

# Введение в кэши:

## микроархитектура, алгоритмы замещения



Терновой Николай,  
инженер Syntacore,  
аспирант НИУ МИЭТ

Тищук Богдана,  
инженер Syntacore

Куроедов Павел,  
инженер Syntacore

## Разрыв в производительности процессоров и памяти

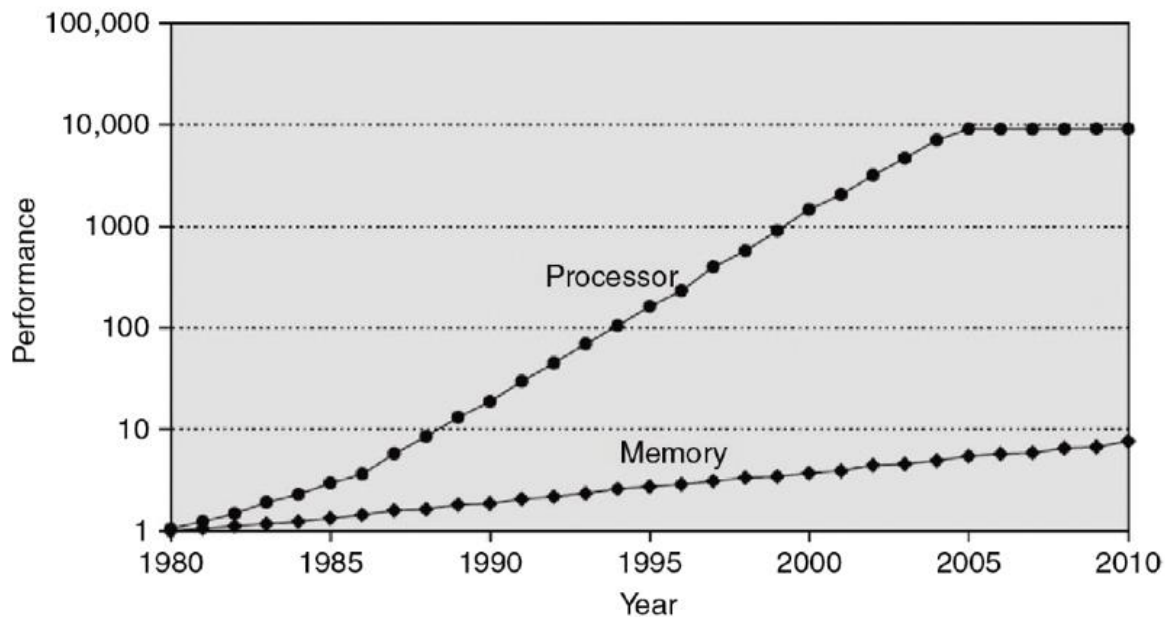
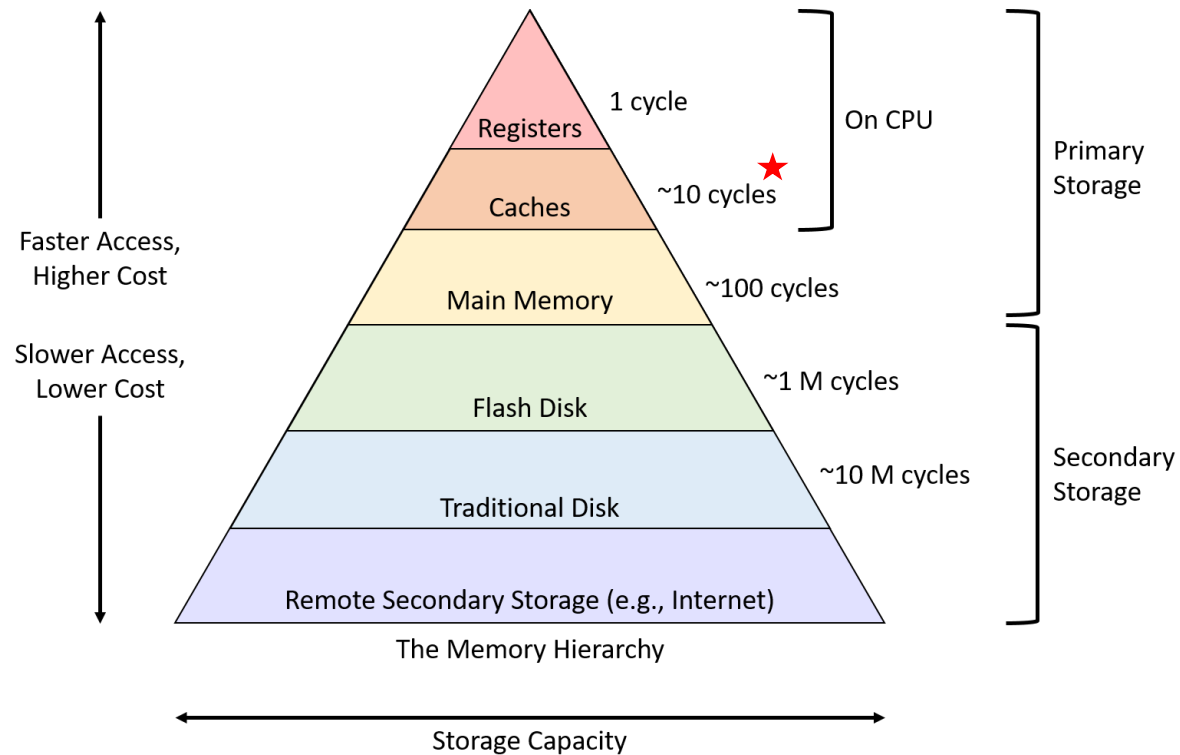


График из книги Hennessy and Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012

	Объем	Задержка	Стоимость/GiB
Flip-flop Registers	1000 bit	20 ps	\$\$\$
SRAM*	10KiB - 10MiB	1-10 ns	1000\$
DRAM	10GiB	80 ns	10\$
NV Flash	100GiB	100 us	1\$
NV Hard Disk	1TB	10 ms	0.1\$

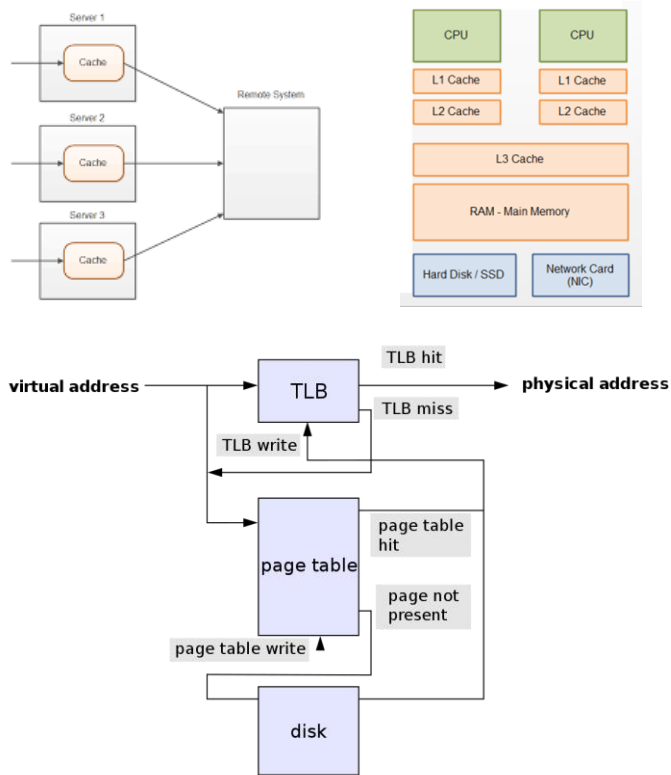
Таблица из книги Hennessy and Patterson, Computer Architecture:  
A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012

# Иерархия памяти



[ИСТОЧНИК](#)

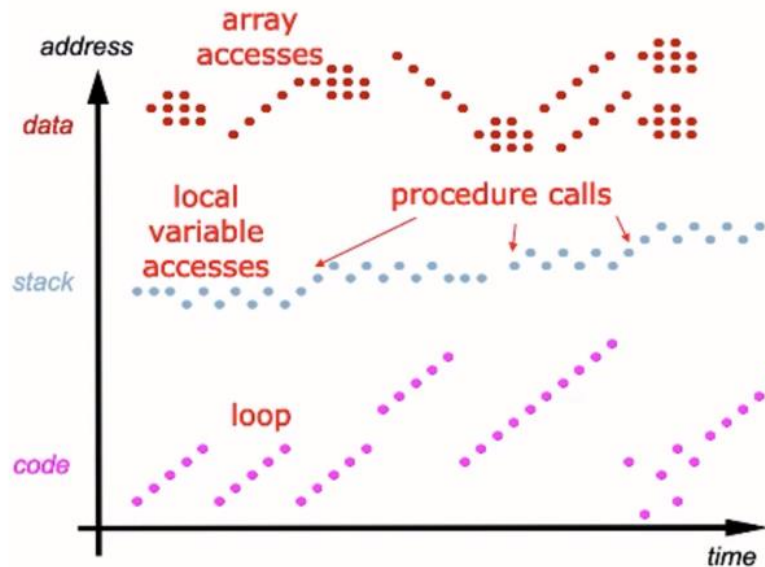
# Кэширование



Кэширование ( в SW и HW ) является одним из механизмов маскирования задержки доступа к какому-либо ресурсу за счет размещения часто используемых данных в памяти с более быстрым временем доступа

- ✓ Многоуровневые кэши в процессорах
- ✓ Многоуровневые TLB
- ✓ Кэширование блоков в SSD/HDD
- ✓ Кэширование в ПО ( браузеры, системы сборки ПО, DNS службы )

# Принципы локальности



- временная локальность означает, что процессор, вероятно, еще раз обратится к тем данным, которые он недавно использовал;
- пространственная локальность означает, что когда процессор обращается к каким-либо данным, то, вероятно, ему понадобятся и расположенные рядом данные;

