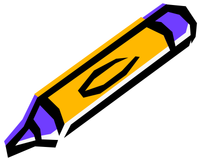
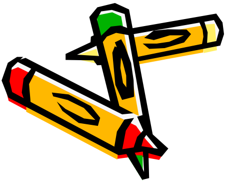
Computer Architecture

Hossein Asadi
Department of Computer Engineering
Sharif University of Technology
asadi@sharif.edu



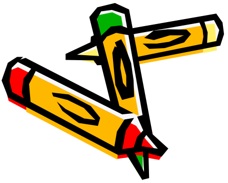
Today's Topics

- Memory & Memory Organization
- Memory Hierarchy
- Principle of Locality
- Cache Memory
  - Directed-mapped
  - Set-associative
  - Fully-associative
  - Cache configuration



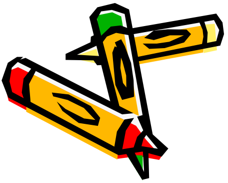
Copyright Notice

- Parts (text & figures) of this lecture adopted from:
  - Computer Organization & Design, The Hardware/Software Interface, 3rd Edition, by D. Patterson and J. Hennessey, MK publishing, 2005.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, CMU, Spring 2009.
  - "Computer Architecture & Engineering" handouts, by Prof. Kubiatowicz, UC Berkeley, Spring 2004.
  - "Intro to Computer Architecture" handouts, by Prof. Hoe, UWisc, Spring 2021.
  - "Computer Arch I" handouts, by Prof. Garzarán, UIUC, Spring 2009.
  - "Intro to Computer Organization" handouts, by Prof. Mahlke & Prof. Narayanasamy, Winter 2008.



