Lecture 8

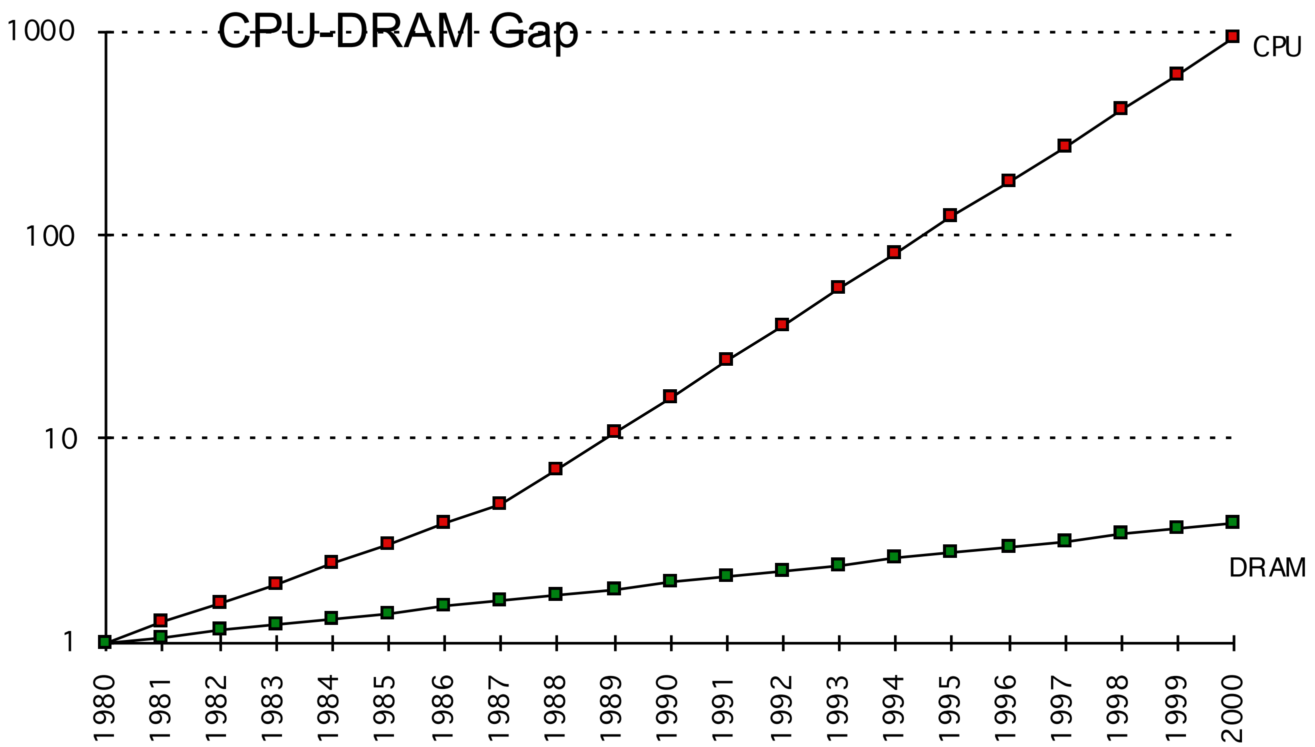
Memory Hierarchy

Memory Hierarchy

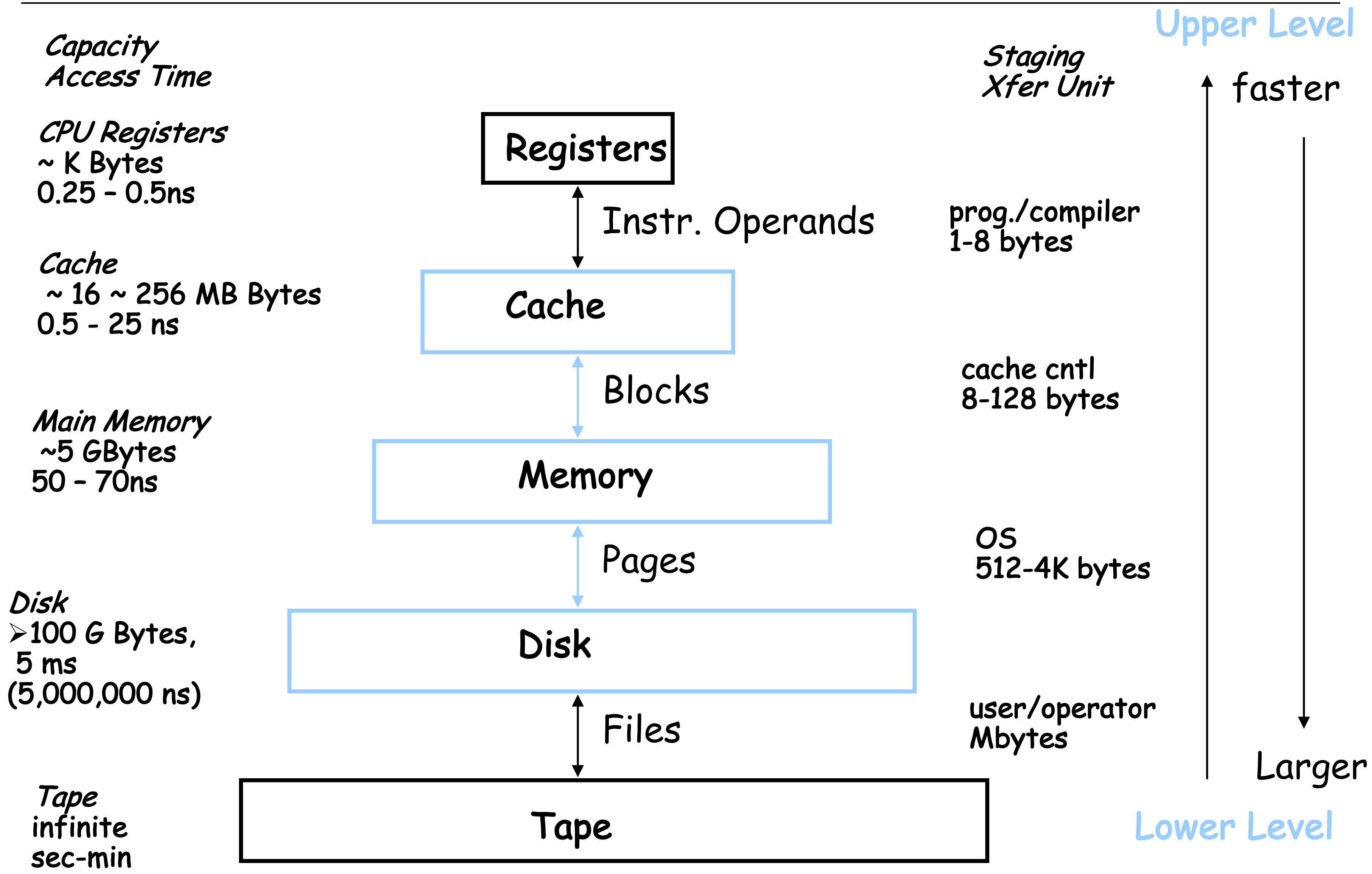
- Keeping useful books close to you
- Keeping useful data close to the CPU
  - Memory Hierarchy

Why Cares about Memory Hierarchy?

- Processor Only Thus Far in Course
  - – CPU cost/performance, ISA, Pipelined Execution



Levels of the Memory Hierarchy



Capacity
Access Time

Staging
Xfer Unit

Upper Level

faster

CPU Registers
~ K Bytes
0.25 – 0.5ns

Registers

Instr. Operands

prog./compiler
1-8 bytes

Cache
~ 16 ~ 256 MB Bytes
0.5 - 25 ns

Cache

Blocks

cache cntl
8-128 bytes

Main Memory
~5 GBytes
50 – 70ns

Memory

Pages

OS
512-4K bytes

Disk
➢100 G Bytes,
5 ms
(5,000,000 ns)

Disk

Files

user/operator
Mbytes

Tape
infinite
sec-min

Tape

Larger

Lower Level

General Principle

- The Principle of Locality:
  - Programs access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time):
    - If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space):
    - If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)

```
For (i= 0; i++;i:<100)
     {
       A[i]=A[i] + x;
       x++;
      }
```

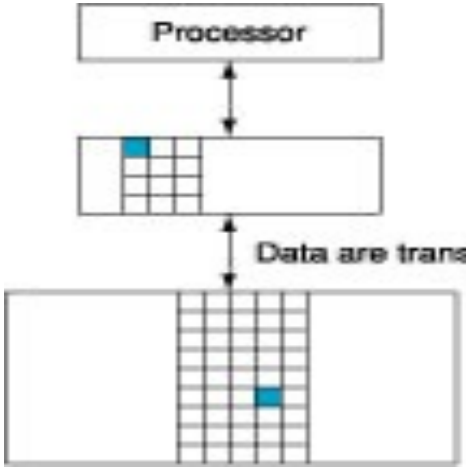
- Locality + smaller HW is faster = memory hierarchy
  - *Levels*: each smaller, faster, more expensive/byte than level below
  - *Inclusive*: data found in top also found in the bottom

Memory Hierarchy: Terminology

- Definitions
  - *Upper* is closer to processor
  - *Block*: minimum unit that present or not in upper level
- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of

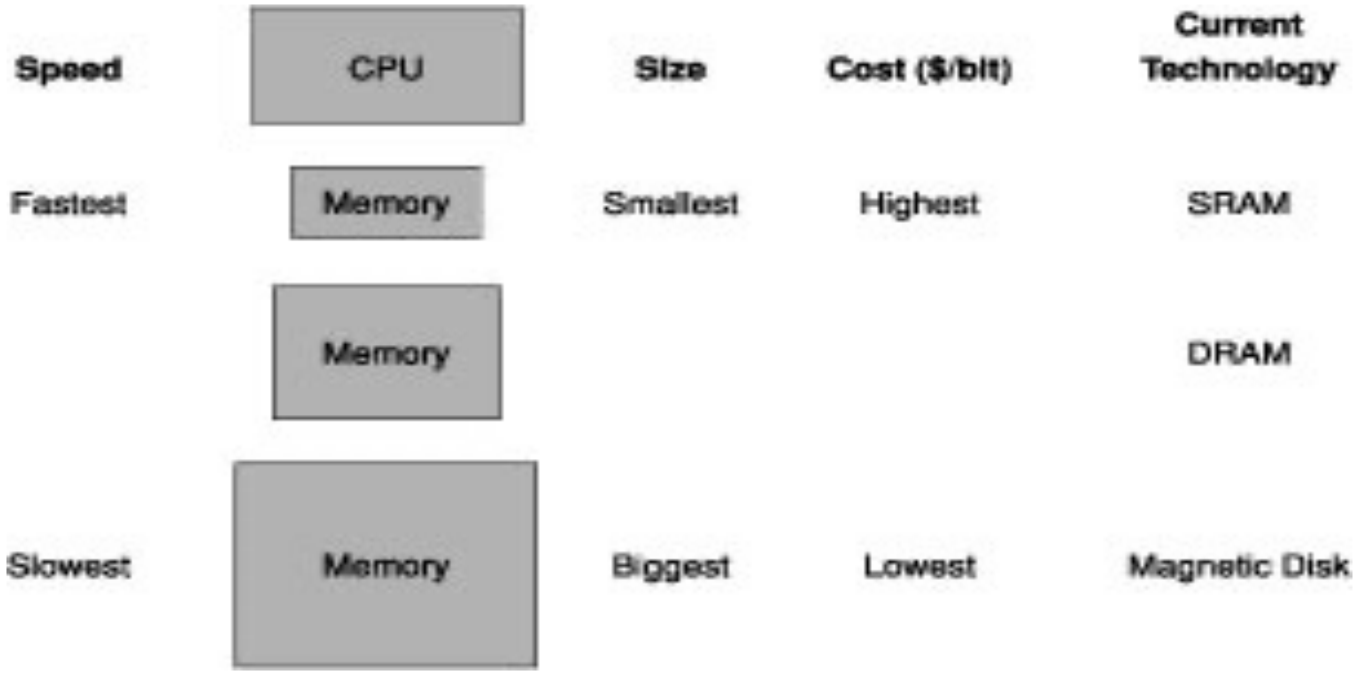
    RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieved from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level +

    Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on Alpha 21264!)



Block

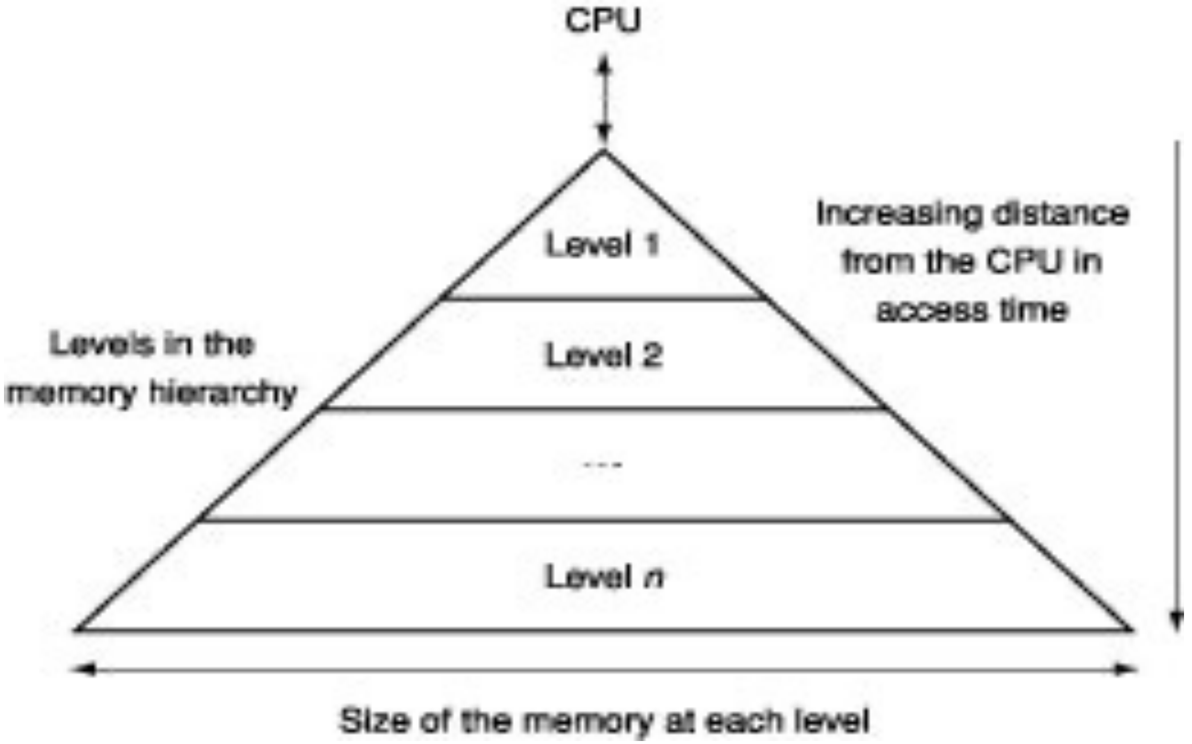
Basic Structure of a Memory Hierarchy



Speed	CPU	Size	Cost ($/bit)	Current Technology
Fastest	Memory	Smallest	Highest	SRAM
	Memory			DRAM
Slowest	Memory	Biggest	Lowest	Magnetic Disk

