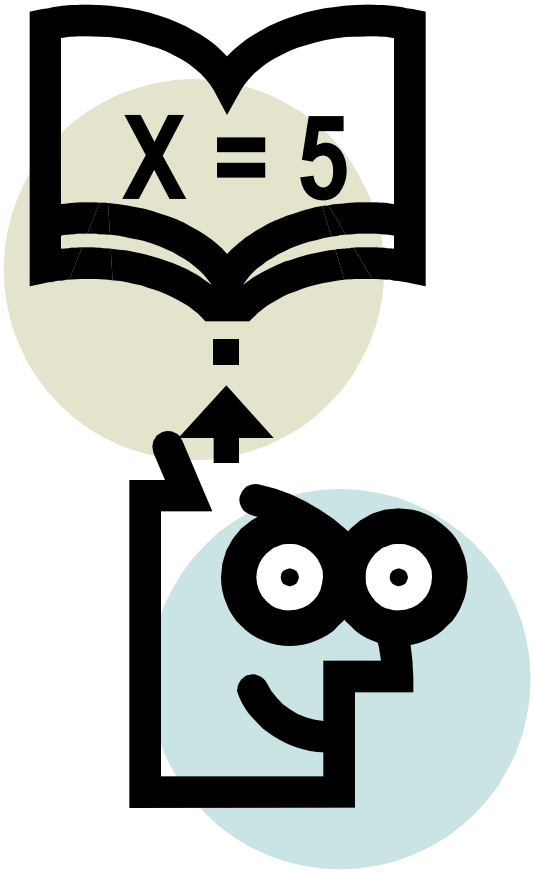
Lecture 14

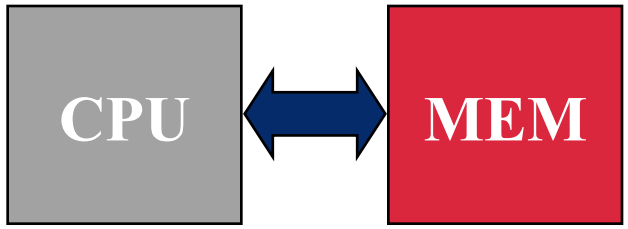
Memory Hierarchies: Caches



Adapted from slides developed by Profs. Hardavellas, Falsafi, Hill, Marculescu, Patterson, Rutenbar and Vijaykumar of Northwestern, Carnegie Mellon, Purdue, Berkeley, UWisconsin

Today's Menu:

- Basic Cache Functionality
- Basic Cache Design Parameters
- Cache Performance (next time?)



Importance of Memory System

- **Every instruction makes at least one memory reference**
  - fetching the instruction
- **Load and store instructions make memory references too**
  - on “average,” about 1/3 of a program’s instructions are loads or stores
  - typical application has 2 loads for every store
    - Example: A = B + C

      Load R1 <-- B

      Load R2 <-- C

      Add R3 <-- R1 + R2

      Store R3 --> A
  - Some software, such as the operating system, executes about the same number of loads and stores

Importance of Memory System (cont.)

- **A program's memory references often determines the ultimate performance of a program**
  - Small, computationally intensive programs make few memory references
    - These program's run time is bounded by the speed of the processor (i.e., ALU, registers, floating point unit, etc.)
    - Examples of computationally intensive programs are:
      - Encryption
      - Digital media (e.g., mpeg, jpeg)
      - Some scientific computations
  - Many other types of applications rely heavily on the memory system
    - Program run time is bounded by the speed of the memory system
    - Examples of memory intensive programs are:
      - Operating system
      - Databases
      - Web browsers

Technology Trends

	Capacity	Speed (latency)
Logic:	2x in 2 years	2x in 3 years
DRAM:	4x in 3 years	2x in 10 years
Disk:	4x in 3 years	2x in 10 years

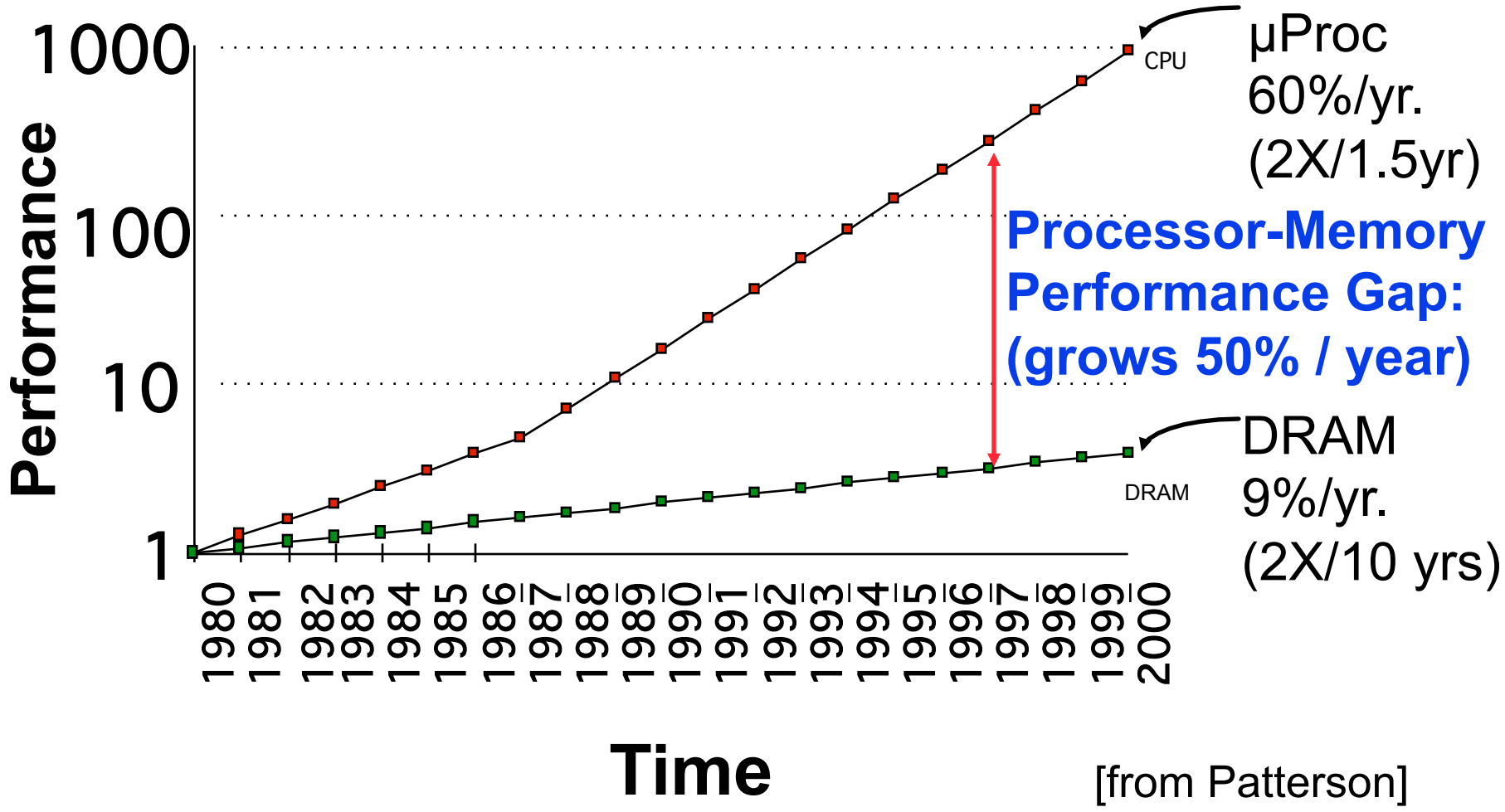
DRAM

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

1000:1!

2:1!

Processor-DRAM Performance Gap



[from Patterson]

Performance Impact of Memory

- **Two examples**
  - First, assume **1 cycle to memory** on every memory access
  - Next, assume **4 cycles to memory** on every load or store access (but still 1 cycle for instruction fetches)

```
lw      r2,0x20
lw      r3,0x30
add     r1,r2,r3
sw      r1,0x40
```

Performance Impact of Memory

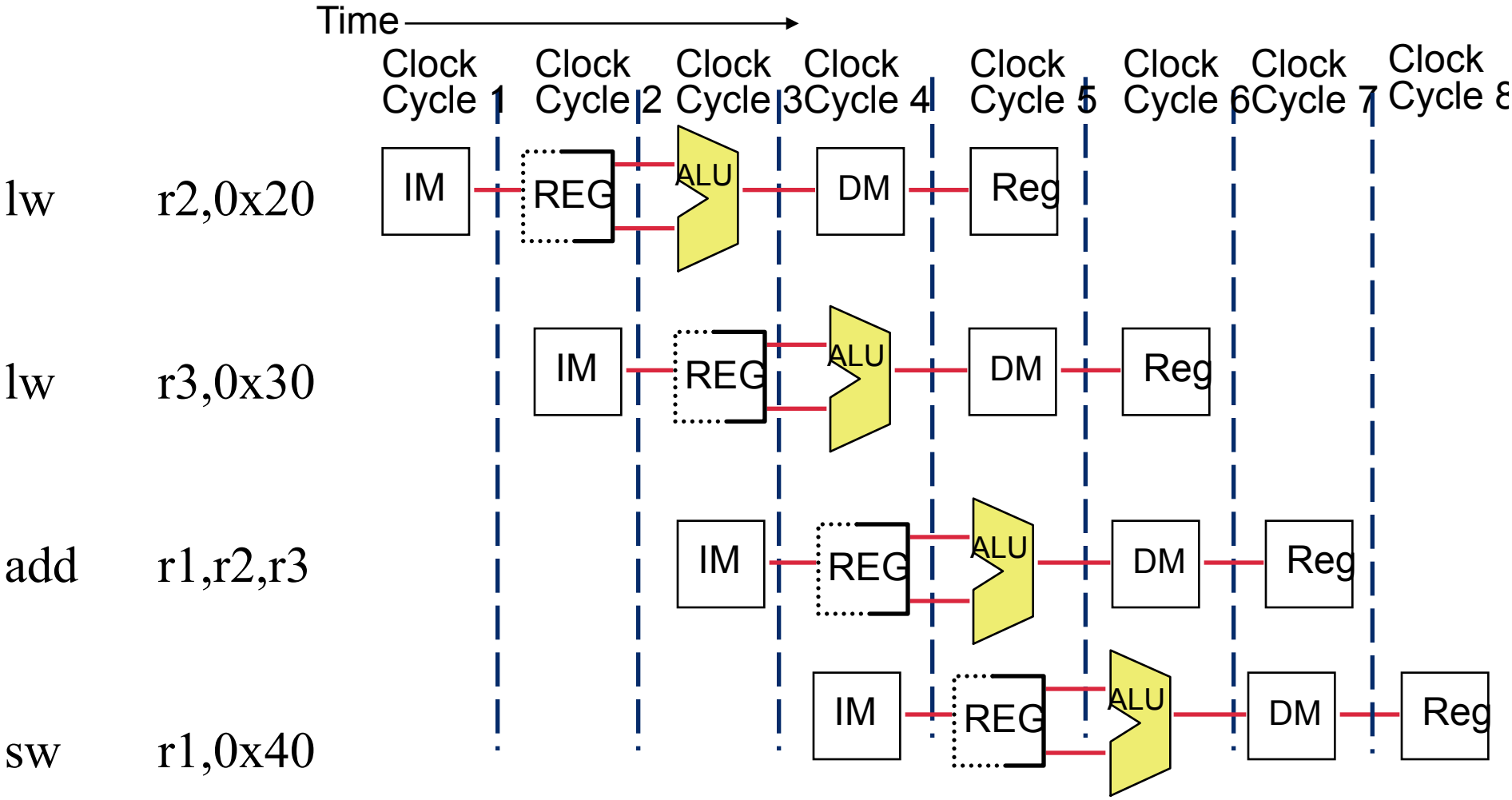
- **Two examples**
  - First, assume **1 cycle to memory** on every memory access
- **Assume a multi-cycle implementation:**
  - sw, add take 4 cycles each
  - lw takes 5 cycles
  - Total # cycles = 5 + 5+ 4+ 4 = 18 cycles

Performance Impact of Memory

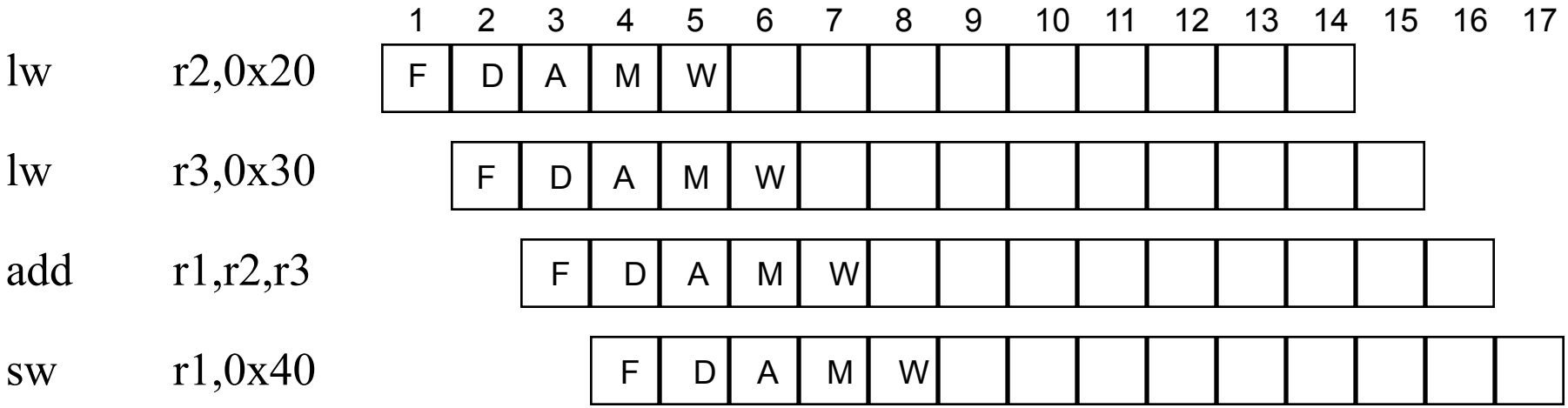
- **Two examples**
  - Next, assume **4 cycles to memory** on every load or store access (but still 1 cycle for instruction fetches)

- **Assume a multi-cycle implementation:**
  - add takes 4 cycles
  - sw takes 3 cycles more: 4 + 3 = 7 cycles
  - lw takes 3 cycles more: 5 + 3 = 8 cycles
  - Total # cycles = 7 + 5+ 8+ 8 = 28 cycles
  - This is a ~60% increase in execution time!

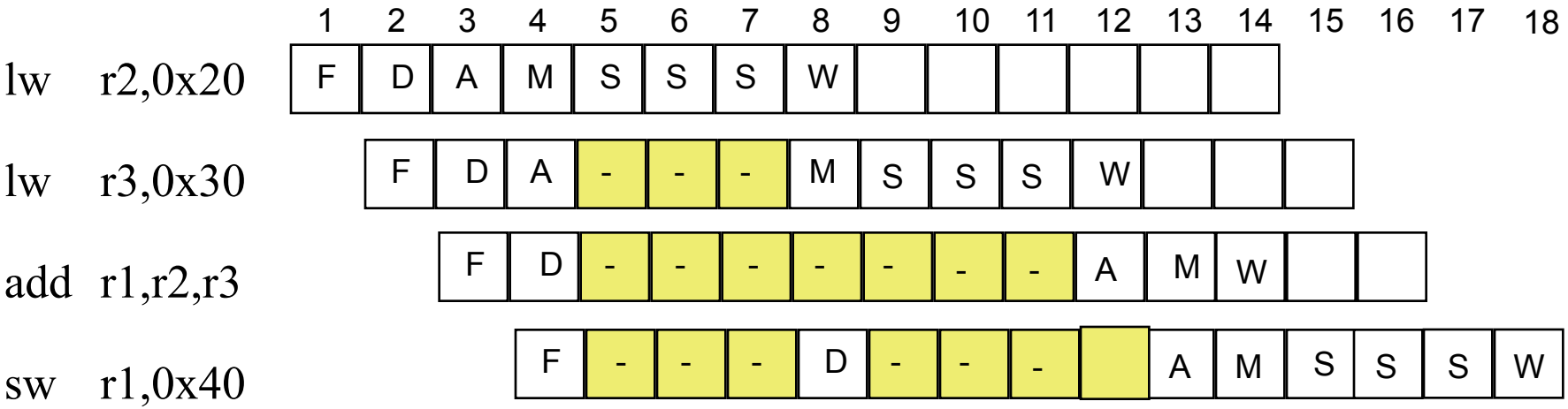
Example 1 (memory is 1 cycle)



Example 1 (again, simpler drawing)



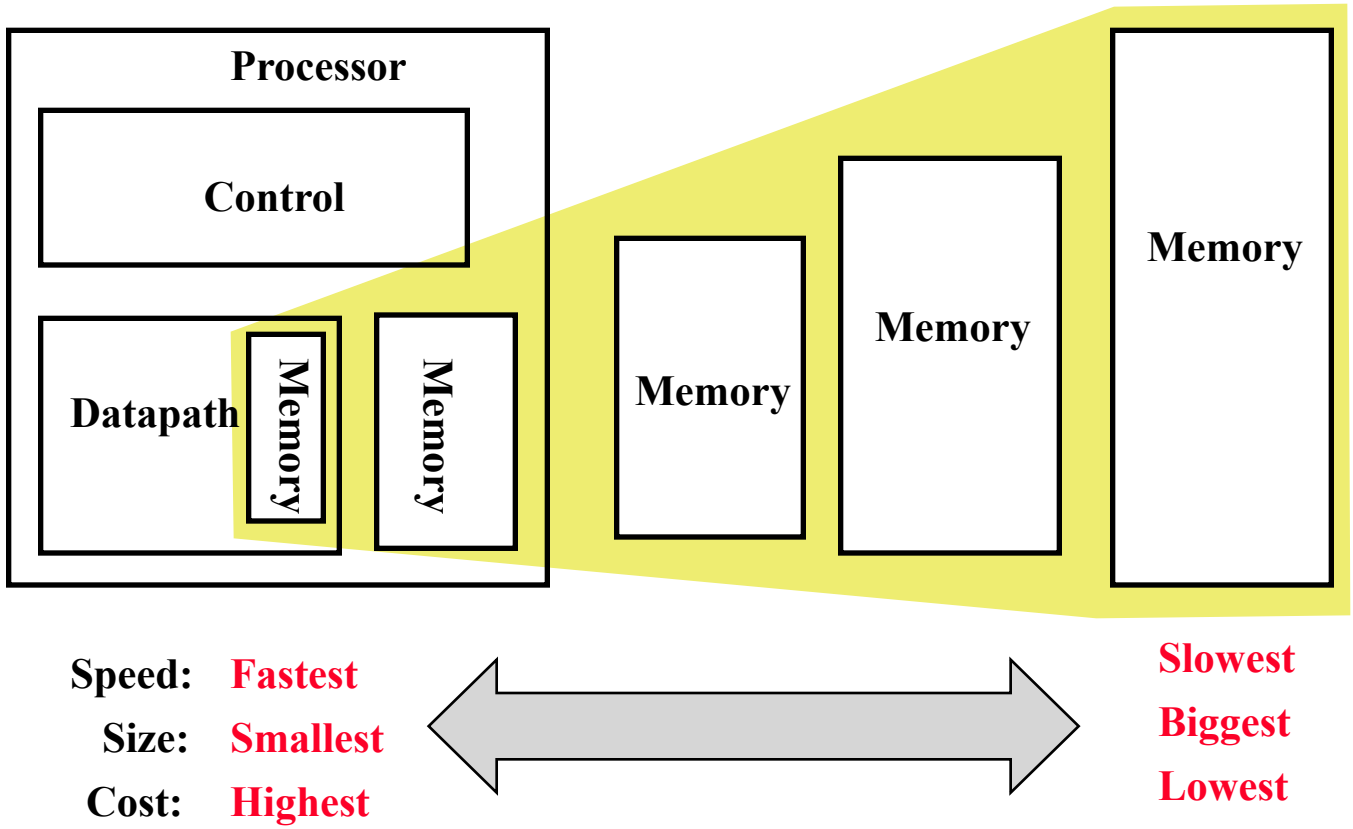
Example 2 (memory is 4 cycles now)



- **With** 1-cycle **memory system, program took** 8 cycles
  - CPI = 8 cycles / 4 instructions = 2.0
  - With lots more instructions, CPI would approach 1.0
- **With** 4-cycle **memory system, program takes** 18 cycles
  - CPI = 18 cycles / 4 instructions = 4.5
  - Doesn't include instruction fetch penalty found in real memory system

Solution: Memory Hierarchy

- Goal: Illusion of a fast, large and cheap memory system.
- Method: Memory hierarchy.



