3     xor %esi, %esi
4     mov $0, %esi
5     loop:
6         cmp $123, %esi
7         je end
8         call printer
9         add $1, %esi
10        jmp loop
11    end:
12    mov $1, %eax
13    xor %ebx, %ebx
14    int $0x80
15    ret
```

Начинаем с нуля, прыгаем в начало `loop`, пока значение в `esi` меньше 123.

## 22-11-07 \todo

### API & ABI

API -- Application Programming Interface -- всякие функции и переменные, хранящиеся в библиотеке.

API используется разработчиком.

ABI -- Application Binary Interface -- "скомпилированный" API. Хранит информацию о месте функции в памяти; какие аргументы функция принимает и возвращает; как чистится память после использования. Еще отвечает за padding и endiannes.

ABI используется компилятором. То, как будет выглядеть ABI, зависит от платформы.

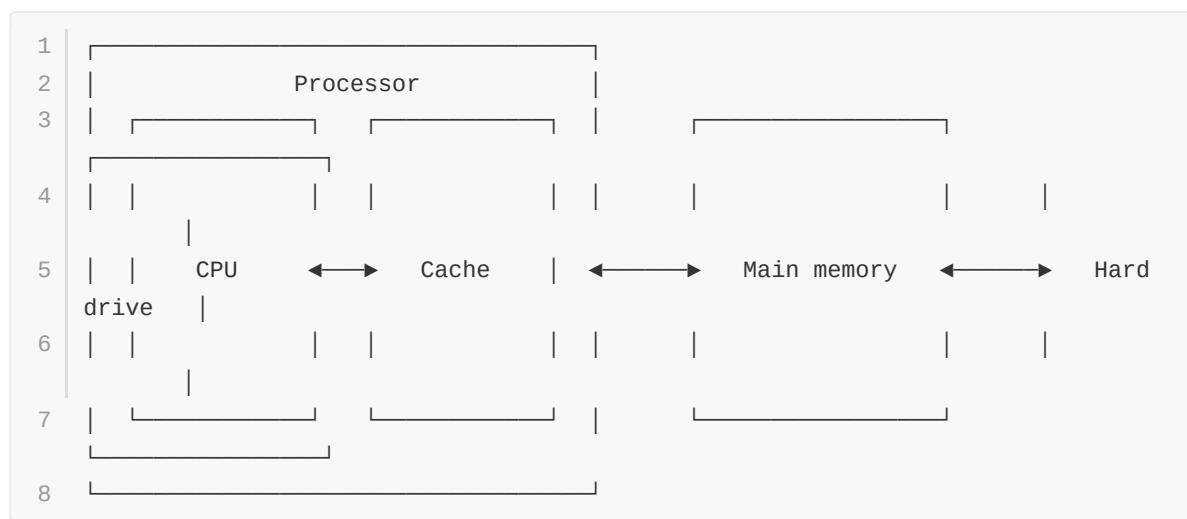
## Совместимость API

Совместимость API не гарантирует переносимость (если изменилась версия ядра, не факт, что будет работать).

Если изменился API, придется переписывать код. Если изменился ABI, нужно перекомпилировать код.

**22-11-21**

## A typical memory hierarchy



Жесткий диск *aka* винчестер -- устройство, на котором хранится информация об операционной системе, драйверах файлах настроек и остальных элементах управления компьютером. Ещё на нём хранится вся информация, созданная пользователем.

Взаимодействие между CPU и main memory происходит через cache.

## Temporal and spatial locality

Временная локальность: использованная информация когда-нибудь будет использоваться снова.

Пространственная локальность: с большой вероятностью будет использоваться информация из той же области.

## Как устроен кэш

В кэше хранятся данные, расположенные по каким-то адресам, причем поддерживается пространственная локальность. Один сохраненный адрес называется *cache block* или *cache line*.

Cache hit -- нужный адрес сохранен в кэше.

Cache miss -- нужный адрес не сохранен в кэше. Вероятность получить cache miss равна 10%.

- compulsory misses -- промахи, которых не избежать. Например, когда к адресу обращаются в первый раз.
- capacity misses -- размер кэша слишком маленький, чтобы хранить все использующиеся данные.
- conflict misses -- в сети переписывается вхождение, которое еще нужно

CPU сначала обращается в кэшу и проверяет, подгружен ли кусок памяти, нужный для запроса, в кэш. Если не лежит, CPU обращается к основной памяти и загружает нужный кусок в кэш.