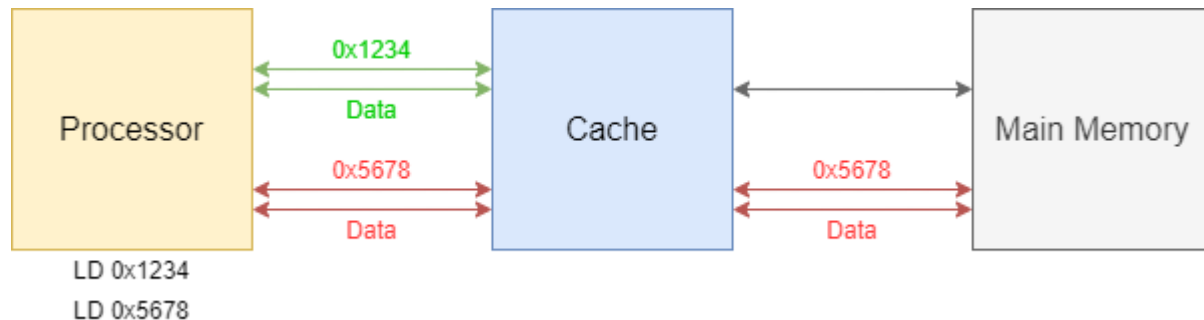
# Принципы локальности

```
#define N = 10
int main()
{
    int test_matrix[N][N] = {7,122,377 ...};
    //          init matrix
    int sum = 0;
    for(int i = 0; i < N; ++i){
        for(int j = 0; j < N; ++j){
            sum += test_matrix[i][j];
        }
    }
    return 0;
}
```

```
0:      00550293      addi x5 x10 5

00000004 <loop>:
4:      00a28063      beq x5 x10 0 <loop>
8:      00452303      lw x6 4 x10
c:      00c52383      lw x7 12 x10
10:     00852e03      lw x28 8 x10
14:     04452e83      lw x29 68 x10
18:     25652f03      lw x30 598 x10
1c:     fff28293      addi x5 x5 -1
20:     fe5ff06f      jal x0 -28 <loop>
```

# Запрос данных процессором



- В первую очередь кэш применяется для ускорению чтению памяти данных, инструкций.
- **Попадание (cache hit)** – запрашиваемый адрес отображен в кэш. Быстрый доступ к данным/инструкции.
- **Промех (cache miss)** – запрашиваемый адрес не отображен (не найден) в кэш.
  - Данные/инструкция запрашиваются из основной памяти.
  - Сохранение данных в кэш для последующего доступа.



# Характеристики кэш-памяти

- Ёмкость –  $C$  (capacity)
  - Число наборов (секций) –  $S$  (set)
  - Длина строки (блока) –  $b$  (block)
  - Количество строк (блоков) –  $B = C/b$
  - Степень ассоциативности –  $N$
- 
- Кэш **состоит из  $S$  наборов**, каждый из которых содержит одну или несколько строк
  - Взаимосвязь между адресом в памяти и расположением в кэш называется **отображением**
  - Каждый адрес в памяти отображается в **один и тот же набор** кэша
- 
- Кэш прямого отображения – Набор  $S$  содержит только один блок –  $S = B$
  - Множественно-ассоциативный кэш – Каждый набор  $S$  состоит из  $N$  строк –  $S = B/N$
  - Полностью ассоциативный кэш – Имеет только один набор  $S = 1$

# Анализ производительности

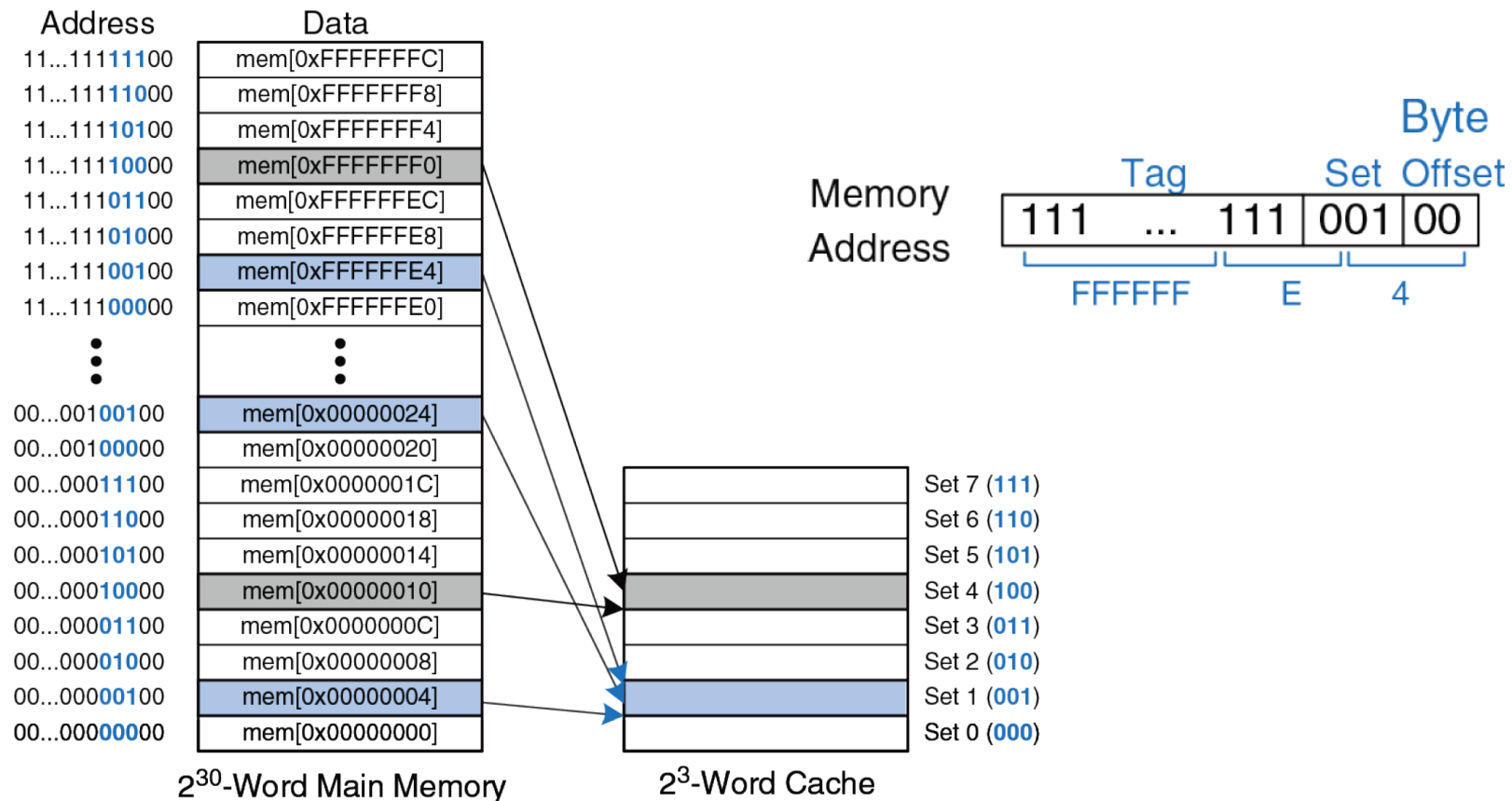
- Доля попаданий (hit rate - **HR**)
- Доля промахов (miss rate - **MR**)
- $MR = \frac{\text{Число промахов}}{\text{Общее число доступов к памяти}} = 1 - HR$
- $HR = \frac{\text{Число попаданий}}{\text{Общее число доступов к памяти}} = 1 - MR$
- **AMAT** - average memory access time
- $AMAT = t_{cache} + MR_{cache} \cdot (t_{MM} + MR_{MM} \cdot t_{VM})$

- **Пример**

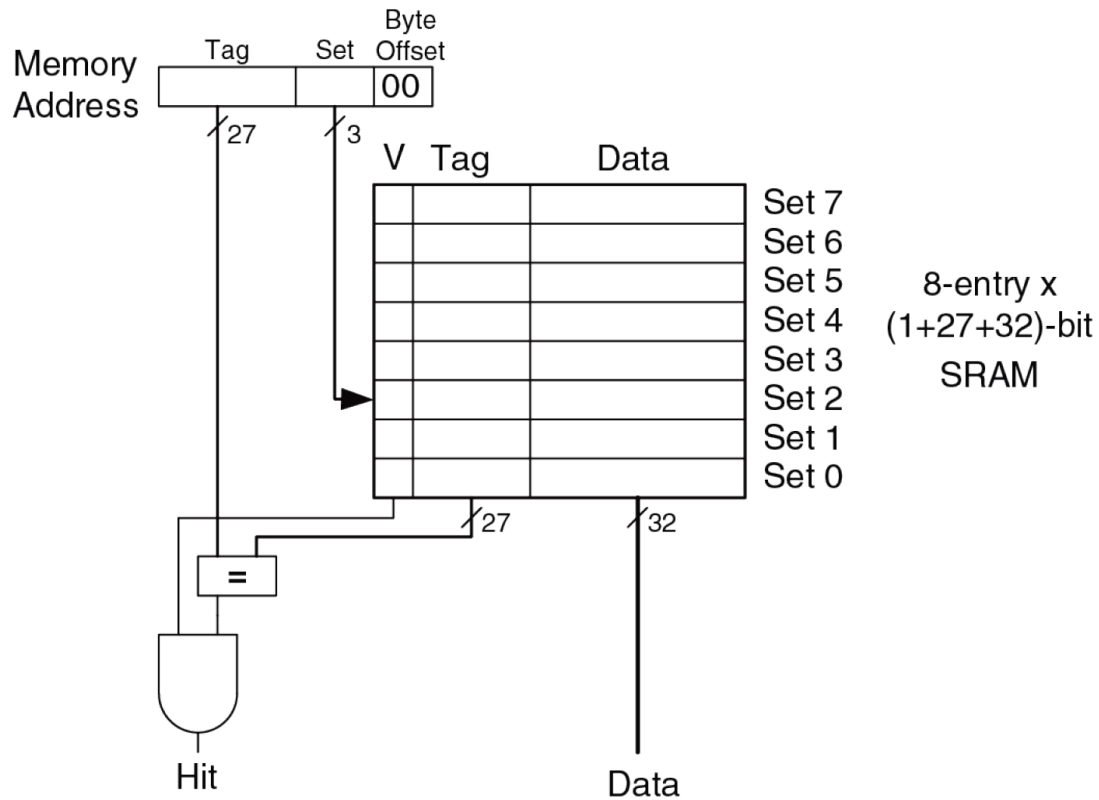
Уровень памяти	Время доступа в тактах	Процент промахов
Кэш-память	1	10%
Оперативная память	100	0%

- $AMAT = 1 + 0.1 \cdot (100) = 11$
- Какой должен быть MR, чтобы снизить AMAT до 1.5 тактов?
- $1 + m \cdot (100) = 1.5 \rightarrow m = 0.5\%$