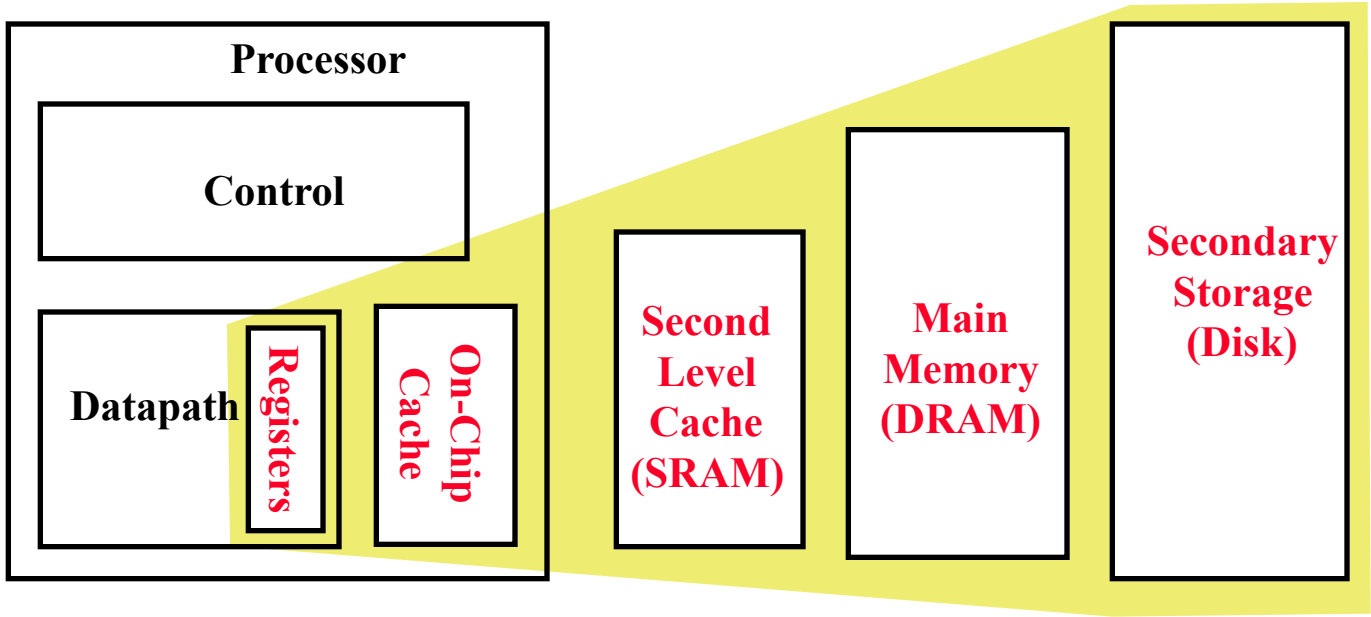
Why Hierarchy Works

- ***The Principle of Locality:***
  - Program accesses a relatively small portion of the address space at any instant of time.

- **Locality and Hierarchy:**
  - Provide the reality of a fast, large, and cheap memory system -- most of the time.

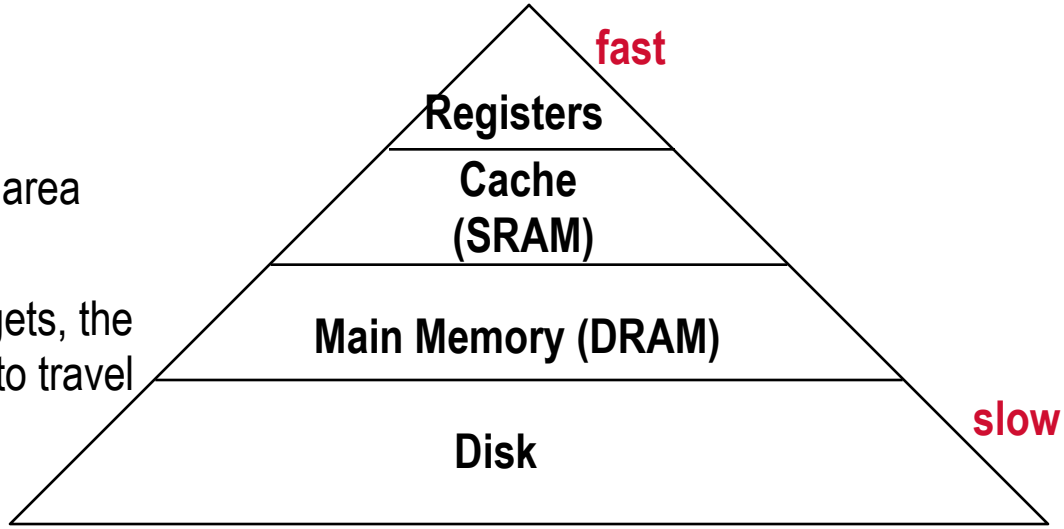
Memory Hierarchy of a Modern Computer System

- **By taking advantage of the principle of locality:**
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Memory Hierarchy and Typical Access Times

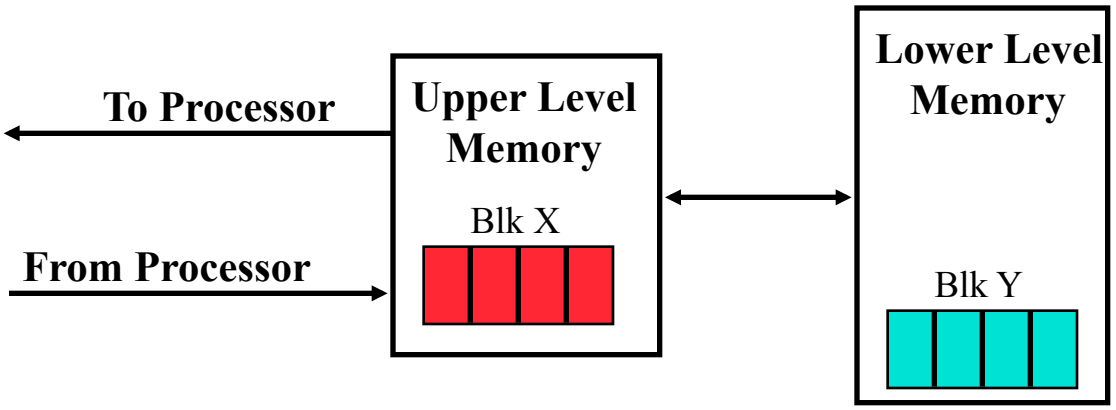
- **Fast memory is usually very expensive and small**
  - Expensive because it requires chip area and special board design
  - Small because the larger memory gets, the longer it takes for electrical signals to travel



Type of Memory	Typical Size	Typical Speed (latency)	Typical Bandwidth
Registers	32 * 8 Bytes	CPU Speed (< 1 ns)	~10 GB/sec
Level 1 Cache	< 64 KB	CPU speed (< 1 ns)	~10 GB/sec
Level 2 Cache	< 16 MB	5 – 20 cycles	25 – 150 GB/sec
Main Memory	< 16 GB	40 – 100 cycles	6 – 25 GB/sec
Disk	> 1 TB	1e6 – 10e6 cycles	3 – 6 Gb/sec
Network	----------	~100's – 100,000 cycles	100 Mb/sec – 10 Gb/sec
Bus	----------	~10 cycles	2 – 6 GB/sec

Memory Hierarchy: How Does it Work?

- **Temporal Locality (Locality in Time):**
  - => If an item is referenced, the **same item** will tend to be referenced again **soon**
  - => Keep most recently accessed data items closer to the processor
- **Spatial Locality (Locality in Space):**
  - => If an item is referenced, **nearby items** will tend to be referenced **soon**
  - => Move recently accessed "blocks" (groups of contiguous words) closer to proc.
- **"Block" (or "line") - minimum unit of data between 2 levels**

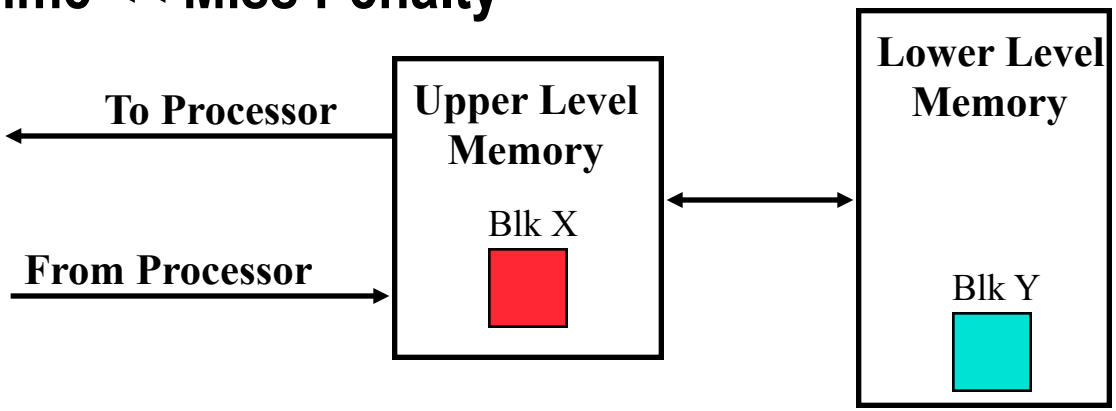


Example: L1-cache

Example: L2-cache

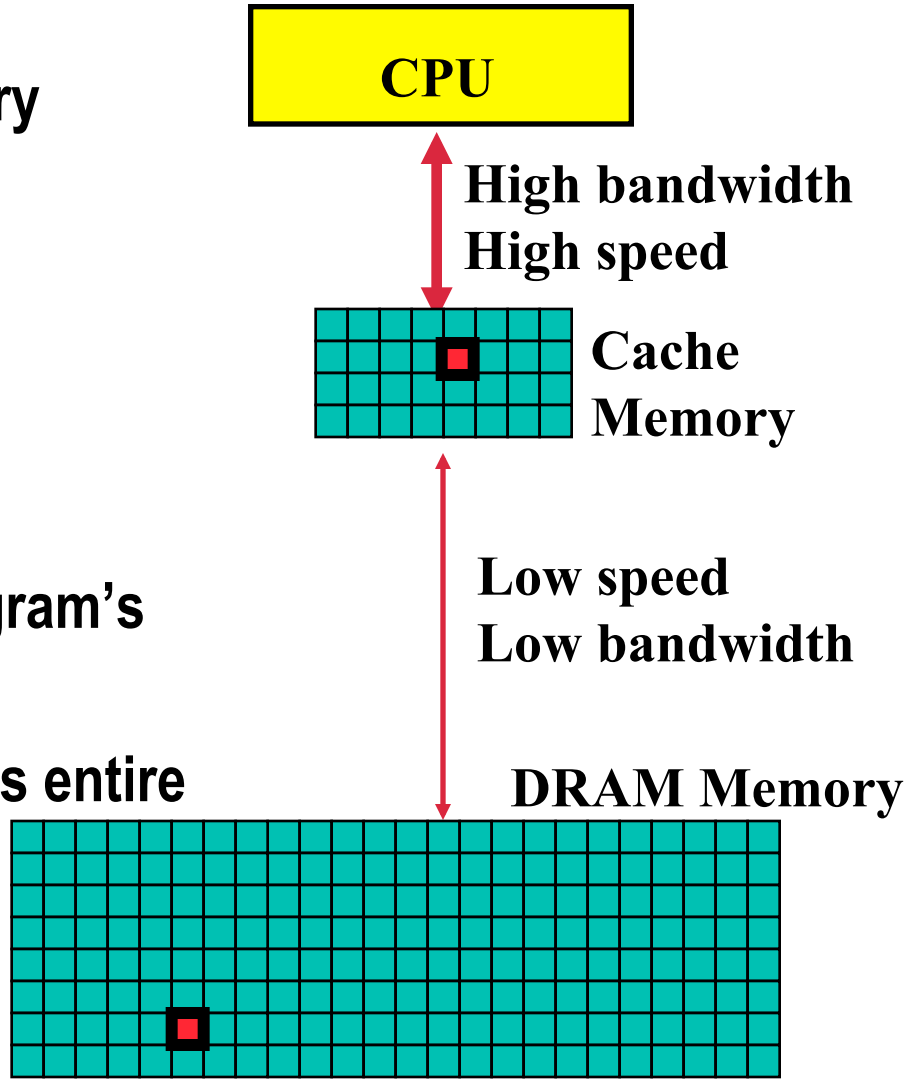
Memory Hierarchy: Terminology

- **Hit: data appears in some block in the upper level (e.g. Block X)**
  - Hit Rate: the fraction of memory accesses 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 (e.g. Block Y)**
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block to the processor
- **Hit Time << Miss Penalty**



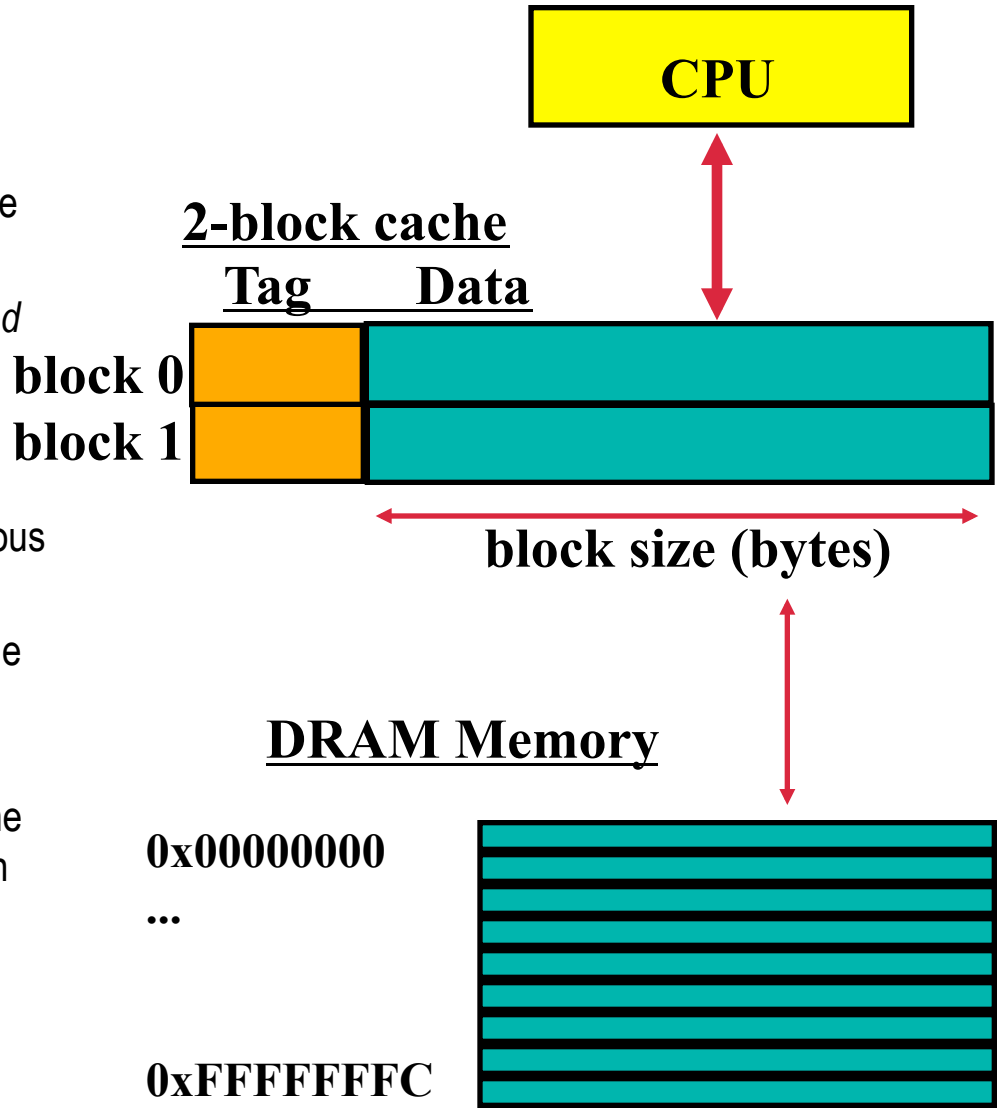
Cache Memory

- **Insert high-speed SRAM memory between DRAM and CPU**
  - Called a "cache"
  - Runs at the speed of processor
    - Typical latency of 1 – 4 cycles
  - Typically 8KB to 64KB in size
- **Caches hold a portion of a program's instructions and data**
- **Usually cannot hold a program's entire "working set"**
  - ie, *all* its instructions and *all* its data