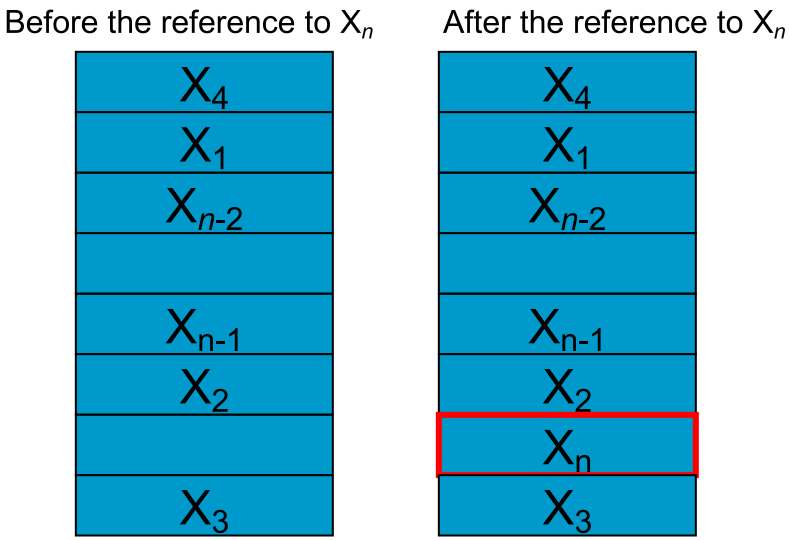
Memory Technology	Typical access time	$ per GB in 2012
SRAM	0.5-5 ns	$500-$1000
DRAM	50 – 70 ns	$10-$20
Flash Memory	5,000 ~ 50,000ns	$0.75 -$1
Magnetic disk	5,000,000-20,000,000 ns	$0.50-$2

Memory Hierarchy


CPU
Level 1
Level 2
...
Level n
Levels in the memory hierarchy
Increasing distance from the CPU in access time
Size of the memory at each level

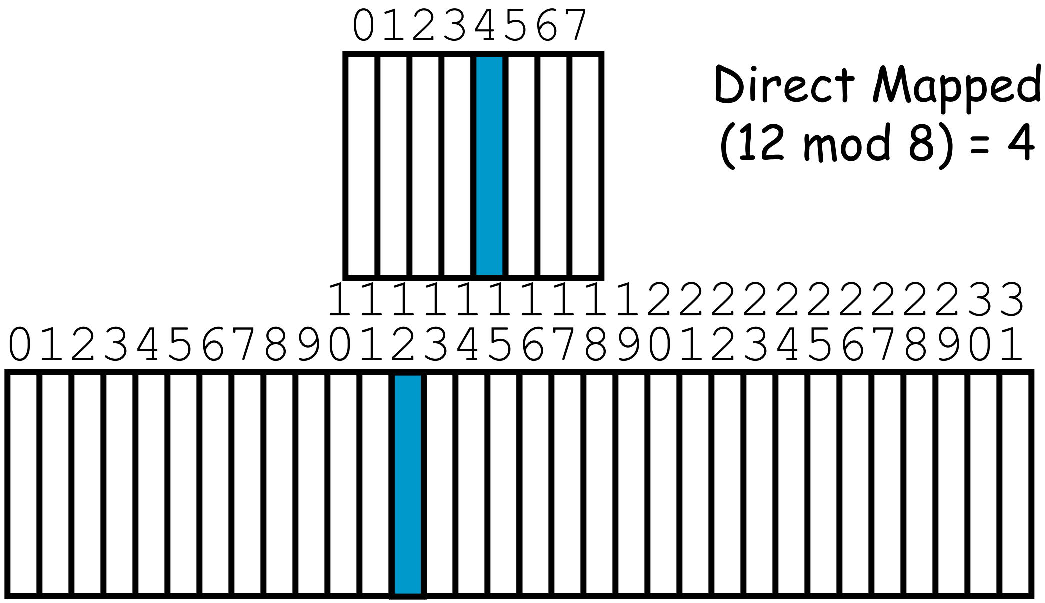
1. Keeping more recently accessed data items closer to the processor -> temporal locality
2. Moving blocks consisting of multiple contiguous words in memory to upper levels of the hierarchy -> spatial locality
3. Data cannot be present in level i unless it is also present in level i+1

Basics of Caches



How do we find X*n* in cache?

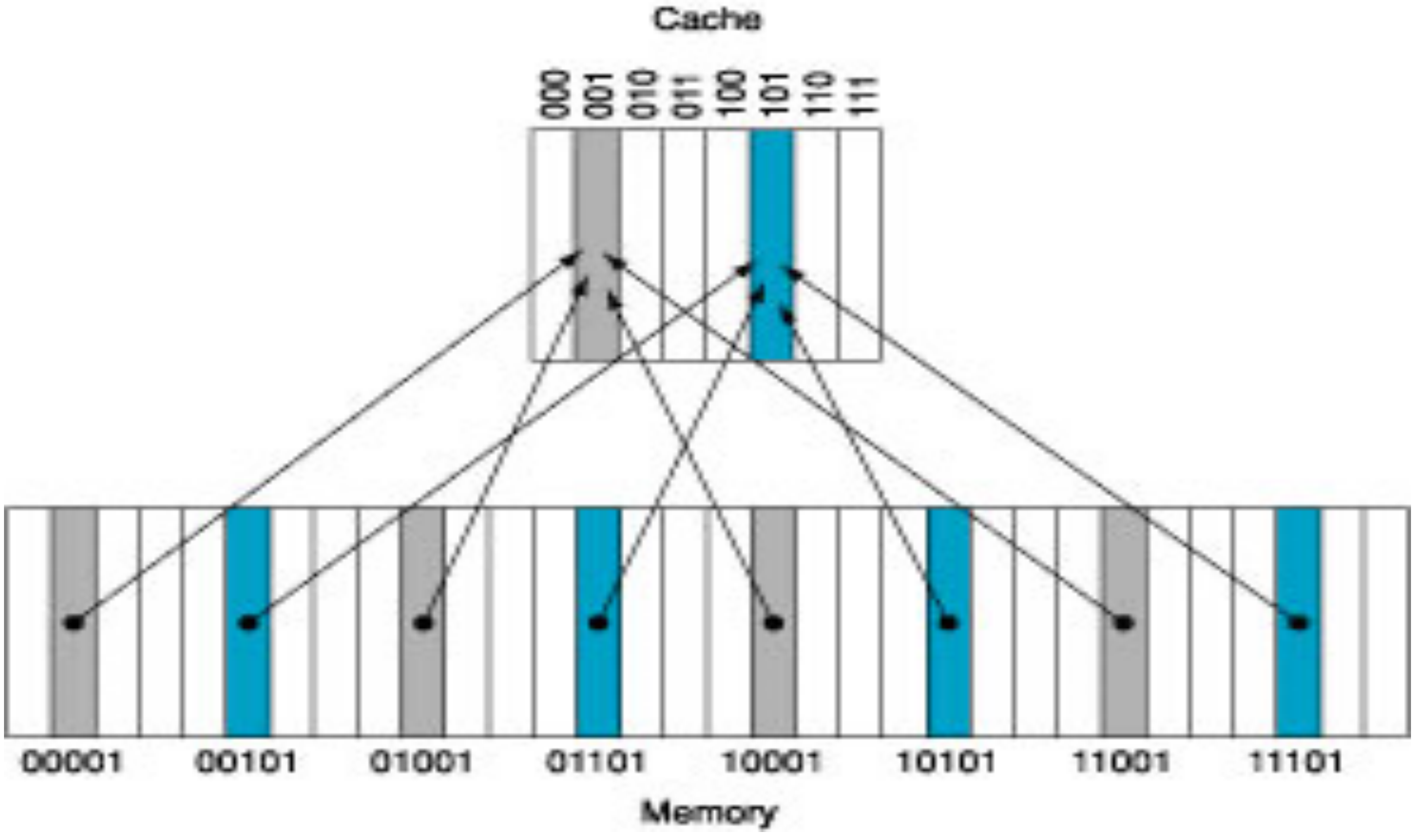
Direct-mapped cache



Direct Mapped
(12 mod 8) = 4

Memory

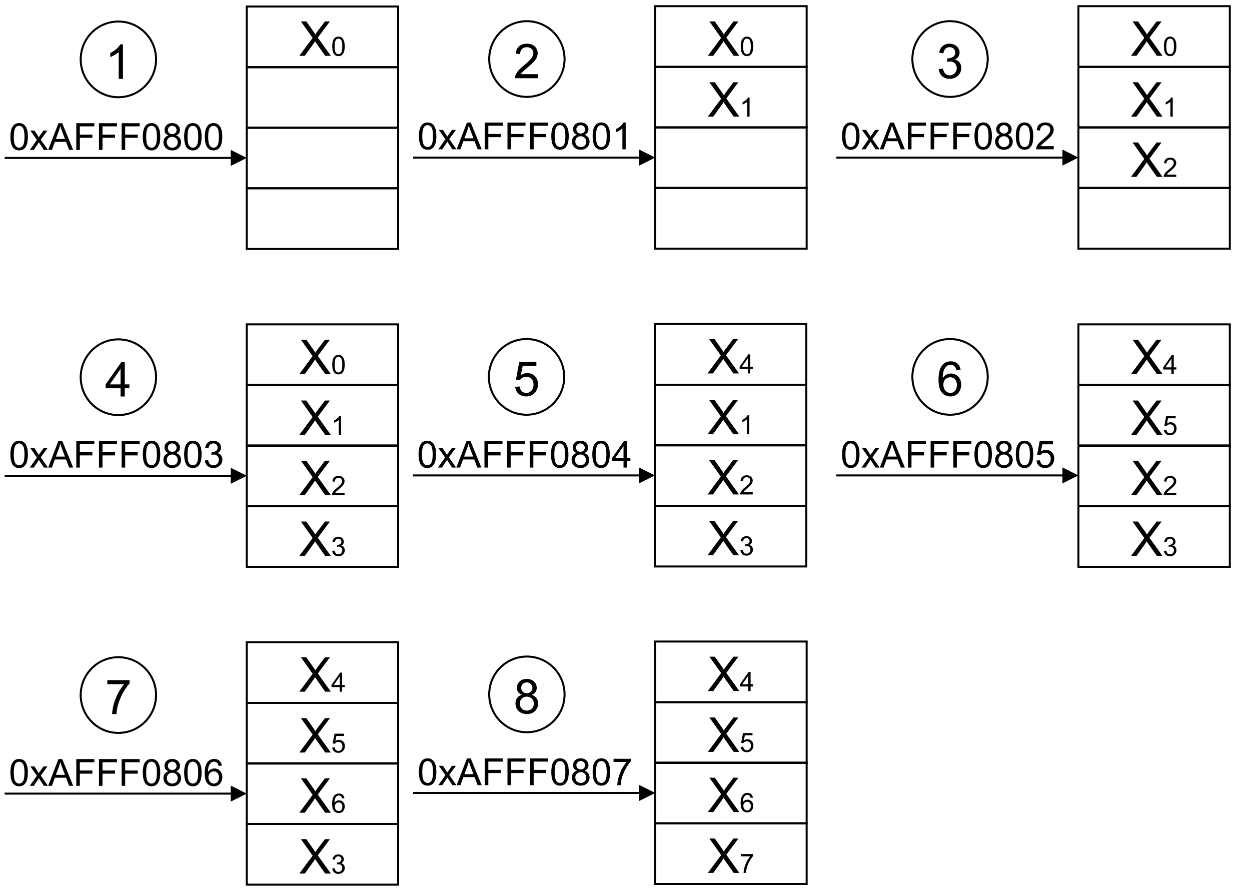
Direct-mapped cache (cont.)



1. Multiple data items map to the same cache location
2. How do we know whether a requested word is in the cache or not?

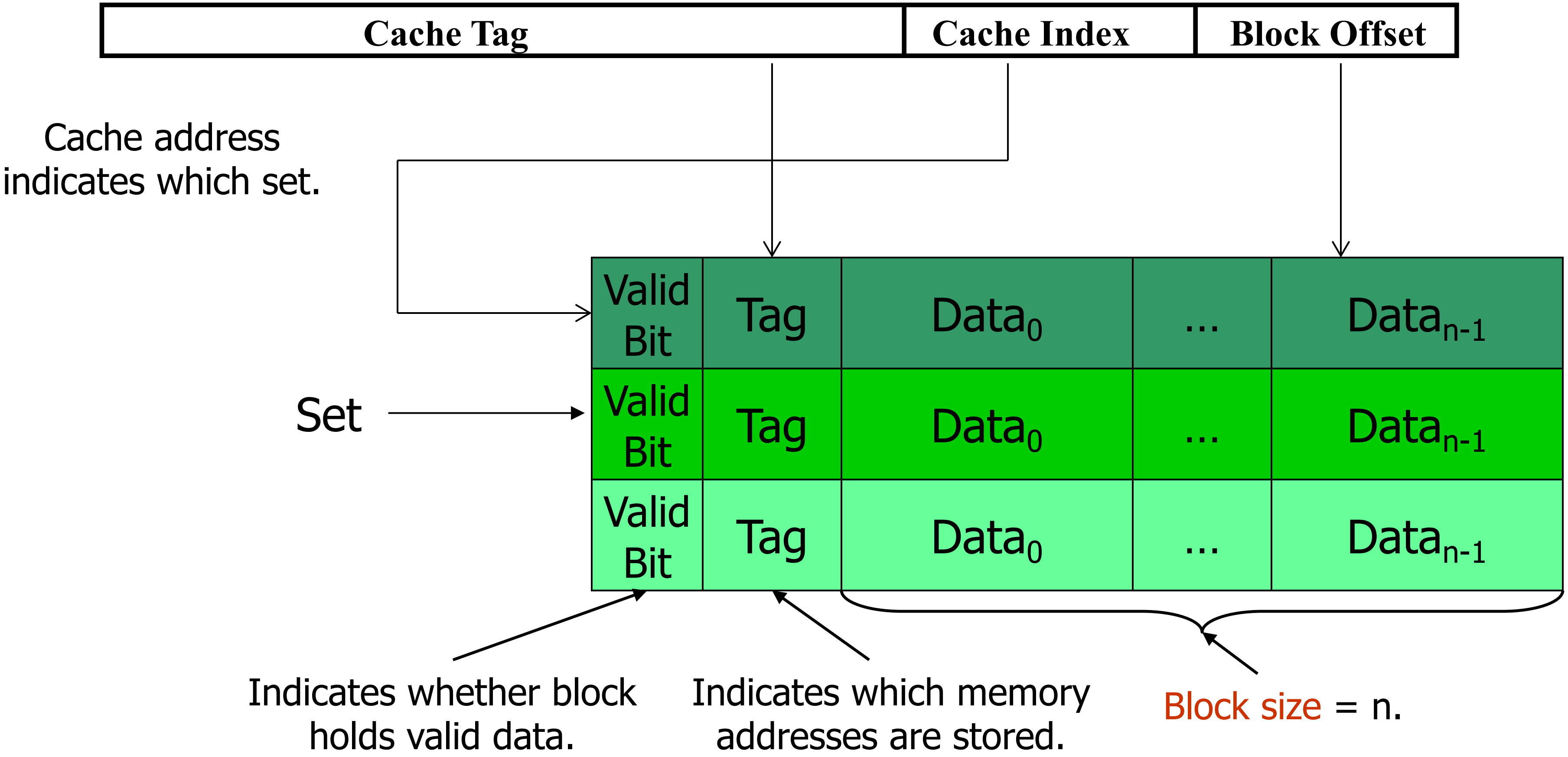
Example

Block Address	Data
0xAFFF0800	X_0
0xAFFF0801	X_1
0xAFFF0802	X_2
0xAFFF0803	X_3
0xAFFF0804	X_4
0xAFFF0805	X_5
0xAFFF0806	X_6
0xAFFF0807	X_7



How is a block found if it is in the upper level?

Memory Address



Cache Tag	Cache Index	Block Offset

Cache address indicates which set.

Set

Valid Bit	Tag	Data$_0$	…	Data$_{n-1}$
Valid Bit	Tag	Data$_0$	…	Data$_{n-1}$
Valid Bit	Tag	Data$_0$	…	Data$_{n-1}$

Indicates whether block holds valid data.