## Типы устройства кэша

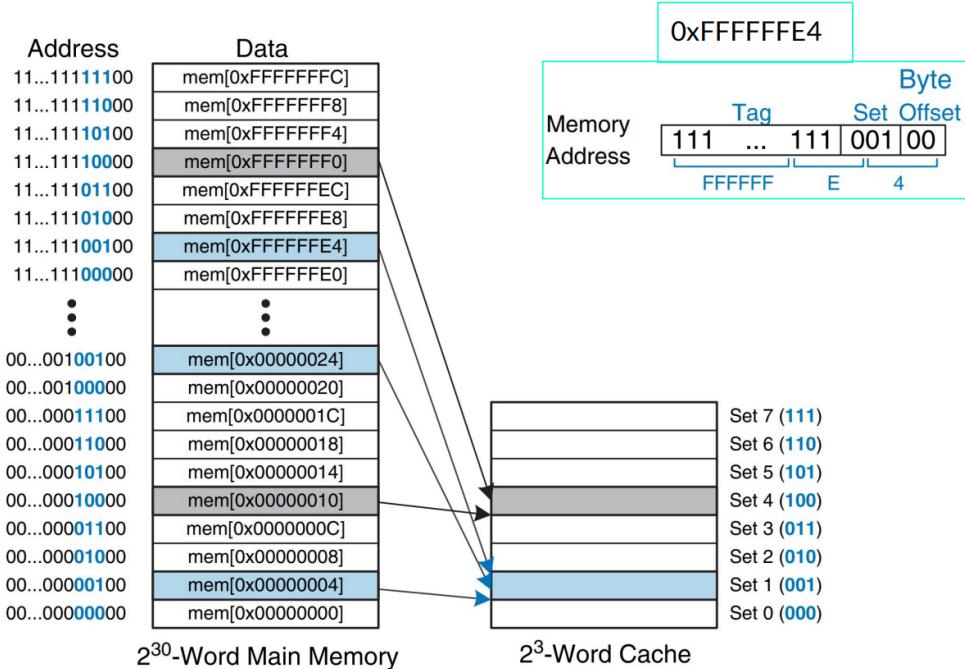
Кэш представляется в виде двумерного массива. Строки называются сетами (sets), столбцы - каналами (ways).

Значение одной ячейки -- тупл вида `<valid bit, tag, data>`. Valid bit -- бит, показывающий, являются ли сохраненные данные валидными. Тег -- часть адреса (об этом подробнее ниже).

Тип устройства		Кол-во каналов	Кол-во сетов
Direct mapped cache	Кэш прямого отображения	1	$B = \text{cache size}$
Multi-way set associative cache	Мульти-канальный ассоциативный кэш	$1 < N < B$	$B/N$
Fully associative cache	Полностью ассоциативный кэш	$B$	1

## 1. Direct mapped cache

Кэш прямого отображения.



Рассмотрим, как устроен адрес.

Последние два бита -- это смещение байта относительно начала в машинном слове.

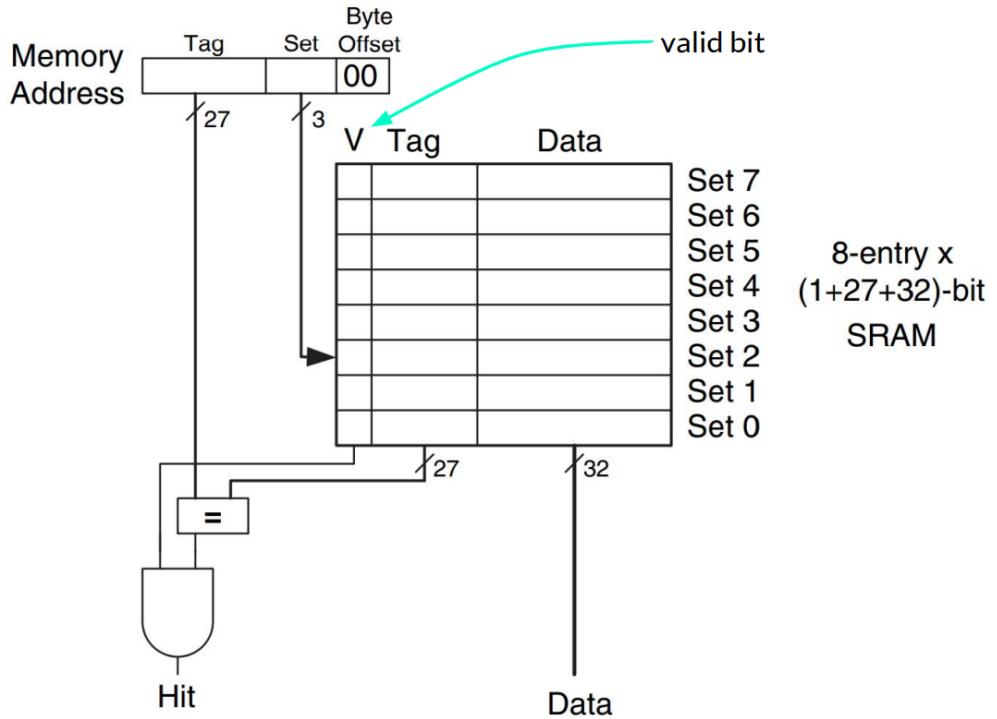
Следующие  $i$  битов -- номер сета в кэше.  $i = \log(\text{cache size})$ . В одном сете лежит один адрес.