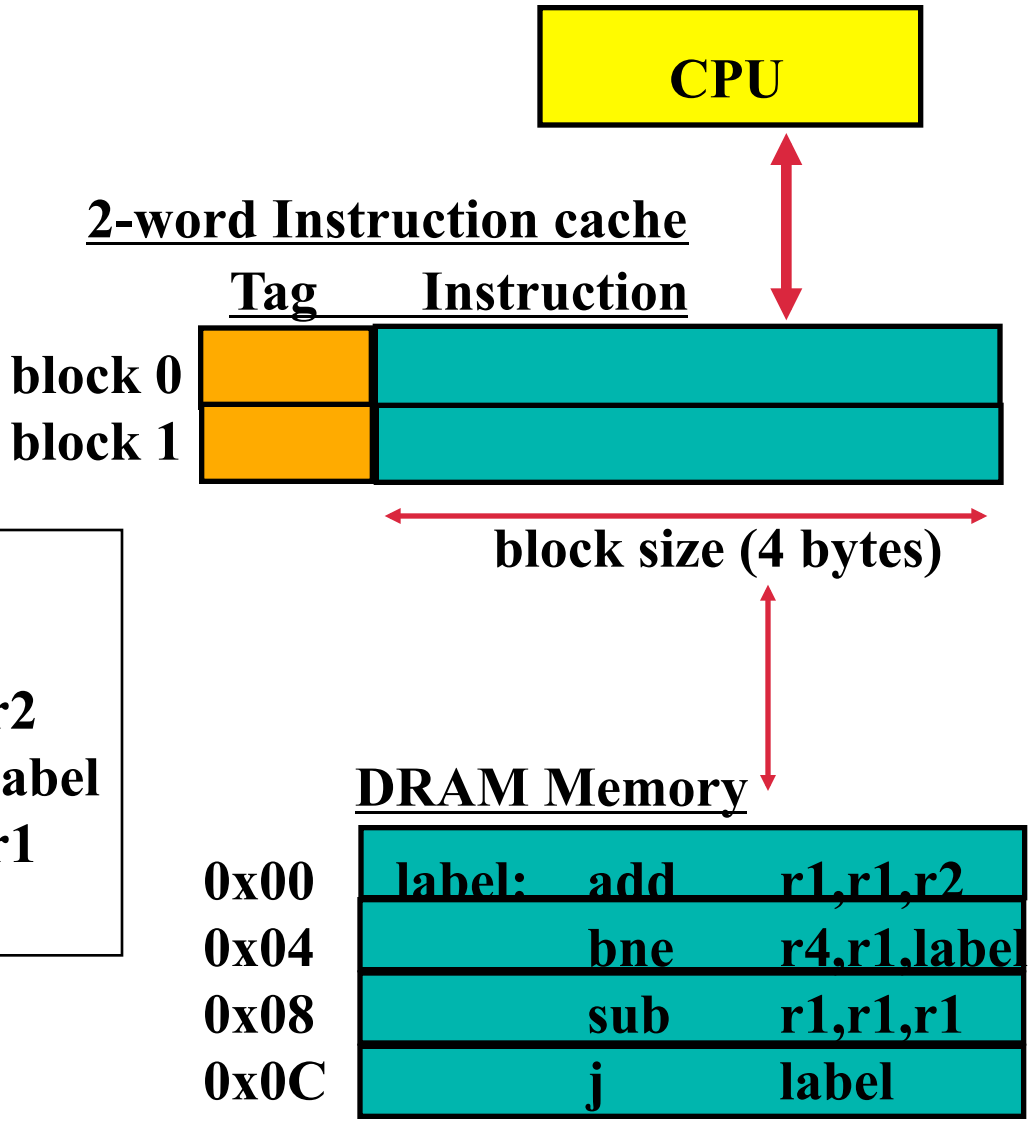
Basic Cache Design

- **Cache only holds a *portion* of a program**
  - Which part of the program does the cache contain?
    - Cache holds *most recently accessed references*
    - Cache is divided into units called *cache blocks* (also known as *cache "lines"*), each block holds a contiguous set of memory addresses
  - How does the CPU know which part of the program the cache is holding?
    - Each cache block has extra bits, called the *cache tag*, which holds the main memory address of the data in the block



Cache Example

- **Assume**
  - r1 == 0, r2 == 1, r4 == 2
  - 1 cycle to cache
  - 5 cycles to main memory
  - instructions take 1 cycle to execute



Sample Program

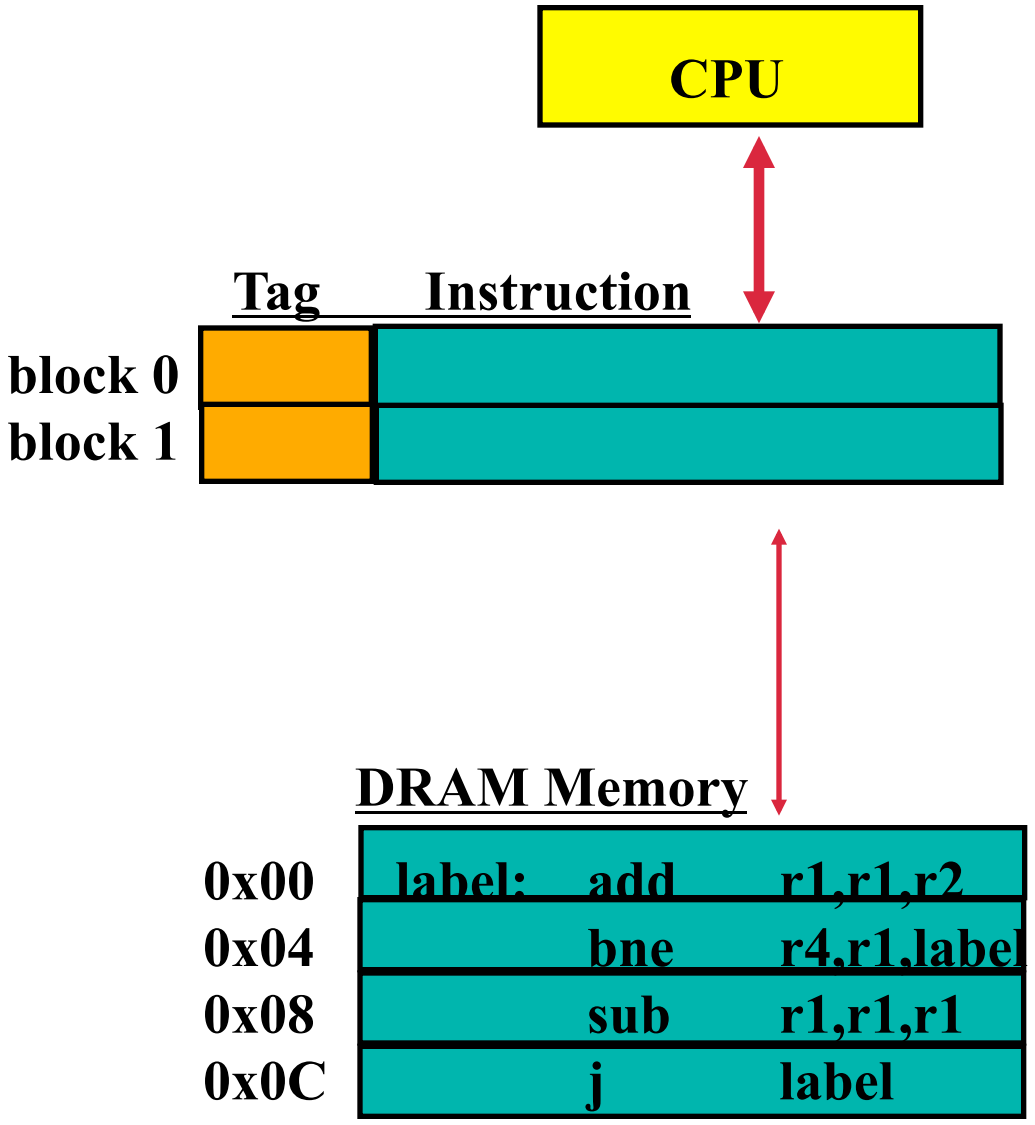
```
0x00    label:  add     r1,r1,r2
0x04            bne     r4,r1,label
0x08            sub     r1,r1,r1
0x0C            j       label
```

Small Cache Example (2)

- Assume
  - r1 == 0, r2 == 1, r4 == 2
  - 1 cycle to cache
  - 5 more cycles to main memory
  - instructions take 1 cycle to execute

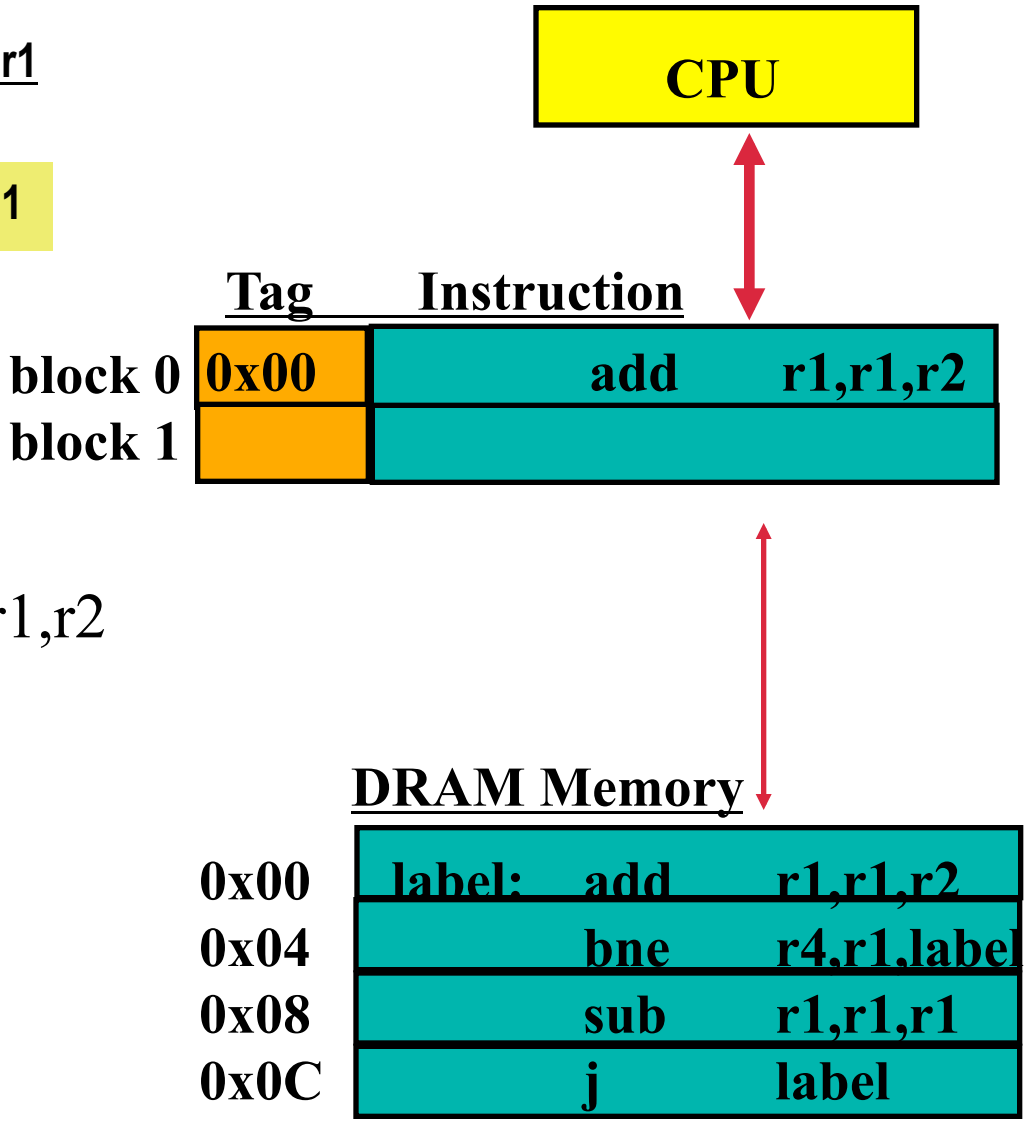
At cycle 1

- **PC == 0x00**
- **Fetch data from memory system**
  - a) **look into cache**
    --> this takes 1 cycle
  - b) **if not there, fetch from main memory and copy into cache**
    --> this takes 5 cycles



Small Cache Example (3)

Cycle	Address	Op/Instruction		r1
1 - 6		FETCH 0x00		
7	0x00	add	r1,r1,r2	1

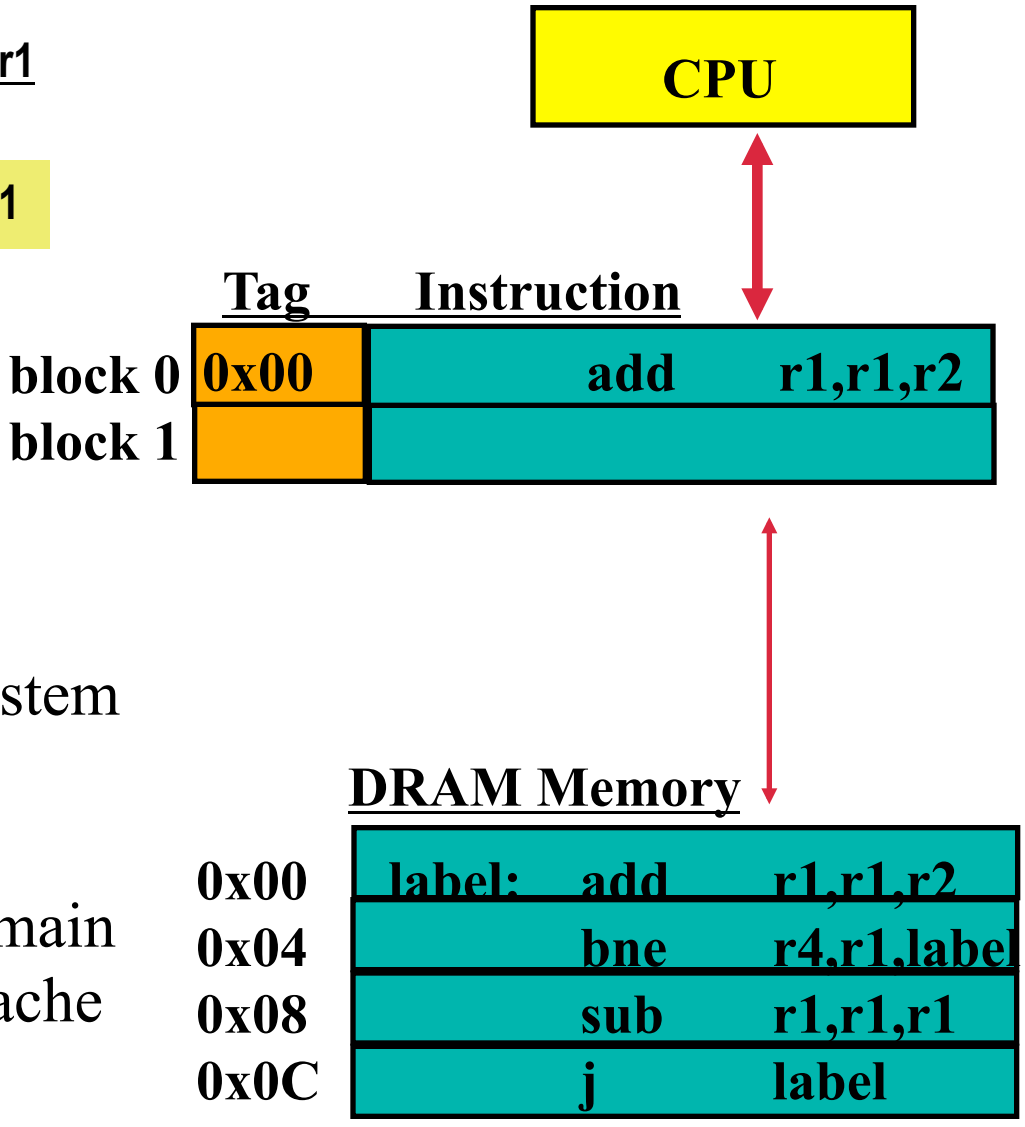


At cycle 7

- Execute instruction add r1,r1,r2

Small Cache Example (4)

Cycle	Address	Op/Instruction		r1
1 - 6		FETCH **0x00**		
7	0x00	add	r1,r1,r2	1
7		FETCH **0x04**		

