CSE 360-Computer Architecture

Lecture-4

Cache Memory

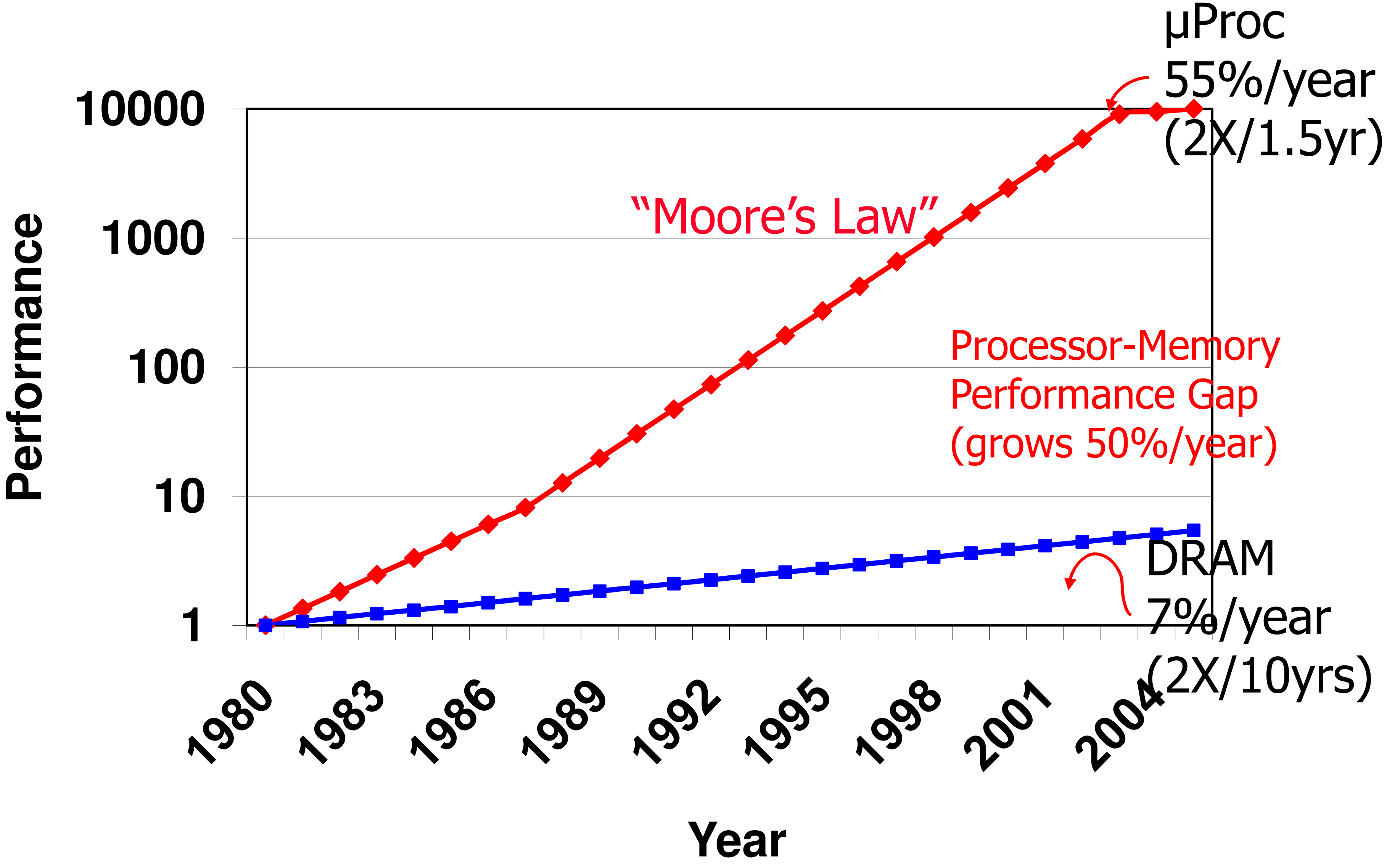
Dr. Shamim Akhter

Computer Science and Engineering

EAST WEST UNIVERSITY

excellence in education

Processor-Memory Performance Gap



µProc
55%/year
(2X/1.5yr)

"Moore's Law"

Processor-Memory
Performance Gap
(grows 50%/year)

DRAM
7%/year
(2X/10yrs)