Остальные биты -- тег.

Из картинки видно, что распределение адресов между сетами кэша перемешанное, а не упорядоченное. Так сделано для того, чтобы не приходилось перезаписывать один и тот же сет.

Допустим, адреса распределяются в порядке увеличения/уменьшения сетов. В сет  $s$  мы записываем адрес  $s$ . Нужно записывать в кэш еще и рядом лежащие адреса, потому что есть пространственная локальность  $\Rightarrow$  выкинем адрес  $a$  и запишем его соседа. Втф. Поэтому при увеличении адресов номера сетов идут не по порядку.

### Реализация



Хотим обратиться по какому-то адресу. Из адреса определяем сет, в котором адрес должен жить. Дальше сравниваем тег адреса, по которому хотим обратиться, с тегом, сохраненным в рассматриваемом сете. Если они равны и **valid bit = 1**, возвращаем сохраненные данные.

**Если такого тега нет**, переписываем значение в сете. Тут выбора нет, потому что биекция между адресом и сетом.

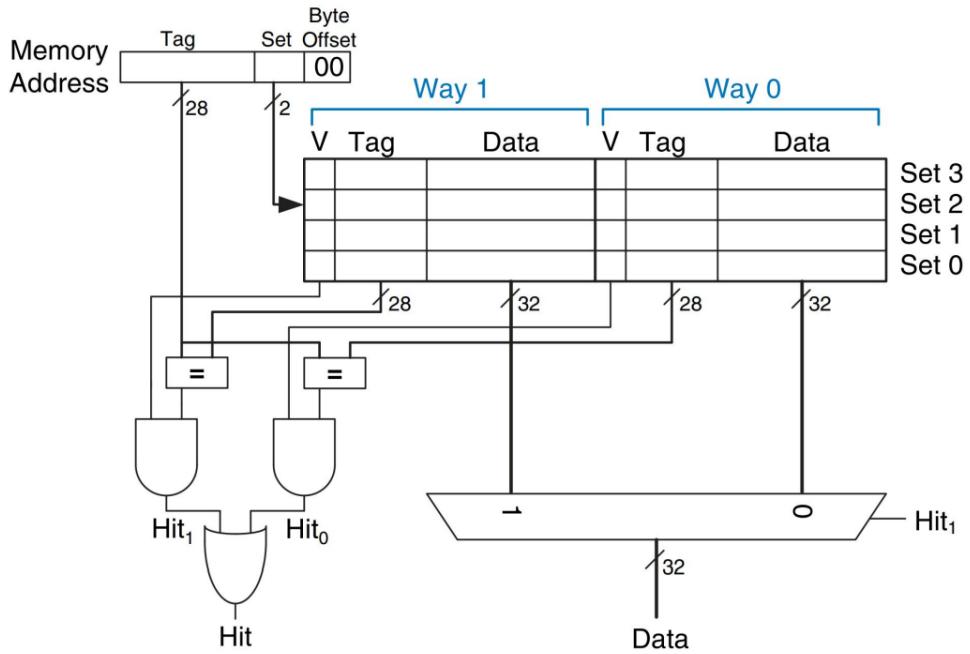
### Какие минусы?

Грустно, если часто обращаемся к адресам, у которых совпадают сети.

## 2. Multi-way set associative cache

$N$ -канальный наборно ассоциативный кэш. Отличается от предыдущего тем, что каждый сет теперь состоит из  $N$  блоков (каналов).  $N$  -- степень ассоциативности.