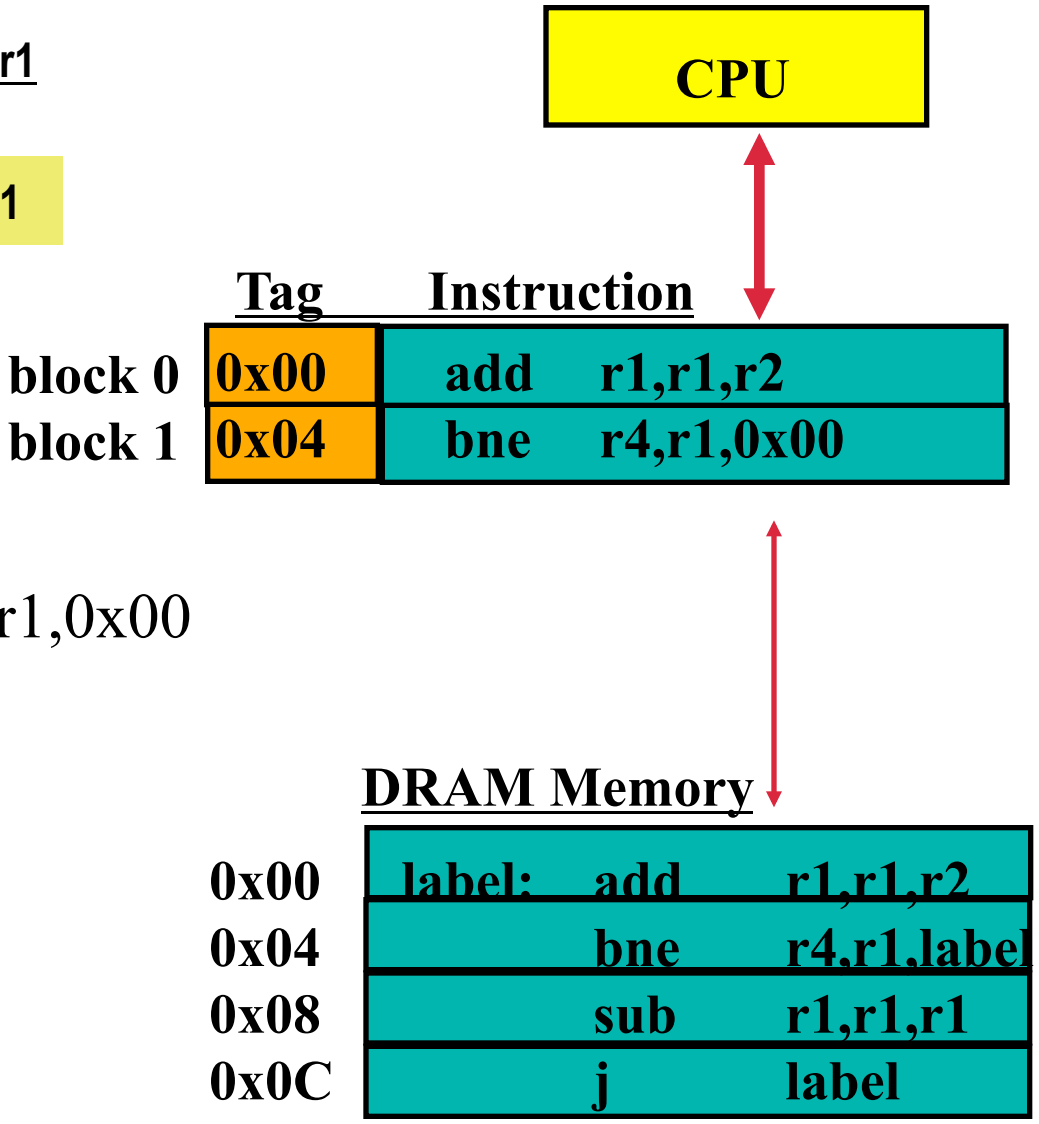


At cycle 7
PC == 0x04

- Fetch data from memory system
  a) look into cache
     --> this takes 1 cycle
  b) if not there, fetch from main memory and copy into cache
     --> this takes 5 cycles

Small Cache Example (5)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	

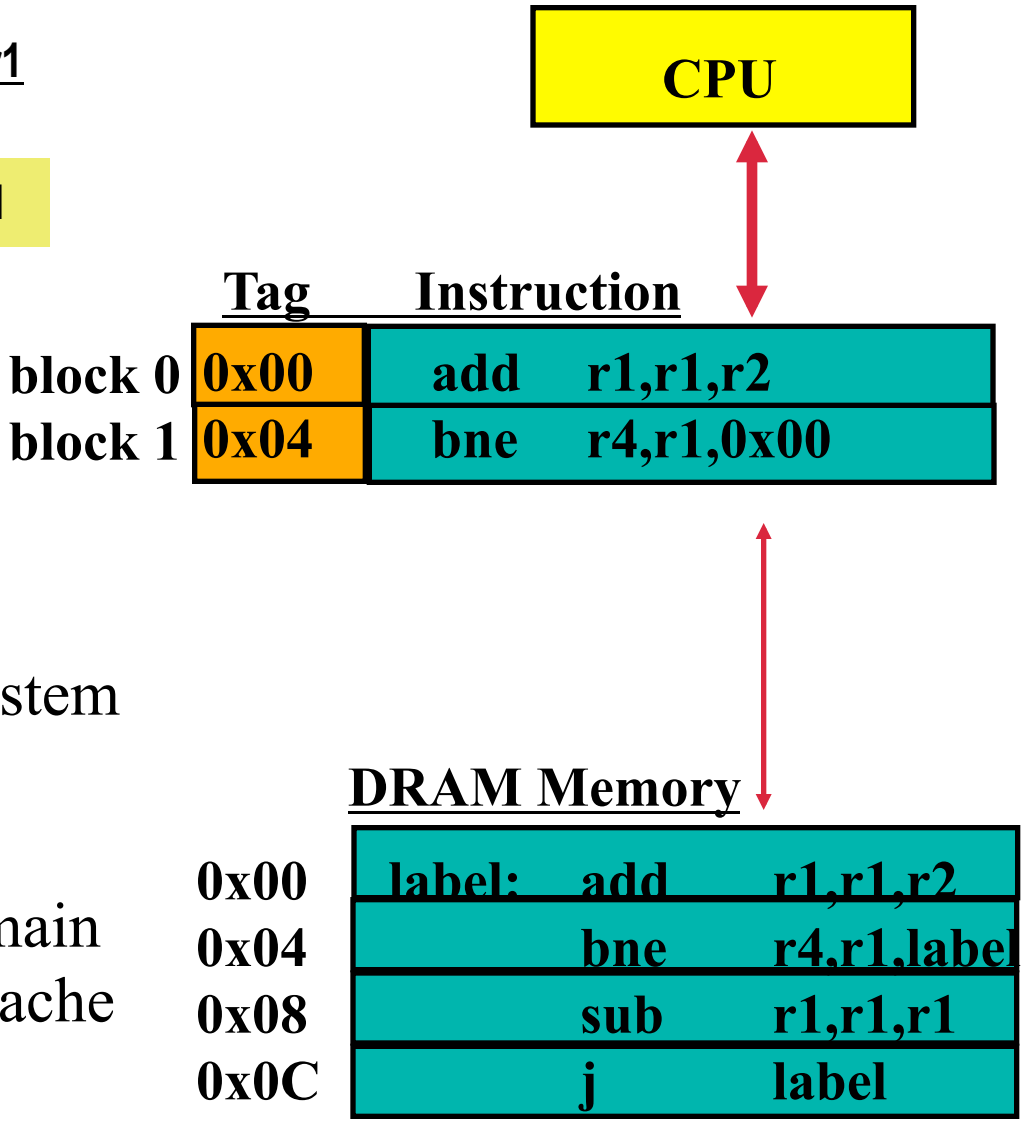


At cycle 13

- Execute instruction bne r4,r1,0x00

Small Cache Example (6)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	

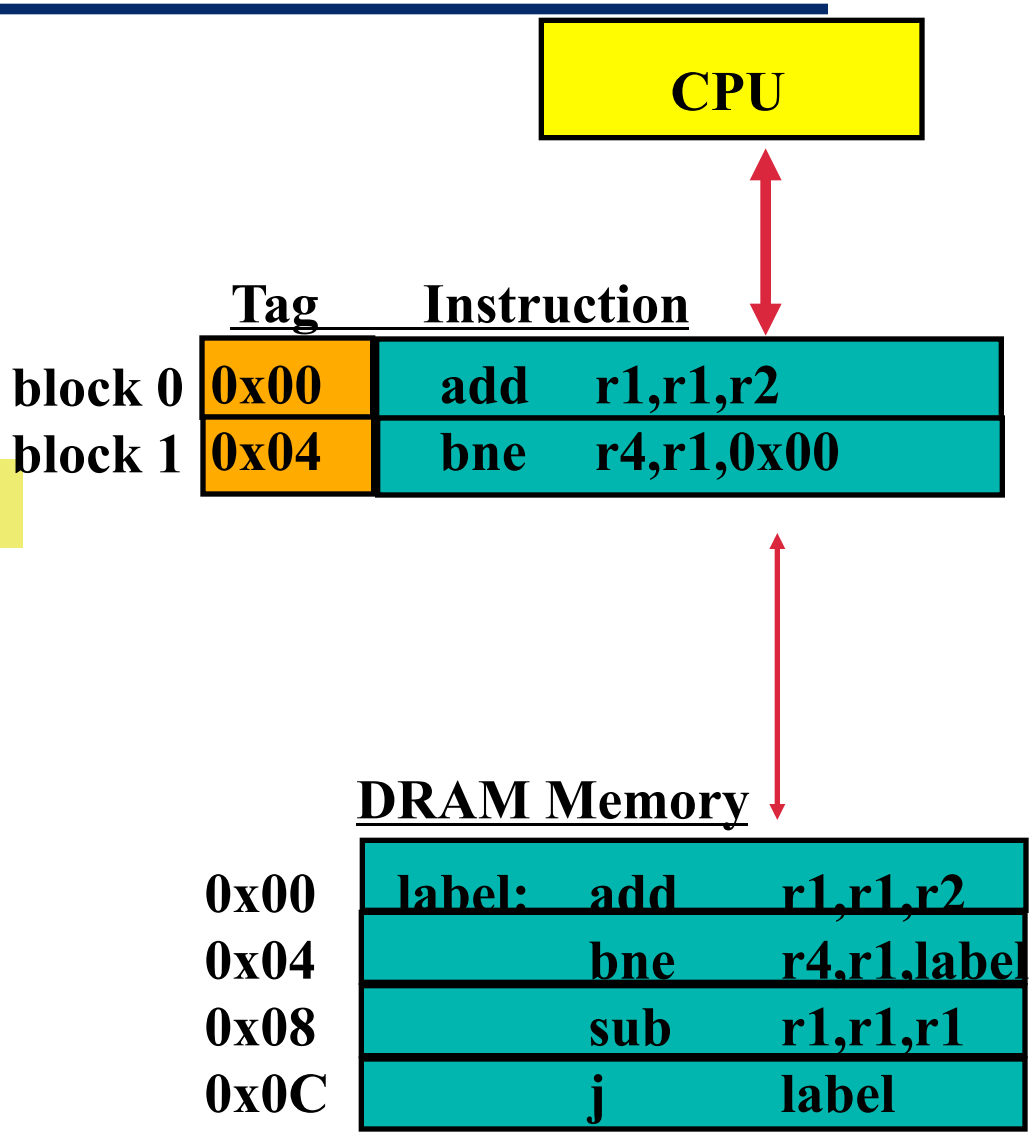


At cycle 13

- PC == 0x00
- Fetch data from memory system
  a) look into cache
     --> this takes 1 cycle
  b) if not there, fetch from main memory and copy into cache
     --> this takes 5 cycles

Small Cache Example (7)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2

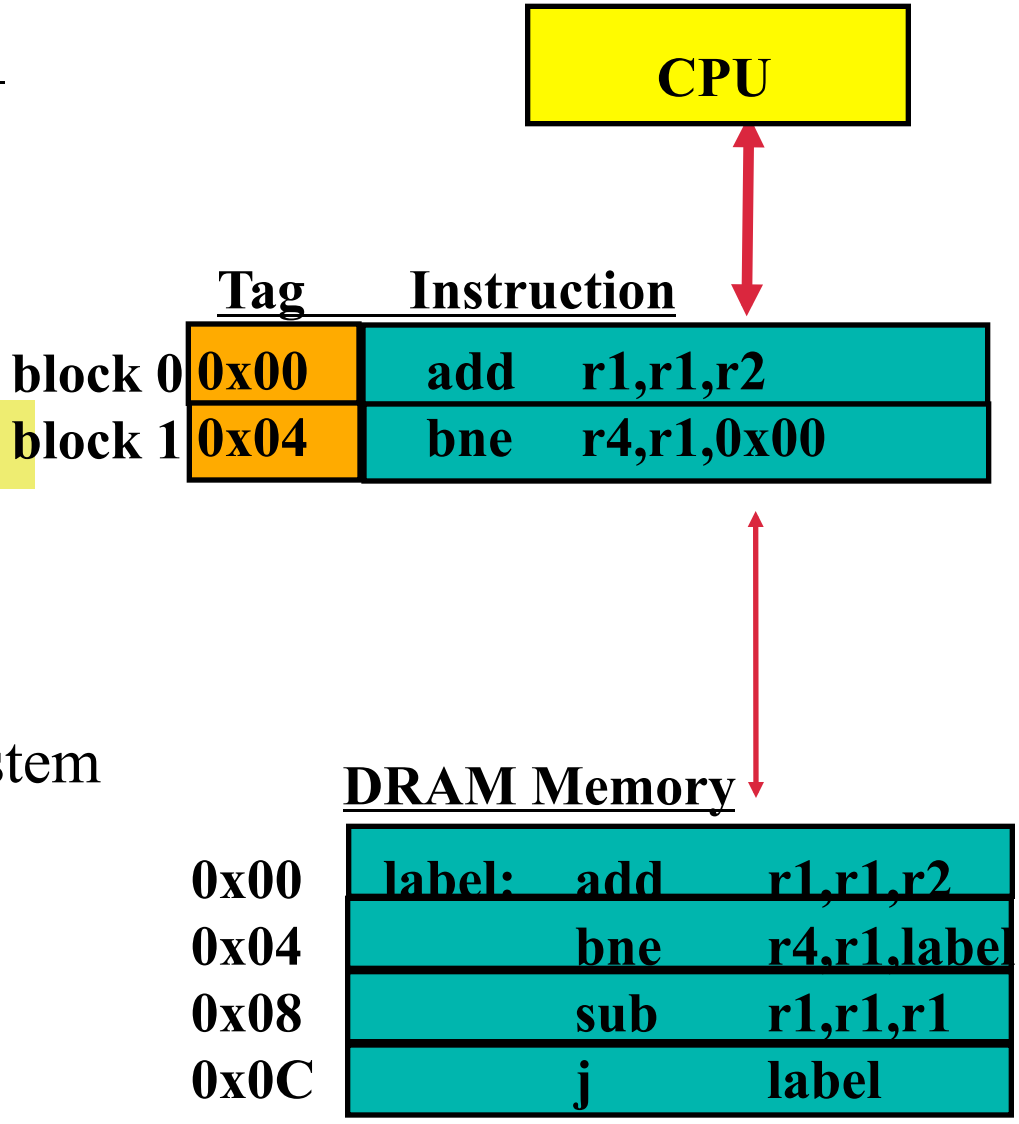


At cycle 14

- Execute add r1,r1,r2

Small Cache Example (8)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	

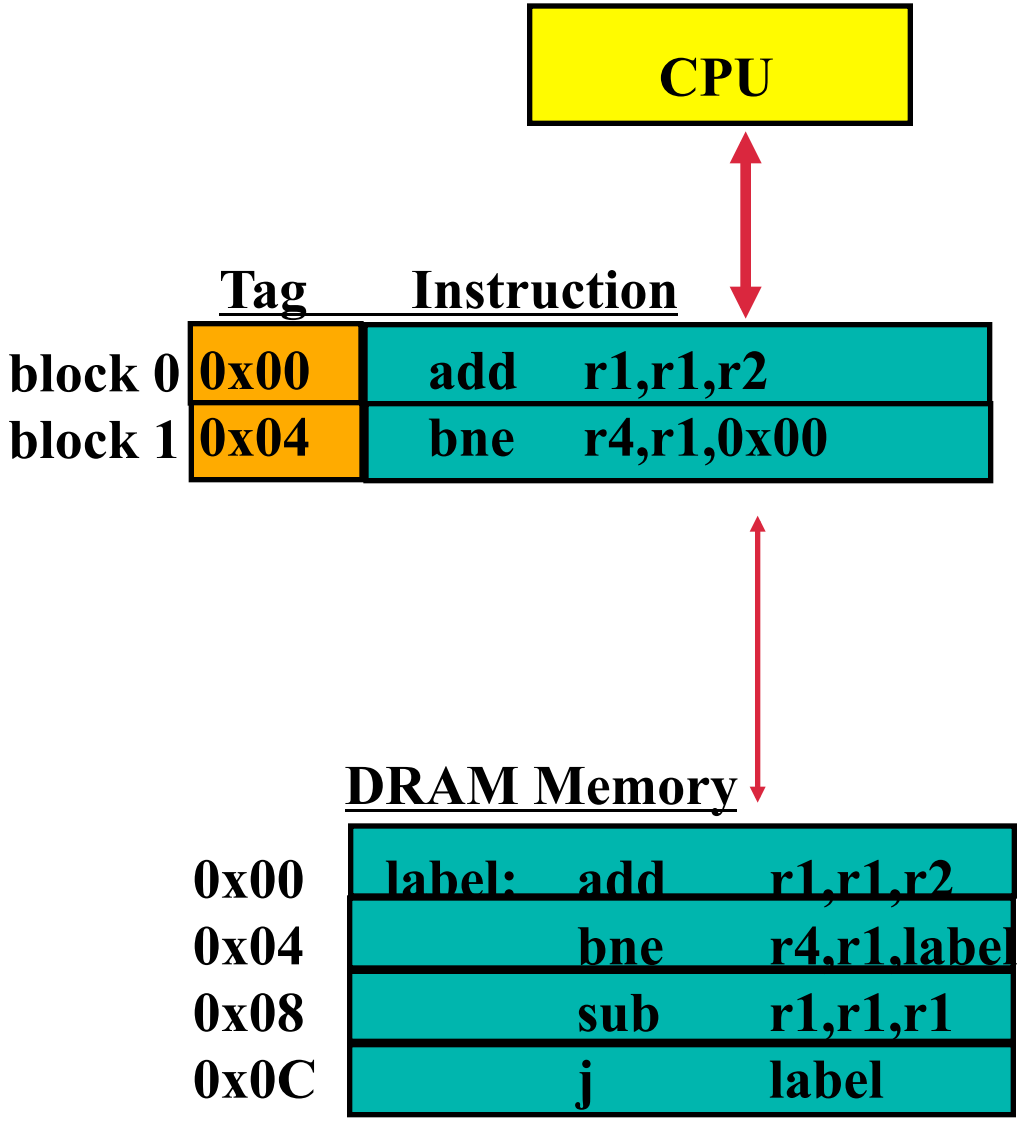


At cycle 14

- PC == 0x04
- Fetch data from memory system
  a) look into cache
  b) if not there, fetch from main memory