Large and Fast: Exploiting Memory Hierarchy

Illusion of unlimited fast memory

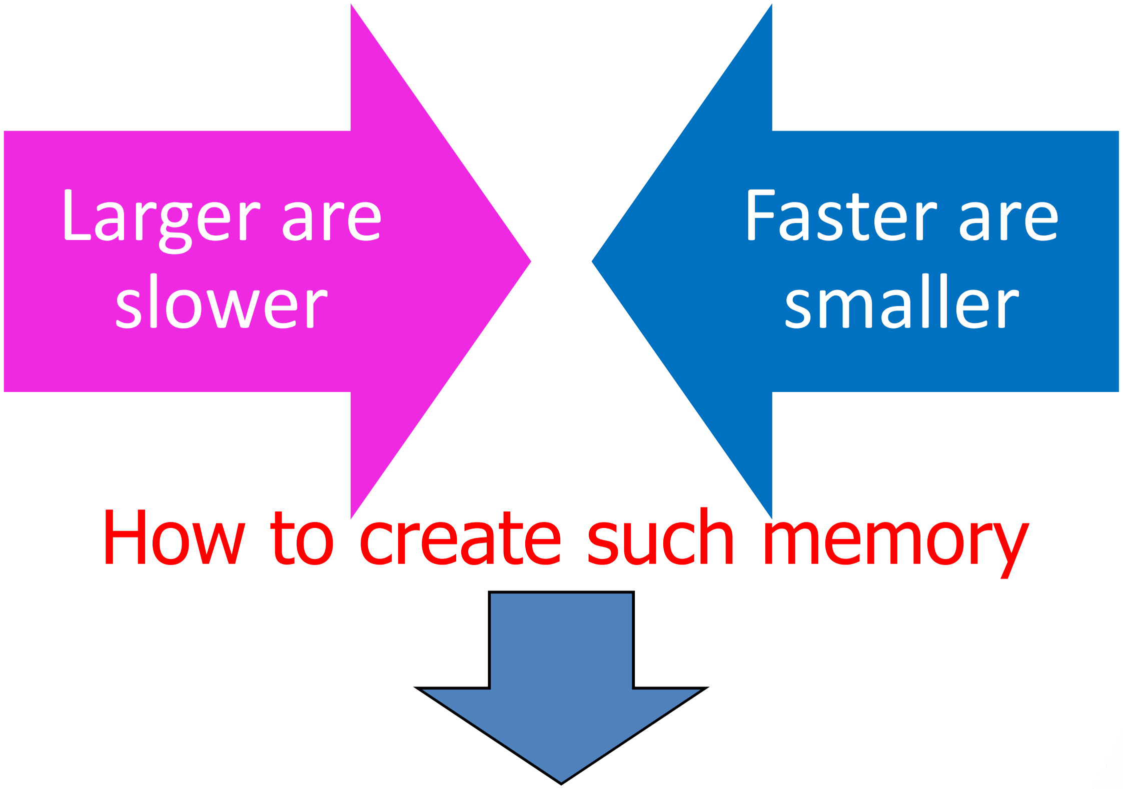
Introduction

Small latency

Unlimited memory

Google Data Center

Memory Facts



Larger are slower

Faster are smaller

How to create such memory

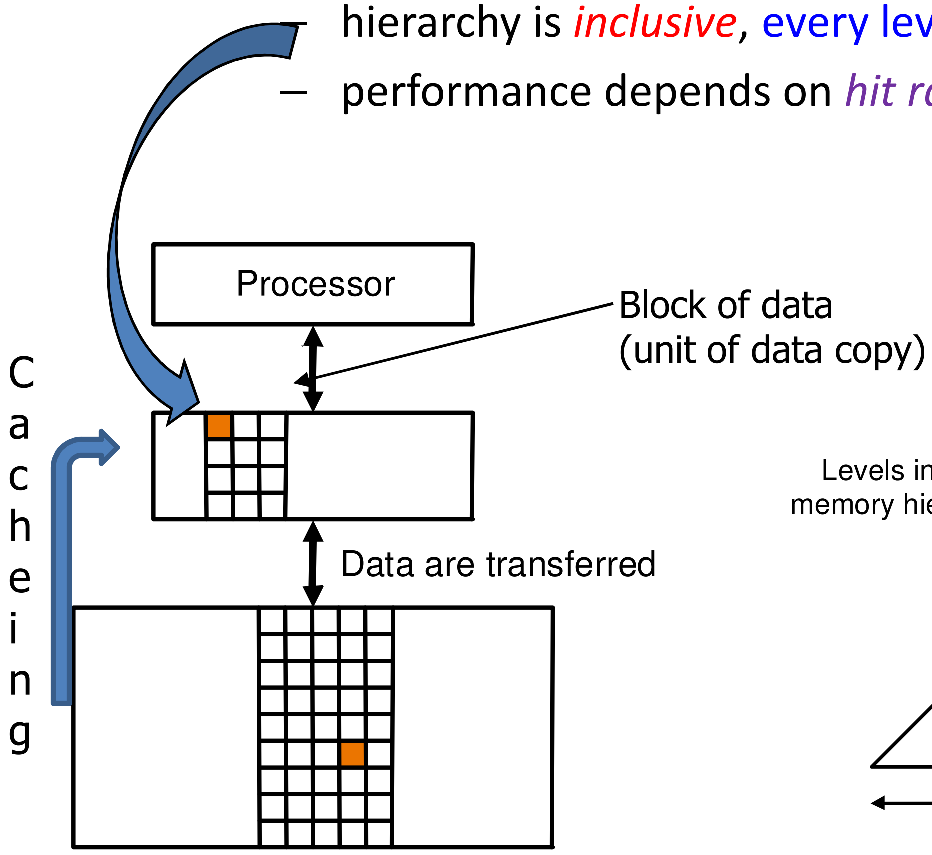
Illusion of being large, cheap and fast

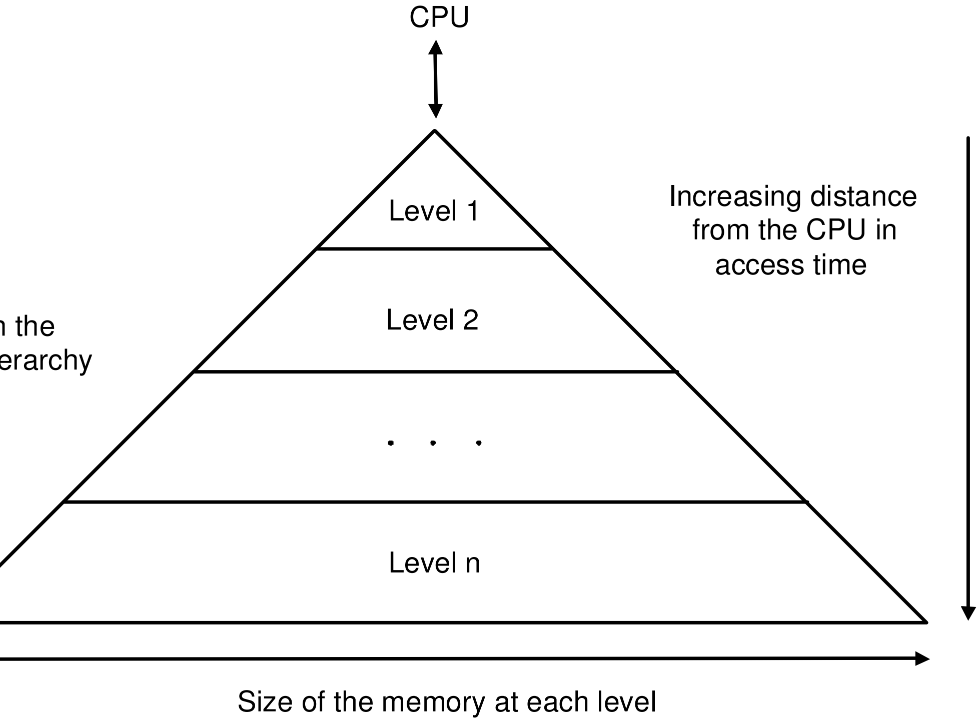
Memory Hierarchy

Memory Hierarchy

- Users want large and fast memories…
  - expensive but they don’t like to pay…

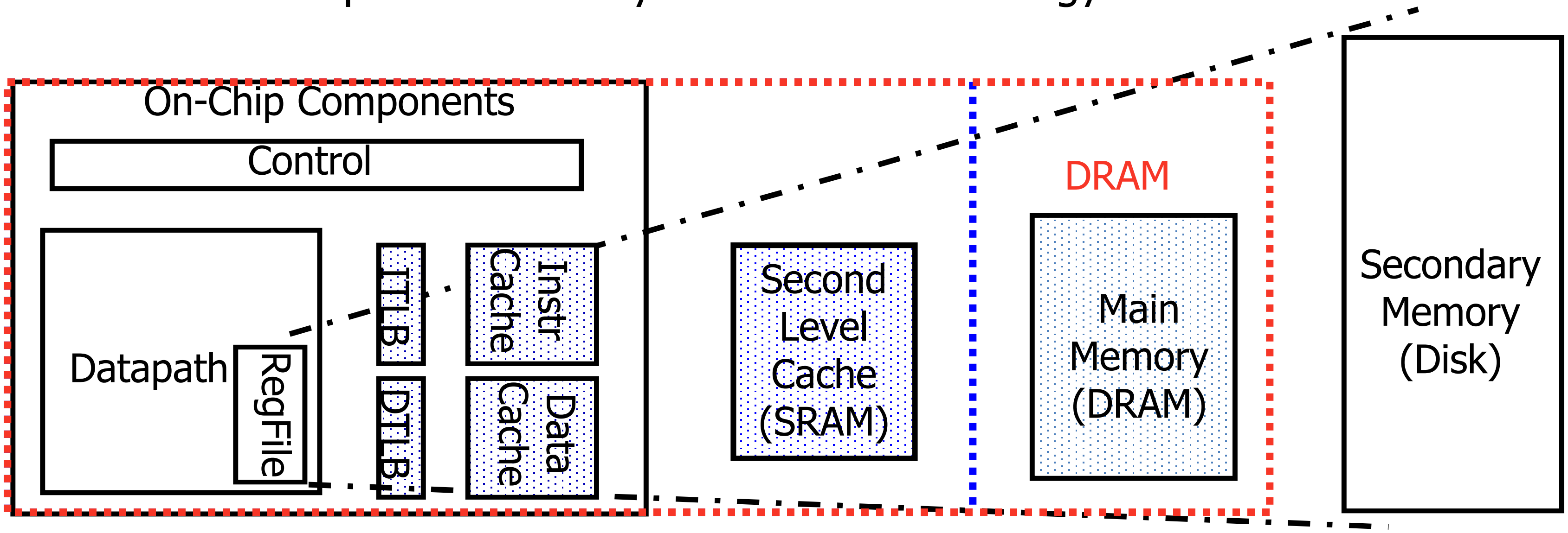
- Make it seems like (illusion)- they have what they want…
  - *memory hierarchy*
  - hierarchy is *inclusive*, every level is subset of lower level
  - performance depends on *hit rates*





A Typical Memory Hierarchy

- By taking advantage of memory hierarchy
  - Can present the user
    - with as much memory as is available
    - in the cheapest technology
    - at the speed offered by the fastest technology



Speed (%cycles):	½	1	10	100	1,000
Size (bytes):	100	K	10K	M	G to T
Cost:	highest				lowest

Cache Memory

A buffer memory that holds the currently most useful program segments and data, in much the same way that registers hold some of the currently active data elements for rapid access by the processor.

@ Behrooz Parhami

How does cache improve system performance?

Locality Principle

Cache applied locality principle to minimize the calls to memory.

Temporal locality

Same item will tend to referenced soon

Spatial locality

Once an instruction/data has been accessed, it /or its neighbors will be referenced soon.

Sequentiality

Reference made to a particular location *s* then it is likely to reference s+1 within the next several references . Sequentiality is a restricted type of spatial locality.

```
for(i = 0; i < 20; i++)
   for(j = 0; j < 10; j++)
         a[i] = a[i]*j;
```

Outer loop is an example of spatial locality.

-It sequentially increments the address of the inner for- loop calls.

A[0]	A[1]	A[2]	A[3]	….	…	A[19]

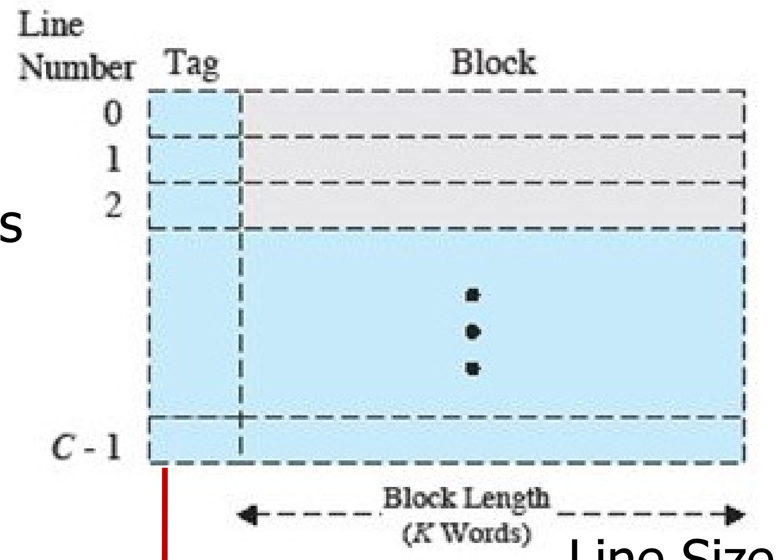
Inside loop demonstrates temporal locality.

- Same memory address is accessed ten times in a row, and multiplied by j each time.
- **Both i and j (loop counters) are very good examples of temporal locality**.