# Кэш прямого отображения [Direct Mapped]



# Кэш прямого отображения [Direct Mapped]



# Пример работы ДМ кэша

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5     beq t0, a0, done
6     lw  t1, 0x4(a0)
7     lw  t2, 0xC(a0)
8     lw  t3, 0x8(a0)
9     addi t0, t0, -1
10    j loop
11
```

Index	V	Tag	Block 0
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

Cache indexing breakdown:

31  0  
■ = Tag ■ = Index ■ = Block ■ = Byte

The diagram shows a 32-bit indexing breakdown from bit 31 to bit 0. Bits 31 through 28 are represented by grey squares (Tag), bits 27 through 24 by red squares (Index), bit 23 by a green square (Block), and bits 22 through 0 by black squares (Byte). The total number of bits shown is 32, with the last bit being 0.

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	0		
3	1	0x00000000	0x00c52383
4	0		
5	0		
6	0		
7	0		

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	1	0x00000000	0x00452303
3	1	0x00000000	0x00c52383
4	0		
5	0		
6	0		
7	0		



# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

## Registers

GPR

Name	Alias	Value
x0	zero	0x00000000
x1	ra	0x00000000
x2	sp	0x7fffffff0
x3	gp	0x10000000
x4	tp	0x00000000
x5	t0	0x00000004

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	1	0x00000000	0x00452303
3	1	0x00000000	0x00c52383
4	0		
5	0		
6	0		
7	0		

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	1	0x00000000	0x00452303
3	1	0x00000000	0x00c52383
4	0		
5	0		
6	0		
7	0		

# Пример работы ДМ кэша

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

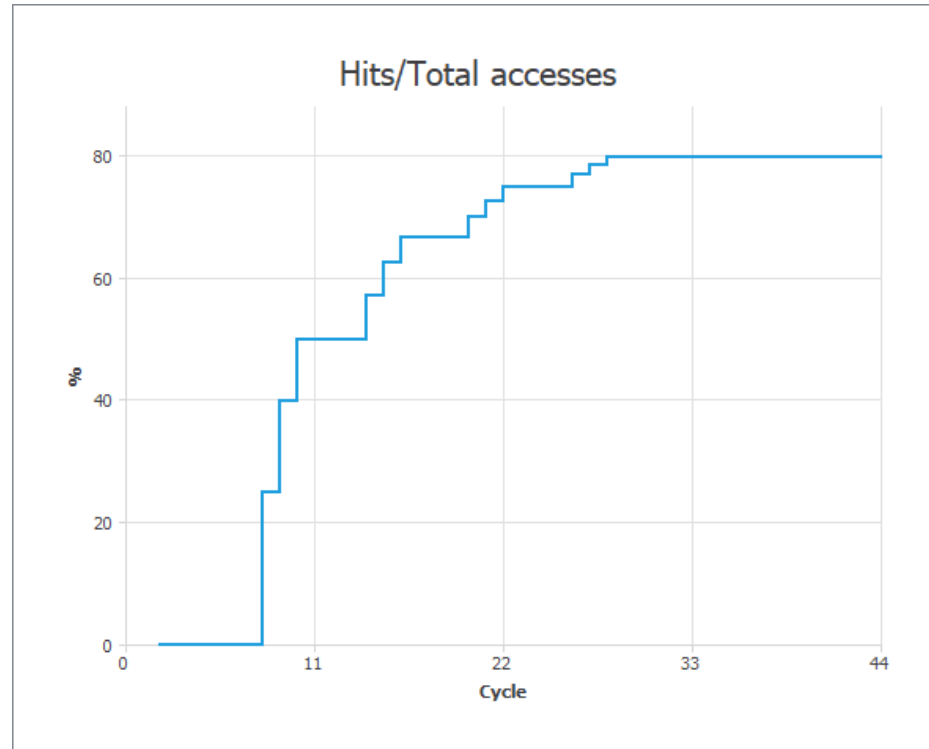
```
c: 00c52383 lw x7 12 x10
```

```
10: 00852e03 lw x28 8 x10
```

```
14: fff28293 addi x5 x5 -1
```

```
18: fedff06f jal x0 -20 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	1	0x00000000	0x00452303
3	1	0x00000000	0x00c52383
4	0		
5	0		
6	0		
7	0		







*Данный пример будет одинаково работать и для других типов кэшей  
(N-way associative, Fully associative)*

# Пример вытеснения (evict)

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5   beq t0, a0, done
6   lw  t1, 0x4(a0)
7   lw  t2, 0x24(a0)
8   addi t0, t0, -1
9   j  loop
```

Cache indexing breakdown:

31  0

 = Tag  = Index  = Block  = Byte

Index	V	Tag	Block
0	0		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

$$24_{16} = 100100_2$$

$$4_{16} = 000100_2$$

# Пример вытеснения (evict)

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 02452383 lw x7 36 x10
```

```
10: fff28293 addi x5 x5 -1
```

```
14: ff1ff06f jal x0 -16 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000000	0x00a28063
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		

# Пример вытеснения (evict)

```
0: 00550293 addi x5 x10 5
```

```
00000004 <loop>:
```

```
4: 00a28063 beq x5 x10 0 <loop>
```

```
8: 00452303 lw x6 4 x10
```

```
c: 02452383 lw x7 36 x10
```

```
10: fff28293 addi x5 x5 -1
```

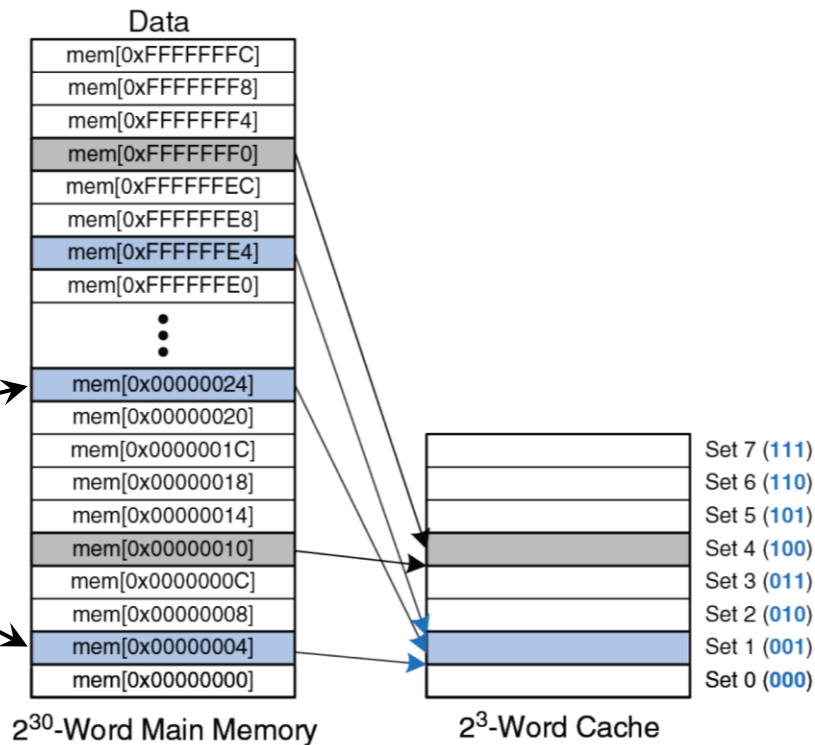
```
14: ff1ff06f jal x0 -16 <loop>
```

Index	V	Tag	Block 0
0	0		
1	1	0x00000001	0x00000000
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		



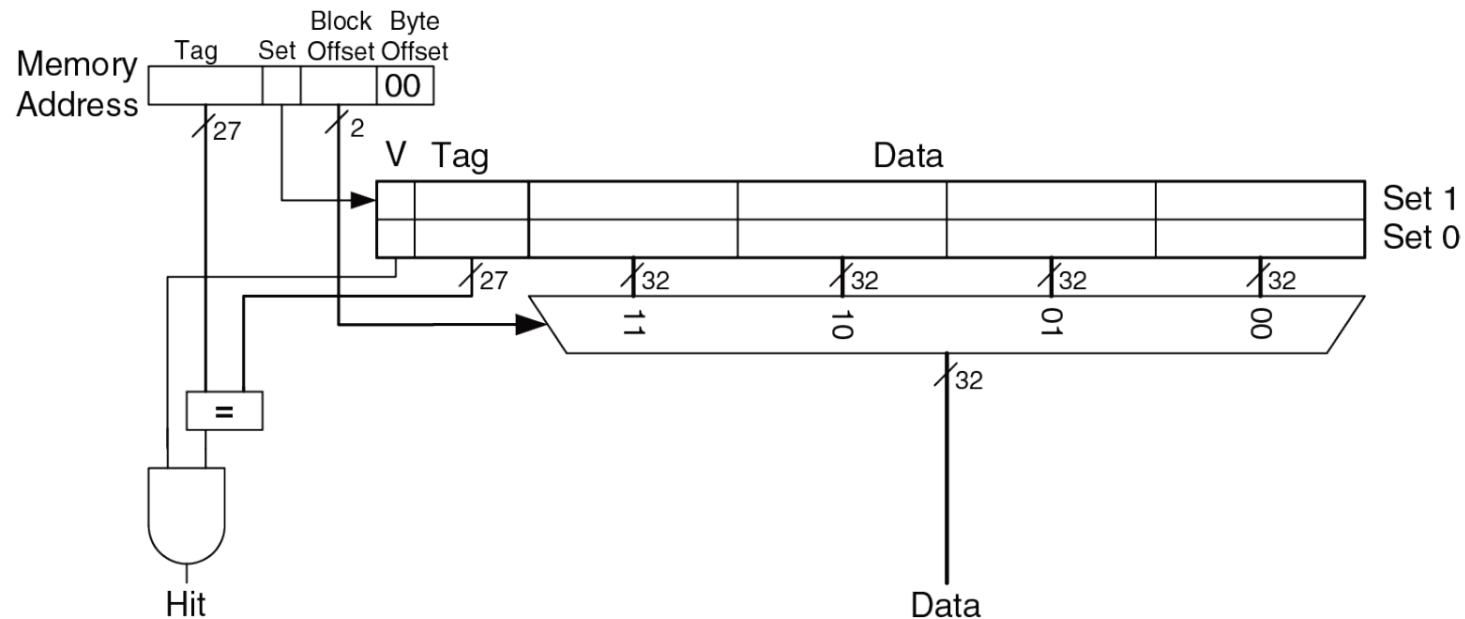
# Пример вытеснения (evict)

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5     beq t0, a0, done
6     lw t1, 0x4(a0)
7     lw t2, 0x24(a0)
8     addi t0, t0, -1
9     j loop
```