Small Cache Example (9)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	

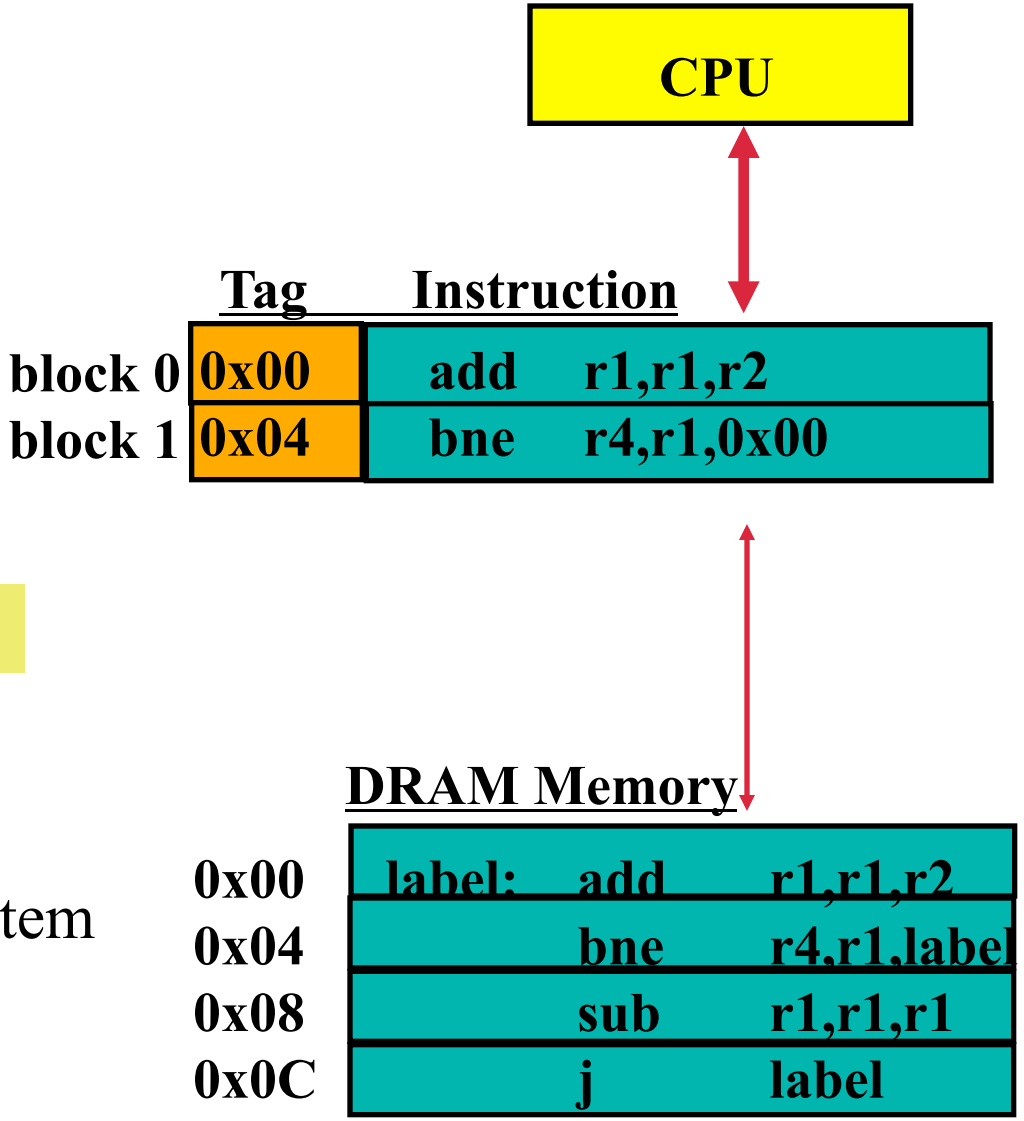


At cycle 15

- Execute bne r4,r1,0x00
  - Branch not taken

Small Cache Example (10)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15		FETCH 0x08	

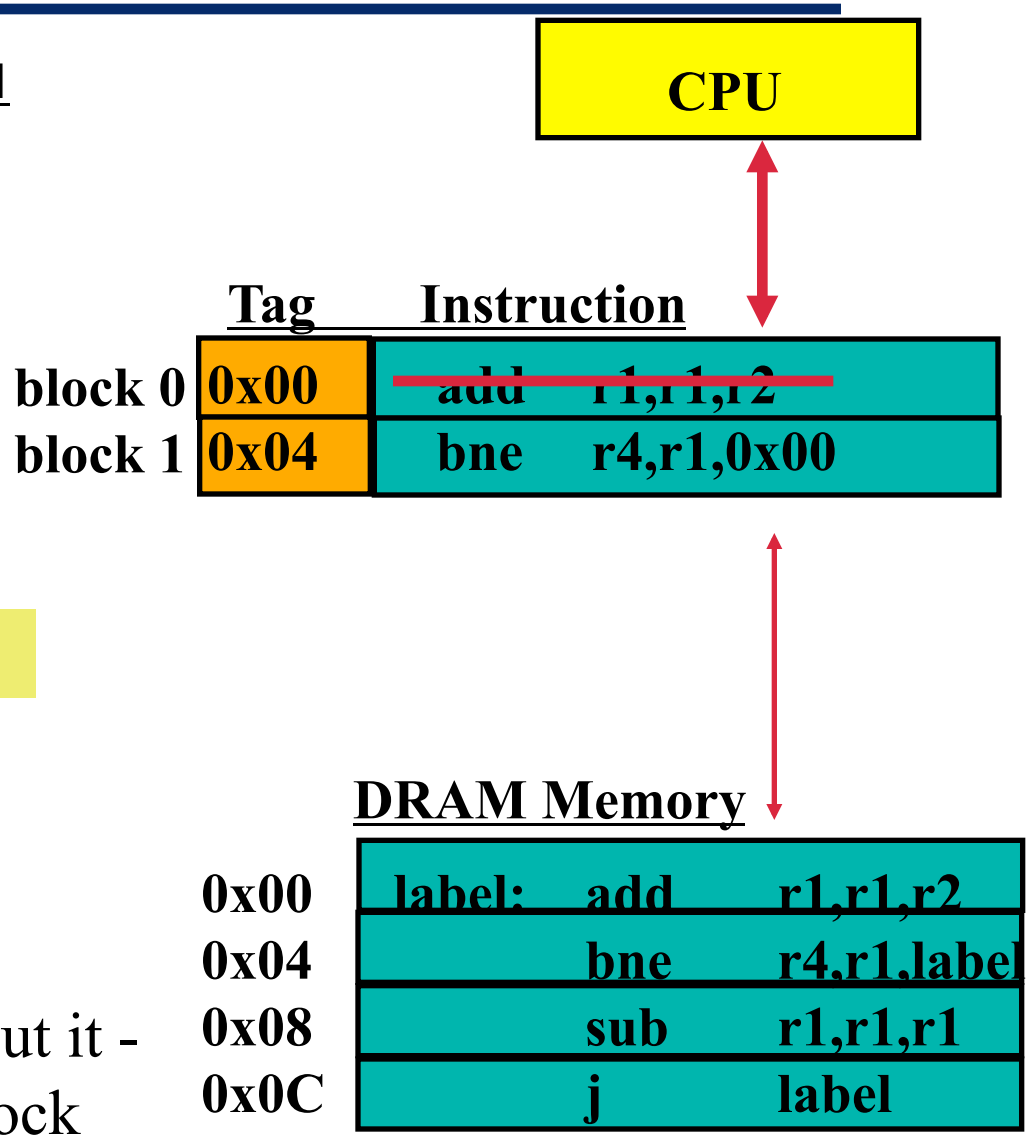


At cycle 15

- PC == 0x08
- Fetch data from memory system
  a) look into cache
  b) if not there, fetch from main memory

Small Cache Example (11)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15-20		FETCH 0x08	



At cycle 20

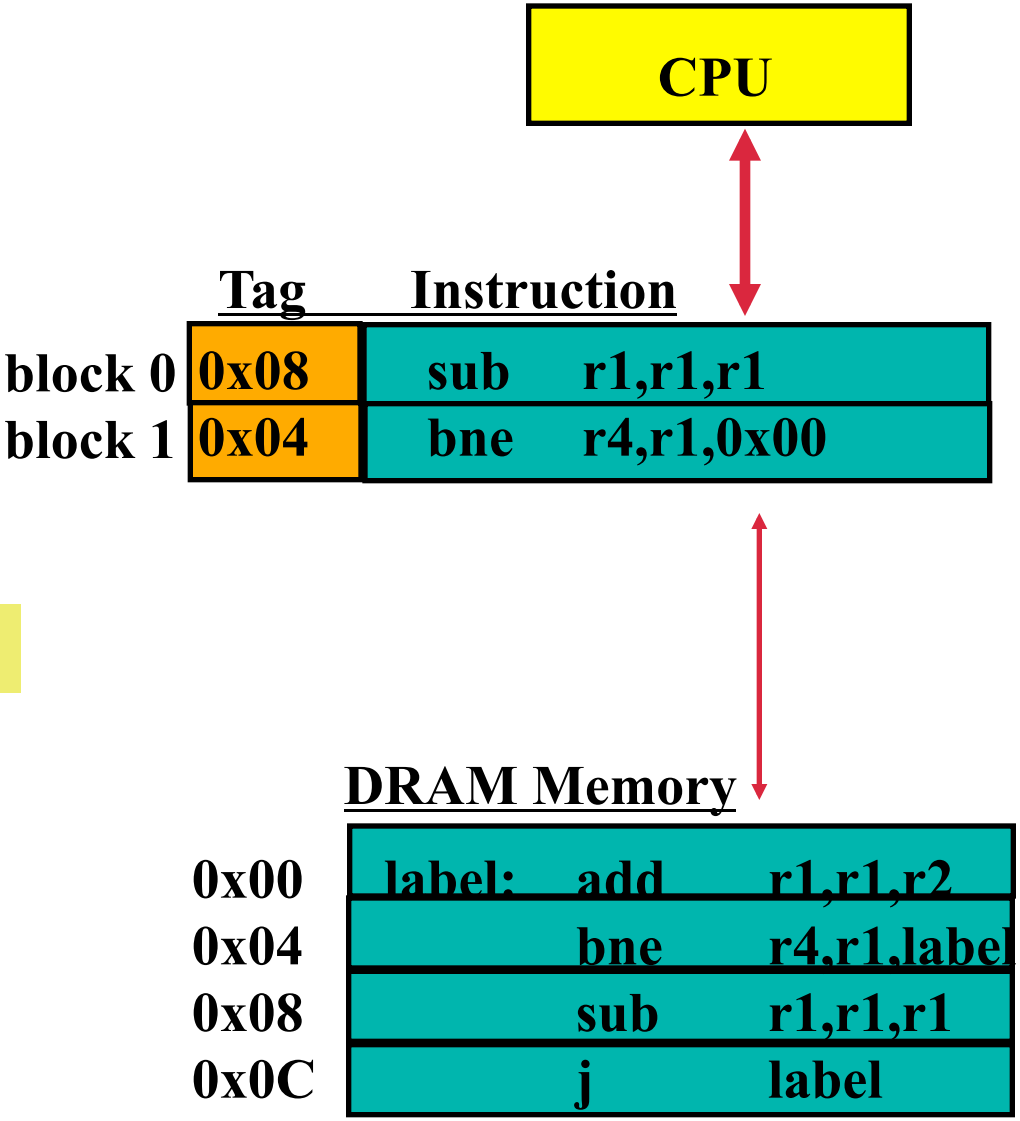
- Put fetched instruction (sub r1,r1,r1) into cache
- Where in cache should we put it - for now, randomly pick a block

Small Cache Example (12)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15 - 20		FETCH 0x08	
21	0x08	sub r1,r1,r1	0

At cycle 21

- Execute sub r1,r1,r1

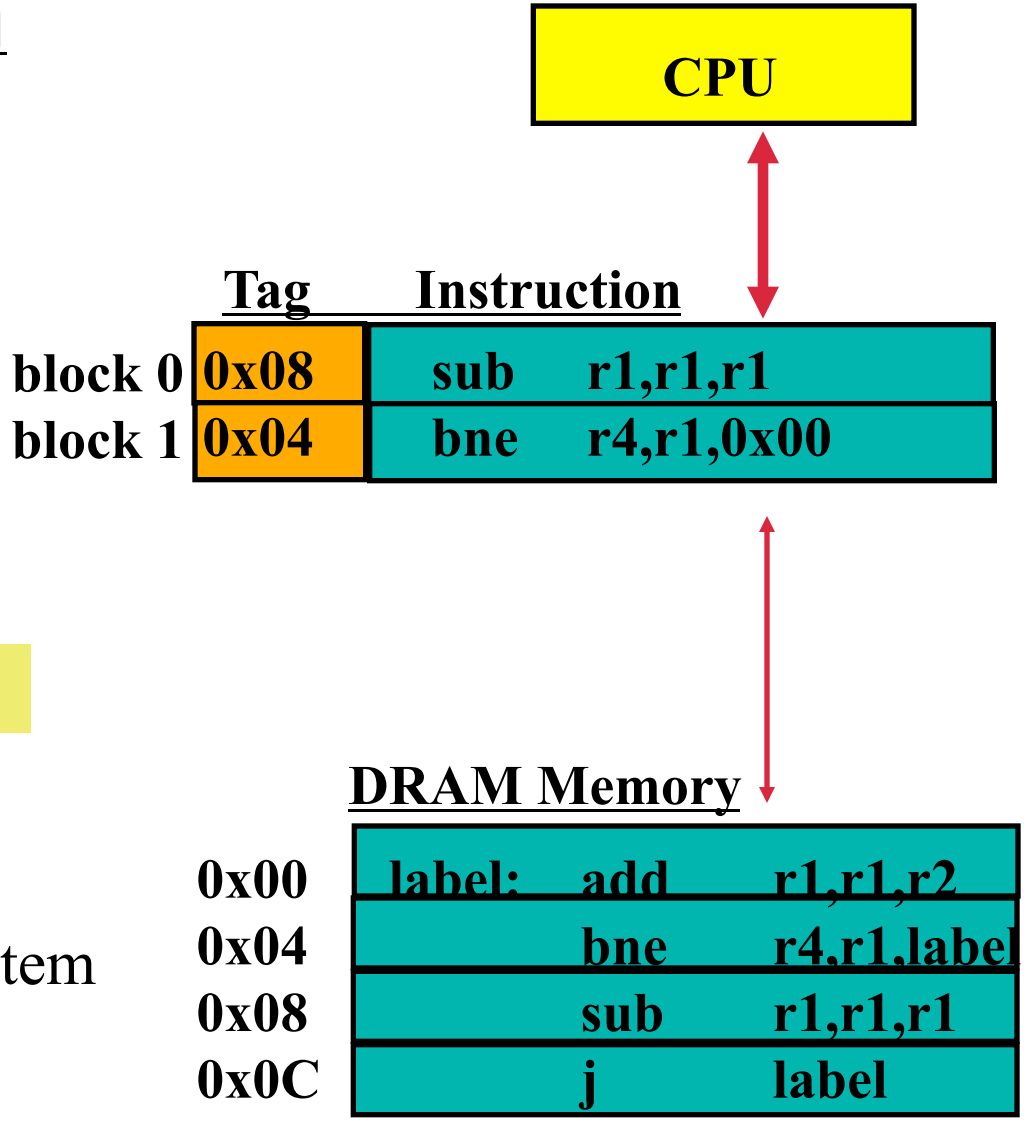


Small Cache Example (13)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15 - 20		FETCH 0x08	
21	0x08	sub r1,r1,r1	0
21		FETCH 0x0C	

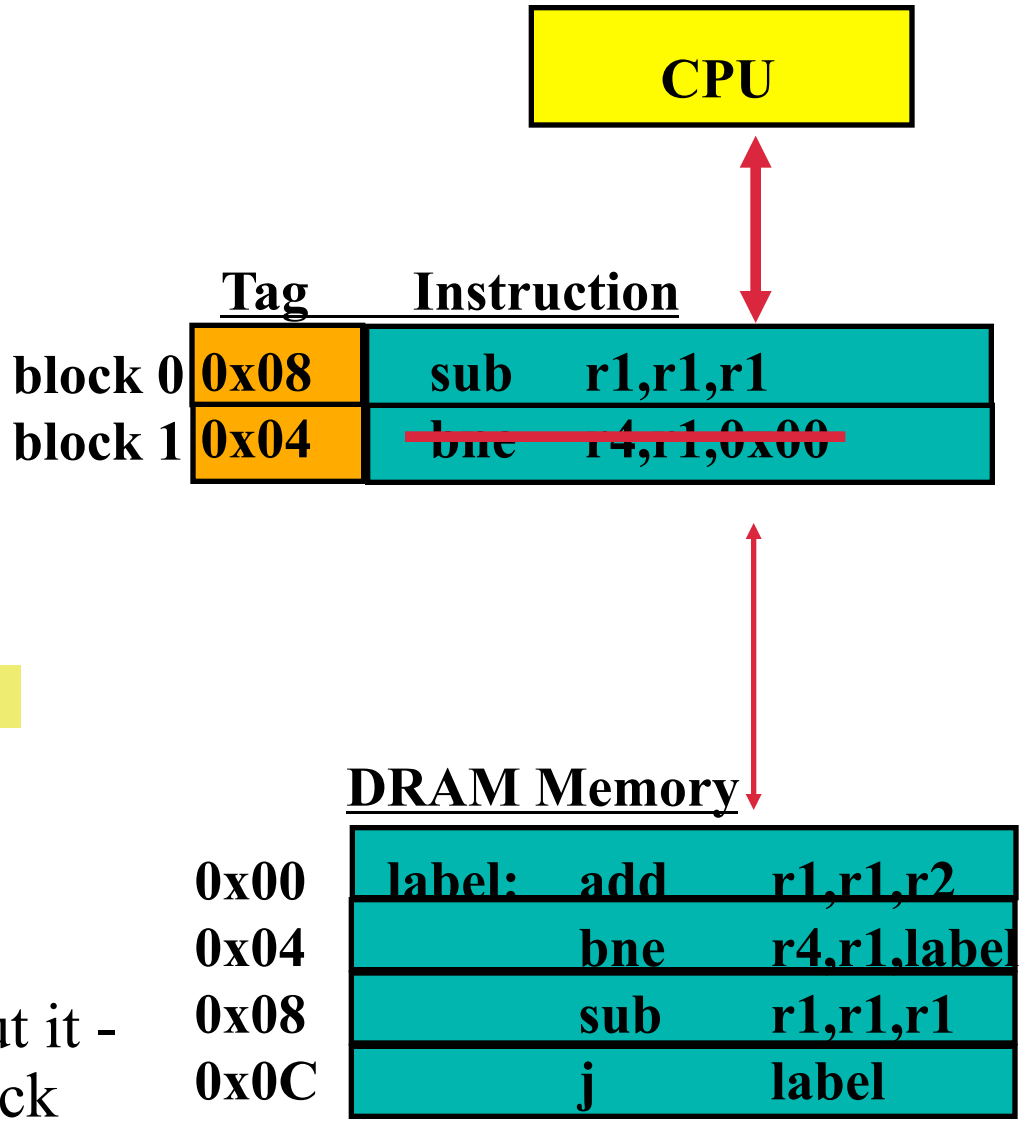
At cycle 21

- PC == 0x0C
- Fetch data from memory system
  a) look into cache
  b) if not there, fetch from main memory