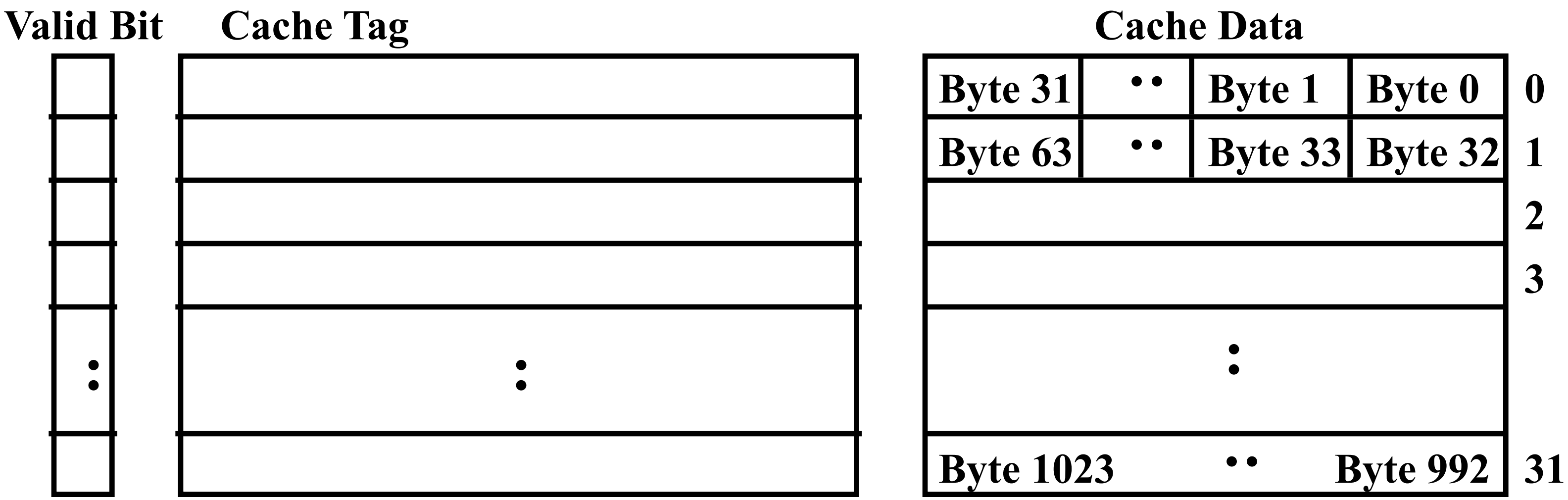
Indicates which memory addresses are stored.

Block size = n.

1 KB Direct Mapped Cache, 32B blocks

- How many sets?

$$\frac{2{\wedge}10}{2{\wedge}5} = 32$$



Valid Bit

Cache Tag

Cache Data

Byte 31 .. Byte 1 Byte 0 0

Byte 63 .. Byte 33 Byte 32 1

2

3

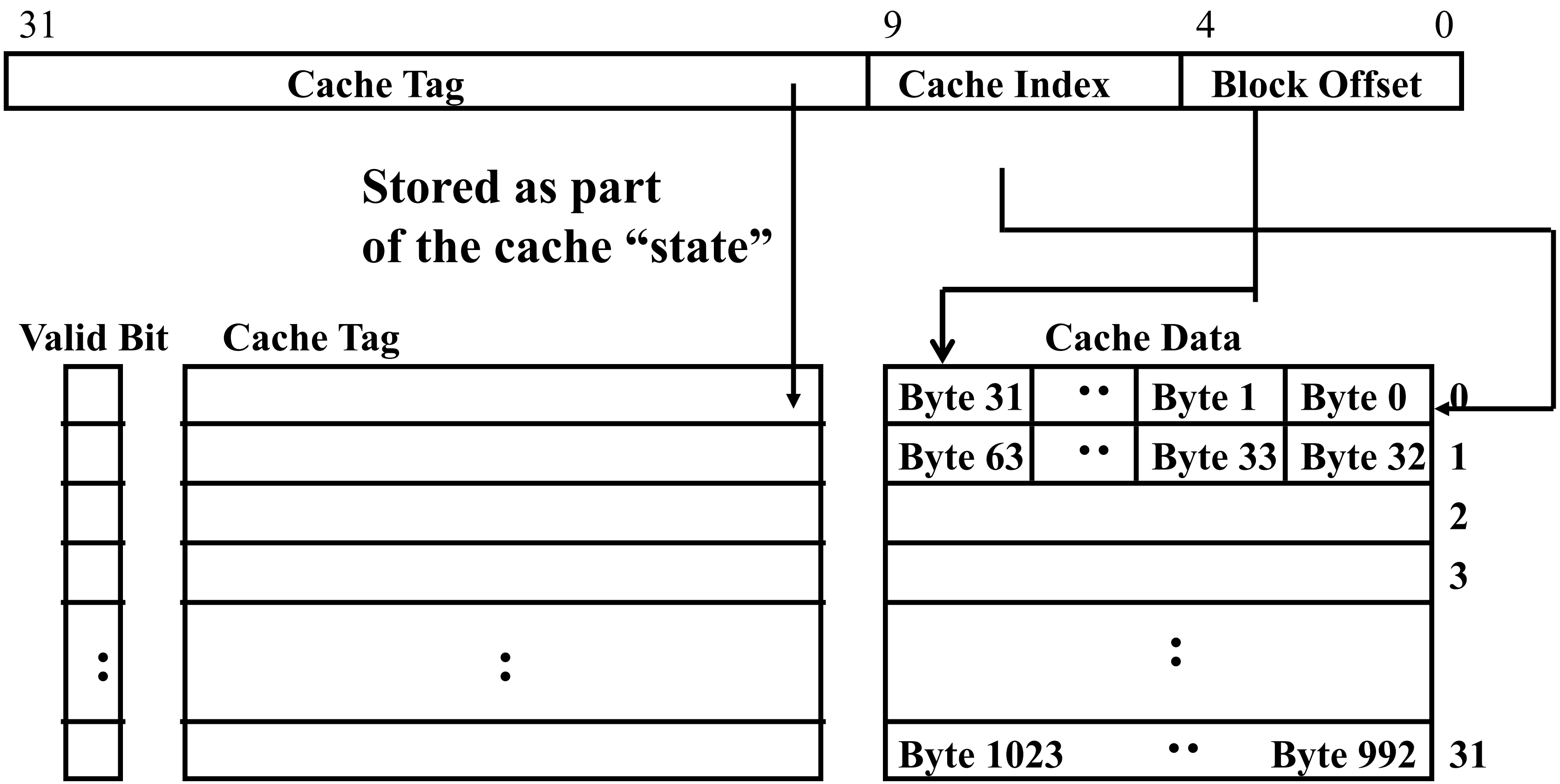
:

Byte 1023 .. Byte 992 31

1 KB Direct Mapped Cache, 32B blocks

- What is the cache address and tag values for 0x00C0A01F?

0000,0000,1100,0000,1010,0000,0001,1111



31 9 4 0

Cache Tag | Cache Index | Block Offset

Stored as part
of the cache "state"

Valid Bit

Cache Tag

Cache Data

Byte 31 •• Byte 1 Byte 0 0

Byte 63 •• Byte 33 Byte 32 1

2

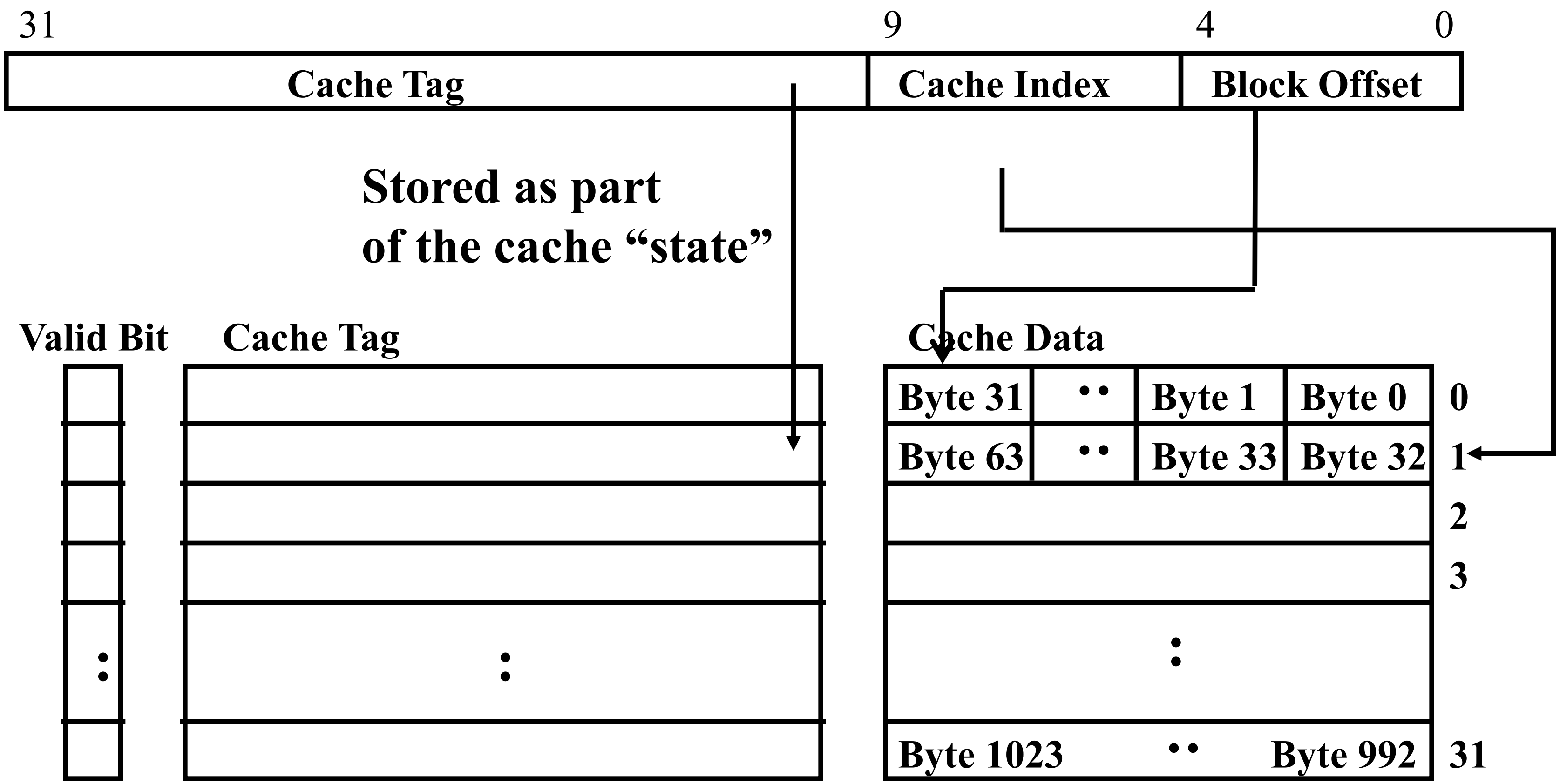
3

:

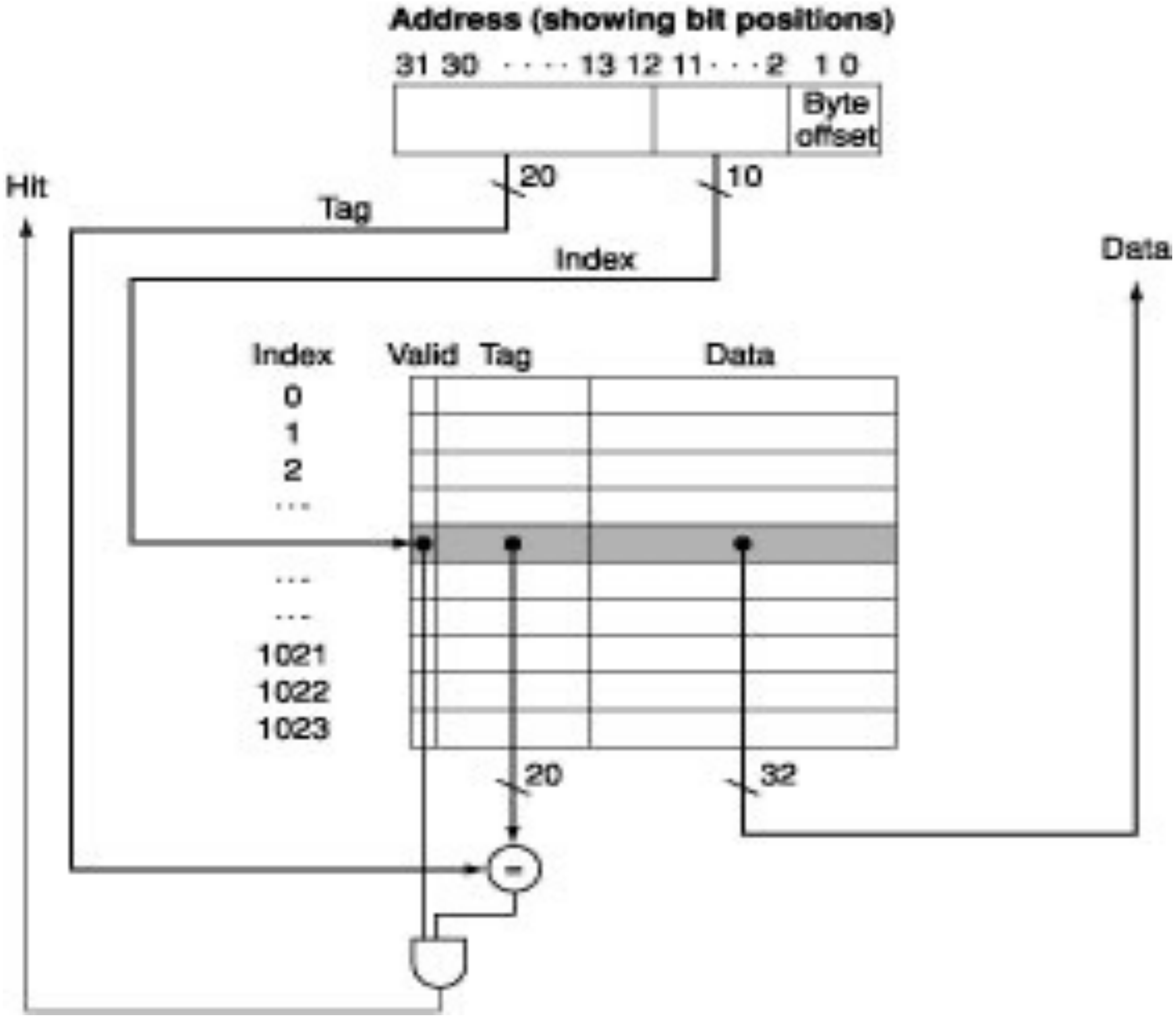
Byte 1023 •• Byte 992 31

1 KB Direct Mapped Cache, 32B blocks

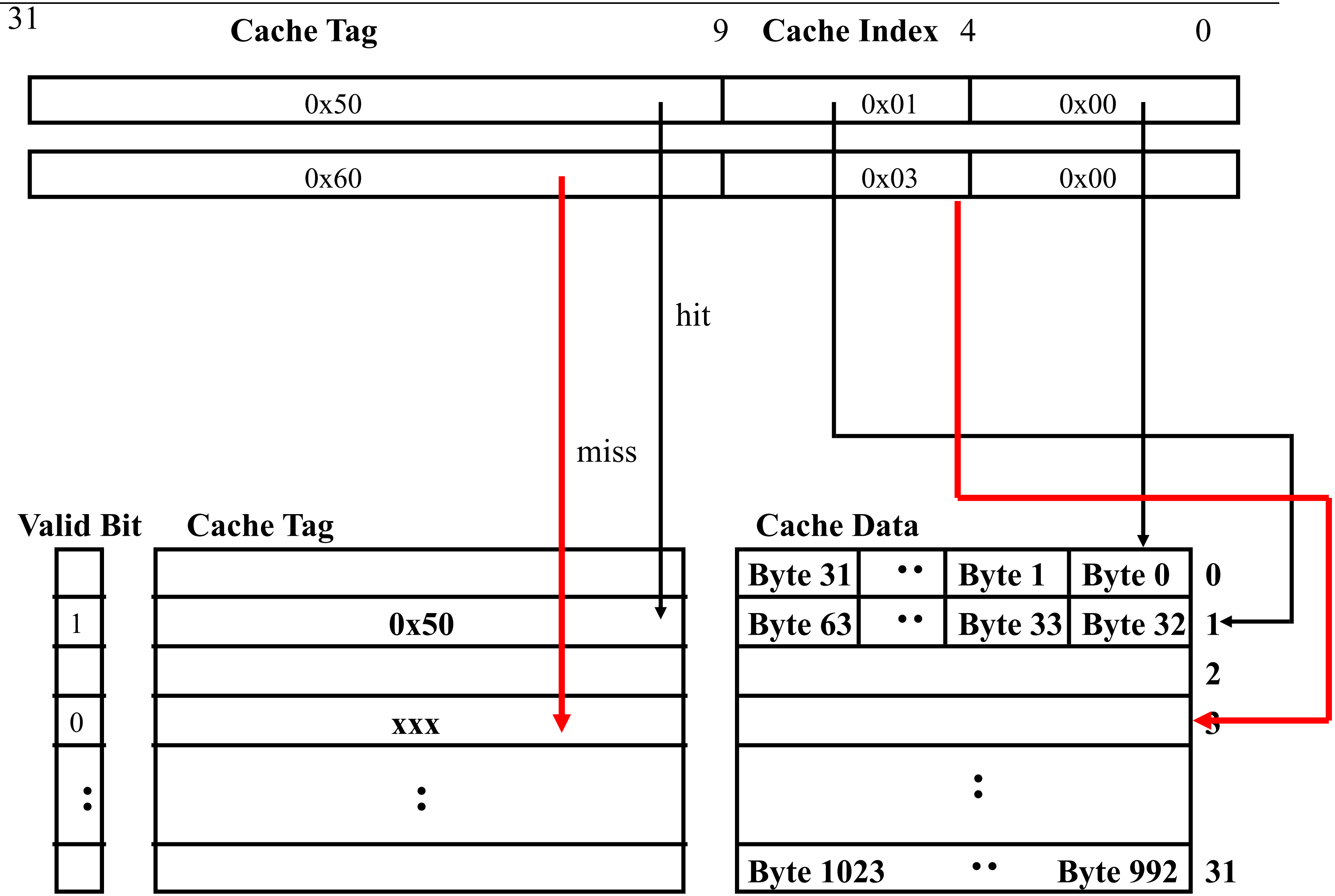
- For a 2 ** N byte direct-mapped cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