A[0]*0	A[0]*1	A[0]*2	A[0]*3	….		A[0]*9

Cache and Main Memory Architecture

m blocks
$m << M$

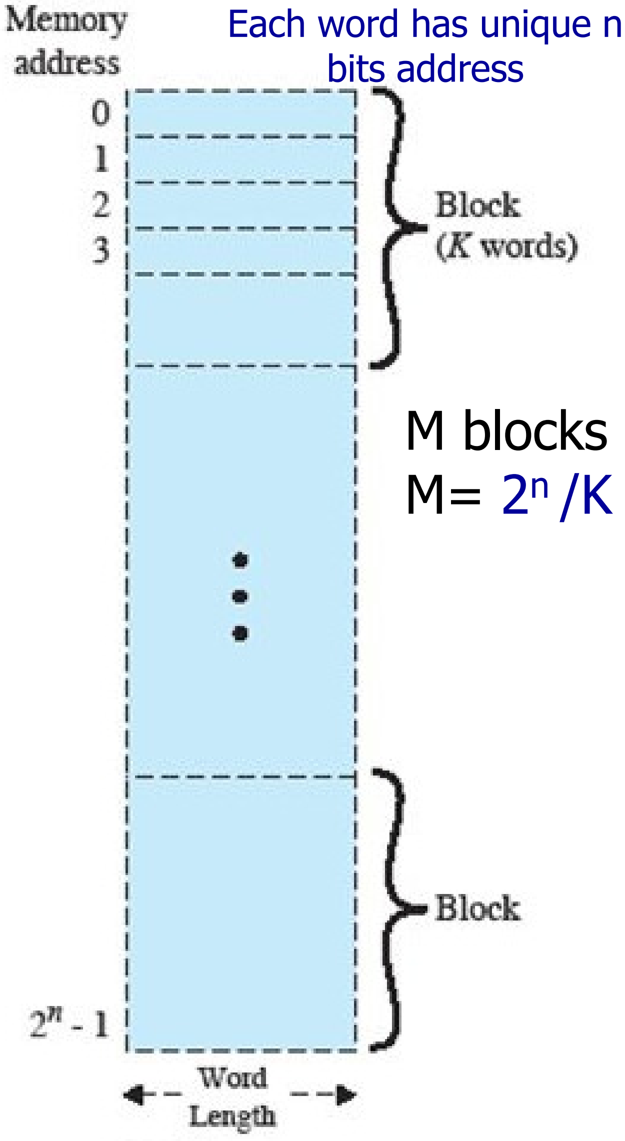


Line Size

(a) Cache

Each line also contains control bits Modify, Valid

2^n addressable words
Each word has unique n bits address

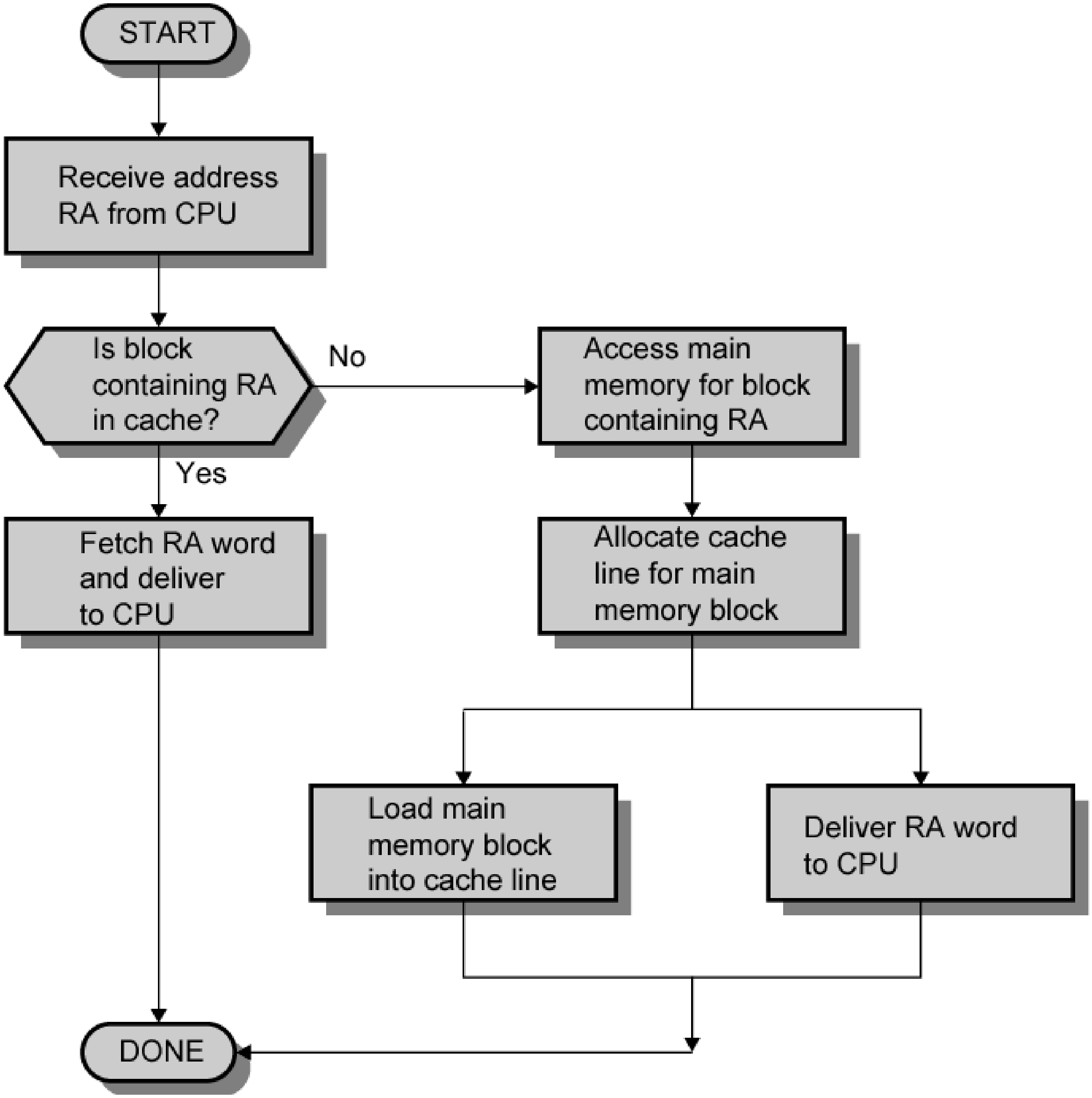


M blocks
$M = 2^n / K$

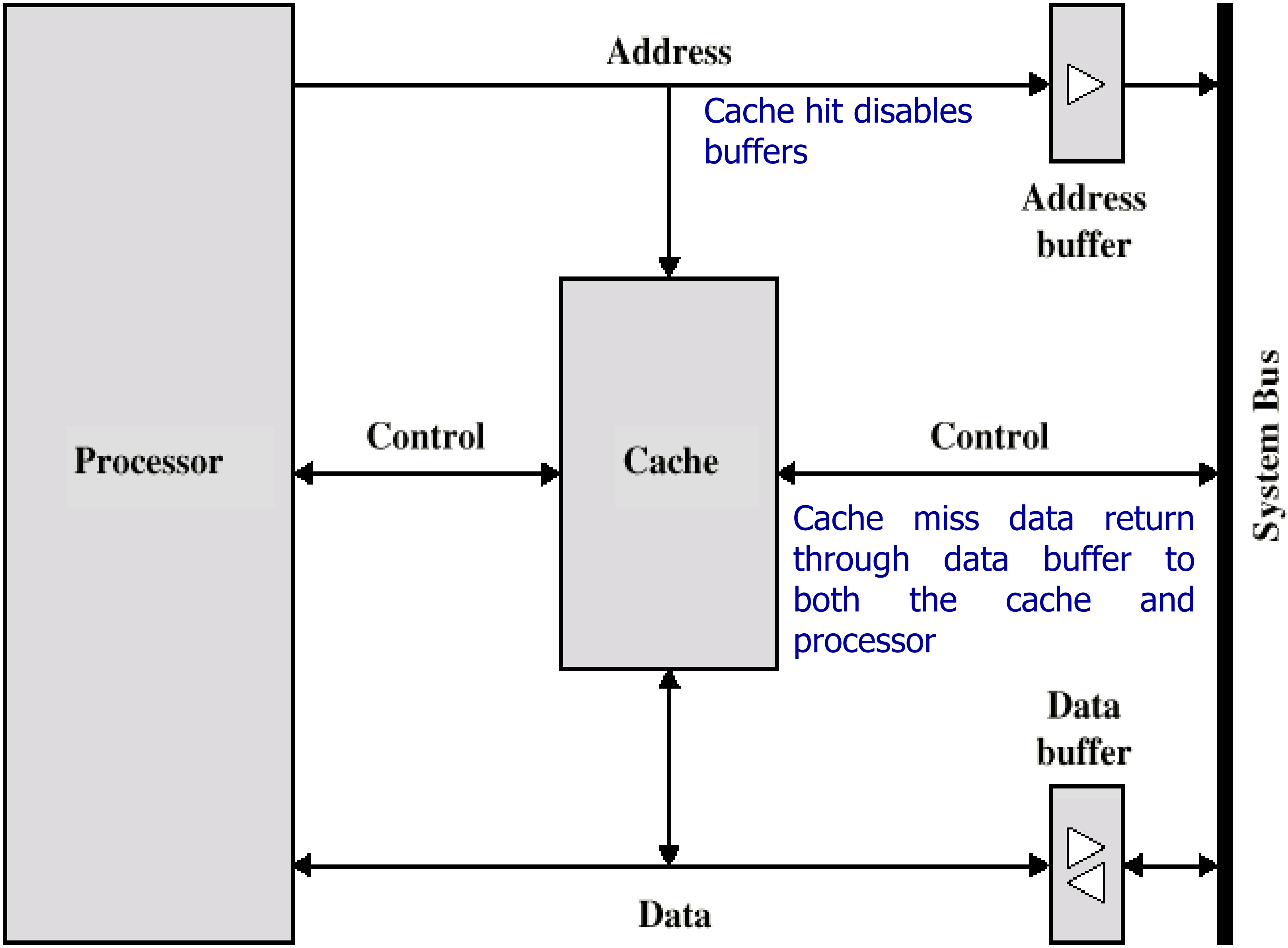
(b) Main memory

Cache Read Operation

RA=Read
Address


START
Receive address RA from CPU
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache line for main memory block
Load main memory block into cache line
Deliver RA word to CPU
DONE

Cache Organization

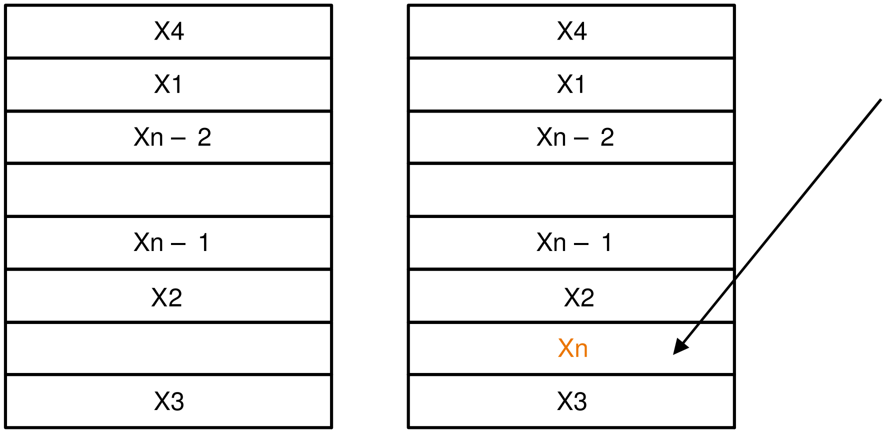


Cache hit disables buffers

Cache miss data return through data buffer to both the cache and processor

Caches

- By simple example
  - assume block size = one word of data = 32 bits



Reference to X_n causes miss so it is fetched from memory

a. Before the reference to Xn

b. After the reference to Xn

Issues:

I. how do we know if a data item is in the cache?
II. if it is, how do we find it?
III. if not, what do we do?

Solution depends on *cache addressing scheme*...