Small Cache Example (14)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15 - 20		FETCH 0x08	
21	0x08	sub r1,r1,r1	0
21-26		FETCH 0x0C	



At cycle 26

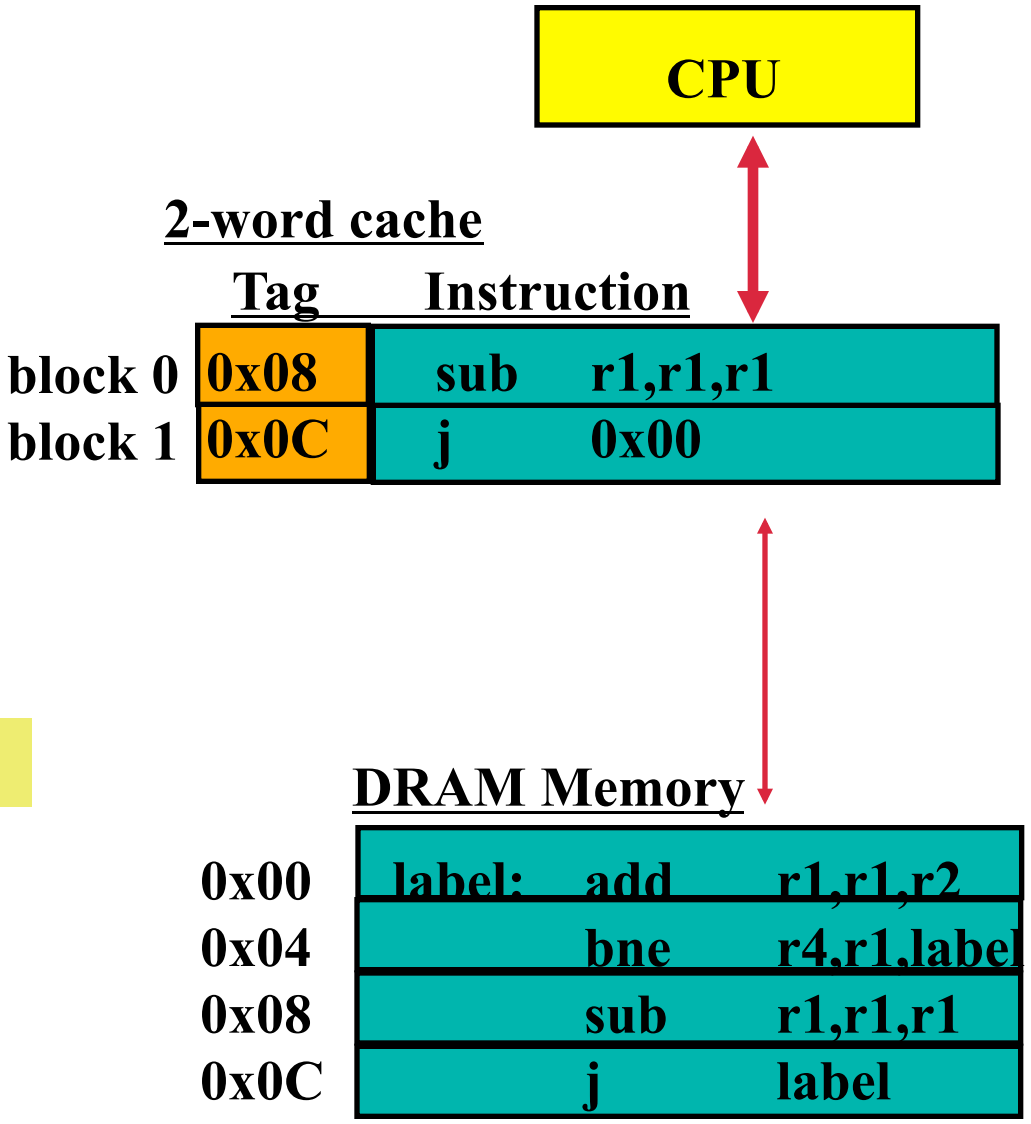
- Put fetched instruction (j 0x00) into cache
- Where in cache should we put it - for now, randomly pick a block

Small Cache Example (15)

Cycle	Address	Op/Instruction	r1
1 - 6		FETCH 0x00	
7	0x00	add r1,r1,r2	1
7-12		FETCH 0x04	
13	0x04	bne r4,r1,0x00	
13		FETCH 0x00	
14	0x00	add r1,r1,r2	2
14		FETCH 0x04	
15	0x04	bne r4,r1,0x00	
15 - 20		FETCH 0x08	
21	0x08	sub r1,r1,r1	0
21-26		FETCH 0x0C	
27		j 0x00	

At cycle 27

- Execute j 0x00



Compare No-cache vs. Cache

NO CACHE

Cycle	Address	Op/Instruction
1 - 5		FETCH 0x00
6	0x00	add r1,r1,r2
6-10		FETCH 0x04
11	0x04	bne r4,r1,label
11 - 15		FETCH 0x00
16	0x00	add r1,r1,r2
16 - 20		FETCH 0x04
21	0x04	bne r4,r1,label
21 - 25		FETCH 0x08
26	0x08	sub r1,r1,r1
26- 30		FETCH 0x0C
31	0x0C	j 0x00

CACHE

Cycle	Address	Op/Instruction
1 - 6		FETCH 0x00
7	0x00	add r1,r1,r2
7-12		FETCH 0x04
13	0x04	bne r4,r1,0x00
13		FETCH 0x00
14	0x00	add r1,r1,r2
14		FETCH 0x04
15	0x04	bne r4,r1,0x00
15 - 20		FETCH 0x08
21	0x08	sub r1,r1,r1
21-26		FETCH 0x0C
27	0x0C	j 0x00

Hit in cache

Oops, missed in the cache

The ABC's (or 1-2-3-4's) of Caches

- **Caching is a general concept used in processors, operating systems, file systems, and applications.**
- **Wherever it is used, there are four basic questions which arise. These include:**
  - Q1: *Where* can a block be placed in a cache?
    (block placement)
  - Q2: *How* is a block found if it is in a cache?
    (block lookup)
  - Q3: *Which* block should be replaced on a miss?
    (block replacement)
  - Q4: *What* happens on a write?
    (write strategy)

Q1: Block Placement

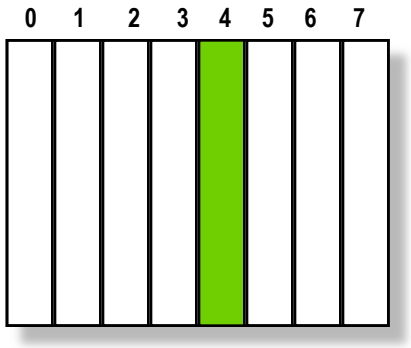
Direct Mapped
block 12 can go only into
block 4 (12 mod 8)

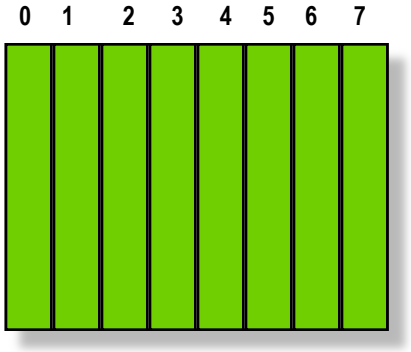
Fully-associative
block 12 can go anywhere

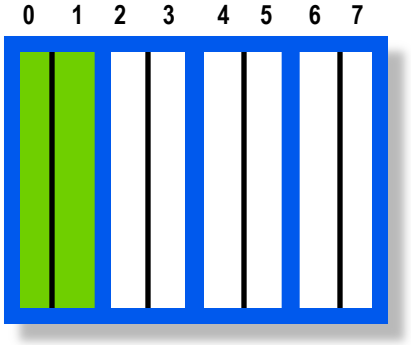
2-way Set-associative
block 12 can go anywhere in set 0
(12 mod 4)

Block Number

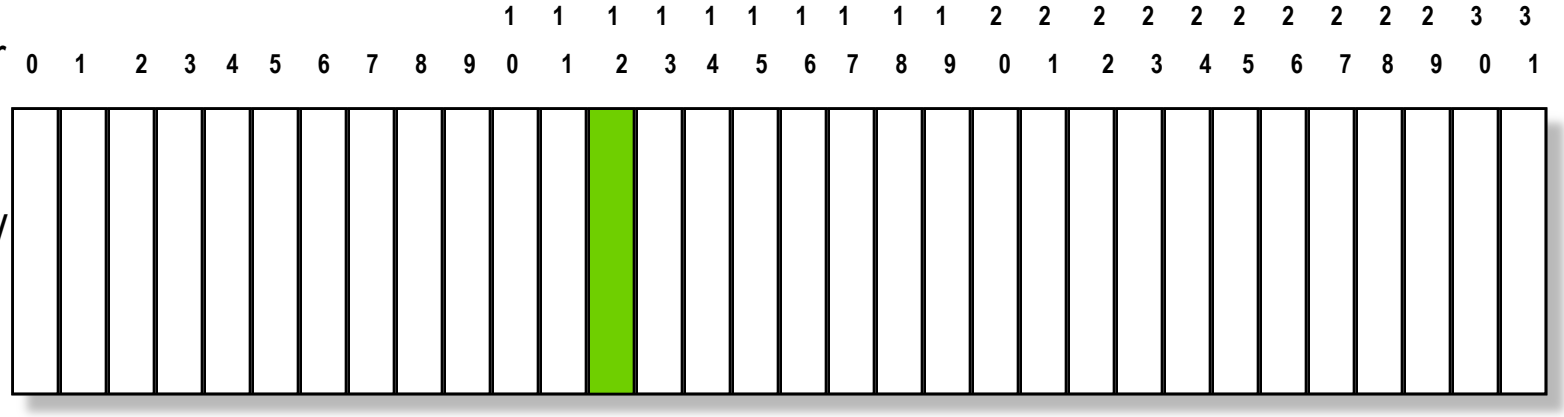
Cache







Block Number



Memory

Q2: Block Lookup

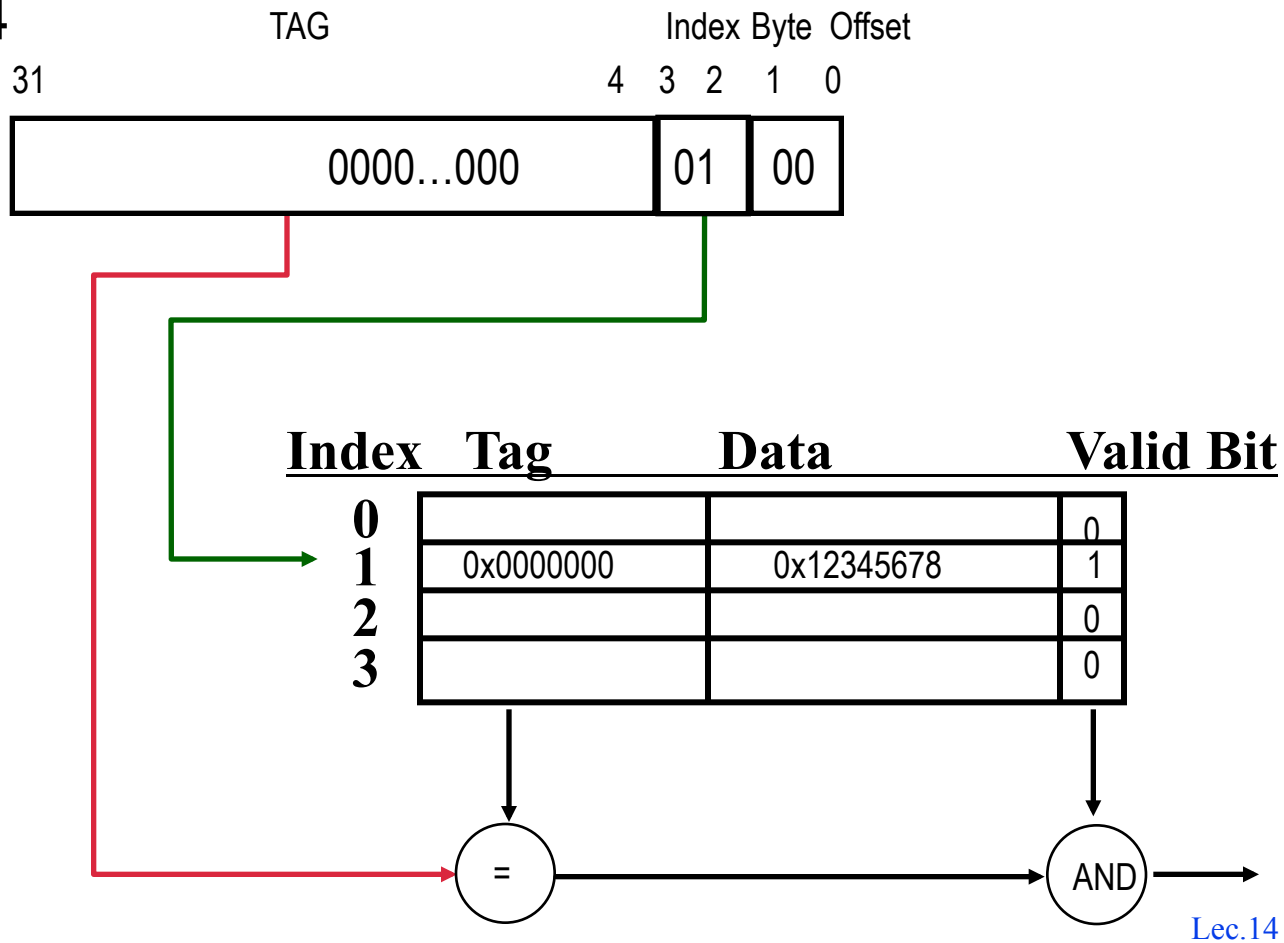
- Every block has an address tag that stores the main memory address of the data stored in the block.
- When checking the cache, the processor will compare the requested memory address to the cache tag -- if the two are equal, then there is a cache hit and the data is present in the cache
- Often, each cache block also has a valid bit that tells if the contents of the cache block are valid

Direct-mapped Cache Example

- Assume cache has 4 blocks and each block is 1 word (4 bytes)

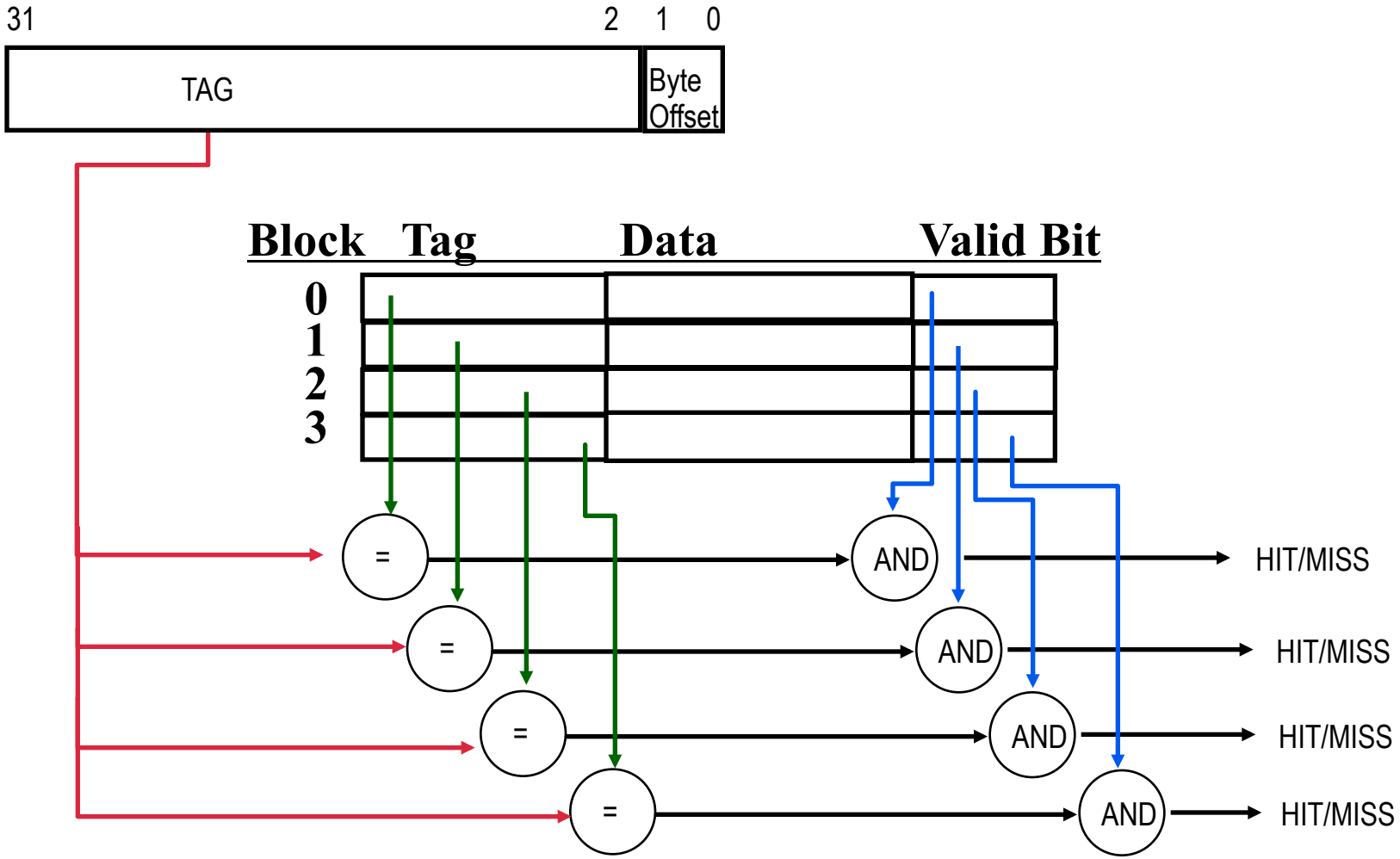
LOAD R1, 0x04

MEMORY Address	Data
0x00	0x00000000
0x04	0x12345678
0x08	0x87654321
0x0C	0x11111111
0x10	0x22222222
0x14	0x33333333
0x18	0x44444444
0x1C	0x55555555
0x20	0x10101010