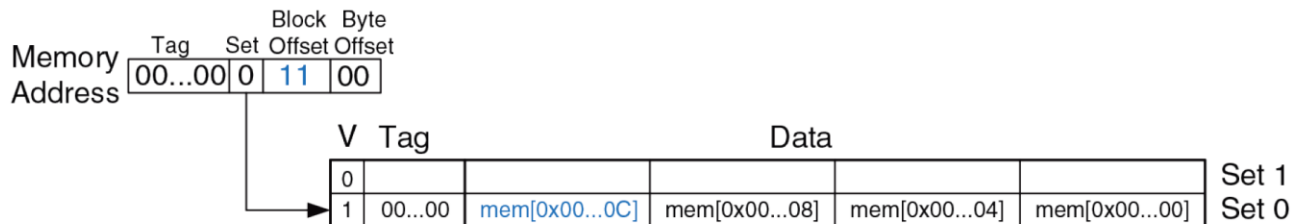
$$MR = \frac{\text{Число промахов}}{\text{Общее число доступов к памяти}} = \frac{10}{10} = 100\%$$

# Размер блока на примере DMC



# Размер блока на примере DMC

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5     beq t0, a0, done
6     lw t1, 0x4(a0)
7     lw t2, 0xC(a0)
8     lw t3, 0x8(a0)
9     addi t0, t0, -1
10    j loop
11
```



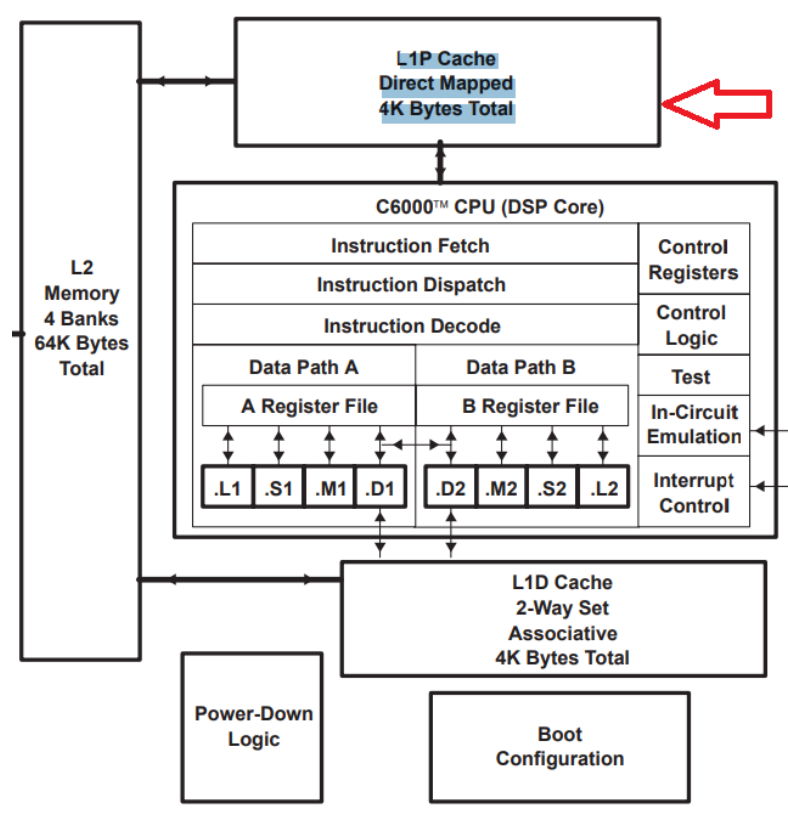
$$MR = \frac{\text{Число промахов}}{\text{Общее число доступов к памяти}} = \frac{1}{15} = 6.67\%$$

# Пример ДМС в реальном микропроцессоре

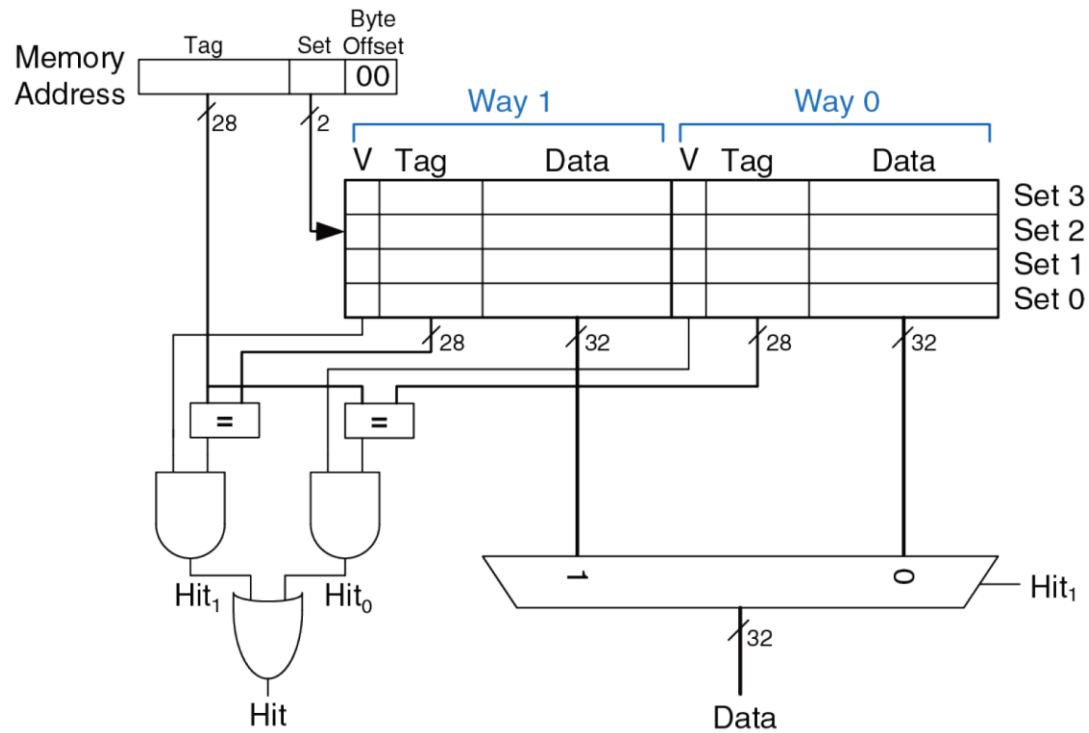
- **L1/L2 Memory Architecture**
  - 32K-Bit (4K-Byte) L1P Program Cache (Direct Mapped)
  - 32K-Bit (4K-Byte) L1D Data Cache (2-Way Set-Associative)
  - 512K-Bit (64K-Byte) L2 Unified Mapped RAM/Cache (Flexible Data/Program Allocation)

Datasheet - [TMS320C6211](#)

Fixed-Point DSP

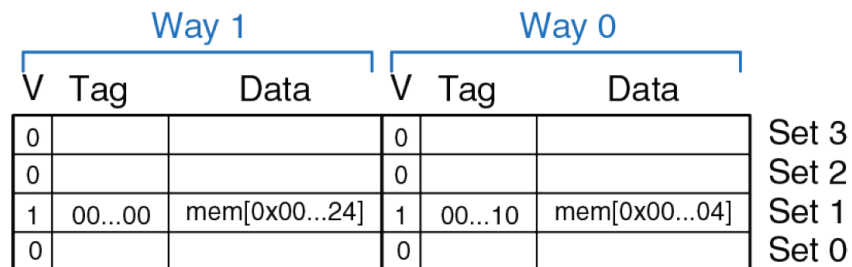


# N-way associative



# N-way associative

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5  beq t0, a0, done
6  lw t1, 0x4(a0)
7  lw t2, 0x24(a0)
8  addi t0, t0, -1
9  j loop
```



# Пример вытеснения (evict)

0: 00550293 addi x5 x10 5

00000004 <loop>:

```
4: 00a28063 beq x5 x10 0 <loop>
8: 00452303 lw x6 4 x10
c: 02452383 lw x7 36 x10
10: fff28293 addi x5 x5 -1
14: ff1ff06f jal x0 -16 <loop>
```

Index	V	Tag	Block
0	0		
	0		
1	0		
	0		
2	0		
	0		
3	0		
	0		
4	0		
	0		
5	0		
	0		
6	0		
	0		
7	0		
	0		

# Пример вытеснения (evict)

```
0:  00500293    addi x5 x0 5
00000004 <loop>:
4:  00028063    beq x5 x0 0 <loop>
8:  00402303    lw x6 4 x0
c:  02402383    lw x7 36 x0
10:  fff28293    addi x5 x5 -1
14:  ff1ff06f    jal x0 -16 <loop>
```

Index	V	Tag	Block 0
0	0		
	0		
1	0		
	1	0x00000000	0x00028063
2	0		
	0		
3	0		
	0		
4	0		
	0		
5	0		
	0		
6	0		
	0		
7	0		
	0		



# Пример вытеснения (evict)

```
0: 00500293    addi x5 x0 5
```

```
00000004 <loop>:
```

```
4: 00028063    beq x5 x0 0 <loop>
```

```
8: 00402303    lw x6 4 x0
```

```
c: 02402383    lw x7 36 x0
```

```
10: fff28293    addi x5 x5 -1
```

```
14: ff1ff06f    jal x0 -16 <loop>
```

Index	V	Tag	Block 0
0	0		
	0		
1	1	0x00000001	0x00000000
	1	0x00000000	0x00028063
2	0		
	0		
3	0		
	0		
4	0		
	0		
5	0		
	0		
6	0		
	0		
7	0		
	0		

# Пример вытеснения (evict)

```
1 .text
2 addi t0, a0, 5
3 done:
4 loop:
5     beq t0, a0, done
6     lw t1, 0x4(a0)
7     lw t2, 0x24(a0)
8     addi t0, t0, -1
9     j loop
```

Index	V	Tag	Block 0
0	0		
	0		
1	1	0x00000000	0x00028063
	1	0x00000001	0x00000000
2	0		
	0		
3	0		
	0		
4	0		
	0		
5	0		
	0		
6	0		
	0		
7	0		
	0		

Index	V	Tag	Block 0
0	0		
	0		
1	1	0x00000000	0x00028063
	1	0x00000001	0x00000000
2	0		
	0		
3	0		
	0		
4	0		
	0		
5	0		
	0		
6	0		
	0		
7	0		
	0		