Двух-канальный ассоциативный кэш:



Хотим обратиться по какому-то адресу. Из адреса определяем сет, в котором адрес должен жить. Потом сравниваем тег адреса, к которому хотим обратиться, с каждым из тегов в сете.

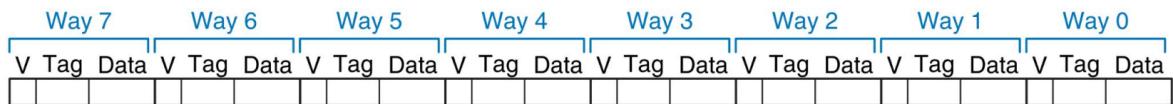
**Если такого тега нет**, то наименее часто использовавшееся вхождение (LRU) заменяется на новые данные.

### 3. Fully associative cache

Полностью ассоциативный кэш.

Используется один сет, у которого  $B$  блоков (каналов).

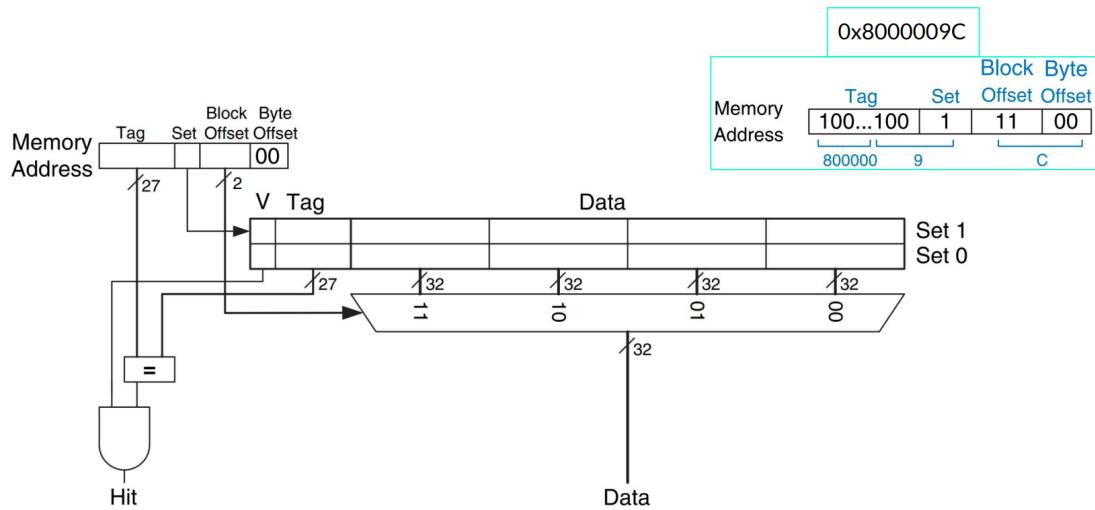
8-канальный полностью ассоциативный кэш:



Хотим обратиться по какому-то адресу. Из адреса определяем тег, по тегу находим нужное вхождение в сете.

**Если такого тега нет**, то заменяется наименее часто использовавшееся вхождение (LRU).

## Пример: 4-канальный кэш с двумя сетами



Теперь в адресе будем хранить смещение внутри строки кэша (block offset).

Сет устроен таким образом: сначала valid bit, потом тег, потом данные по адресам с таким тегом.

Особенность этой схемы в том, что теперь в одном сете хранятся данные из подряд идущих адресов. Таким образом решена проблема

[кэша прямого отображения](#), и больше не нужно перемешивать сеты между адресами.

## LRU replacement

Способ замены данных в кэшах с несколькими каналами.

Нужно знать, сколько раз обращались к каждому адресу. Добавим поле  $U$  в каждое из вхождений. Когда обращаемся к сетю, значение  $U$  увеличивается у каждого вхождения.

**Pseudo-LRU:** если каналов слишком много, то не выгодно поддерживать актуальные поля  $U$  для каждого вхождения. Вместо этого поделим таблицу кэша пополам и будем запоминать, какая из половин последний раз менялась.

## **Изменение данных по адресу и обновление кэша**

Второй способ лучше, потому что меньше обращений к основной памяти  $\Rightarrow$  меньше времени в общем.

### **1. Write-through**

При изменении данных в кэше сразу менять их в основной памяти

### **2. Write-back**

Введем дополнительный dirty bit ( $D$ ). Данные переписаны  $\Rightarrow D = 1$ , иначе  $D = 0$ .

Данные переписываются в основной памяти, только когда из кэша убирается адрес с  $D = 1$ .