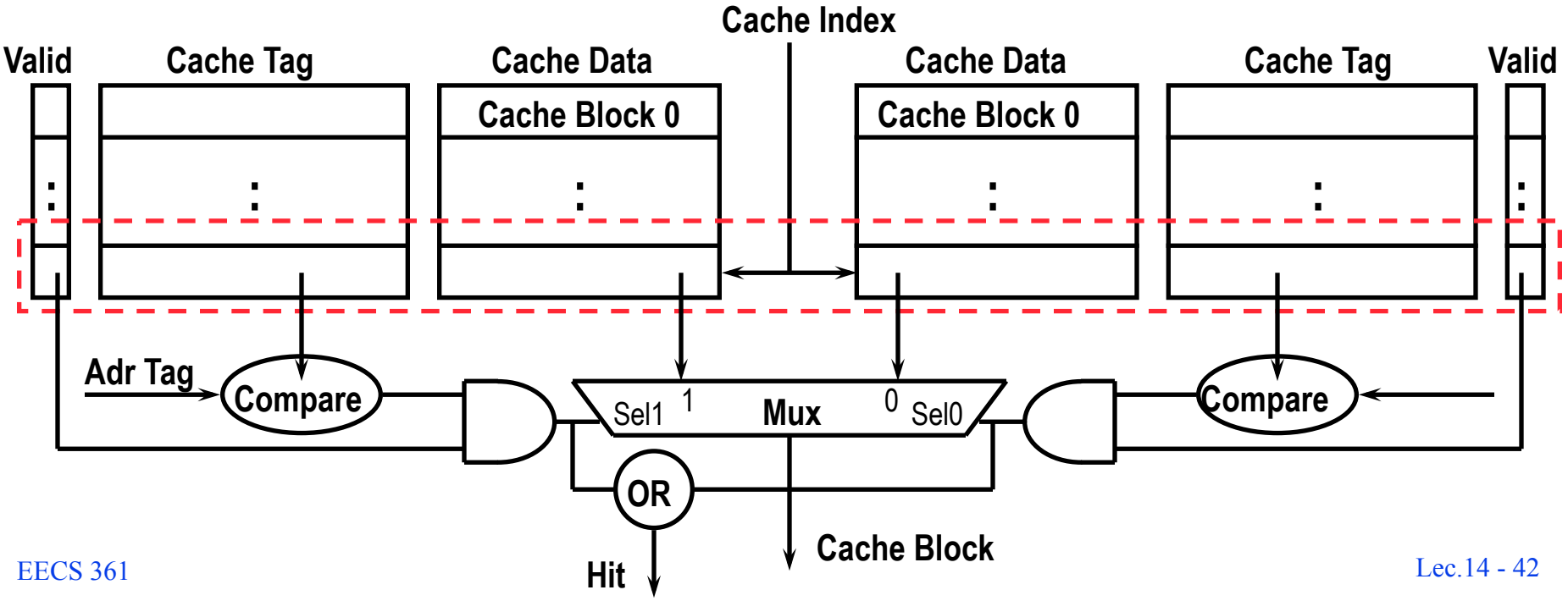
Fully-Associative Cache

- Assume cache has 4 blocks and each block is 1 word (4 bytes)



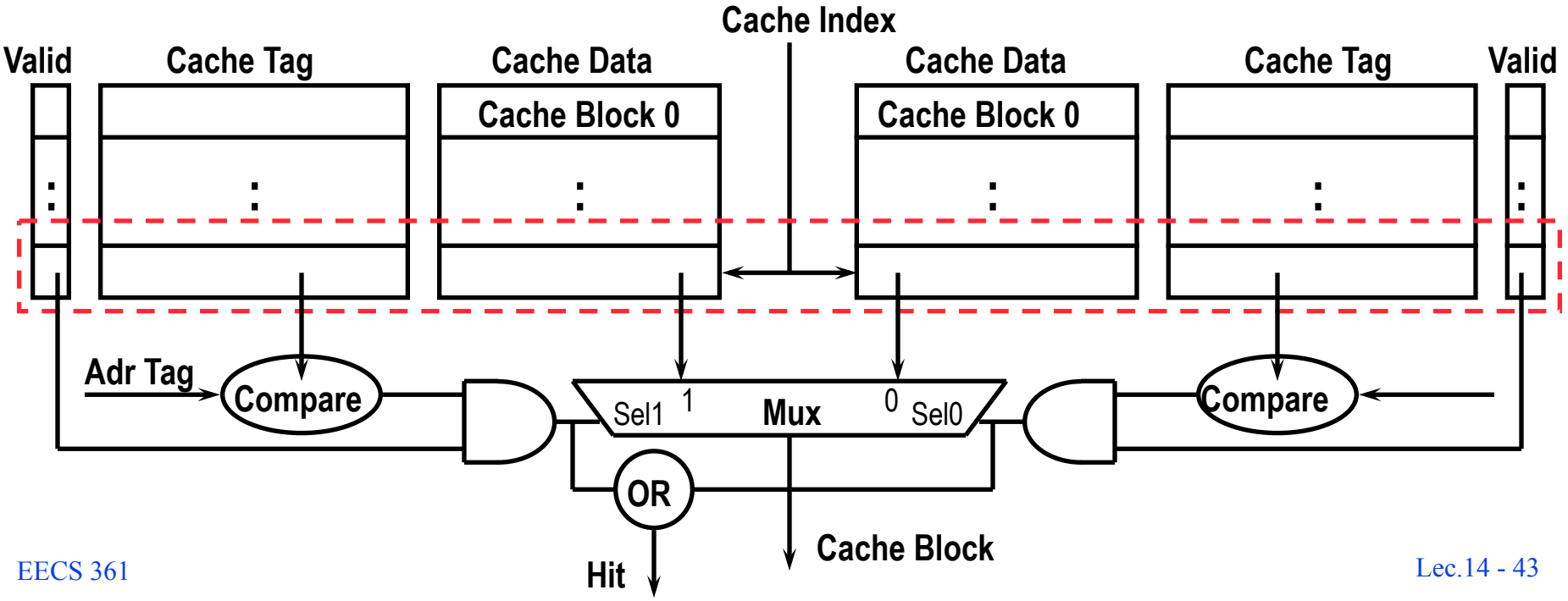
A Two-way Set Associative Cache

- **N-way set associative: N entries for each Cache Index**
  - N direct mapped caches operates in parallel
- **Example: Two-way set associative 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



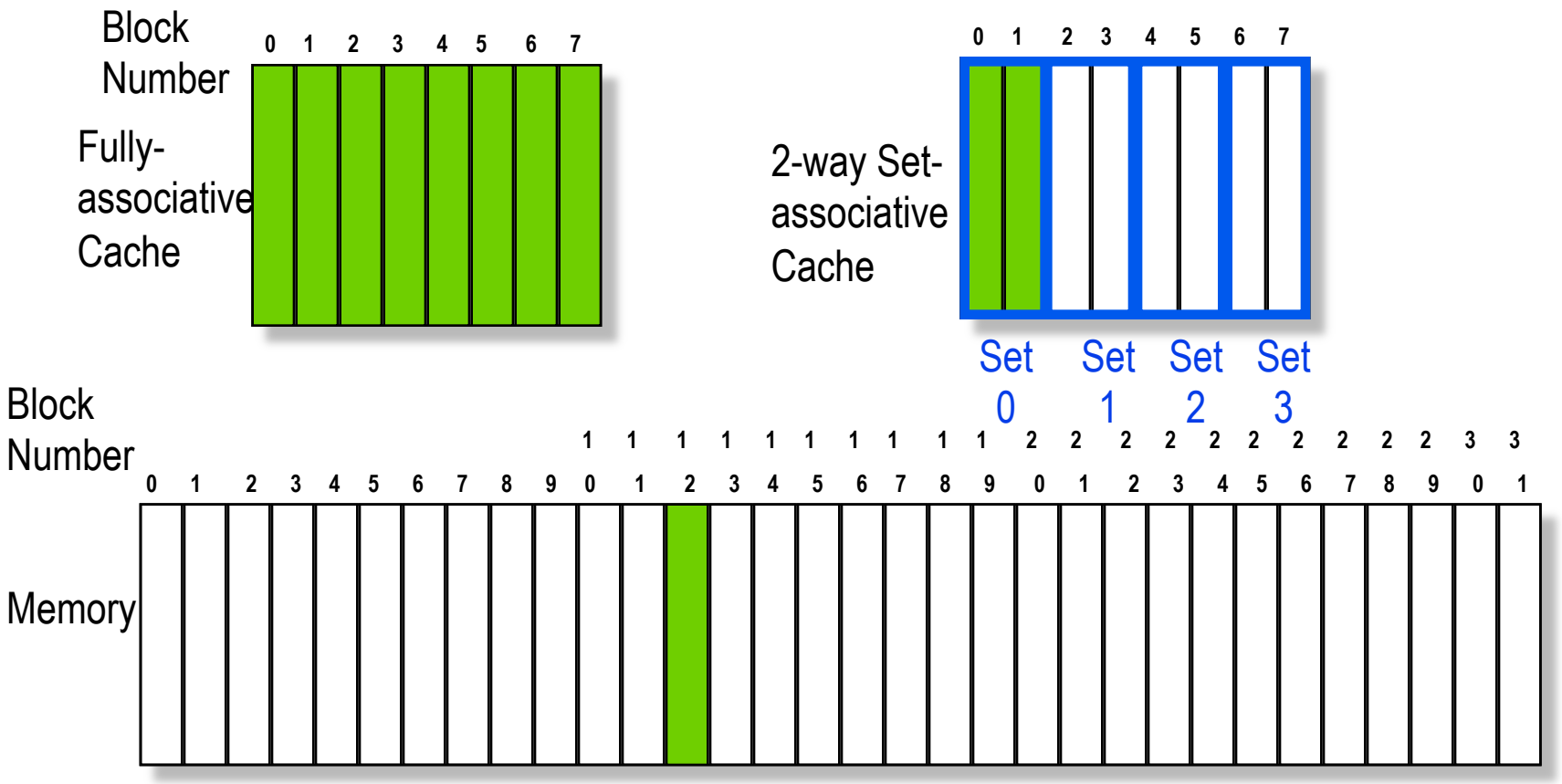
Disadvantage of Set Associative Cache

- 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: assume a hit and continue. Recover later if miss.



Q3: Block Replacement

- In a direct-mapped cache, there is only one block that can be replaced
- In set-associative and fully-associative caches, there are N blocks (where N is the degree of associativity



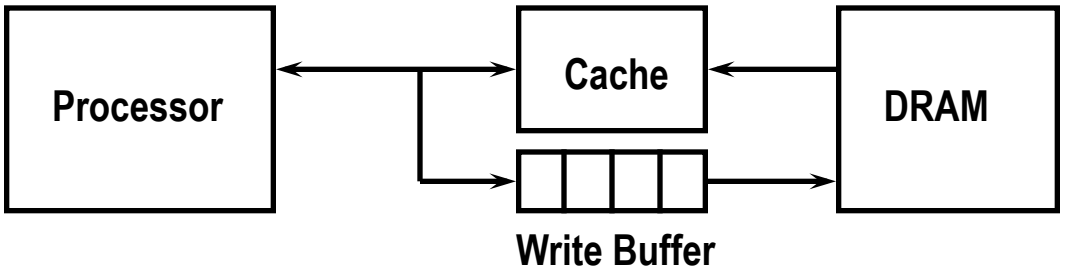
Q3: Block Replacement (cont.)

- **Several different replacement policies can be used**
  - Random replacement - randomly pick any block
    - Easy to implement in hardware, just requires a random number generator
    - Spreads allocation uniformly across cache
    - May evict a block that is about to be accessed
  - Least-recently used (LRU) - pick the block in the set which was least recently accessed
    - Assumed more recently accessed blocks more likely to be referenced again
    - Easy to implement for 2-way set-associative caches
      - Just have a single bit which gets set when the block is accessed, unset when the other block in the set is accessed
    - Much more difficult to implement for set-associativity greater than 2
      - Must resort to pseudo-LRU
        - One bit specifies most recently used block and replacement algorithm (randomly) picks from the remaining blocks in the set

Q4: Write Strategy

- **When data is written into the cache (on a store), is the data also written to main memory?**
  - If data is written to memory, the cache is called a ***write-through cache***
    - PRO: Easy hardware
    - PRO: Good speed on read misses
  - If data is NOT written to memory, the cache is called a ***write-back cache***
    - For a write-back cache, need to remember which blocks have been written ("dirty bit")
    - When you replace a "dirty" block, it must first be written back into the main memory
    - PRO: Good for "write-intensive" apps

Write Buffer for Write Through



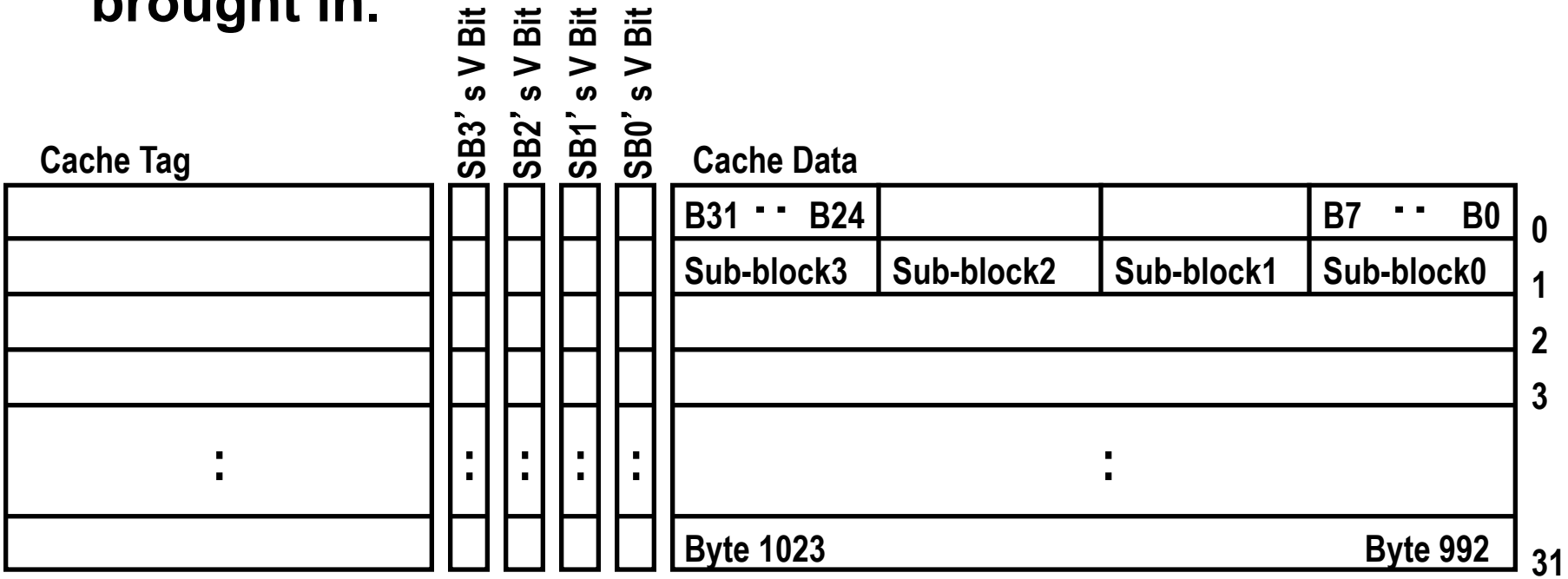
- **A Write Buffer is needed between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- **Memory system designer's nightmare:**
  - Store frequency (w.r.t. time) > 1 / DRAM write cycle
  - Write buffer saturation

Q4: Write Strategy (Write Allocation)

- **Should you cache a block if it is a write-miss?**
  - Maybe it will not be read again
  - Maybe it will replace a block that could be read again
- **Write allocate:**
  - On ANY miss (write or read) move the missed block into the cache
- **Write no-allocate:**
  - Only move missed block into the cache in case of read-miss

What is a Sub-block?

- **Sub-block:**
  - A unit within a block that has its own valid bit
  - Example: 1 KB Direct Mapped Cache, 32-B Block, 8-B Sub-block
    - Each cache entry will have: 32/8 = 4 valid bits
- **Write miss: only the bytes in that sub-block are brought in.**



Summary of Cache Questions

- **Q1: Where can a block be placed in the cache?**
  - Direct mapped, associative, fully-associative [Block Placement]
- **Q2: How is a block found if it is in the cache?**
  - indexing (direct mapped), limited search (associative), full search (fully-associative) [Block Lookup]
- **Q3: Which block should be replaced on a cache miss?**
  - random, least-recently used (LRU) [Block Replacement]
- **Q4: What happens on a write?**
  - write-through or write-back [Write Strategy (Write Allocation)]

A Summary on Sources of Cache Misses

- **Compulsory (cold start, first reference): first access to a block**
  - "Cold" fact of life: not a whole lot you can do about it
- **Conflict (collision):**
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- **Capacity:**
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size
- **Invalidation: other process (e.g., I/O) updates memory**

Metric for Cache Performance - Miss Rate (SPEC 92)

- **32 Byte block size, direct-mapped caches**
  - **Option 1**: 2 separate caches, one for instructions, one data; **a Split cache**
  - **Option 2:** a single cache with both instructions & data; **a Unified cache**
  - Each holds the same # of total bytes

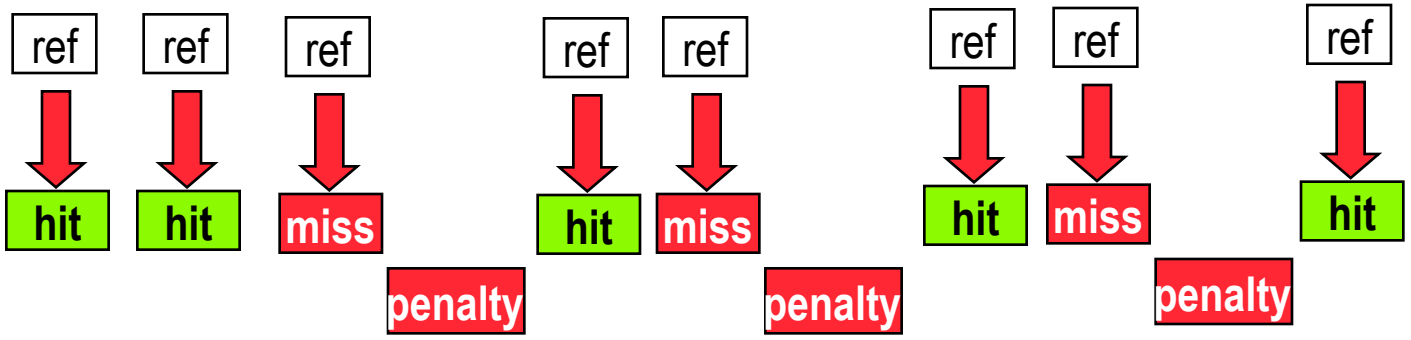
Cache Size	Instruction Cache	Data Cache	Unified Cache
1KB	3.06%	24.61%	13.34%
2KB	2.26%	20.57%	9.78%
4KB	1.78%	15.94%	7.24%
8KB	1.10%	10.19%	4.57%
16KB	0.64%	6.47%	2.87%
32KB	0.39%	4.82%	1.99%
64KB	0.15%	3.77%	1.35%
128KB	0.02%	2.88%	0.95%