Cache Addressing Schemes

Direct Mapped

- One block per set

N-way Associative

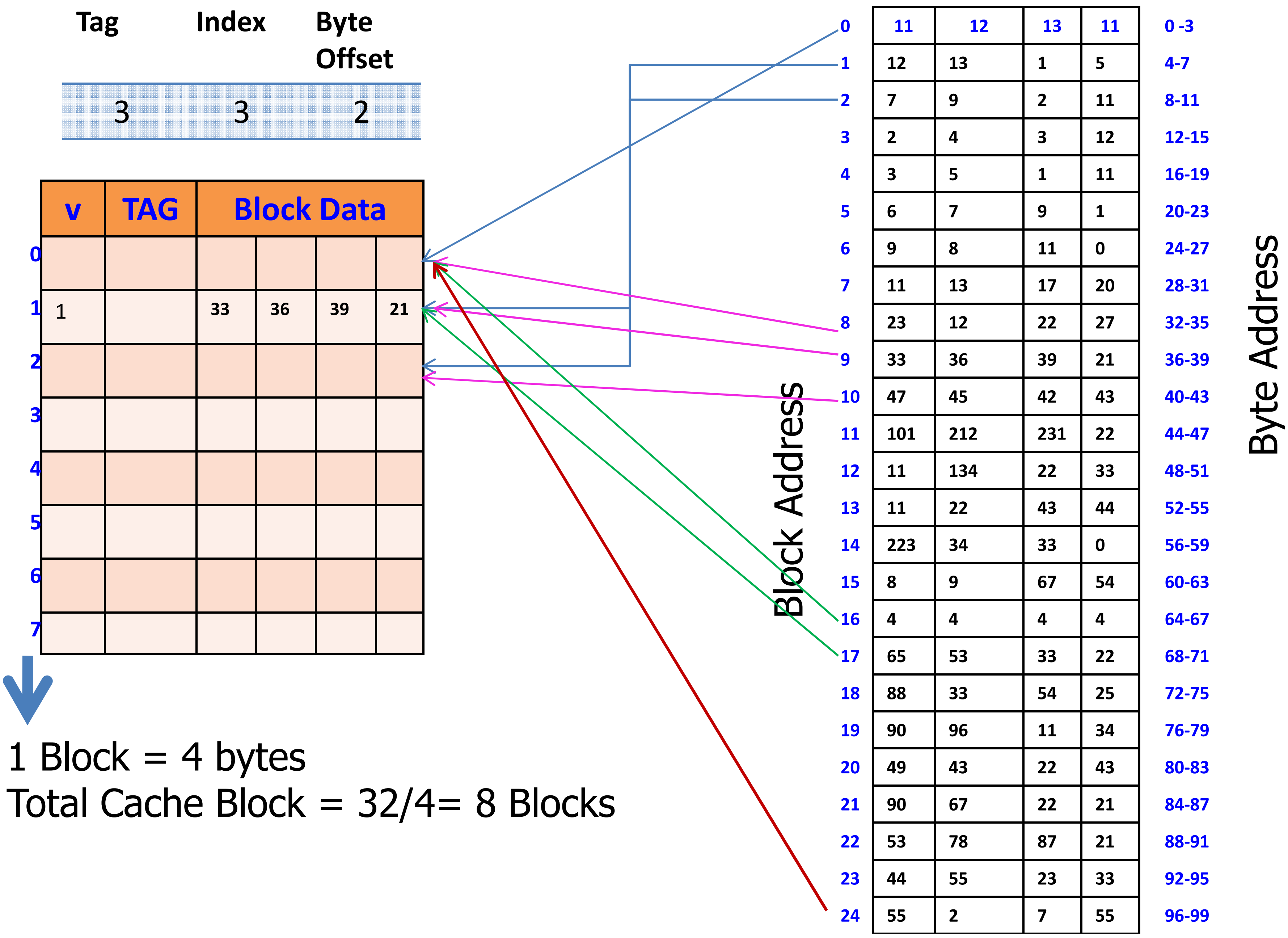
- N Blocks per set

Fully Associative

- All Blocks in one set

Direct- Mapped 32 bytes Cache

Tag	Index	Byte Offset
3	3	2



	v	TAG	Block Data			
0						
1	1		33	36	39	21
2						
3						
4						
5						
6						
7						

Block Address					Byte Address
0	11	12	13	11	0 -3
1	12	13	1	5	4-7
2	7	9	2	11	8-11
3	2	4	3	12	12-15
4	3	5	1	11	16-19
5	6	7	9	1	20-23
6	9	8	11	0	24-27
7	11	13	17	20	28-31
8	23	12	22	27	32-35
9	33	36	39	21	36-39
10	47	45	42	43	40-43
11	101	212	231	22	44-47
12	11	134	22	33	48-51
13	11	22	43	44	52-55
14	223	34	33	0	56-59
15	8	9	67	54	60-63
16	4	4	4	4	64-67
17	65	53	33	22	68-71
18	88	33	54	25	72-75
19	90	96	11	34	76-79
20	49	43	22	43	80-83
21	90	67	22	21	84-87
22	53	78	87	21	88-91
23	44	55	23	33	92-95
24	55	2	7	55	96-99

1 Block = 4 bytes
Total Cache Block = 32/4= 8 Blocks

Questions: Direct Mapped Cache

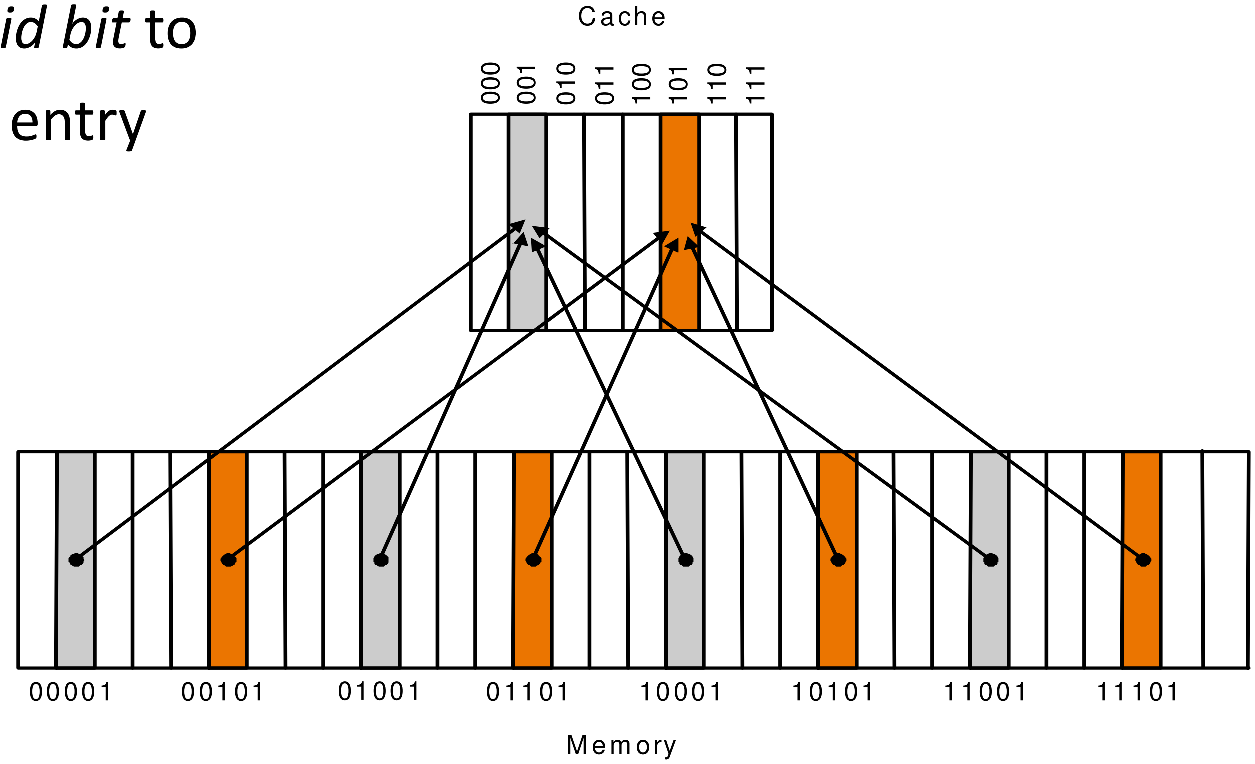
1. How do you define the main block and its corresponding cache block?
2. How to check the data availability in cache or not?

Solution 1: Simple

Cache block# = main block# % Total Cache block

Direct Mapped Cache

- **Addressing scheme in *direct mapped* cache:**
  - cache block address = memory block address % cache size (*unique*)
  - if cache size = 2^m,
    - cache address = lower m bits of n-bit memory address
    - remaining upper n-m bits kept as *tag bits* at each cache block
  - also need a *valid bit* to recognize valid entry



How to check the data availability in cache or not?

- Need to decide the address formats
  - 256(2^8)bytes main memory
  - 4 bytes block size
  - # of blocks 256/4= 64
- Cache size 8 blocks
- Lets ref. **39 address** from main memory
  - Bit address 00100111

Why do we need Tag?

Memory Address Formats

Block #	Byte Offset
6	2

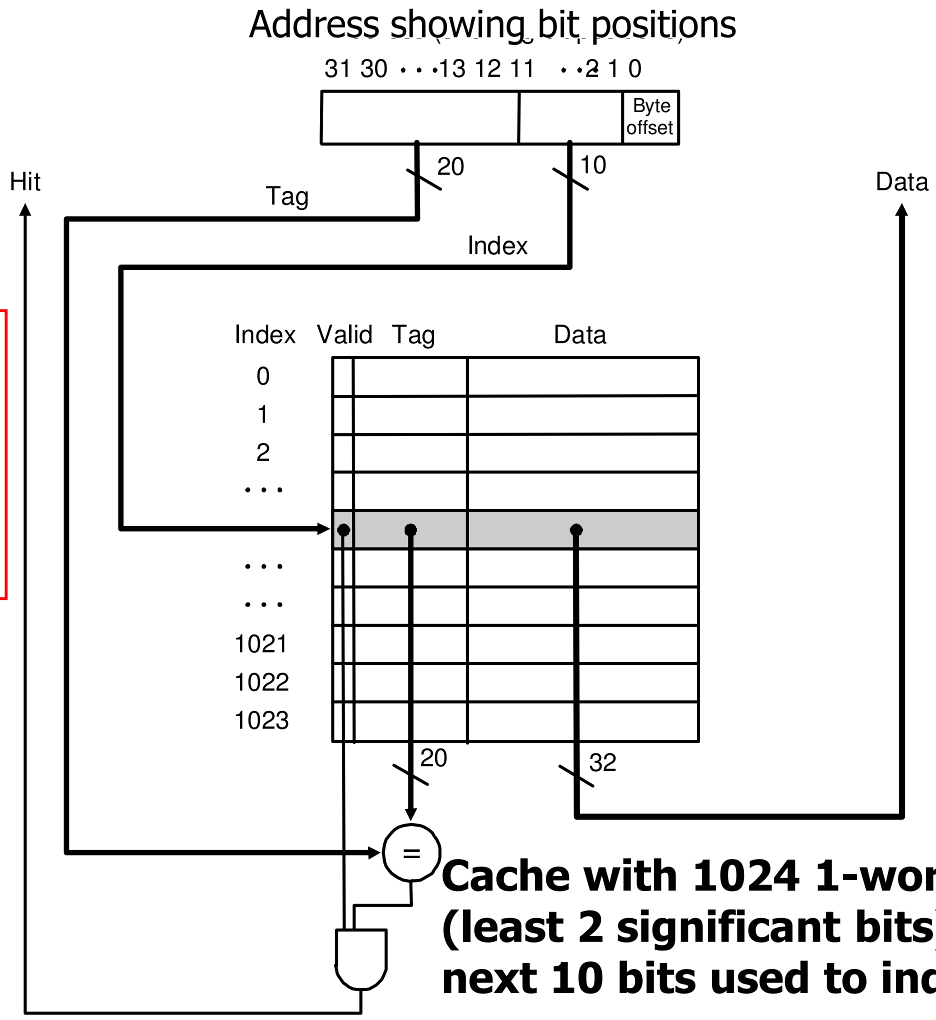
Cache Address Formats

Tag	Index	Byte Offset
3	3	2

Direct Mapped Cache

- MIPS style:

Recently address words replace less recently referenced words.