# Пример N-way associative в реальном процессоре

**CPU** | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name: Intel Core i7 4770K

Code Name: Haswell Max TDP: 84.0 W

Package: Socket 1150 LGA

Technology: 22 nm Core Voltage: 1.252 V

Specification: Intel® Core™ i7-4770K CPU @ 3.50GHz

Family: 6 Model: C Stepping: 3

Ext. Family: 6 Ext. Model: 3C Revision: C0

Instructions: MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, EM64T, AES, AVX, AVX2, FMA3

Clocks (Core #0)

Core Speed: 4242.05 MHz

Multiplier: x 43.0 (8 - 43)

Bus Speed: 98.65 MHz

Rated FSB:

Cache

L1 Data	4 x 32 KBytes	8-way
L1 Inst.	4 x 32 KBytes	8-way
Level 2	4 x 256 KBytes	8-way
Level 3	8 MBytes	16-way

Selection: Socket #1 Cores: 4 Threads: 8

**CPU** | Caches | Mainboard | Memory | SPD | Graphics | Bench | About

Processor

Name: AMD Ryzen 7 1700

Code Name: Summit Ridge Max TDP: 65 W

Package: Socket AM4 (1331)

Technology: 14 nm Core Voltage: 1.373 V

Specification: AMD Ryzen 7 1700 Eight-Core Processor

Family: F Model: 1 Stepping: 1

Ext. Family: 17 Ext. Model: 1 Revision: ZP-B1

Instructions: MMX(+), SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4A, x86-64, AMD-V, AES, AVX, AVX2, FMA3, SHA

Clocks (Core #0)

Core Speed: 3990.48 MHz

Multiplier: x 40.0

Bus Speed: 99.76 MHz

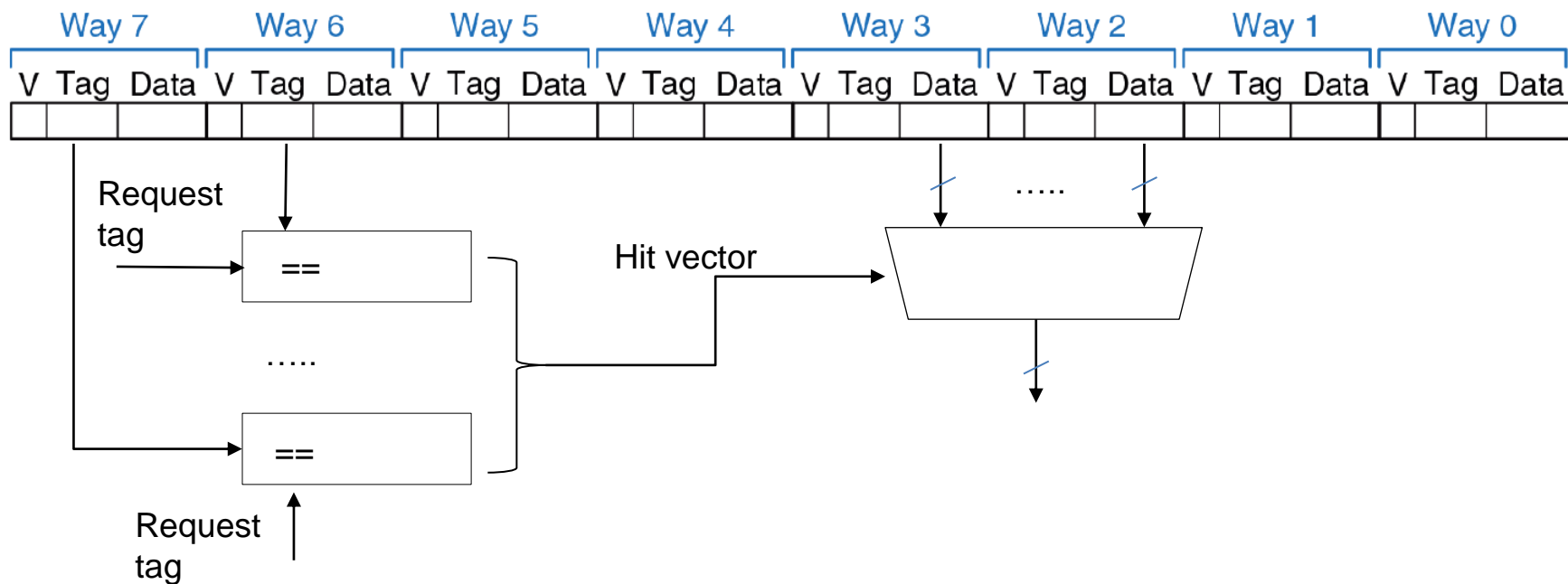
Rated FSB:

Cache

L1 Data	8 x 32 KBytes	8-way
L1 Inst.	8 x 64 KBytes	4-way
Level 2	8 x 512 KBytes	8-way
Level 3	2 x 8192 KBytes	16-way

Selection: Processor #1 Cores: 8 Threads: 16

# Fully-associative

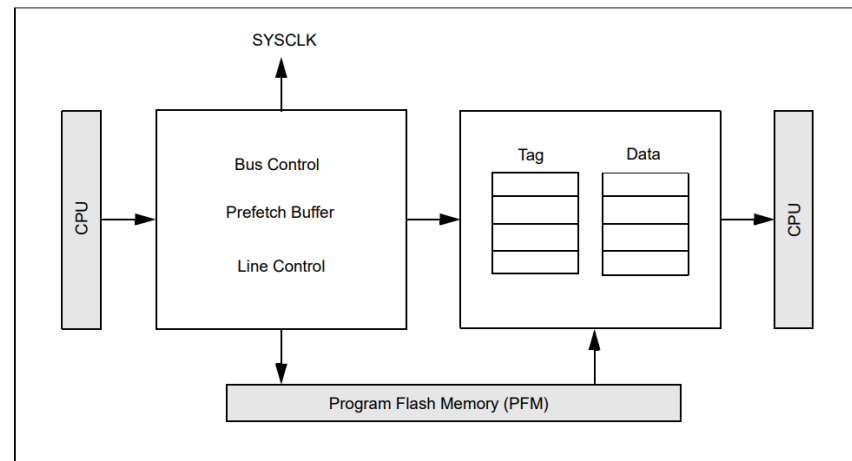


# Пример Fully-associative cache в реальном процессоре

The following are key features of the Prefetch module:

- 4x16 byte fully-associative lines
- One line for CPU instructions
- One line for CPU data
- Two lines for peripheral data
- 16-byte parallel memory fetch
- Configurable predictive prefetch
- Error detection and correction

Datasheet - [PIC32MZ](#)

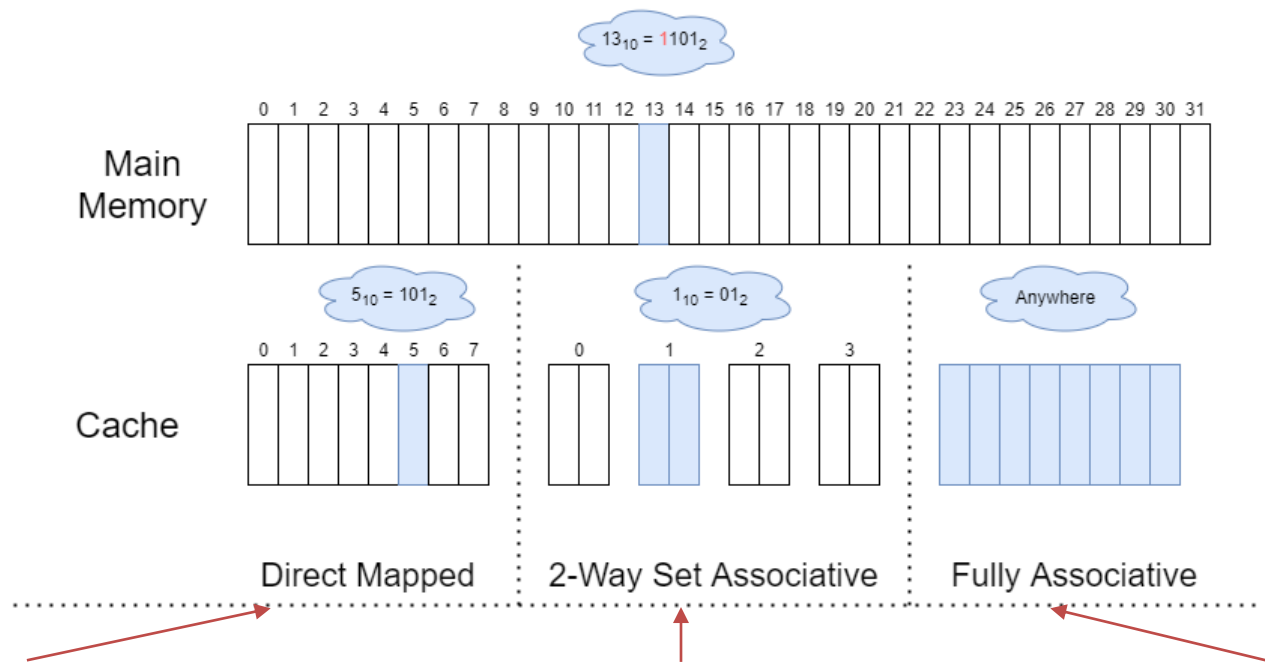


# Размер блока на примере DMC

```
1  /*--- pi.c PROGRAM RANPI */
2  #include <stdio.h>
3  void myadd(float* sum, float* addend) {
4
5      *sum = *sum + *addend;
6  }
7  int main(int argc, char* argv[]) {
8      float ztot, yran, ymult, ymod, x, y, z, pi, prod;
9      long int low, ixran, itot, j, iprod;
10     ztot = 0.0;
11     low = 1;
12     ixran = 1907;
13     yran = 5813.0;
14     ymult = 1307.0;
15     ymod = 5471.0;
16     itot = 100;
17
18     for (j = 1; j <= itot; j++) {
19         iprod = 27611 * ixran;
20         ixran = iprod - 74383 * (long int)(iprod / 74383);
21         x = (float)ixran / 74383.0;
22         prod = ymult * yran;
23         yran = (prod - ymod * (long int)(prod / ymod));
24         y = yran / ymod;
25         z = x * x + y * y;
26         myadd(&ztot, &z);
27         if (z <= 1.0) {
28             low = low + 1;
29         }
30     }
31
32     pi = 4.0 * (float)low / (float)itot;
33     // Print result
34     printf("Result: %f\n", pi);
35     // Move result to some pre-determined register
36     asm("mv x27, %[v]"
37         : /* Output registers */
38         : [v] "r"(pi) /* Input registers */
39         : /* Clobber registers */
40     );
41     return 0;
42 }
43
```

Cache type	Hits	Misses	HR
direct mapped cache [16 sets]	48589	236908	17%
2-way associative cache [8 sets]	47482	238015	16,6%
full associative cache [16 way]	48545	236952	17%
direct mapped cache [8 sets] Block size 2 word	151847	133650	53%

# Сравнение архитектур кэша



Запрашиваемый адрес сравнивается только с **одним** тегом.