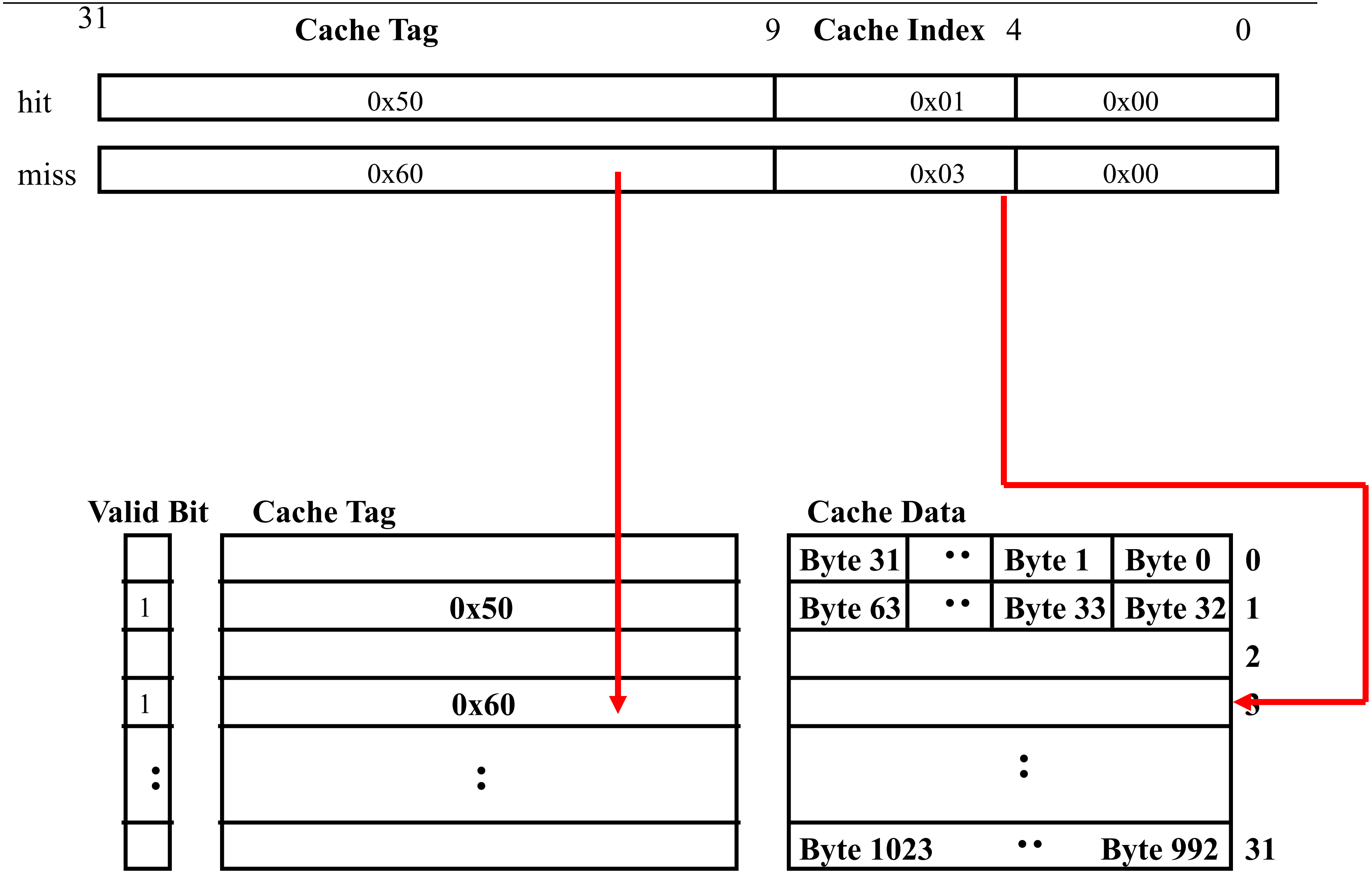
Cache Access



Example: 1 KB Direct Mapped Cache, 32B blocks



Example: 1 KB Direct Mapped Cache, 32B blocks



31 Cache Tag 9 Cache Index 4 0

hit 0x50 0x01 0x00

miss 0x60 0x03 0x00

Valid Bit Cache Tag Cache Data

1 0x50 Byte 63 .. Byte 33 Byte 32 1

Byte 31 .. Byte 1 Byte 0 0

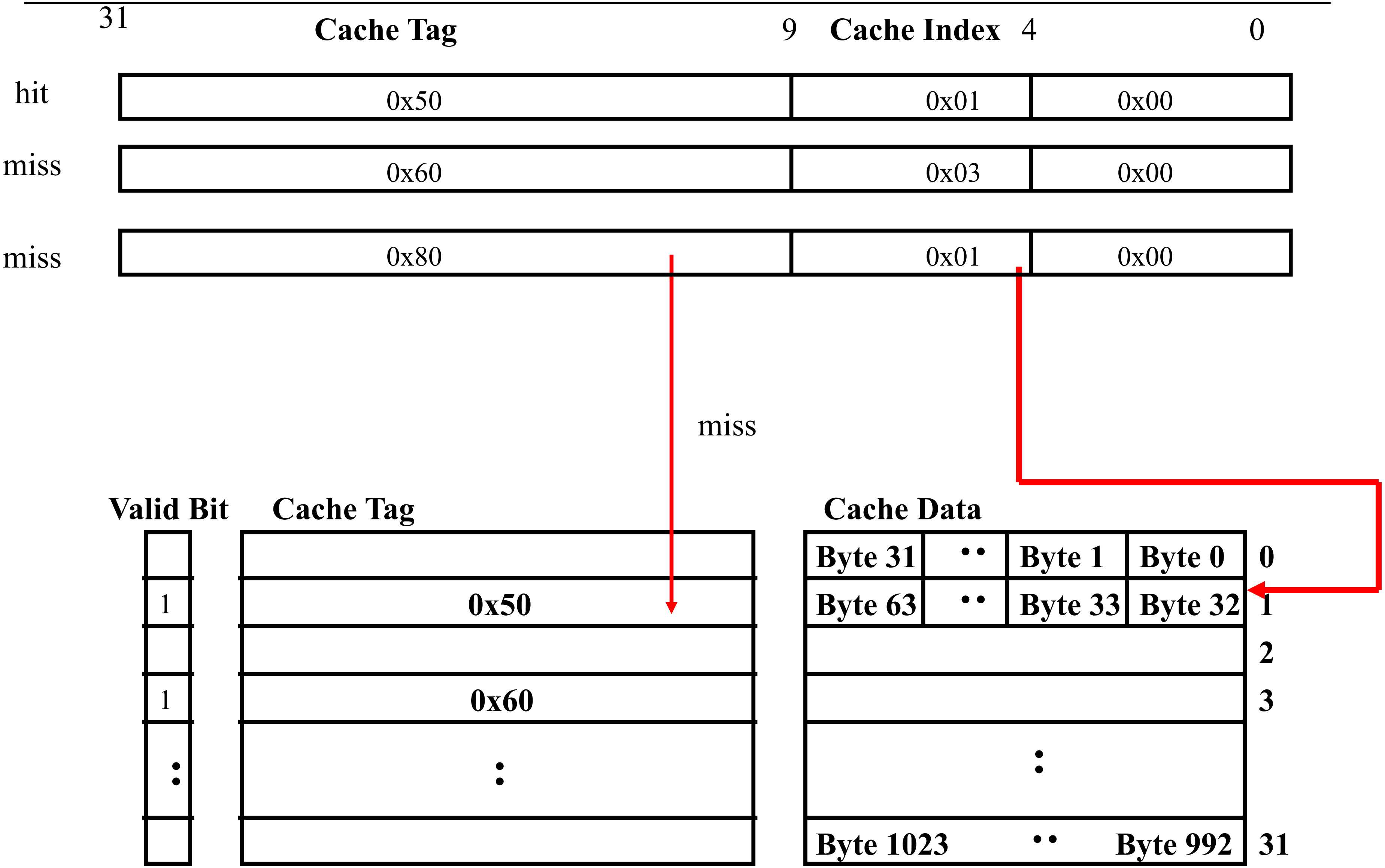
2

1 0x60 3

:

Byte 1023 .. Byte 992 31

Example: 1 KB Direct Mapped Cache, 32B blocks

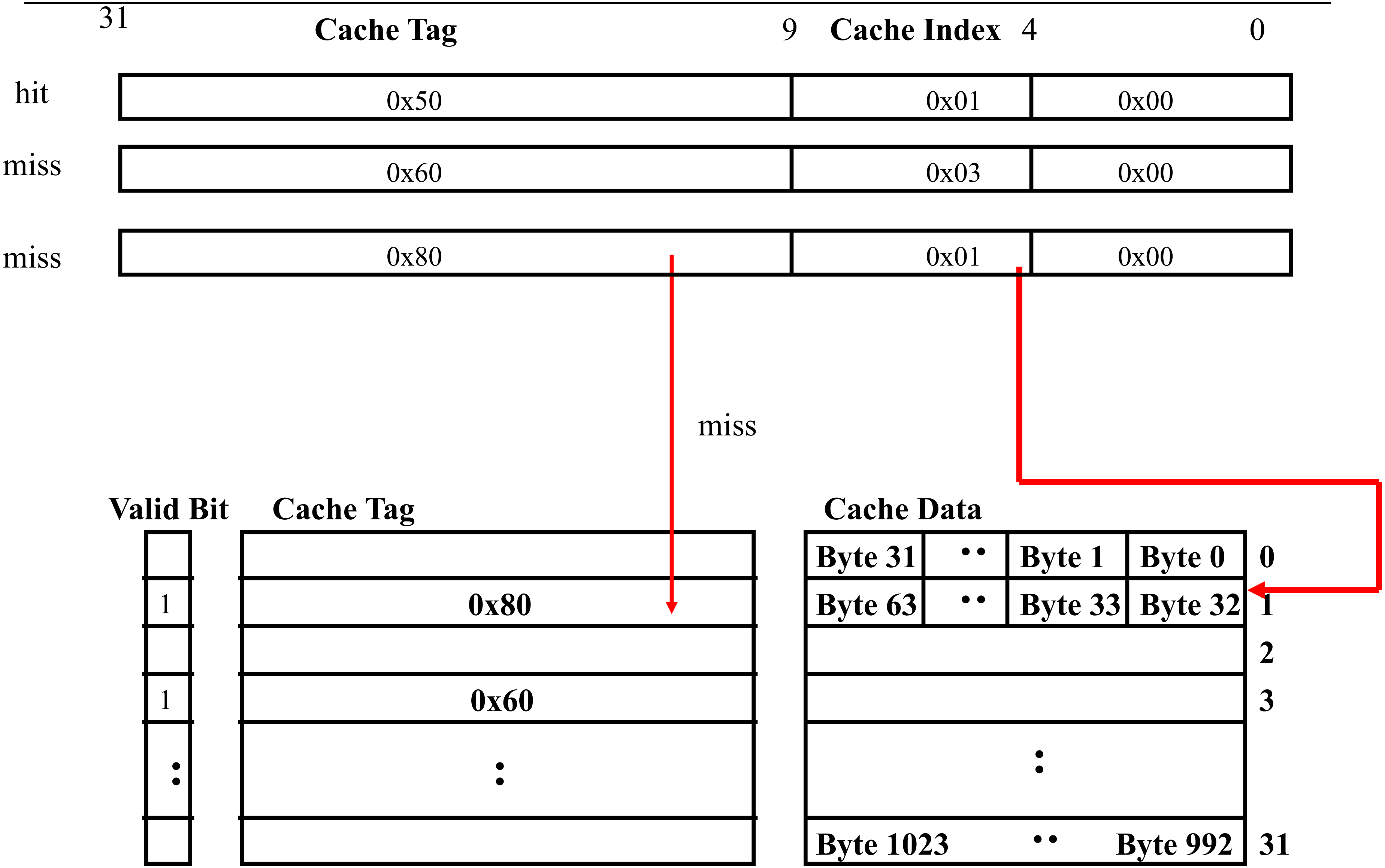


	31 Cache Tag 9	Cache Index 4	0
hit	0x50	0x01	0x00
miss	0x60	0x03	0x00
miss	0x80	0x01	0x00

miss

Valid Bit	Cache Tag	Cache Data				
		Byte 31	••	Byte 1	Byte 0	0
1	0x50	Byte 63	••	Byte 33	Byte 32	1
						2
1	0x60					3
:	:	:				
		Byte 1023	••		Byte 992	31

Example: 1 KB Direct Mapped Cache, 32B blocks



	31 Cache Tag 9	Cache Index 4	0
hit	0x50	0x01	0x00
miss	0x60	0x03	0x00
miss	0x80	0x01	0x00

miss

Valid Bit	Cache Tag	Cache Data				
		Byte 31	••	Byte 1	Byte 0	0
1	0x80	Byte 63	••	Byte 33	Byte 32	1
						2
1	0x60					3
:	:	:				
		Byte 1023	••		Byte 992	31

Exercise

- Show the cache contents of an eight-word direct-mapped caches (1-word block size) after each reference for the following address trace (word addressing) :

10110_{two}, 11010_{two}, 10110_{two}, 110101_{two}, 10000_{two}, 00011_{two}, 10000_{two}, 100010_{two}

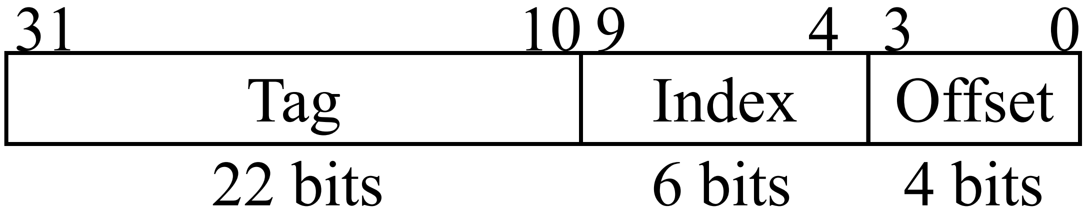
Exercise

- How many total bits are required for a direct-mapped cache with 16 KB of data and 4-word blocks, assuming a 32-bit address?
  - # of sets = ?
  - # of data bits for each set =?
  - # of tag bits for each set = ?
  - Valid bit for each set = 1
  - total cache bits = # of set x (valid bit (1-bit) + tag bits + data bits)

Exercise

- Consider a cache with 64 blocks and a block size of 16 bytes. What block number does byte address 1200 map to?