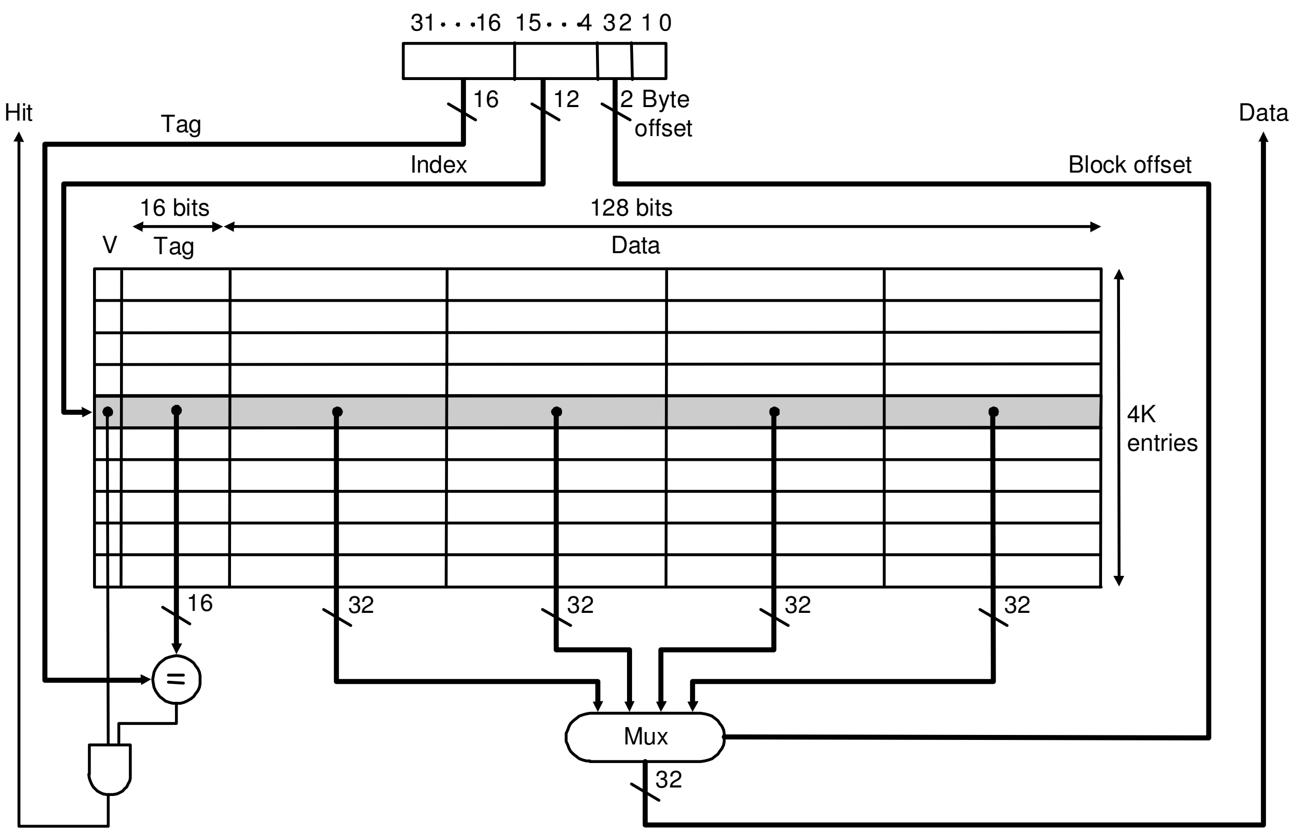


Cache with 1024 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 10 bits used to index into cache

Direct Mapped Cache: Taking Advantage of Spatial Locality

- Taking advantage of spatial locality with *larger* blocks:

Address showing bit positions



Cache with 4K 4-word blocks: *byte offset* (least 2 significant bits) is ignored, next 2 bits are *block offset*, and the next 12 bits are used to index into cache

Example Problem

- In a Direct Mapped Cache:
- Consider a cache with **64 blocks and a block size of 16 bytes**. What block # does the byte address 1200 map to?

- The block is given by

  (Block address) module (# of cache blocks)
- Where the block address is

  Byte address/Bytes per block

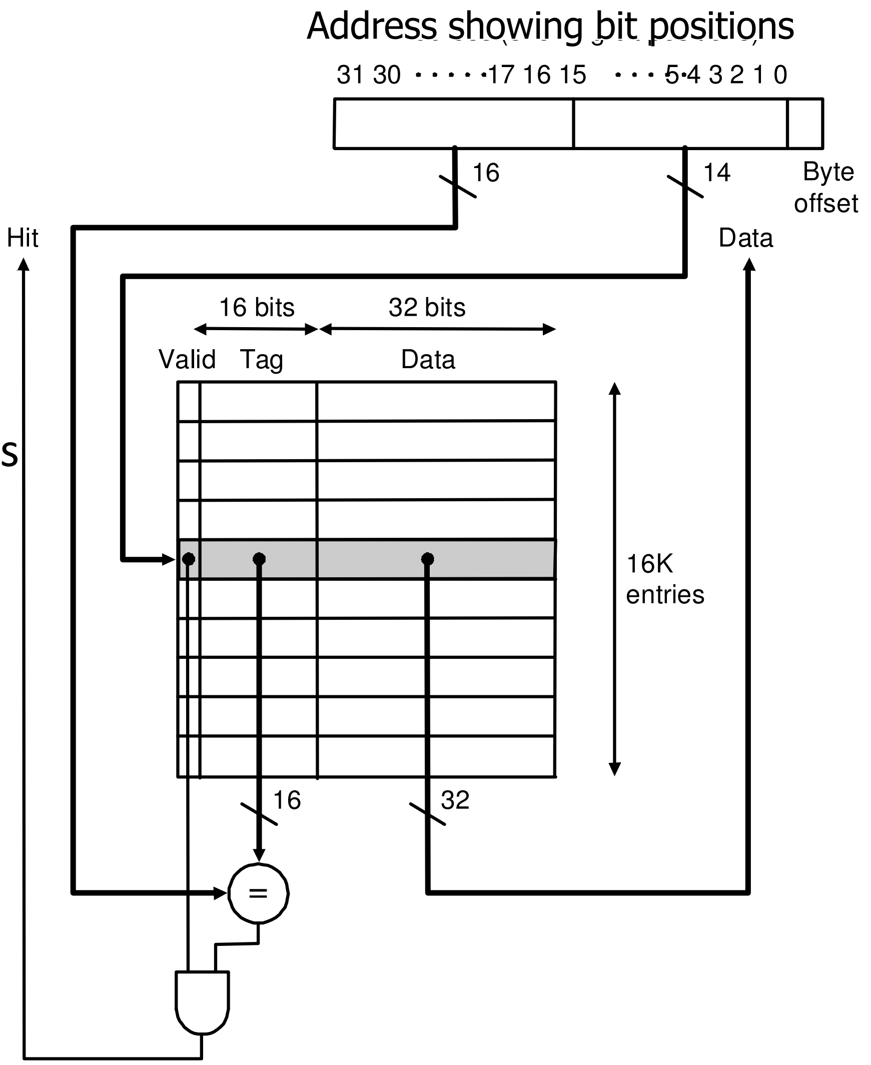
- Thus, with 16 bytes per block, byte address 1200 is block address

  Floor(1200/16)= 75
- which maps to cache block # **(75 module 64) =11**

Example: DECStation 3100 Cache (MIPS R2000 processor)

Cache with 16K 1-word blocks: *byte offset* (least 2 significant bits) is ignored and next 14 bits used to index into cache

- How many total bits are required for a direct-mapped cache with 64 KB of data in total and one-word blocks, assuming a 32-bit address?