Искомые данные могут находиться только в **определенной** линии кэша

Запрашиваемый адрес сравнивается только с **N** тегов одновременно

Искомые данные могут храниться ровно в **одном наборе**, но в любом из **N** кэш-линий, принадлежащих этому набору

Запрашиваемый адрес сравнивается с **каждым** тегом одновременно

Искомые данные могут находиться только в **любой** линии кэша

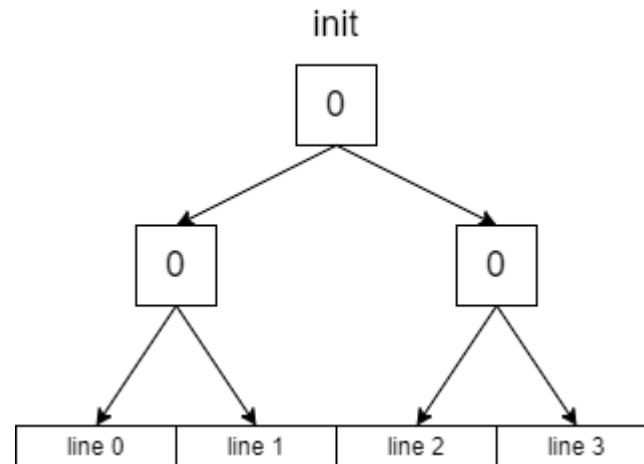
# Алгоритмы замещения данных

- **LRU** (Least Recently Used) – наиболее давнего использования
- **PLRU** (Pseudo-Least Recently Used) – псевдо наиболее давнего использования
- **FIFO** (First In First Out) – замещение в порядке очереди
- **LFU** (Least Frequently Used) – наименее частого использования
- **RND** (Random Replacement) – замена случайной строки
- **CLOCK** – циклический список с указателем

Size	Associativity								
	Two-way			Four-way			Eight-way		
	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

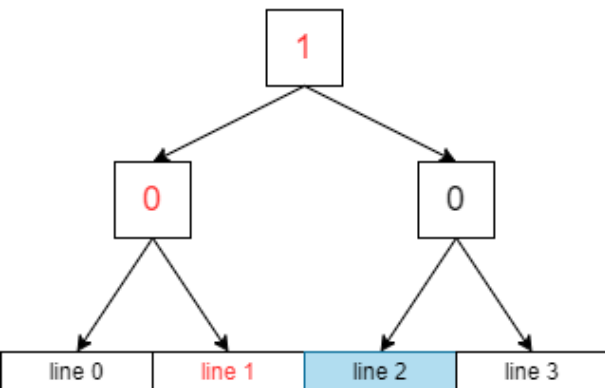


- Идея – бинарное дерево
  - 0 левая ветвь
  - 1 правая ветвь
- **Преимущества:**
  - Простота реализации
  - Требуется N-1 бит на набор [set]
- **Недостатки :**
  - Узел на вершине дерева содержит только один бит и не может достаточно точно отразить историю “дерева”
  - декодирование битов - это последовательный процесс



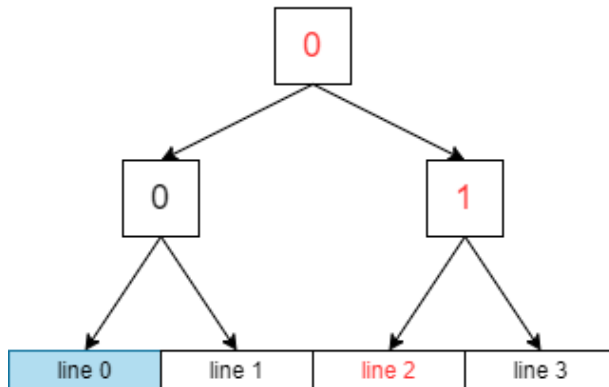
# Имплементация plru

access line 1



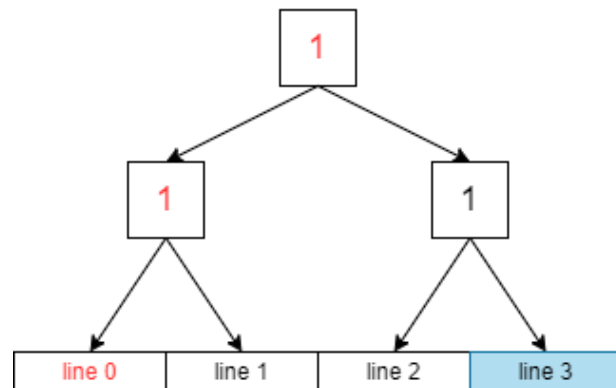
LRU line 2

access line 2



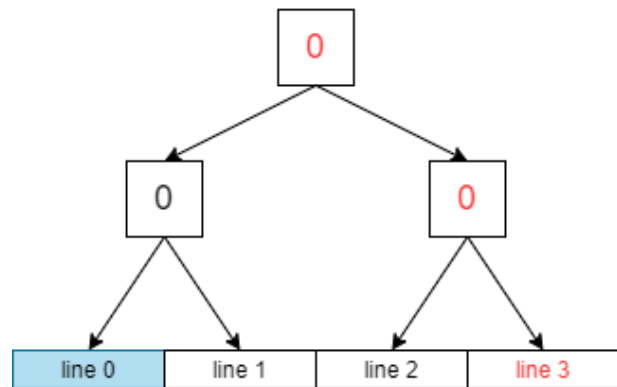
LRU line 0

access line 1



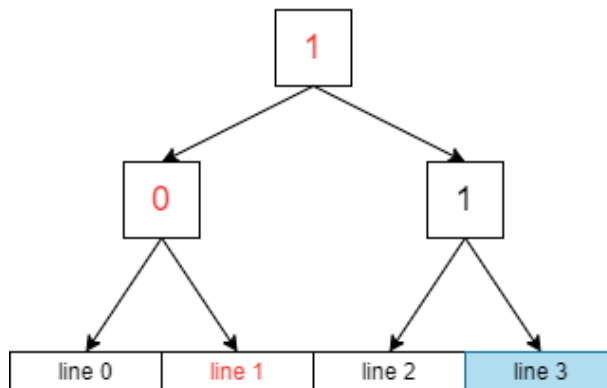
LRU line 3

access line 0



LRU line 0

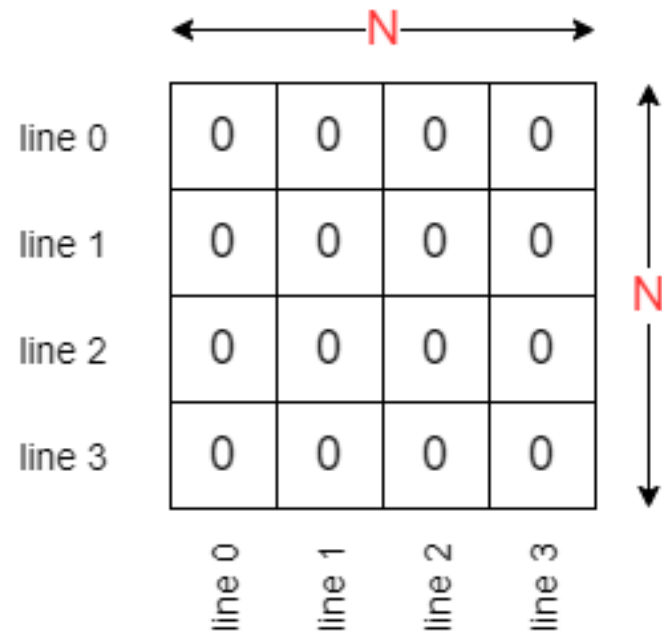
access line 3



LRU line 3

# Имплементация Real LRU

- Идея
  - установить в соответствующей строке все единицы
  - установить соответствующий столбец во все нули
  - LRU линия будет представлена нулевой строкой
- Плюсы:
  - Сохраняется история доступа
- Недостатки:
  - Необходимо хранить матрицу битов  $N^2$  для каждого набора [set]



# Имплементация Real LRU

access line 1

line 0	0	0	0	0
line 1	1	0	1	1
line 2	0	0	0	0
line 3	0	0	0	0
	line 0	line 1	line 2	line 3

access line 2

0	0	0	0
1	0	0	1
1	1	0	1
0	0	0	0

access line 0 [LRU line 3]