- Block address = $\lfloor 1200/16 \rfloor$ = 75
- Block number = 75 modulo 64 = 11

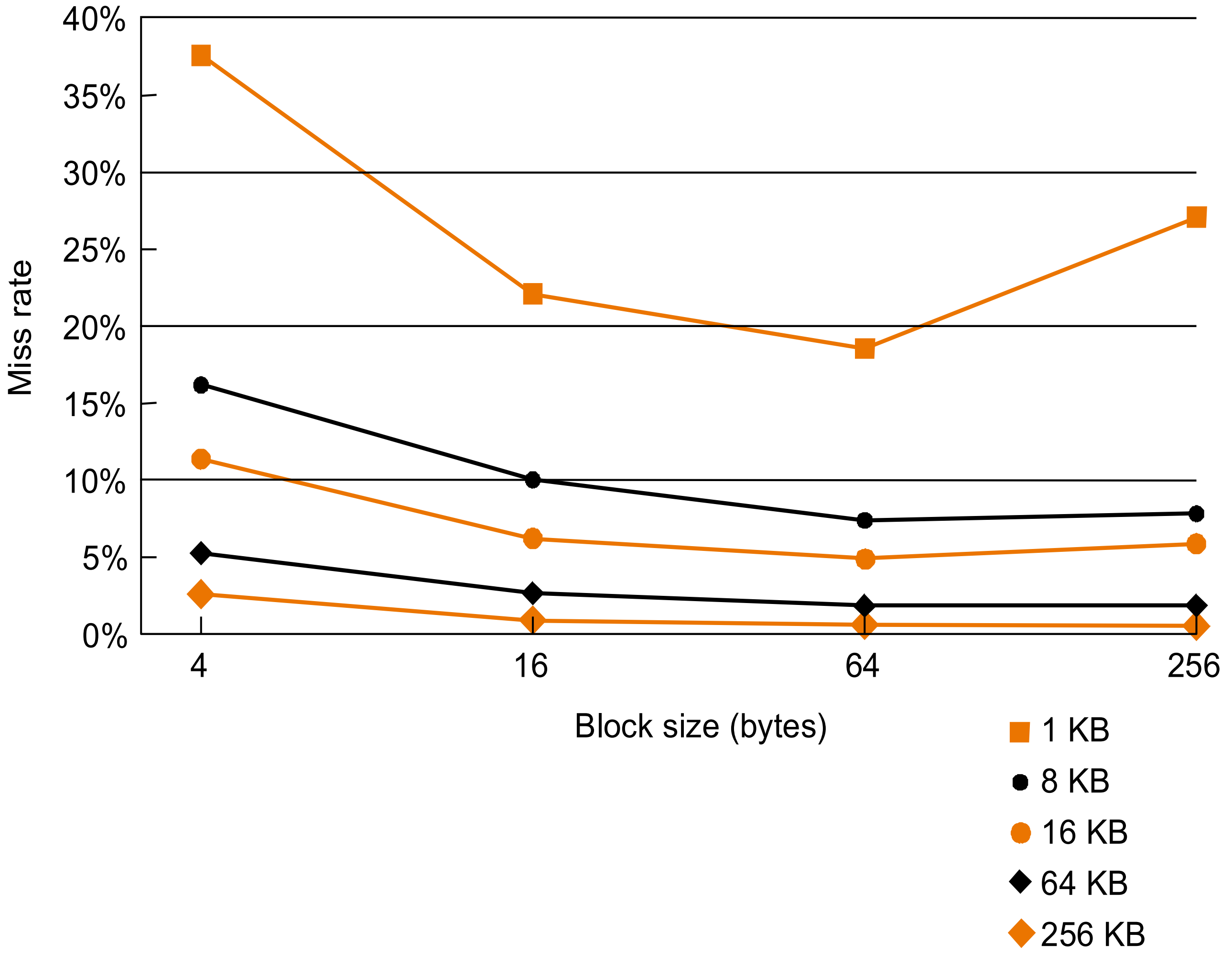


31 10 9 4 3 0

Tag | Index | Offset

22 bits 6 bits 4 bits

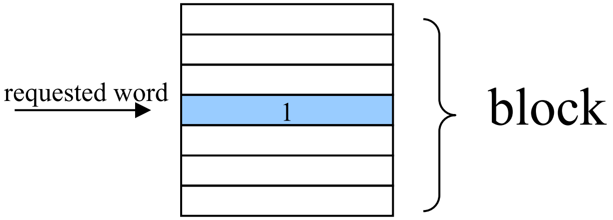
Block Size



40%
35%
30%
25%
20%
15%
10%
5%
0%
Miss rate
4
16
64
256
Block size (bytes)
1 KB
8 KB
16 KB
64 KB
256 KB

Block Size (cont.)

- Advantage of larger block size
  - take advantage of spatial locality
- Disadvantage
  - Too few blocks in cache => high competition
  - Longer cache miss penalty
    - Early restart
      - Resume execution as soon as the requested word of the block is returned
    - Requested word first (critical word first)



Handling Cache Misses

Cache miss =>

Stall the entire pipeline & fetch the requested word

Steps to handle an instruction cache miss:

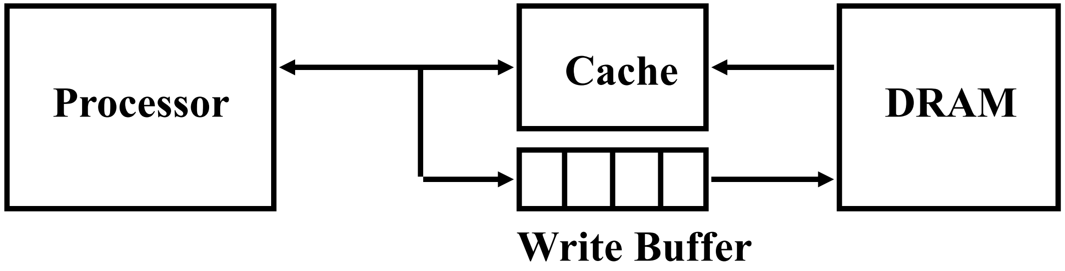
1. Send the instruction address to the memory.
2. Instruct main memory to perform a read and wait for the memory to complete its access.
3. Write the cache entry, putting the data from memory in the data portion of the entry, writing the upper bits of the address into the tag field, and turning the valid bit on.
4. Restart the instruction execution at the first step, which will refresh the instruction, this time finding it in the cache.

Note that the control of the cache on data access is essentially identical as Instruction access shown above.

Handling Writes

- *Write through*—The information is written to both the block in the cache and to the block in the lower-level memory.
- *Write back*—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT:
    - Good: read misses cannot result in writes & data coherency
    - Bad: write stall
  - WB:
    - no repeated writes to same location
    - Write new data to cache & write modified block to the lower level of memory hierarchy

Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system designer's nightmare:
  - Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - Write buffer saturation
- Note: many write-back caches also include write buffers that are used to reduce the miss penalty

Write Miss Policy

- Why there is a “write miss policy”, but no “read miss policy”?

Write Miss Policy

- Write allocate (fetch on write)
  - The block is loaded on a write miss
- No-write allocate (write-around)
  - The block is modified in the lower level and not loaded into the cache

	Write through	Write back
Write allocate	hit: write to cache/memory miss: load block into cache; write to cache/memory	hit: write to cache, set dirty bit. miss: load block into cache; write to cache;set dirty bit
Write around	hit: write to cache/memory miss: write to memory	hit: write to cache, set dirty bit. miss: write to memory

Example: Intrinsity FastMath Processor

- Intrinsity FastMATH
  - embedded microprocessor using the MIPS architecture
  - 12-stage pipeline
  - Separate instruction/data caches (split cache), 16 KB, 16-word blocks
  - Offer both write-through and write-back
  - One-entry write buffer.
- Miss rate of Intrinsity FastMATH for SPEC2000:
  - Instruction miss rate : 0.4%
  - Data miss rate: 11.4%
  - Effective combined miss rate : 3.2%
- Q1: Why is data miss rate higher than instruction miss rate?
- Q 2: Why is combined miss rate is lower than data miss rate?

Split cache vs. Combined cache

- Combined cache – higher cache hit rate & lower cache bandwidth
- Split cache – lower cache hit rate & higher cache bandwidth

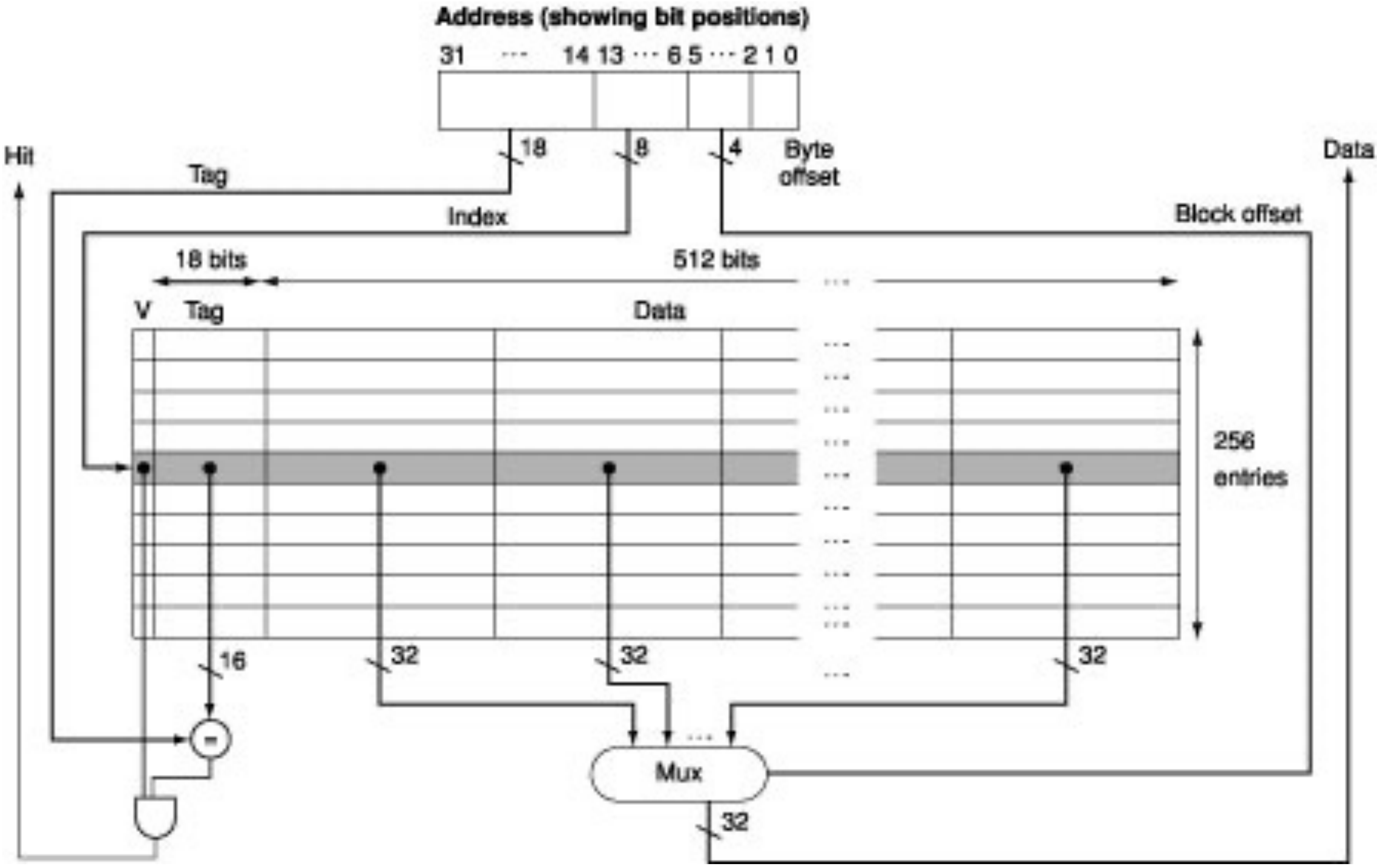
I

D

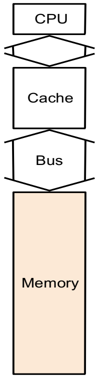
I+D

Example: Intrinsity FastMath Processor (cont.)

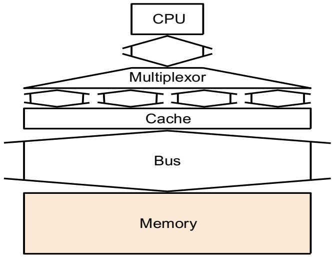
- The 16 KB caches in the Intrinsity FastMATH each contain 256 blocks with 16 words per block.



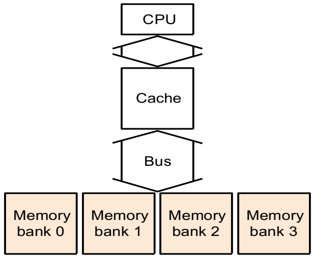
Memory Design to Support Cache



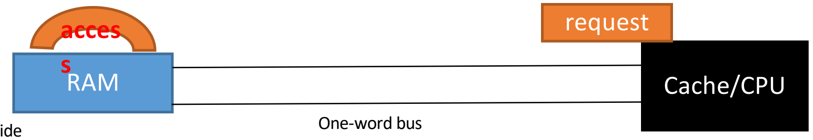
a. One-word-wide memory organization



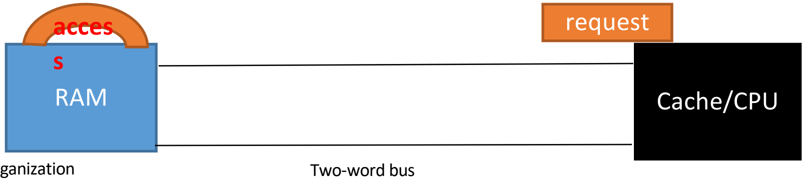
b. Wide memory organization



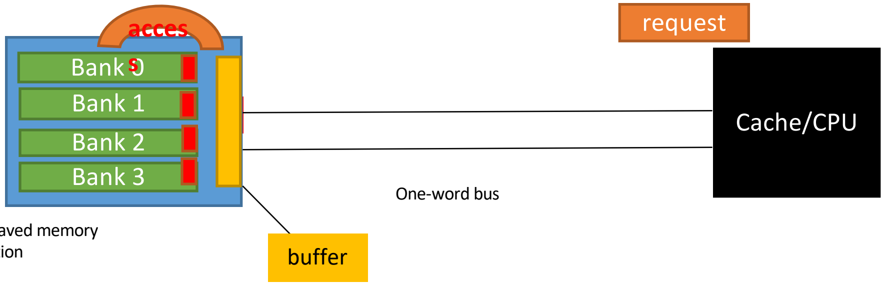
c. Interleaved memory organization



a. One-word-wide memory organization



b. Wide memory organization

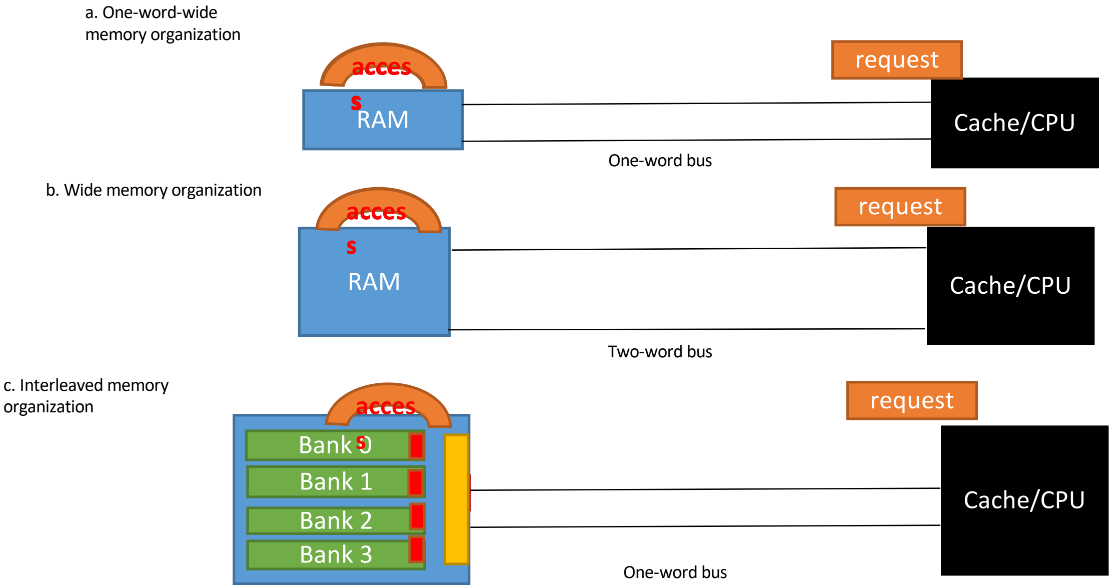


c. Interleaved memory organization

Assume

1 memory bus cycle to send the address
15 memory bus cycles for each DRAM access
1 memory bus cycle to send a word of data
4-word block & on-word-wide memory bank