- **Miss rate neglects cycle time implications**

Better Metric: Average Memory Access time

- **Average time to access memory (AMAT)**

Average memory access time **=** Hit time **+ (**Miss Rate **X** Miss Penalty**)**
where miss penalty **is the extra time it takes to handle a miss**
(i.e., the time beyond the required 1 cycle hit cost)



Example: **8** references, **5** hits, **3** misses
Average Mem Access Time =[(**5** X hit time) + (**3** X miss time) + (**3** X miss penalty)] / **8**
=[(**8** x hit time since miss=hit=1 cycle) + (**3** X miss penalty)] / **8**
= (hit time) + (**3 / 8**)(miss penalty)
= (hit time) + (**miss rate**)(miss penalty)

hit = **miss**
penalty >= **hit**

Average Memory Access Time Example

- **Recall:**
  Average memory access time = Hit time + (Miss Rate X Miss Penalty)

- **Example 1:**
  - Machine has 1 cycle hit cost, 10 cycle miss penalty (11 cycles total for a miss)
  - Program has 10% miss rate

  Average memory access time = 1.0 + 10% * 10 = 2.0

- **Example 2:**
  - Same machine as example 1
  - Program has 100 memory references: 90 hit, 10 miss

  90 hits * 1 cycle + 10 misses * 11 cycles = 200 cycles

  Average memory access time = # of cycle / # of refs = 200 cycles / 100 refs = 2.0

Example

- **Which system has a lower miss rate?**
  - 16KB instruction cache & 16 KB data cache

    vs.
  - 32KB unified instruction+data cache

  - Assume a hit takes **1 cycle**, a miss costs **50 cycles** and a load or store takes **1 extra clock cycle** on the unified cache
    - Why the extra cycle?
    - Because the unified cache only has one port to satisfy simultaneous requests

  - Assume that **75%** of the memory references are instruction fetches

Metric for Cache Performance - Miss Rate

- **SPEC 92 benchmark suite**
  - 32byte blocks, direct-mapped cache: use this table to get **miss rates** we need

Cache Size	Instruction Cache	Data Cache	Unified Cache
1KB	3.06%	24.61%	13.34%
2KB	2.26%	20.57%	9.78%
4KB	1.78%	15.94%	7.24%
8KB	1.10%	10.19%	4.57%
16KB	0.64%	6.47%	2.87%
32KB	0.39%	4.82%	1.99%
64KB	0.15%	3.77%	1.35%
128KB	0.02%	2.88%	0.95%

Example (cont.)

- **Solution**
  - Using the miss rates from the SPEC92 benchmark suite
    - For the split caches, the miss rate is:

      (75% * 0.64%) + (25% * 6.47%) = 2.10%

    - For the unified cache, the miss rate is (taken directly from the table) 1.99%
  - Given result, a designer optimizing miss rate would build a **32KB unified cache**
  - But, if we consider the cycle time penalty for a unified cache, we can compute the average memory access time:

    = % instruction * (instr hit time + instr miss rate * miss penalty) +
    %data * (data hit time + data miss rate * miss penalty)

    - For the split caches

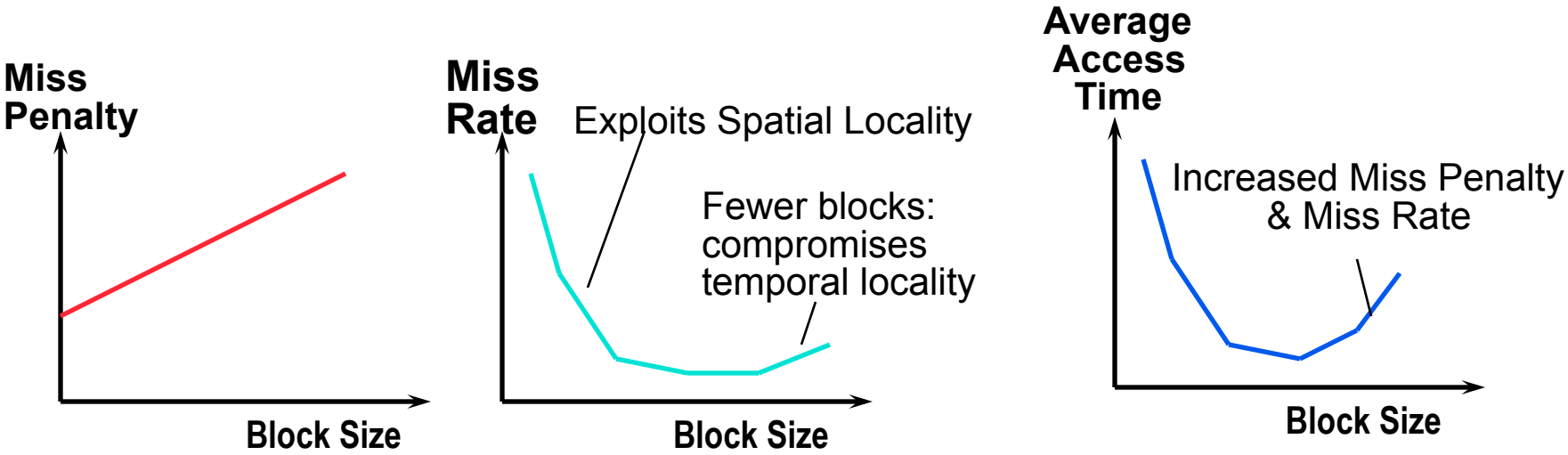
      = 75% * (1 + 0.64% * 50) + 25% * (1 + 6.47% * 50) = 2.05 cycles

    - For the unified cache

      = 75% * (1 + 1.99% * 50) + 25% * (1 + **1** + 1.99% * 50) = 2.24 cycles
  - Given this result, a designer optimizing the memory access time would build the **split cache system**

Block Size Tradeoff

- **In general, larger block size takes advantage of spatial locality BUT:**
  - Larger block size means larger miss penalty:
    - Takes longer time to fill up the block
  - If block size is too big relative to cache size, miss rate will go up
- **Average Access Time:**
  - = Hit Time + Miss Penalty x Miss Rate



Example

- **What's the impact on performance (CPU time) when the following cache behavior is included**
  - 50 cycle miss penalty
  - All instructions normally take 2.0 cycles (excluding memory stalls)
  - Miss rate is 2.0%
  - Average of 1.33 memory references per instruction

  Recall that:

CPU time = IC **X (** $CPI_{execution}$ **+** Memory stall cycles/Instruction**) X** Clock cycle time

Where: IC = Instruction Count

Example (cont.)

- **What's the impact on performance (CPU time) when the following cache behavior is included**
  - 50 cycle miss penalty
  - All instructions normally take 2.0 cycles (excluding memory stalls)
  - Miss rate is 2.0%
  - Average of 1.33 memory references per instruction

  Recall that:

IC X ($CPI_{execution}$ + Memory stall cycles/Instruction) X Clock cycle time

= IC X (2.0 + 0.02 X 1 .33 X 50) **X Clock cycle time**

= IC X 3.33 X Clock cycle time

Example (cont.)

IC X ($CPI_{execution}$ + Memory stall cycles/Instruction) X Clock cycle time

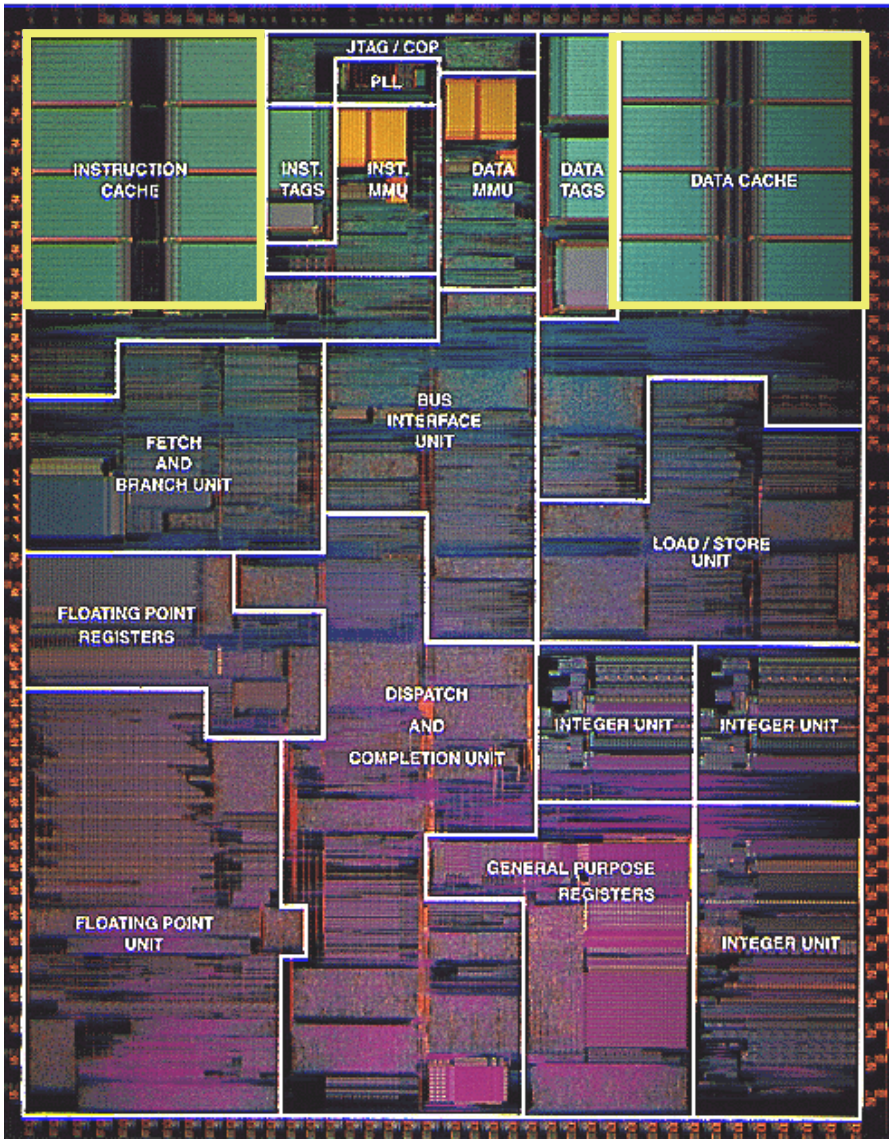
= IC X (2.0 + 0.02 X 1 .33 X 50) **X Clock cycle time**

= IC X 3.33 **X Clock cycle time**

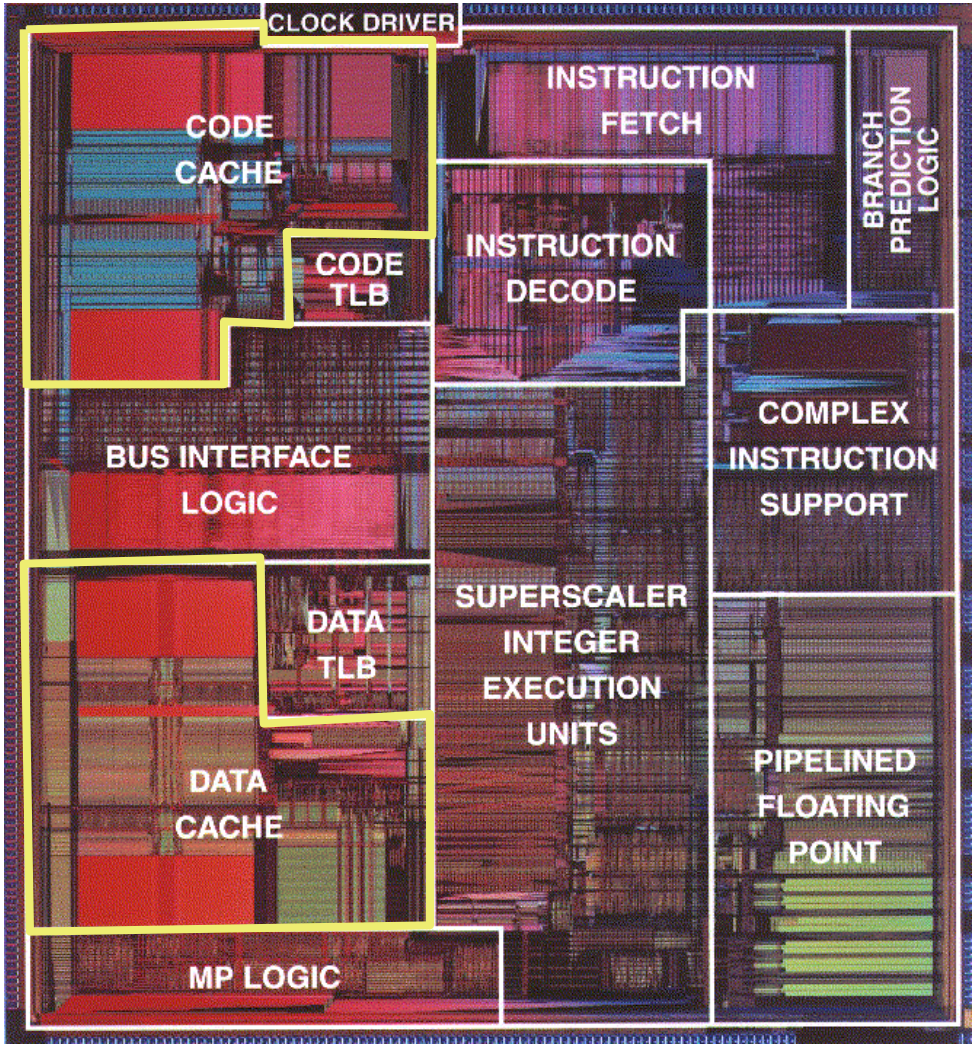
- **Two important results to keep in mind:**
  - The **lower** the $CPI_{execution}$, the **higher** the **relative impact** of a cache miss penalty
  - Comparing two machines with identical memory systems, the machine with the higher clock rate will have the larger number of clock cycles per miss and hence the memory portion of its CPI will be higher.

Real Life Examples: PowerPC

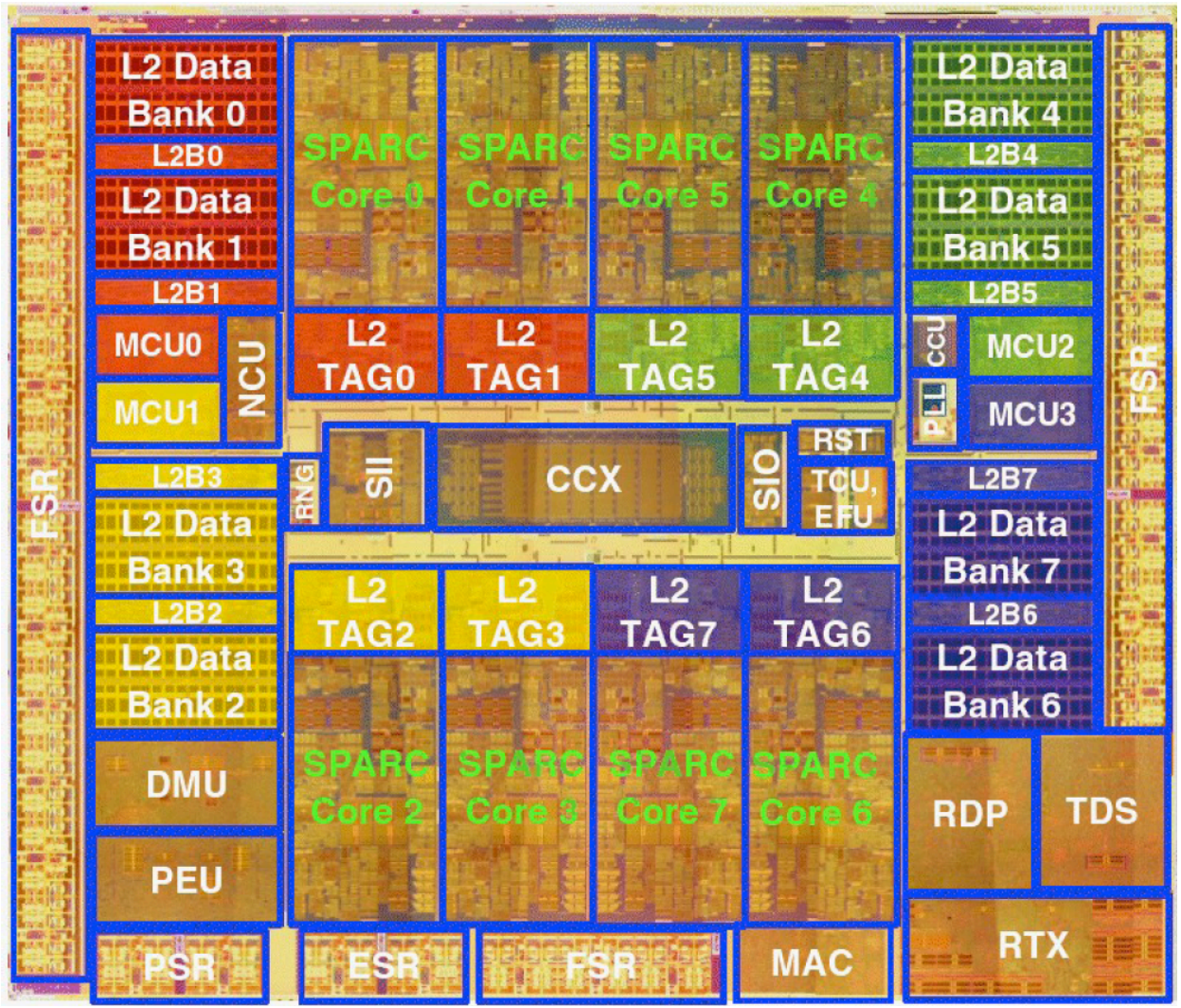
Motorola's PowerPC™ 604 RISC Microprocessor



Real Life Examples: Pentium Pro



Real Life Examples: Sun UltraSPARC T2 (Niagara-2)



Summary: Levels of the Memory Hierarchy