0	1	1	1
0	0	0	1
0	1	0	1
0	0	0	0

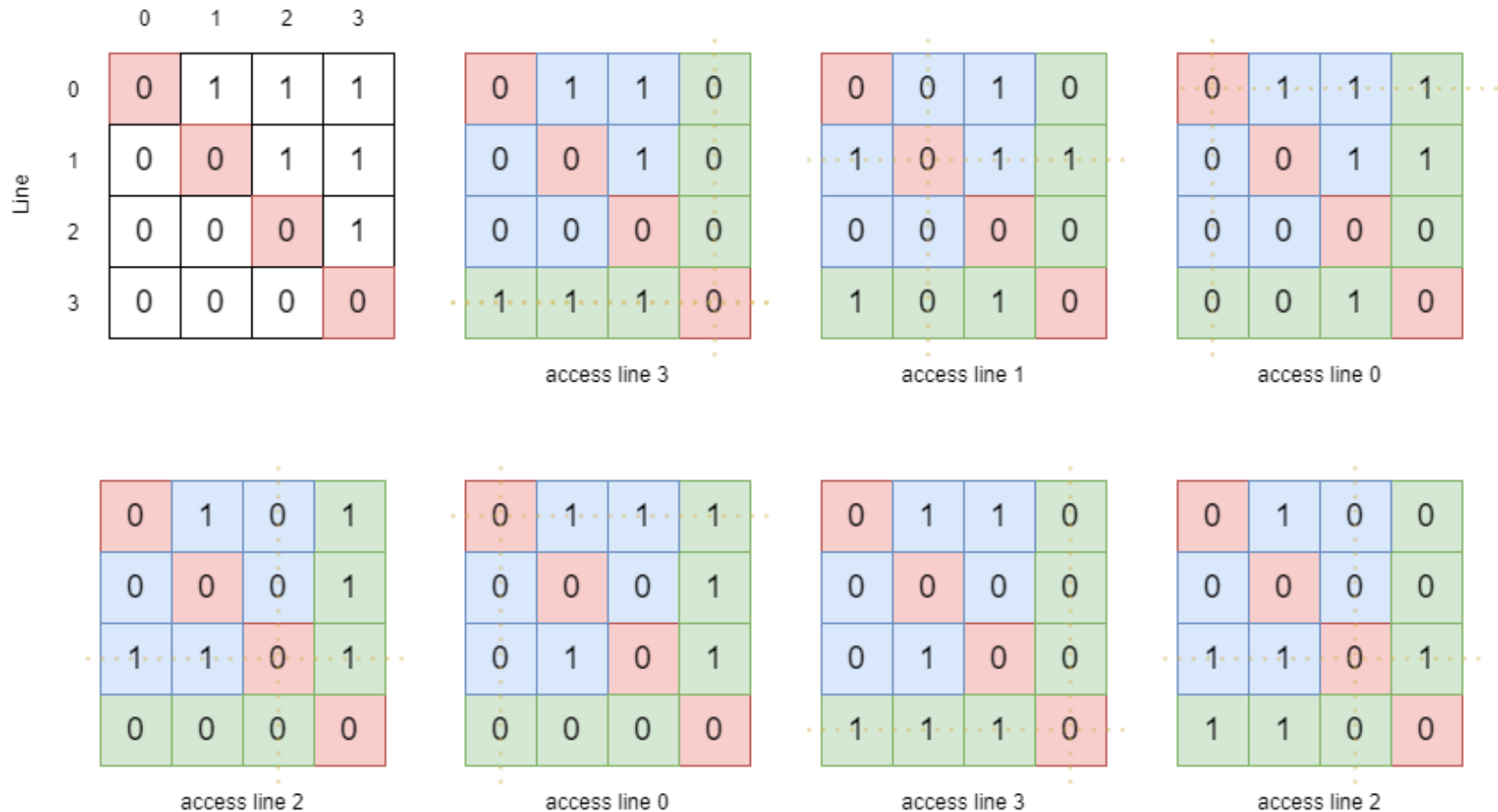
access line 1 [LRU line 3]

0	0	1	1
1	0	1	1
0	0	0	1
0	0	0	0

access line 3 [LRU line 2]

0	0	1	0
1	0	1	0
0	0	0	0
1	1	1	0

# Имплементация LRU



- Идея
  - *reference matrix method*  
(математическое описание  
сможете найти в книжке  
*Hacker's Delight*  
7-9 An LRU Algorithm)
  - Для матрицы размером  $N * N$  эквивалентно выражение  
 $a_{i,j} = -a_{j,i}$
- Плюсы:
  - Необходимо хранить не  $N^2$ , а  
 $N * (N - 1)/2$  битов для каждого  
набора [set]



	0	1	2	3
0	0;0	0;1	0;2	0;3
1	1;0	1;1	1;2	1;3
2	2;0	2;1	2;2	2;3
3	3;0	3;1	3;2	3;3

- **Неизбежные** (**compulsory**) – самое первое обращение к блоку, которого нет в кэше. Данный блок должен быть загружен в кэш.
- **Емкостные** (**capacity**) – если кэш не может содержать все блоки, необходимые во время выполнения программы (недостаточный объем), емкостные промахи, наряду с **неизбежными** промахами, будут происходить из-за того, что блоки удаляются, а затем снова загружаются.
- **Конфликтные** (**conflict**) – если стратегия размещения блока не полностью ассоциативна, конфликтные промахи (наряду с **неизбежными** и **емкостными**) будут происходить из-за того, что блок может быть удален, а потом снова загружен, если несколько блоков отображаются на один набор и обращения к разным блокам чередуются во времени.



## При попадании

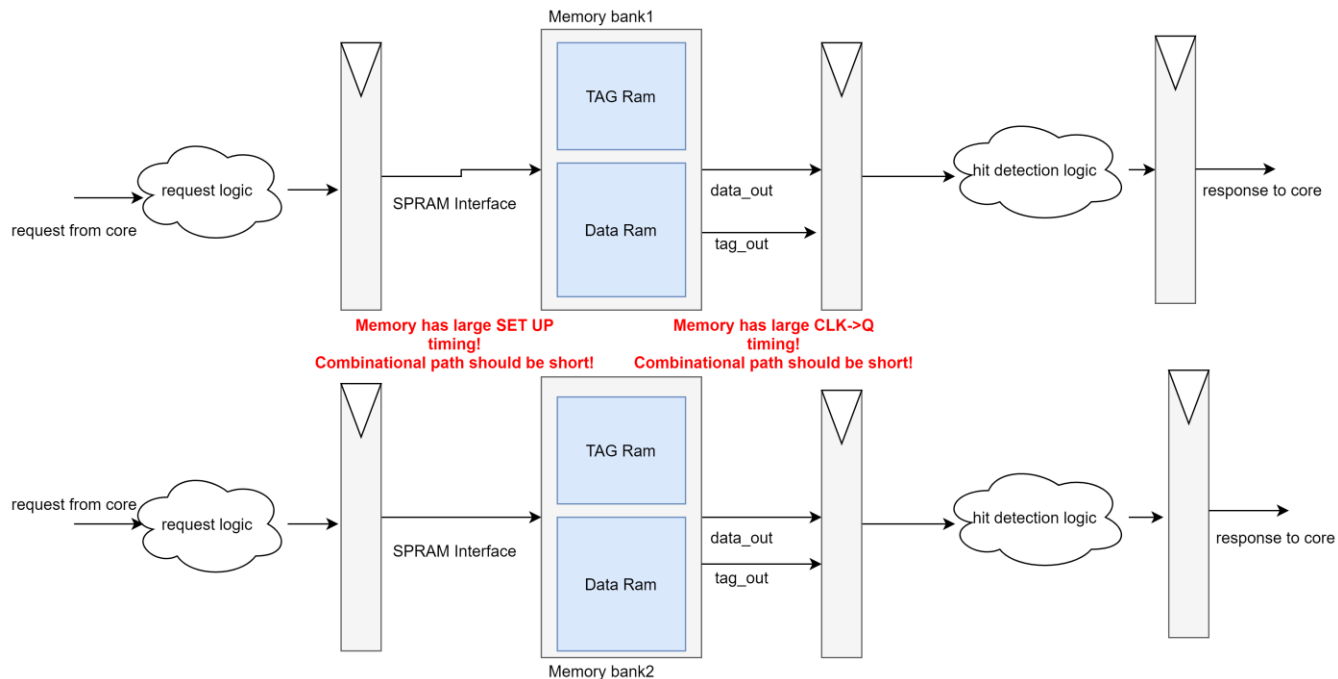
- ✓ **Write through** – при записи данные попадают сразу в кэш и в память ( кэш следующего по иерархии уровня )
- ✓ **Write back** – при записи данные попадают только в кэш, память обновляется при вытеснении кэш строки

## При промахе

- ✓ **Write allocate** – при промахе по записи в кэше аллоцируется (размещается) ячейка, содержащая данные, которые должны быть записаны
- ✓ **Write non-allocate** – при промахе по записи данные сразу пишутся в память, в кэше ячейка для этих данных не заводится

# Детали проектирования

- ✓ В современных чипах время от поступления запроса в L1 кэш до получения данных — **не 1 такт!**



- ✓ Разбиение доступа по банкам

<b>TAG</b>	<b>MEMORY Index</b>	<b>MEMORY Bank</b>	<b>CL Offset</b>
------------	---------------------	--------------------	------------------

- ✓ Требования по частоте делают многие теоретически – эффективные решения слабо реализуемыми на практике

Из – за этого не применяют fully-associative кэши большого размера

- ✓ Памяти в кэше могут накапливать спонтанные ошибки

Для избежания использования невалидных данных во многих кэшах применяют ECC

- ✓ Кэширование не должно нарушать модель памяти архитектуры

Кэши в современных процессорах достаточно сложны. Поддержка глобального порядка операций с памятью, регламентируемого моделью памяти конкретной архитектуры при высоких требованиях к производительности и частоте становится нетривиальной задачей со множеством нюансов для проектирования

- ✓ Могут ли быть conflict misses в fully-associative кэше?
- ✓ Чем банк отличается от way? Где будет меньше conflict miss при одинаковом объеме: а) 2 банка DM кэша б) 1 банк 2-way кэша
- ✓ Каким образом влияет политика замещения в кэше на производительность?
- ✓ Может ли кэш ухудшать производительность ? Если да, то в каких случаях?
- ✓ Чем отличается вытеснение кэш-линии от инвалидации кэш-линии?
- ✓ Вытеснение линии из кэша, при том, что есть другие свободные кэш – строки. Когда такое возможно?
- ✓ Отличие процедуры вытеснения для Write-back и Write-through кэша
- ✓ Почему LRU сложно применять на кэшах с большой ассоциативностью?
- ✓ Преимущества и недостатки большой гранулярности ( размера кэш-линии )