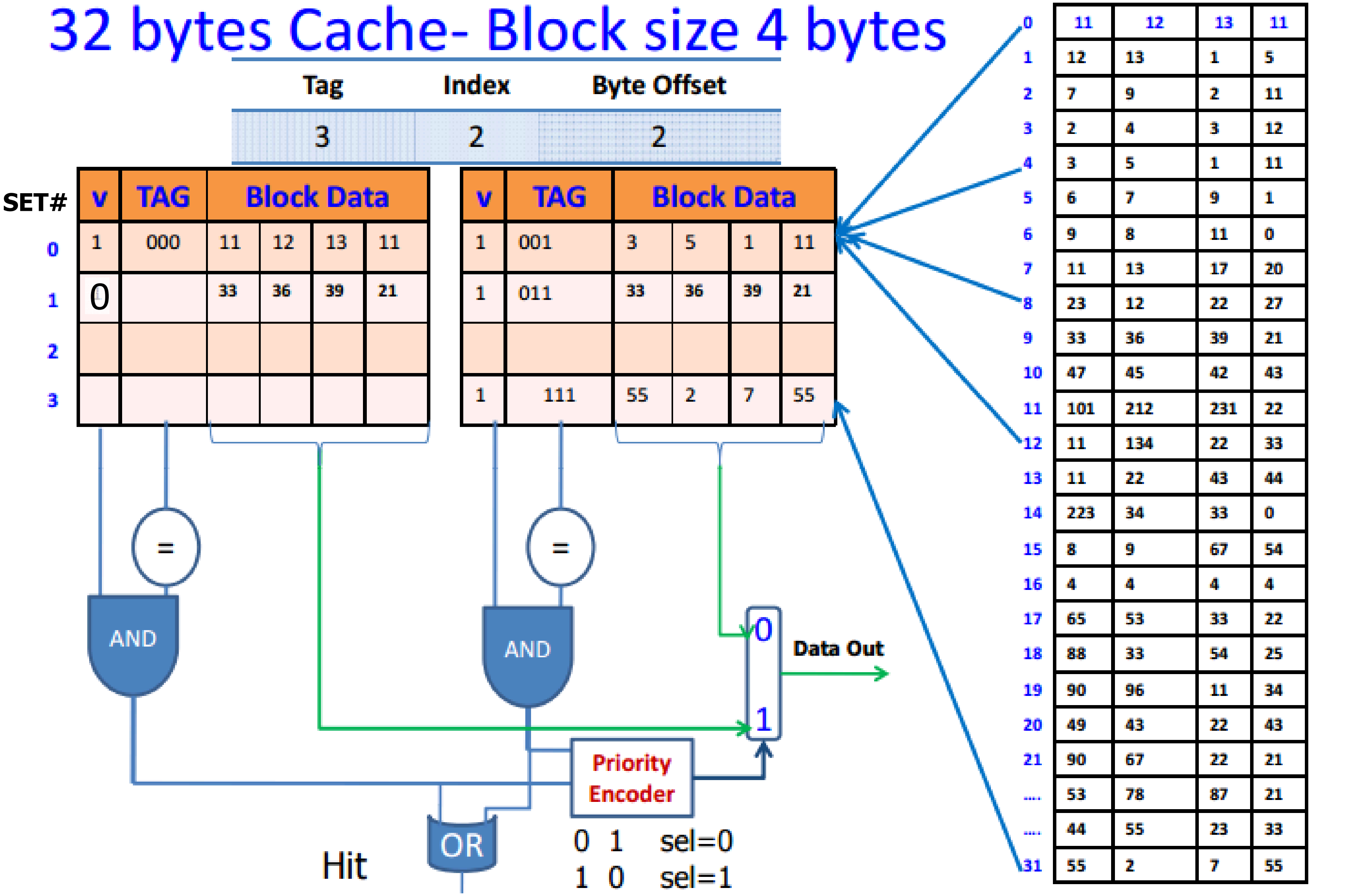


- 64 KB=16K Word=16x1024=2^{14} words=2^{14} blocks
- Thus, Tag= (32-14-2)=16 bits
- Each 2^{14} words has 16bits tag+ 1 bit validity bit
- Total Size=2^{14} x (32+17) = 2^{10} x 784 = 784 Kbits=98KB
- Cache size~1.5 times bigger than total storage data

2-Way Set Associative- 2 blocks per set

32 bytes Cache- Block size 4 bytes

Tag	Index	Byte Offset
3	2	2



SET#	v	TAG	Block Data				v	TAG	Block Data			
0	1	000	11	12	13	11	1	001	3	5	1	11
1	0		33	36	39	21	1	011	33	36	39	21
2												
3							1	111	55	2	7	55

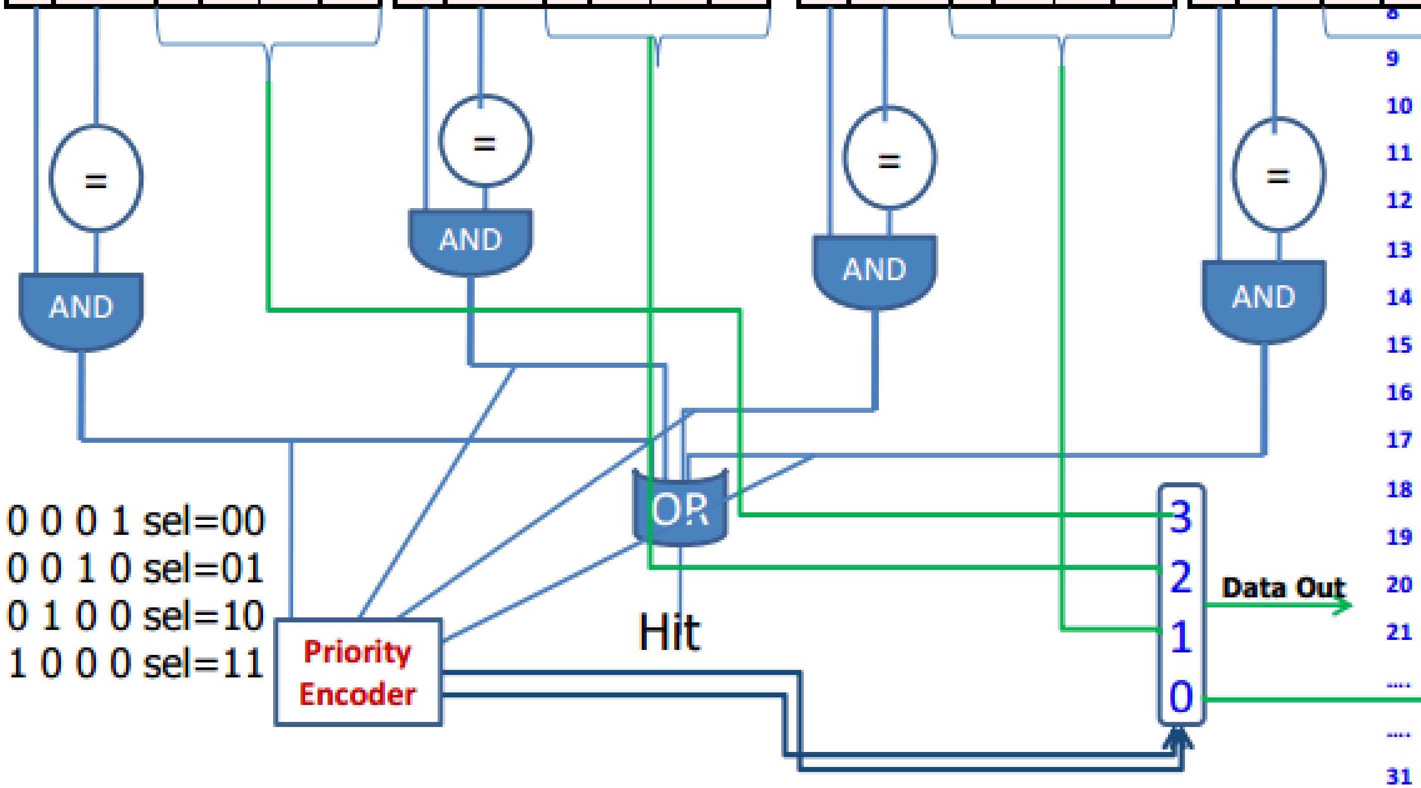
Block	Byte Offset
5	2

0	11	12	13	11
1	12	13	1	5
2	7	9	2	11
3	2	4	3	12
4	3	5	1	11
5	6	7	9	1
6	9	8	11	0
7	11	13	17	20
8	23	12	22	27
9	33	36	39	21
10	47	45	42	43
11	101	212	231	22
12	11	134	22	33
13	11	22	43	44
14	223	34	33	0
15	8	9	67	54
16	4	4	4	4
17	65	53	33	22
18	88	33	54	25
19	90	96	11	34
20	49	43	22	43
21	90	67	22	21
....	53	78	87	21
....	44	55	23	33
31	55	2	7	55

4-Way Set Associative- 4 blocks per set
32 bytes Cache- Block size 4 bytes

Tag	Index	Byte Offset
4	1	2

SET#	v	TAG	Block Data				v	TAG	Block Data				v	TAG	Block Data				v	TAG	Block Data			
0	1	0000	1	3	1	5	1	0001	7	9	2	1	1	1010	4	3	2	4						
1	1	1010	9	6	2	1	1	0111	8	9	5	7	1	1111	5	2	7	5						

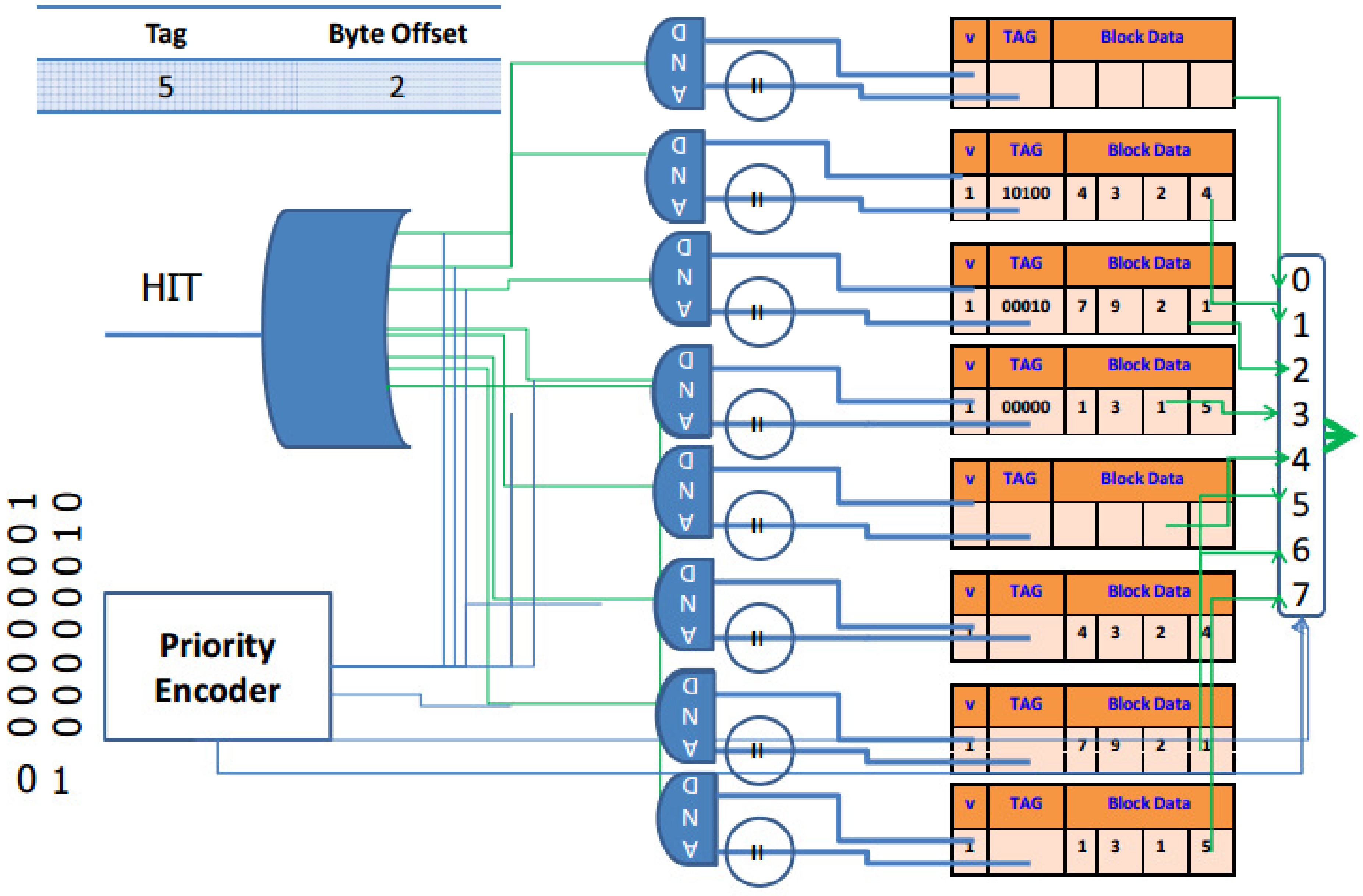


0 0 0 1 sel=00
0 0 1 0 sel=01
0 1 0 0 sel=10
1 0 0 0 sel=11

Block	Byte Offset
5	2

0	1	3	1	5
1	2	1	1	5
2	7	9	2	1
3	2	4	3	12
4	3	5	1	11
			9	1
			11	0
			17	20
8				
9				
10				
11				
12	11	134	22	33
13	11	22	43	44
14	223	34	33	0
15	8	9	7	5
16	4	4	4	4
17	65	53	33	22
18	88	33	54	25
19	90	96	11	34
20	4	3	2	4
21	9	6	2	1
....	53	78	87	21
....	44	55	23	33
31	5	2	7	5