What is the cache miss penalty?


a. One-word-wide memory organization

b. Wide memory organization

c. Interleaved memory organization

$$\overbrace{1}^{\text{Send request}} + \overbrace{4\text{x}15}^{\text{Number of access DRAM}} + \overbrace{4\text{x}1}^{\text{Number of send data back}}$$

1 + 2x15 + 2x1

1 + 1x15 + 4x1

Cache Performance

- CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time
- Memory stall clock cycles = Read-stall cycles + write-stall cycles
- Read-stall cycles = # of Read x Read miss rate X Read miss penalty
- Write-stall cycles = (# of Writes X Write miss rate X Write miss penalty) +
  Write buffer stalls.
- Memory-stall clock cycles = # Memory accesses X Miss rate X Miss penalty

$$= \frac{\text{Instructions}}{\text{program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

- Average memory access time = Hit time + Miss rate x Miss penalty

Example

- I-Cache miss rate = 2% & D-Cache miss rate = 4%
- Base CPI 2.0
- Miss penalty = 100 cycles
- Frequency of loads and stores is 36%.
- Compare the performance with a perfect cache

Assume total instruction count = I

- I-cache stall cycles =
- D-cache stall cycles =
- CPU time with stalls =

- CPU time with perfect cache =

CPU time = cycles x clock-cycle
= (CPU cycles + memory stall cycles) x clock-cycle

Example

- I-Cache miss rate = 2% & D-Cache miss rate = 4%
- Base CPI 2.0
- Miss penalty = 100 cycles
- Frequency of loads and stores is 36%.
- Compare the performance with a perfect cache

- I-cache stall cycles = I x 2% x 100 = 2.00 X I
- D-cache stall cycles =
- CPU time with stalls =

  =
- CPU time with perfect caches

  =

  =

CPU time = cycles x clock-cycle

= (CPU cycles + memory stall cycles) x clock-cycle

Cache Performance with Increased Clock rate

- How much faster will the computer be with 2x clock rate, assuming the same miss rate as the previous example.

2x clock rate → Miss penalty = 200 cycles

- I-cache stall cycles =
- D-cache stall cycles =
- CPU time with stalls =

2x clock rate → clock cycle = clock cycle / 2

$$\frac{Performance\ with\ fast\ clock}{Performance\ with\ slow\ clock} = \frac{Execution\ time\ with\ slow\ clock}{Execution\ time\ with\ fast\ clock}$$

$$=$$

$$=$$

Cache Performance with Increased Clock rate

- How much faster will the computer be with 2x clock rate, assuming the same miss rate as the previous example.

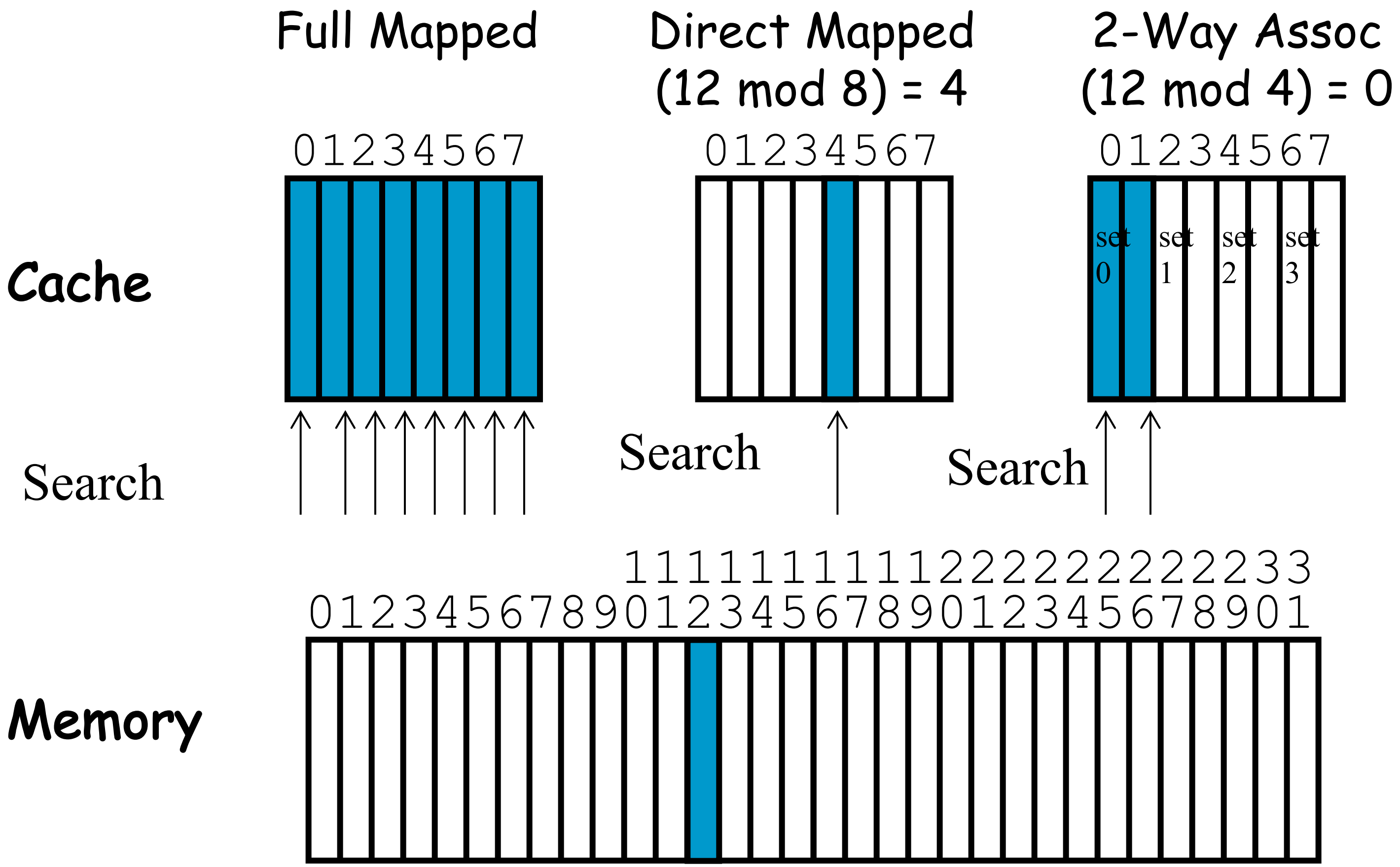
- I-cache stall cycles = I x 2% x 200 = 4.00 X I
- D-cache stall cycles =
- CPU time with stalls =

$$\frac{\textit{Performance with fast clock}}{\textit{Performance with slow clock}} = \frac{\textit{Execution time with slow clock}}{\textit{Execution time with fast clock}}$$

$$=$$

$$=$$

Reducing Cache Misses by More Flexible Placement of Blocks



Possible Associativity Structures

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

An 8-block cache

Address Conflict

```
float dot_prod(float x[SIZE],
               float y[SIZE])
{
  float sum = 0.0;
  int   i;

  for (i = 0; i < SIZE; i++)
    sum += x[i]*y[i];

  return sum;
}
```

Assume:

Direct-mapped cache.

`x[i]` and **`y[i]`** map to same blocks.

? What is the hit rate? ?

Under these assumptions, every access is a cache miss.
Hit rate = 0%.

What can we do? Increasing the associativity

Exercise

- Three small caches, each consisting of four one-word blocks
  - Direct mapped cache
  - Two-way set associative cache
  - Fully associative cache
- Find the number of misses for each cache for the following sequence
  - 0,8,0,6,8

Exercise (cont.)

- The direct mapped cache

Block Address	Cache Set
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		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]	

Exercise (cont.)

- The two-way set associative cache

Block Address	Cache Set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

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]		

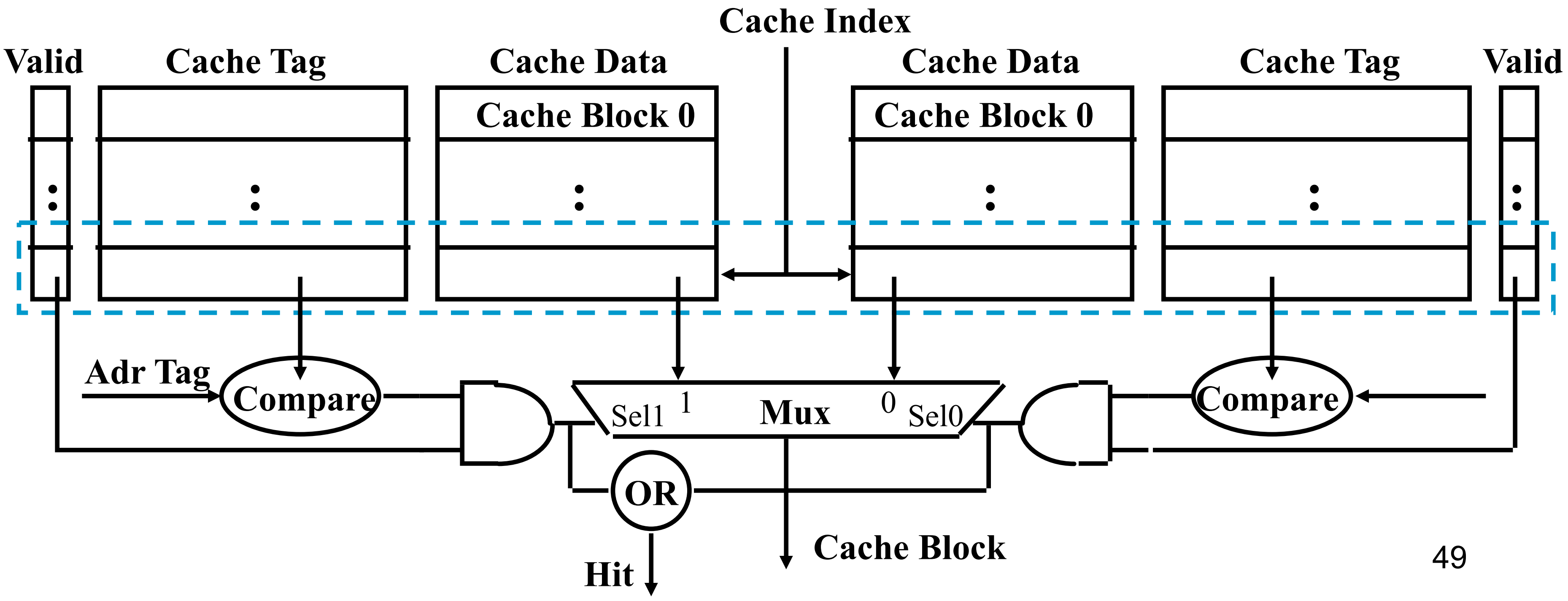
Exercise (cont.)

- The fully associative cache

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		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]	

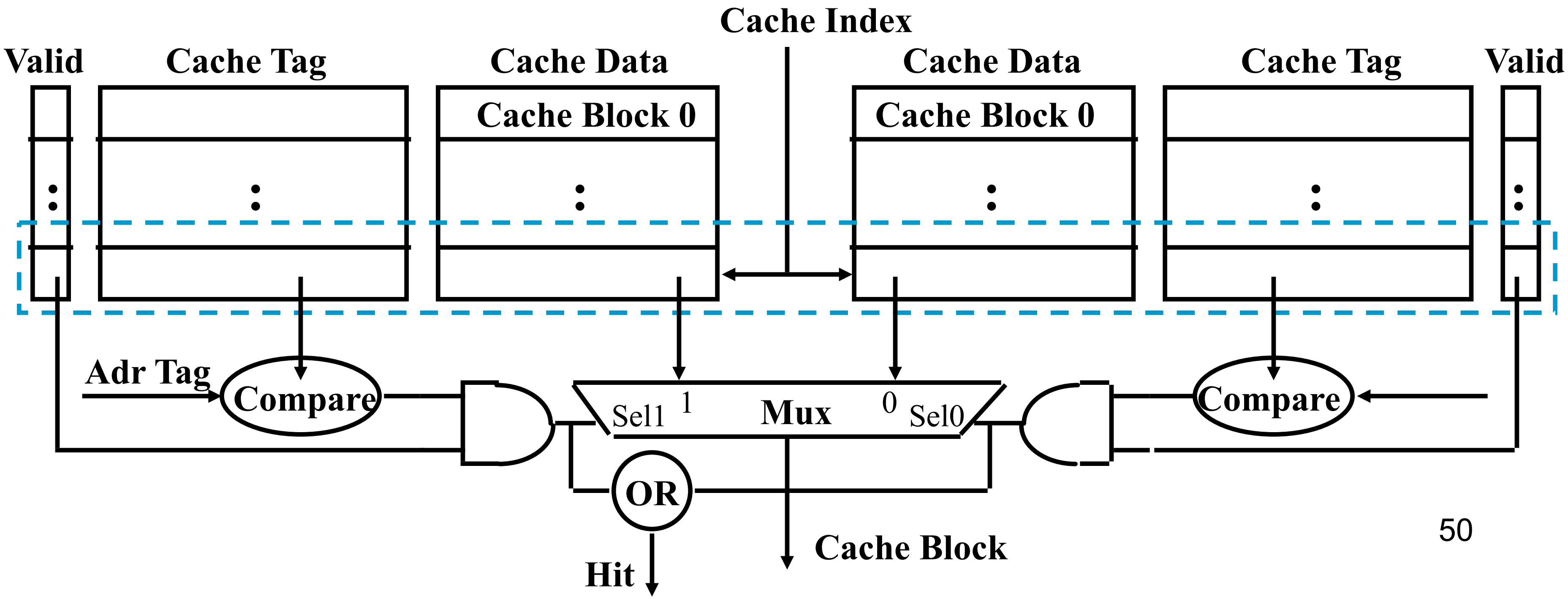
2-way Set Associative Cache

- 1K, 2-way,32B block
  - 16 sets, each set has 2 blocks
  - Index = 4 bits, tag = 23 bits
- Increasing associativity shrinks index, expands tag
- How to find a data in 2-way cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