# Ссылки на источники и материалы

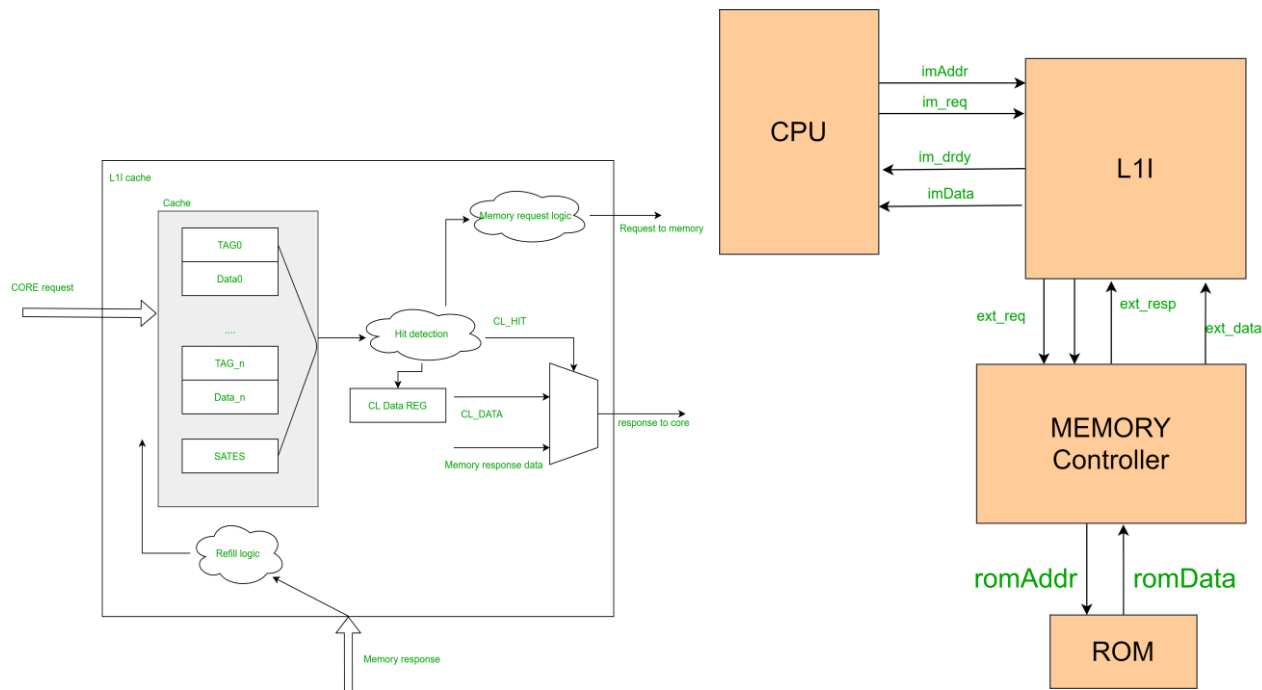
1. [Лекции курса 6.004 MIT](#)
2. [Лекции инженера Intel Александра Титова](#)
3. [Симулятор Ripes](#)
4. [Advanced Material: Implementing Cache Controllers](#)
5. Книга: Цифровая схемотехника и архитектура компьютера
6. [Следующие шаги в черной магии процессоростроения после того, как вы освоили Харрис & Харрис](#)
7. Курс от David Wentzlaff на [coursera](#)

# План лабораторной работы

- Изучить функционал testbench
- Изучить функционал .do скрипта, добавить по аналогии произвольные сигналы на waveform
- Ознакомиться с параметрами конфигурации памяти и кэша
- Оценить время исполнение программы с включенным и выключенным кэшем
- Изучить способы параметризации длины строки кэш линии

# L1 Cache SchoolRISC-v

- ✓ Fully-associative
- N-line L1I cache
- ✓ 1 cycle latency
- ✓ Caching mechanism can be disabled
- ✓ PLRU eviction



schoolIRISCV\_ICache-quartus\_prj > schoolIRISCV > testbench

Имя	Дата изменения	Тип	Размер
work	15.01.2022 2:51	Папка с файлами	
make.do	15.01.2022 2:21	Файл "DO"	2 КБ
testbench.v	15.01.2022 2:51	Файл "V"	4 КБ



```
vlib work
vmap work
```

```
vlog    ../src/sr_cpu.vh
vlog    ../src/sm_hex_display.v
vlog    ../src/sm_register.v
vlog    ../src/sm_rom.v
vlog    ../src/sr_cpu.v
vlog    ../src/srv_icache.sv
vlog    ../src/srv_mem.sv
vlog    ../src/sm_top.v
vlog    testbench.v
```

```
vsim    sm_testbench
```

```
add wave -color #ee66ff -radix hex -group CPU \
/sm_testbench/sm_top/sm_cpu/regAddr \
/sm_testbench/sm_top/sm_cpu/regData \
/sm_testbench/sm_top/sm_cpu/im_req \
/sm_testbench/sm_top/sm_cpu/imAddr \
/sm_testbench/sm_top/sm_cpu/imData \
/sm_testbench/sm_top/sm_cpu/im_drdy \
/sm_testbench/sm_top/sm_cpu/addr_o \
/sm_testbench/sm_top/sm_cpu/data_o \

# cycle cnt
add wave -color #cccc00 -radix unsigned -group CYCLE_CNT \
/sm_testbench/sm_top/i_cycle_cnt/cycleCnt_o \
sm_testbench/sm_top/i_cycle_cnt/en_i \

run -all
```

```
# C:/Users/Nick/Desktop/schoolRISCV_ICache-quartus_prj/schoolRISCV/testbench
VSIM(paused)> ls
# make.do
# testbench.v
# vsim.wlf
# work
```

```
do make.do
```

```
VSIM(paused)> do make.do
```



sim - Default

Instance

- sm\_testbench
  - disasmInstr
  - sm\_top
    - f0
    - f1
    - f2
    - sm\_clk\_divider
    - reset\_rom
    - mem\_ctrl
    - sm\_cpu
    - sm\_jcache
    - i\_cycle\_cnt
      - #ASSIGN#184
      - #ALWAYS#187
      - #ALWAYS#111
      - #ALWAYS#118
      - #INITIAL#62
      - #ALWAYS#120
      - #ALWAYS#139
  - std
  - #vsim\_capacity#

Design unit

Design

Objects

Name

- Tt
- CACHE\_EN
- clk
- rst\_n
- regAddr
- cpuClk
- cycle

Wave - Default

Msgs

0

St1

St1

(Cache\_IO)

(Mem\_Cntrl)

(CPU)

(CYCLE\_CNT)

Now 26750000 ps

Cursor 1 0 ps

26749200 ps 26749400 ps

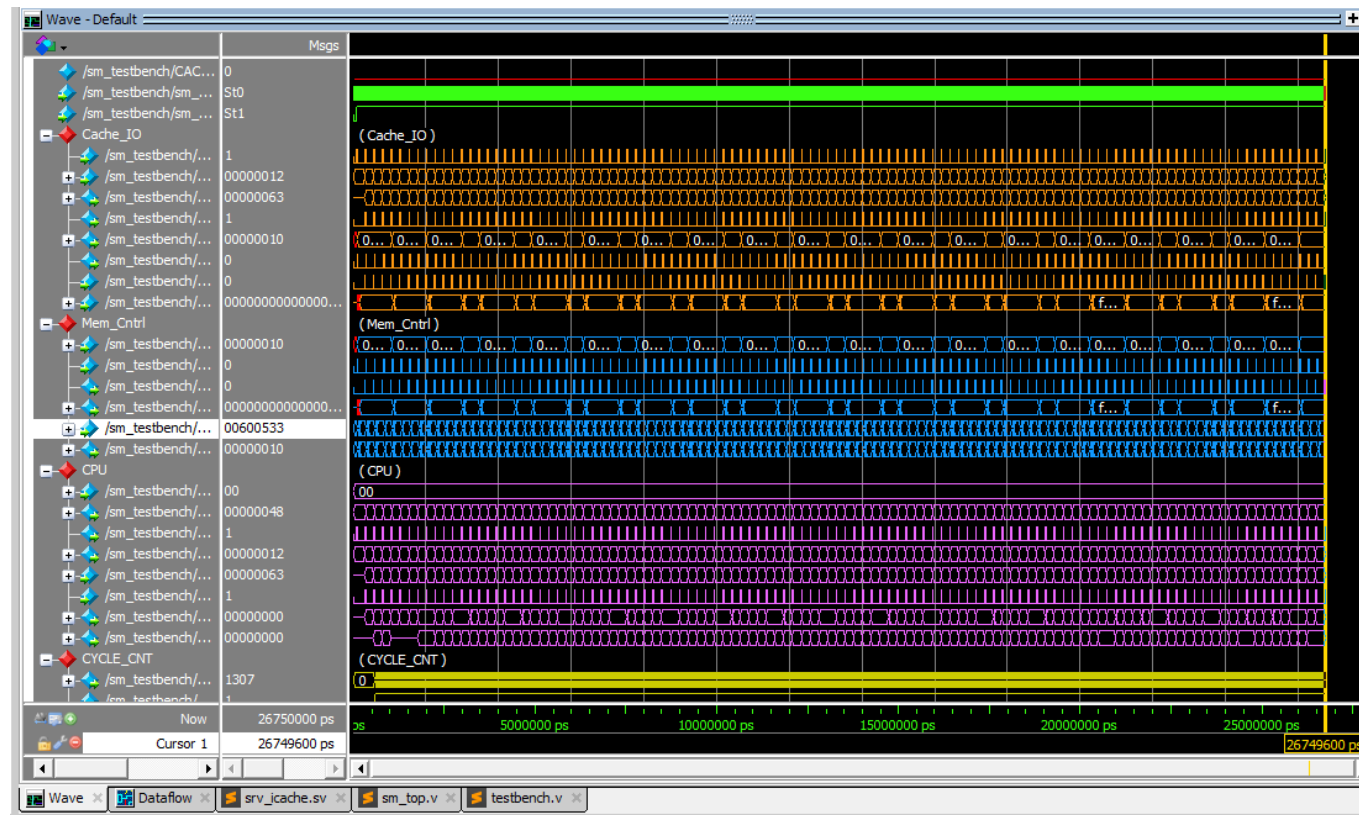
Project Memory List sim

Wave Dataflow srv\_jcache.sv sm\_top.v testbench.v

1

2

3



```

10
11 //hardware top level module
12 module sm_top
13 #(
14     parameter CACHE_EN = 1'b1
15 )
16 (
17     input          clkIn,
18     input          rst_n,
19     input [ 3:0 ]  clkDevide,
20     input          clkEnable,
21     output         clk,
22     input [ 4:0 ]  regAddr,
23     output [31:0]  regData,
24     output [31:0]  cycleCnt_o
25 );
26 //metastability input filters
27 wire [ 3:0 ] devide;
28 wire        enable;
29 wire [ 4:0 ] addr;
30 //instruction memory
31 wire [31:0] imAddr;
32 wire [31:0] imData;
33 wire        im_req;
34 wire        im_drdy;
35 wire [31:0] ext_addr;
36 wire        ext_req;
37 wire        ext_rsp;
38 wire [127:0] ext_data;
39 wire [31:0]  rom_data;

```

```

module srv_mem (
input logic clk,
input logic rst_n,

input  logic [31:0] ext_addr_i,
input  logic        ext_req_i,
output logic        ext_rsp_o,
output logic [127:0] ext_data_o,

input logic [31:0] rom_data_i,
output logic [31:0] rom_addr_o
);

localparam DEPTH      = 1024;
localparam AWIDTH     = 10;
localparam RD_NUM     = 128/32;
localparam MEM_DELAY  = 10;

```

Спасибо за внимание!