Fully Set Associative- All Blocks in One Set



Tag	Byte Offset
5	2

v	TAG	Block Data			
1	10100	4	3	2	4
1	00010	7	9	2	1
1	00000	1	3	1	5
1		4	3	2	4
1		7	9	2	1
1		1	3	1	5

- If the total size is kept the same,
  - increasing associatively increases the # of blocks per set (# of comparators)
  - Increase factor of 2 in associatively
    - doubles the # of blocks per set and halves the # of sets.
    - decreases the size of the index by 1 bit
    - increases the size of the tag by 1 bit.
  - Fully associative cache
    - Only one set and all blocks must be checked in parallel
    - No index
    - Tag address address-block offsets
  - Direct-mapped cache
    - Single comparator is needed (only one block)

What Block Should be Replaced on a Cache READ Miss?

- When a miss occurs in an associative cache- which block to replace?
  - Random: Candidate blocks are randomly selected.
  - Least recently used (LRU): Candidate block is the one that has been unused for the longest time.
  - First in First Out (FIFO): Selects the oldest rather than the LRU block.

Example Problems

- Find the number of misses for a cache with four 1-word blocks given the following sequence of memory block accesses:

  0, 8, 0, 6, 8

  for each of the following cache configurations

1. **direct mapped**
2. **2-way set associative (use LRU replacement policy)**
3. **fully associative**

Solution

- 1 (direct-mapped)

Block address	Cache block
0	0 (= 0 *mod* 4)
6	2 (= 6 *mod* 4)
8	0 (= 8 *mod* 4)

Block address translation in direct-mapped cache

Address of memory	**Hit or**	**Contents of cache blocks after reference**			
block accessed	**miss**	**0**	**1**	**2**	**3**
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

- 5 misses

Solution (cont.)

- 2 (two-way set-associative)

Block address	Cache set
0	0 (= 0 *mod* 2)
6	0 (= 6 *mod* 2)
8	0 (= 8 *mod* 2)

mod set number

Block address translation in a two-way set-associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	**Set 0**	**Set 1**	**Set 1**
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

- Four misses

LRU page replacement

Solution (cont.)

- 3 (fully associative)

Address of memory	Hit or	Contents of cache blocks after reference			
block accessed	miss	Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

- 3 misses

What Happens on a Write?

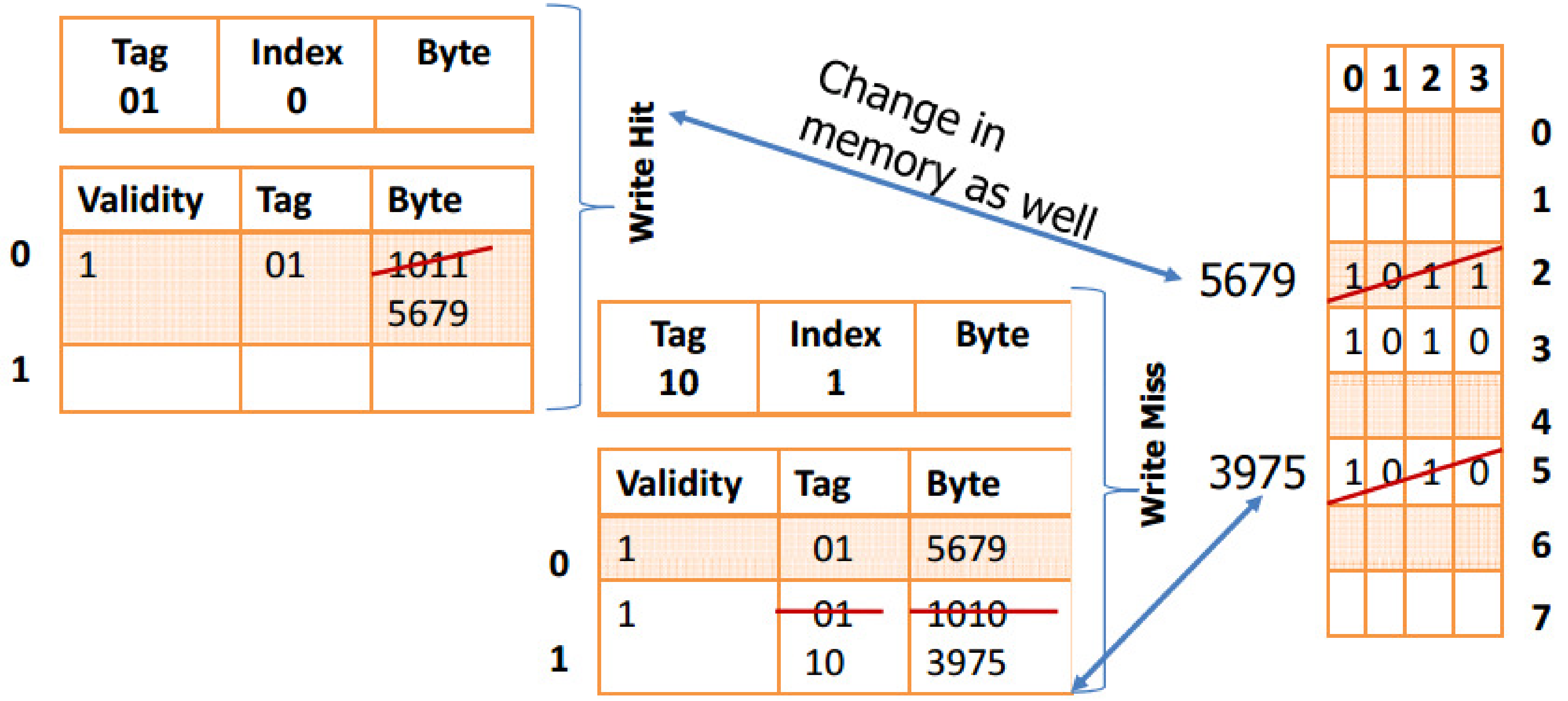
- Two approaches
  - Write-Through Scheme
  - Write-back Scheme

Cache Write Hit/Miss (Ex. store op)

1-word blocks

- **Write-through scheme**
  - on write hit: replace data in cache and memory blocks with every write hit to avoid inconsistency
  - on write miss: update tag, valid bit, write the word into cache and memory

✗ obviously no need to read missed word from memory!



Tag	Index	Byte
01	0	

	Validity	Tag	Byte
0	1	01	~~1011~~ 5679
1			

Tag	Index	Byte
10	1	

	Validity	Tag	Byte
0	1	01	5679
1	1	~~01~~ 10	~~1010~~ 3975

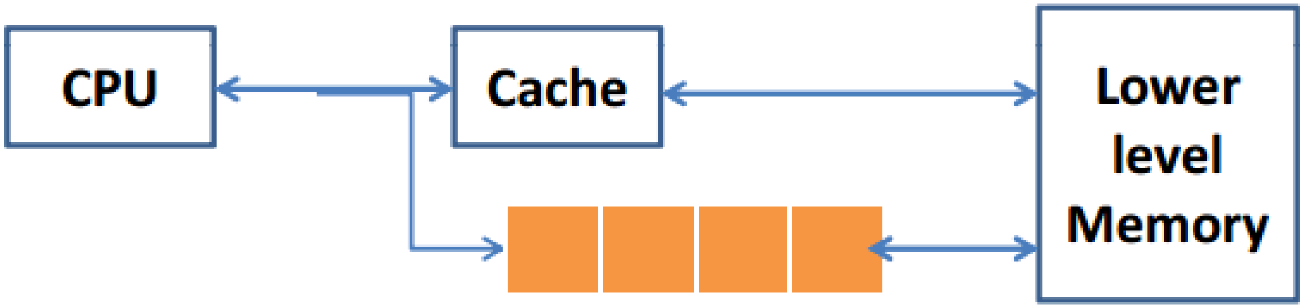
		0	1	2	3
0					
1					
2	5679	~~1~~	~~0~~	~~1~~	~~1~~
3		1	0	1	0
4					
5	3975	~~1~~	~~0~~	~~1~~	~~0~~
6					
7					