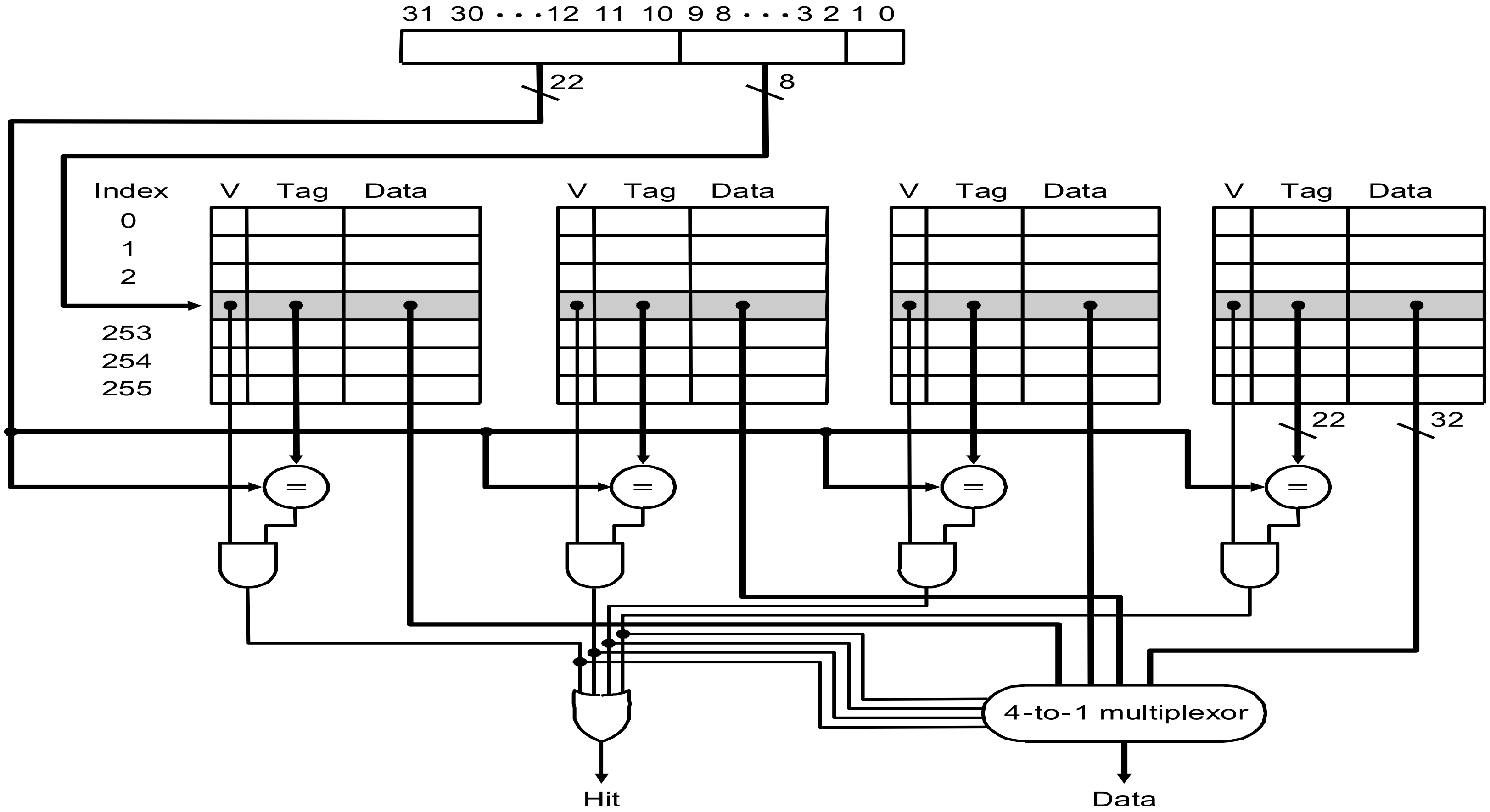


Disadvantage of Set Associative Cache

- N-way Set Associative Cache vs. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, cache block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



A 4-Way Set-Associative Cache



Replacement Policy: Choosing Which Block to Replace

- Easy for direct mapped
- Set associative or fully associative:
  - Random
  - FIFO
  - LRU (Least Recently Used):
    - Hardware keeps track of the access history and replace the block that has not been used for the longest time

Effects of Associativity

Associativity	Data miss rate
1	10.3%
2	8.6%
4	8.3%
8	8.1%

Data Source: Spec2000 benchmarks

Tag Size vs. Associtivity

- With the same cache capacity, increasing associtivity increases or decreasing tag bits?

(direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

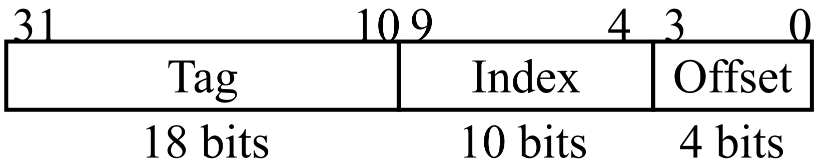
Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

Exercise: Size of Tags vs. Set Associativity

- Find # of sets, total # of tag bits for direct-mapped, two-way and four way. (Capacity- 4K blocks, four-word block, 32 bit address)
  - tag+index = 32- 4 = 28
  - The direct mapped cache
    - 4K sets
    - $\log_2(4K)$ = 12 bits of index
    - Total (28-12) × 4K = 16 × 4K = 64 Kbits of tag
  - The two-way set-associative cache
    - 2K sets
    - Total (28-11) × 2 × 2K = 34 × 2K = 68 Kbits of tag
  - The four-way set-associative cache
    - 1K sets
    - Total (28-10) × 4 × 1K = 72 × 1K = 72 Kbits of tag
  - The fully associative cache
    - One set with 4K blocks
    - Tag is 28 bits
    - Total 28 × 4K × 1 = 112K tag bits

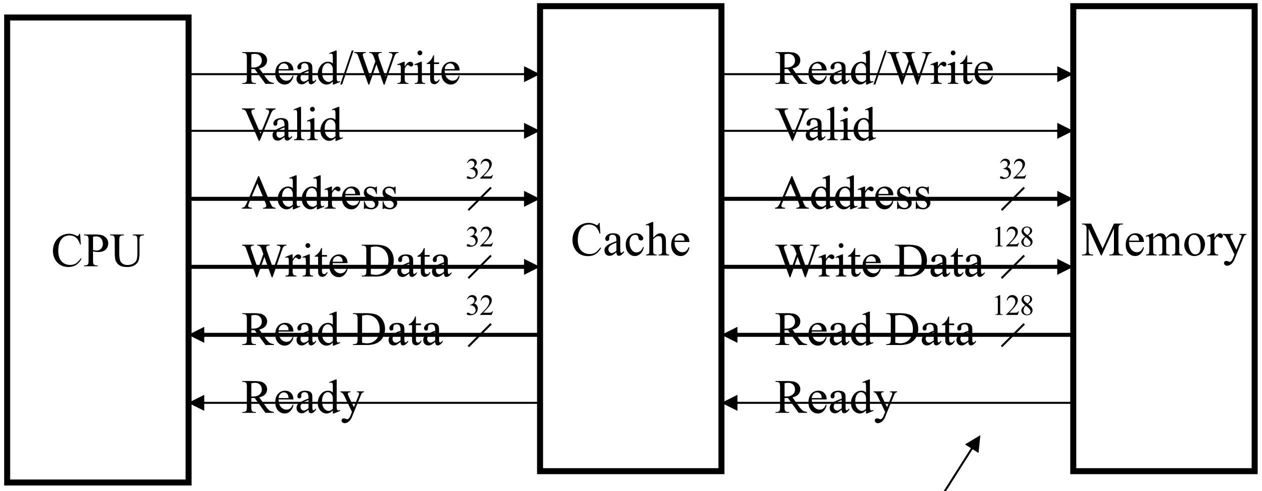
Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete



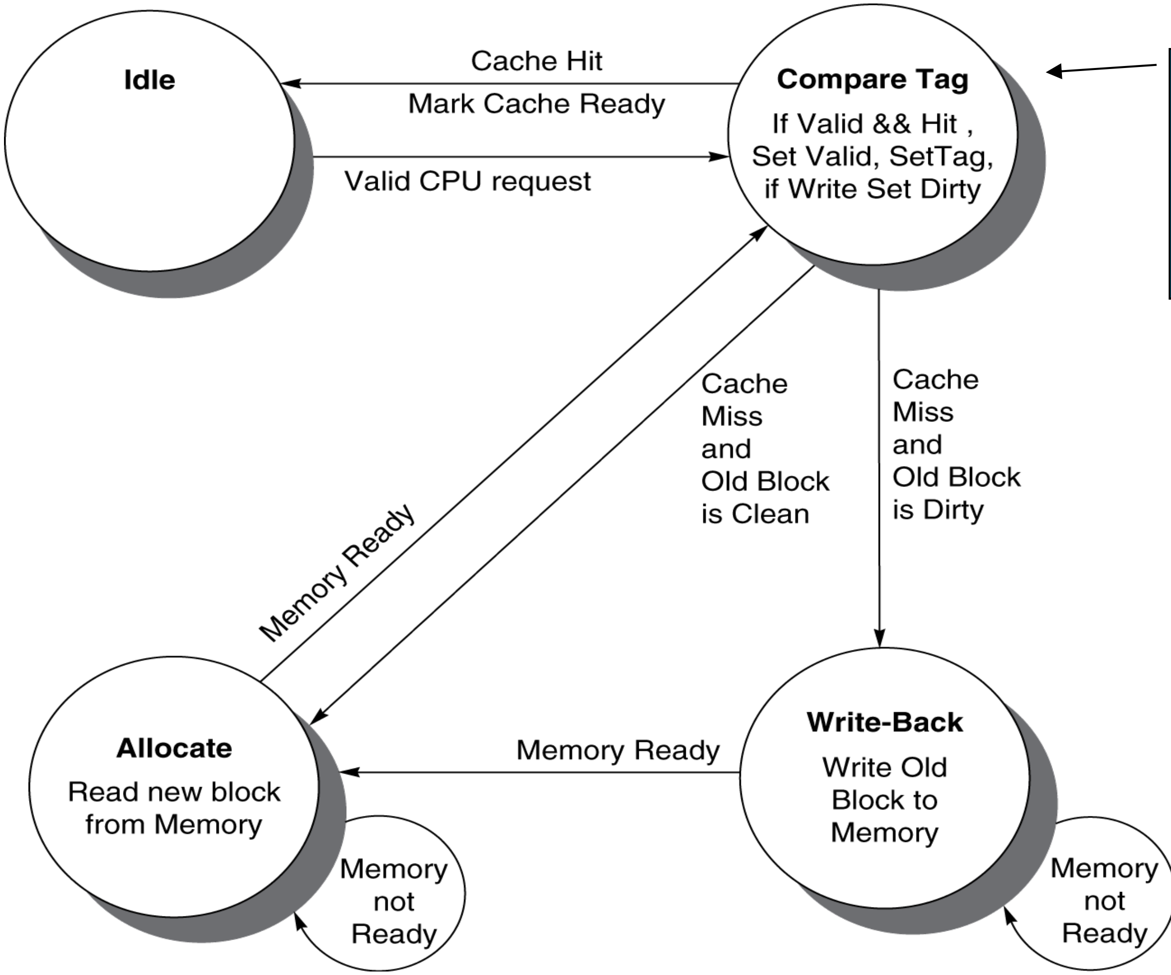
31 … 10	9 … 4	3 … 0
Tag	Index	Offset
18 bits	10 bits	4 bits

Interface Signals



Multiple cycles per access

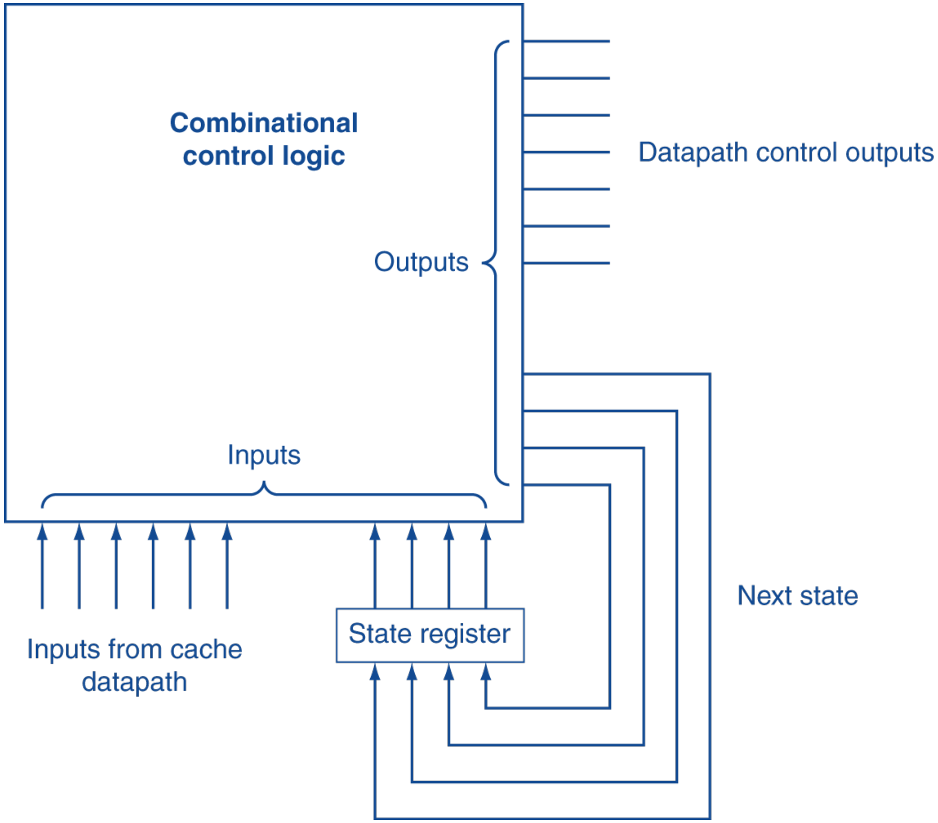
Cache Controller FSM



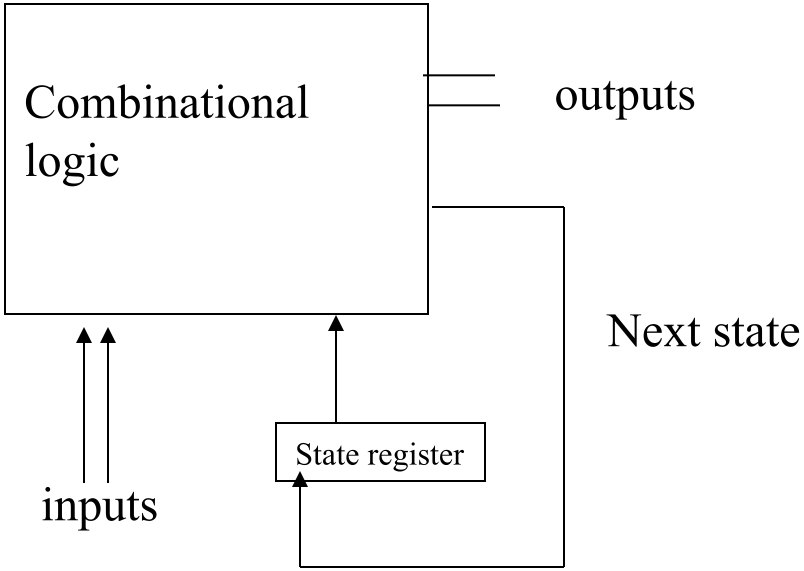
Could partition into separate states to reduce clock cycle time

Finite State Machines

- Use a FSM to sequence control steps
- Set of states, transition on each clock edge
  - State values are binary encoded
  - Current state stored in a register
  - Next state
    = f_n (current state,
    current inputs)
- Control output signals
  = f_o (current state)



Finite State Implementation



How to implement the combinational logic?

- PLA (Programmable Logic Array)
- ROMs

Miss Penalty Reduction: Multi- Level Cache

- Larger cache vs. CPU time
  - Add another level of cache
  - The L2 cache is much larger than L1

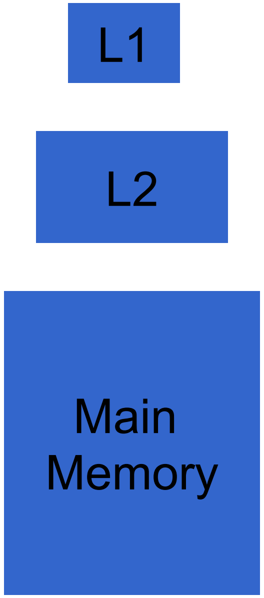
- L2 Equations

  AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x Miss Penalty$_{L1}$

  Miss Penalty$_{L1}$ = Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$

  AMAT = Hit Time$_{L1}$ + Miss Rate$_{L1}$ x (Hit Time$_{L2}$ + Miss Rate$_{L2}$ x Miss Penalty$_{L2}$)

- Definitions:
  - *Local miss rate*— misses in this cache divided by the total number of memory accesses *to this cache* (Miss rate$_{L2}$)
  - *Global miss rate*—misses in this cache divided by the total number of memory accesses *generated by the CPU* (Miss Rate$_{L1}$ x Miss Rate$_{L2}$)



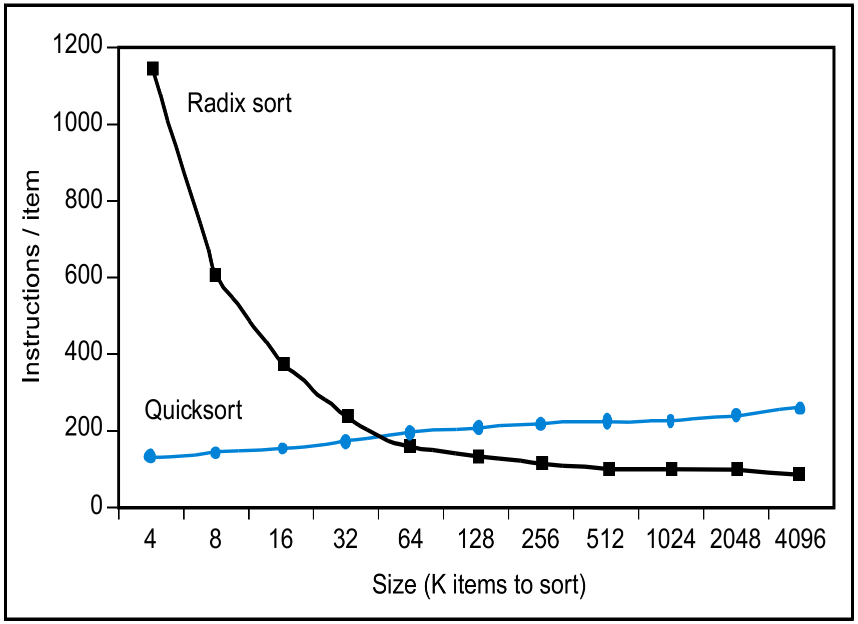
Performance of Multilevel Caches

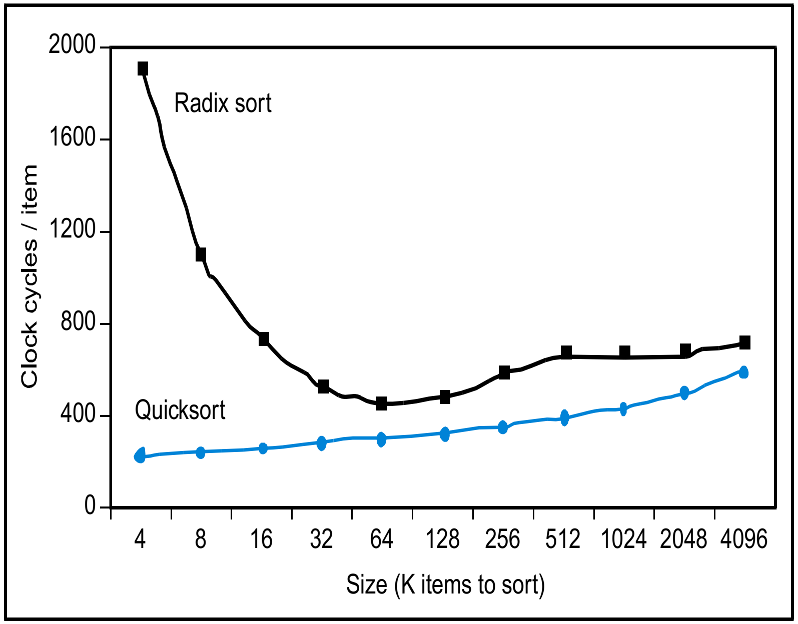
- Suppose we have a processor with the following parameters:
  - Base CPI=1.0, if hit in the L1 cache. L1 cache miss rate is 2%. Clock rate is 5 GHz. Memory access time is 100 ns, including all the miss handling.
  - Miss penalty to main memory is 100ns/0.2 = 500 clock cycle.
  - Total CPI = 1.0 + 2%×500 = 11.0
- If we add a L2 cache that has 5 ns access time. L2 global miss rate = 0.5%
  - Miss penalty to L2 is 5ns/0.2 = 25 clock cycle
  - Total CPI = 1.0+ 2%×25 + 0.5%×500 = 4.0

L2 Cache Design Principle

- L2 not tied to CPU clock cycle
  - Different design consideration from L1
  - Hits are less important than misses
  - Larger cache, higher associativity and larger blocks

Caches vs. Performance

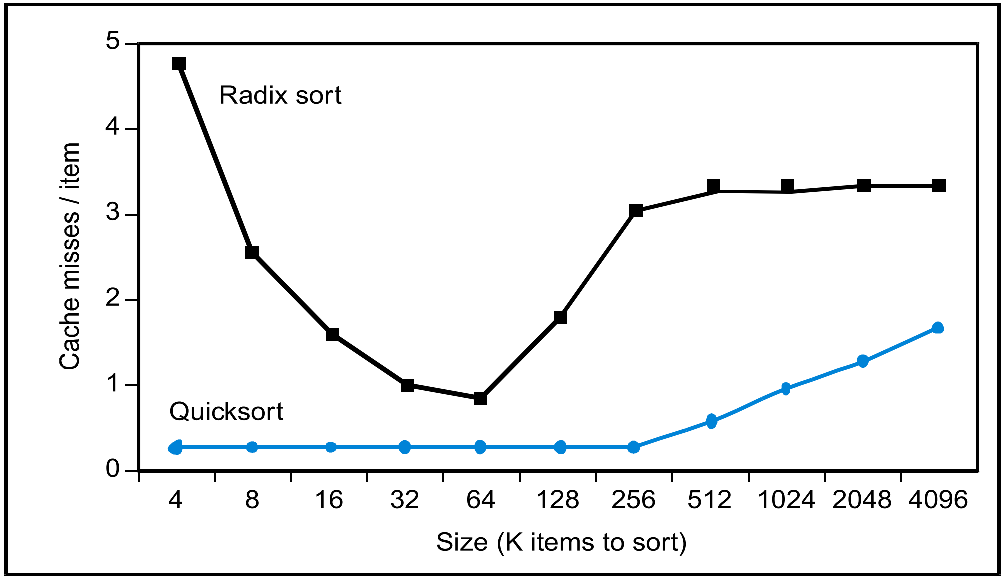




Theoretical behavior of Radix sort vs. Quicksort (instruction/item)

Observed behavior of Radix sort vs. Quicksort (clock cycles/item)

Caches vs. Performance (cont.)

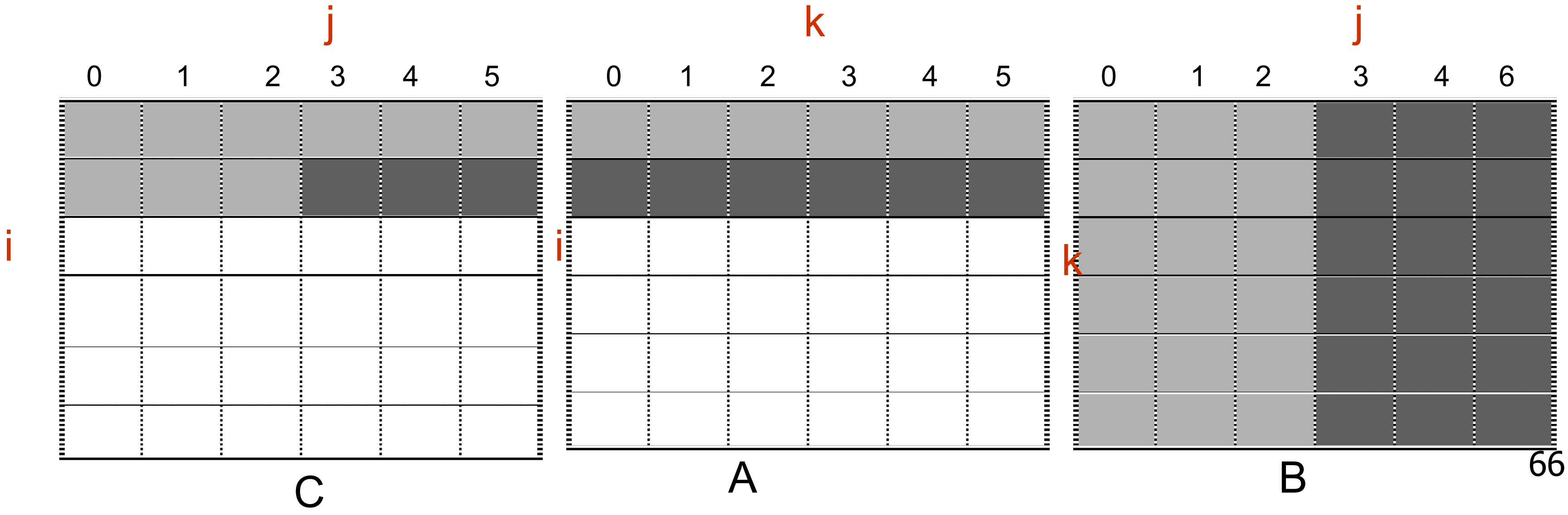


Cache behavior
Radix sort vs. Quicksort
(cache misses /item)

- Memory system performance is often critical factor
  - Multilevel caches, pipelined processors, make it harder to predict outcomes
  - Need experimental data

Software Optimization via blocking:DGEMM

```
for (int j = 0; j < n; ++j)
{
   double cij = C[i+j*n];  /* cij = C[i][j] */
   for( int k = 0; k < n; k++ )
            cij += A[i+k*n] * B[k+j*n];  /* cij += A[i][k]*B[k][j] */
   C[i+j*n] = cij;  /* C[i][j] = cij */
}
```

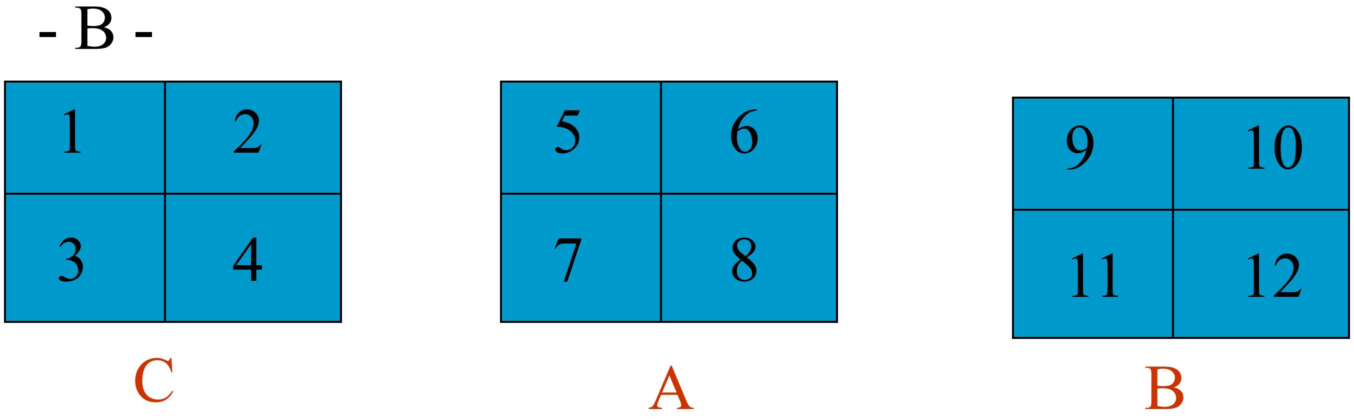


Blocking Example

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i+j*n];  /* cij = C[i][j] */
  for( int k = 0; k < n; k++ )
            cij += A[i+k*n] * B[k+j*n];  /* cij += A[i][k]*B[k][j] */
  C[i+j*n] = cij;  /* C[i][j] = cij */
}
```

- Cache accesses 2 x N^3 + N ^ 2
- Worst cache misses: 2 x N^3 + N^2
- Blocking: compute on BxB submatrix that fits in the cache

Blocking Example:

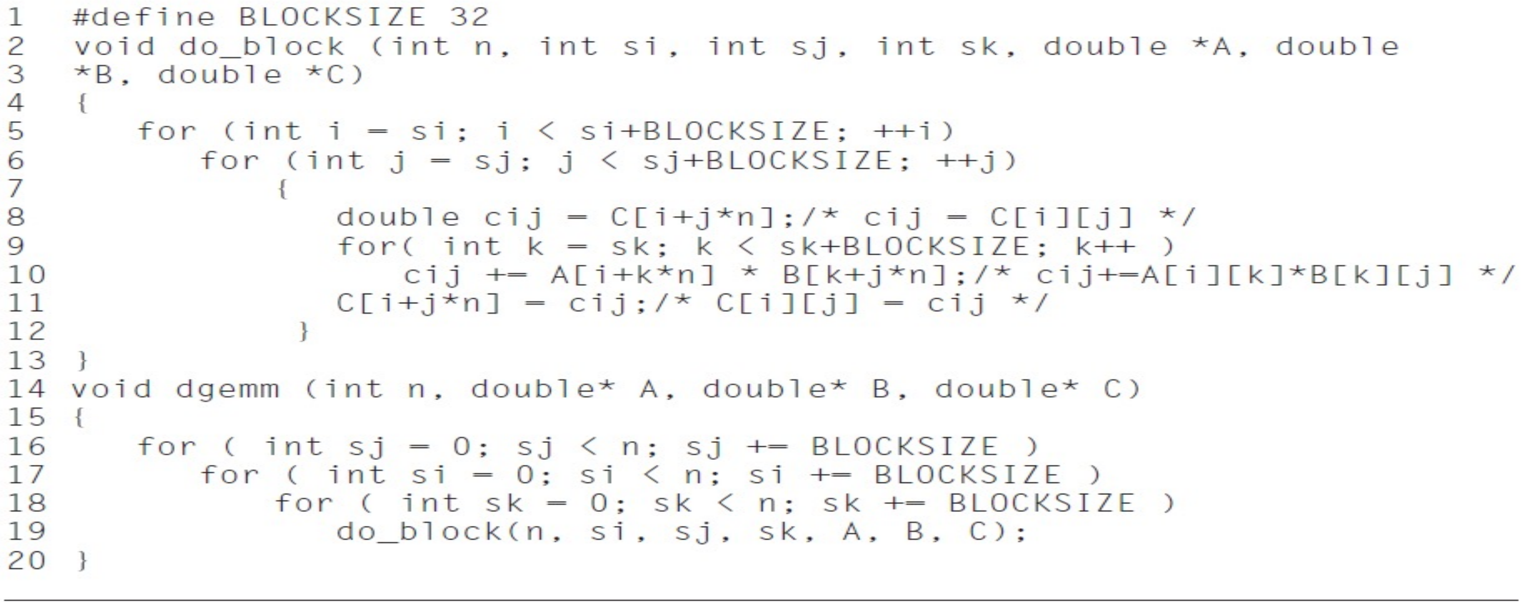


(1, 5, 9)
(1, 6, 11)
(3, 7, 9)
(3, 8, 11)
(2, 5, 10)
(2, 6, 12)
(4, 7, 10)
(4, 8, 12)

1. Assume B x B can hit in the cache
2. Ignore conflict misses

# of misses = 2 x N^3/ B + N^2

Blocking Example



```
#define BLOCKSIZE 32
void do_block (int n, int si, int sj, int sk, double *A, double
*B, double *C)
{
   for (int i = si; i < si+BLOCKSIZE; ++i)
      for (int j = sj; j < sj+BLOCKSIZE; ++j)
          {
             double cij = C[i+j*n];/* cij = C[i][j] */
             for( int k = sk; k < sk+BLOCKSIZE; k++ )
                cij += A[i+k*n] * B[k+j*n];/* cij+=A[i][k]*B[k][j] */
             C[i+j*n] = cij;/* C[i][j] = cij */
           }
}
void dgemm (int n, double* A, double* B, double* C)
{
   for ( int sj = 0; sj < n; sj += BLOCKSIZE )
      for ( int si = 0; si < n; si += BLOCKSIZE )
          for ( int sk = 0; sk < n; sk += BLOCKSIZE )
             do_block(n, si, sj, sk, A, B, C);
}
```