Disadvantage of Write-through Memory Write

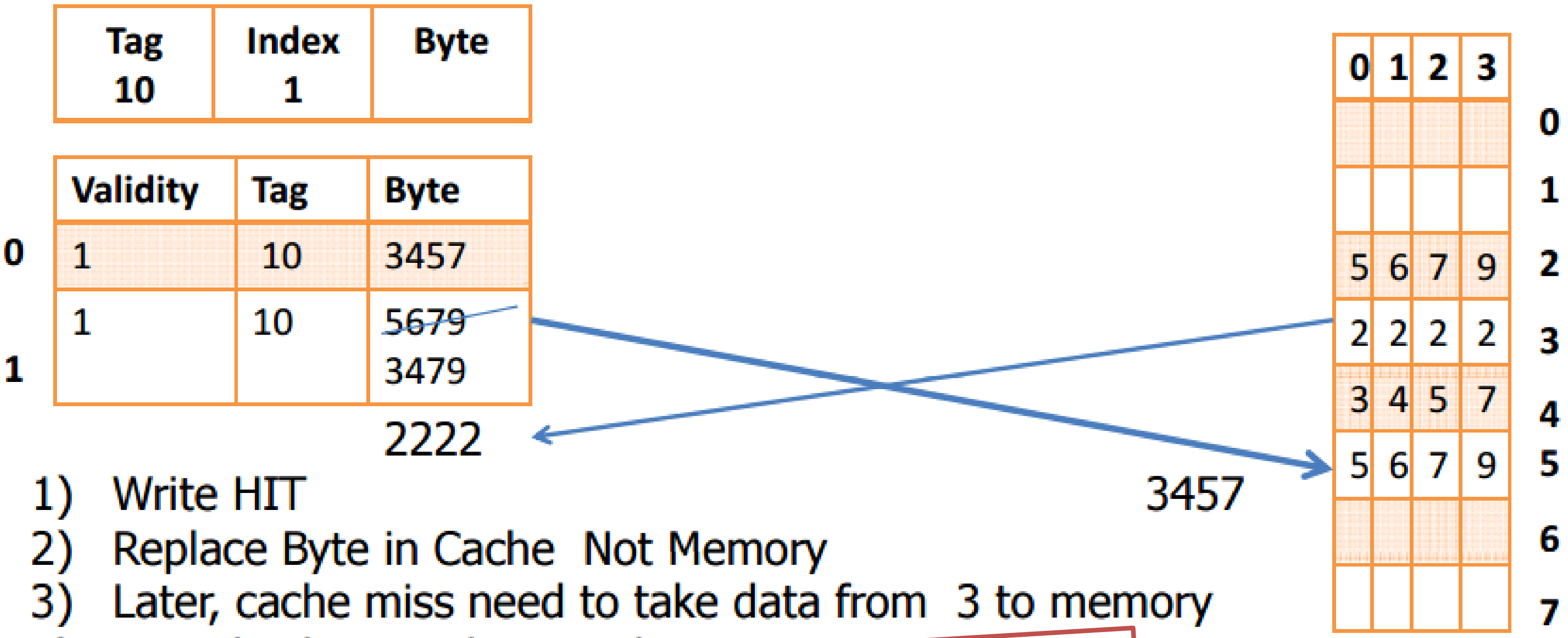
– Write-through is slow because of always required memory write

- performance is improved with a **write buffer** where words are stored while waiting to be written to memory – processor can continue execution until write buffer is full.
- when a word in the write buffer completes writing into memory that buffer slot is freed and becomes available for future writes

- DEC 3100 write buffer has 4 words



- **Write-back scheme**
  - write the data block only into the cache and write-back the block to memory only **when it is replaced in cache**
  - more efficient than write-through, more complex to implement



Tag 10	Index 1	Byte

	Validity	Tag	Byte
0	1	10	3457
1	1	10	~~5679~~ 3479

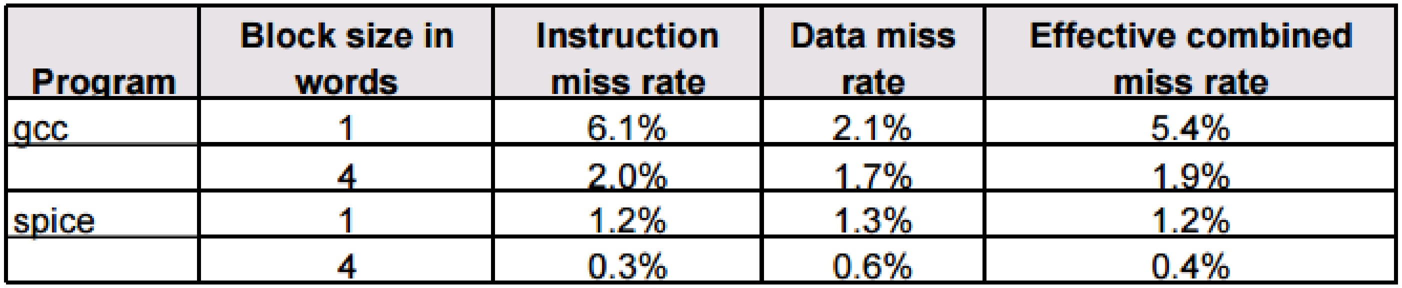
0	1	2	3	
				0
				1
5	6	7	9	2
2	2	2	2	3
3	4	5	7	4
5	6	7	9	5
				6
				7

1) Write HIT
2) Replace Byte in Cache Not Memory
3) Later, cache miss need to take data from 3 to memory
4) Write back 3479 then read 2222

Read miss

Improving Cache Performance

- Use split caches for instruction and data because there is more spatial locality in instruction references:

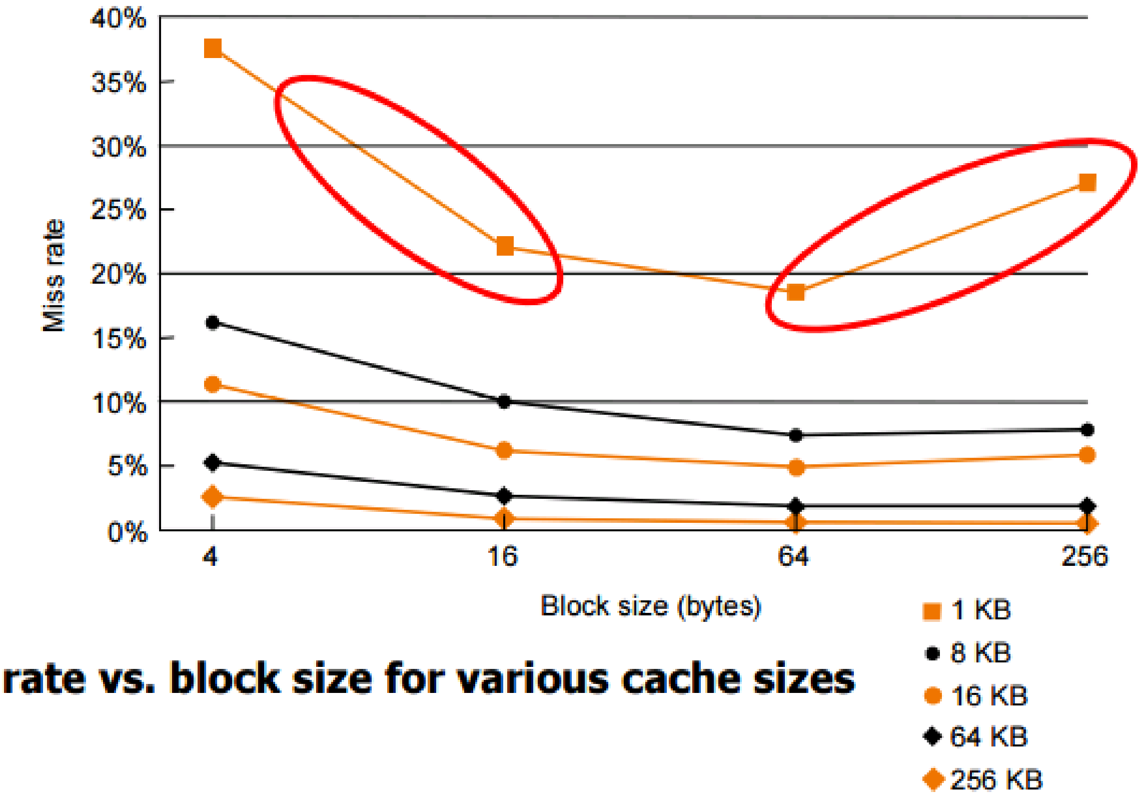


Program	Block size in words	Instruction miss rate	Data miss rate	Effective combined miss rate
gcc	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
spice	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

Miss rates for gcc and spice in a MIPS R2000 with one and four word block sizes

Direct Mapped Cache
Taking Advantage of Spatial Locality

- Miss rate falls at first with increasing block size as expected, but, as block size becomes a large fraction of total cache size, miss rate may go up because
  - there are few blocks
  - competition for blocks increases
  - blocks get ejected before most of their words are accessed (*thrashing* in cache)



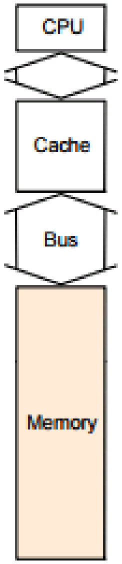
Miss rate vs. block size for various cache sizes

Improving Cache Performance by Increasing Bandwidth

Interconnectivity

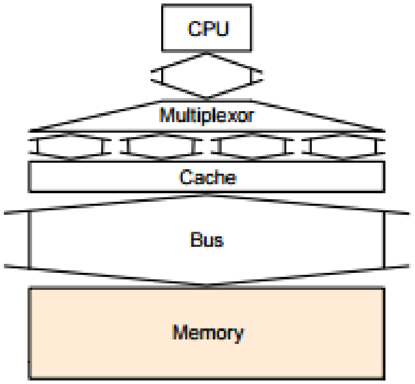
- Assume:
  - cache block of 4 words
  - 1 clock cycle to send address to memory address buffer (1 bus trip)
  - 15 clock cycles for each memory data access
  - 1 clock cycle to send data to memory data buffer (1 bus trip)

4 words cache block



a. One-word-wide memory organization

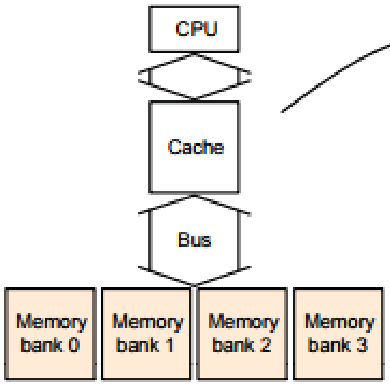
1 + 4*15 + 4*1 = 65 cycles



b. Wide memory organization

4 word wide memory and bus

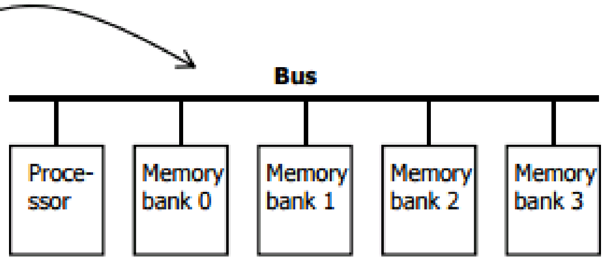
1 + 1*15 +1*1 = 17 cycles



c. Interleaved memory organization

4 word wide memory only

1 +1*15 + 4*1 = 20 cycles



Interleaved memory units compete for bus

i%n – n-way interleaved

Miss penalties

Memory wide 2 words and bus