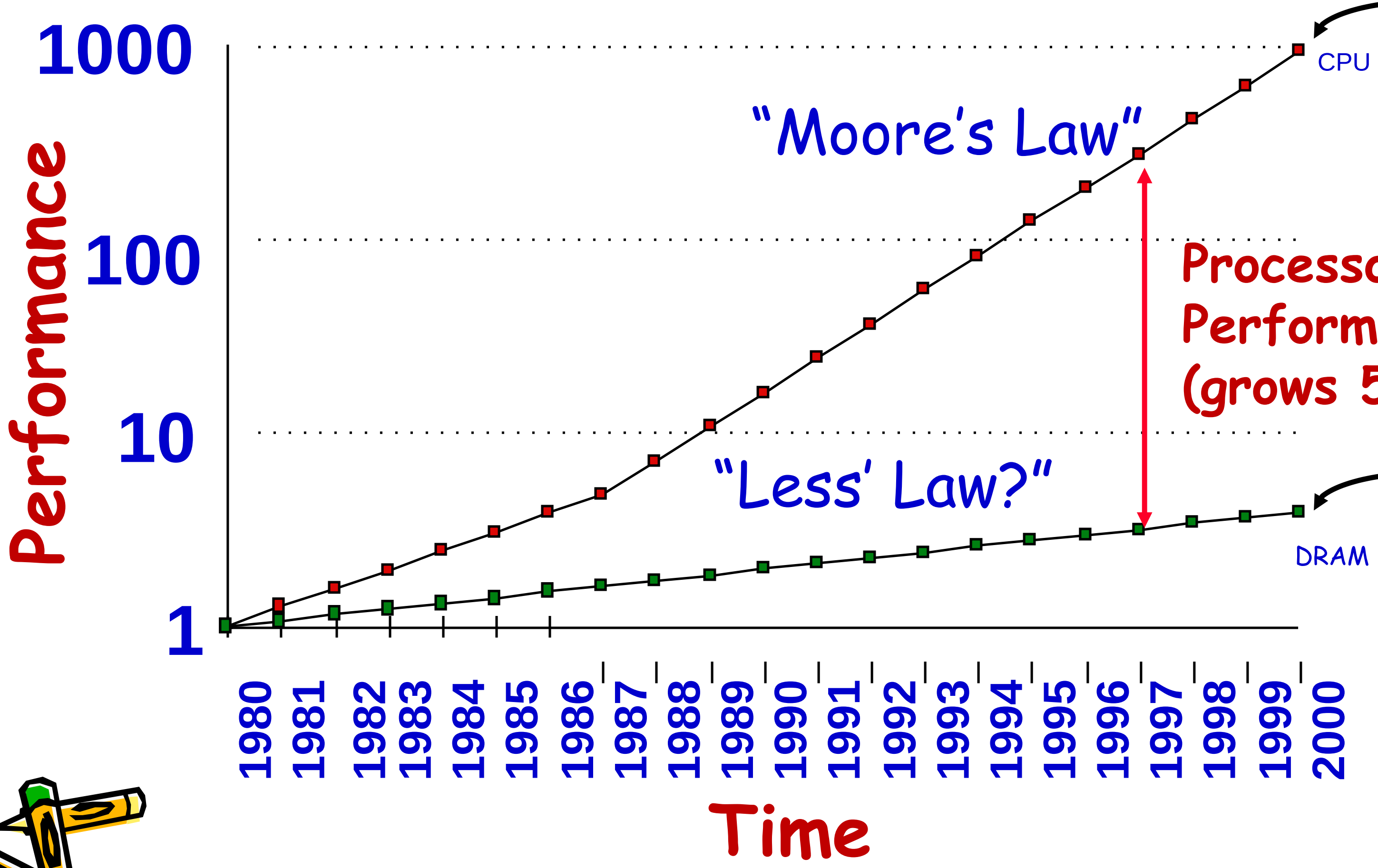
Ideal Memory

- Processors
  - Would run one instruction per cycle if:
    - Every memory access takes one cycle
    - Every request to memory is successful
- Our Ideal Memory?
  - Very large
  - Can be accessed in one clock cycle
- Reality
  - Any GB-size memory running at Ghz?



Why Memory a Big Deal?

Processor-DRAM Memory Gap (latency)



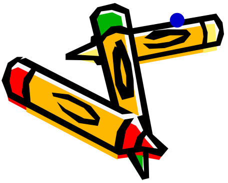
µProc
60% per yr.
(2X/1.5yr)

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

DRAM 9%
per yr.
(2X/10 yrs)

The Law of Storage

- Bigger is Slower
  - FFs, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Faster is More Expensive ($ and chip area)
  - SRAM < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
- *Note* these sample values scale with time



Question is:

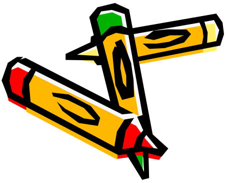
- How to Make Memory?
  - Bigger,
  - Faster, &
  - Cheaper?

Principle of Locality

- Locality
  - One's recent past is a very good predictor of his/her near future
- Temporal Locality:
  - If you just did something, it is very likely that you will do same thing again soon
- Spatial Locality:
  - If you just did something there, it is very likely you will do something similar/related around again

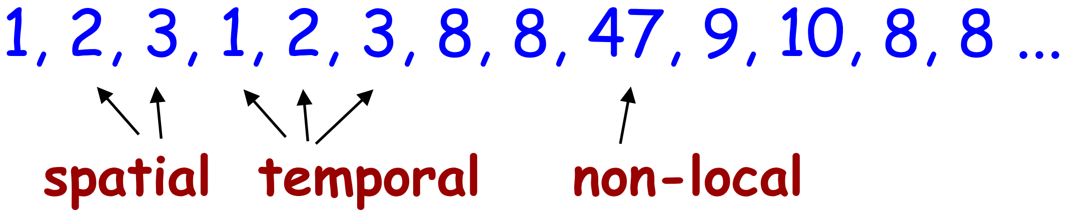
Locality in Memory

- Locality in Memory
  - A “typical” program has a lot of locality in memory references
  - Programs are sequential and composed of “loops”
- Temporal:
  - A program tends to reference same memory location many times and all within a small window of time
- Spatial:
  - A program tends to reference a cluster of memory locations at a time

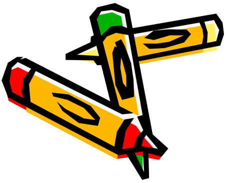


Locality in Memory (cont.)

- Example 1:
  - This sequence of addresses has both types of locality



1, 2, 3, 1, 2, 3, 8, 8, 47, 9, 10, 8, 8 ...

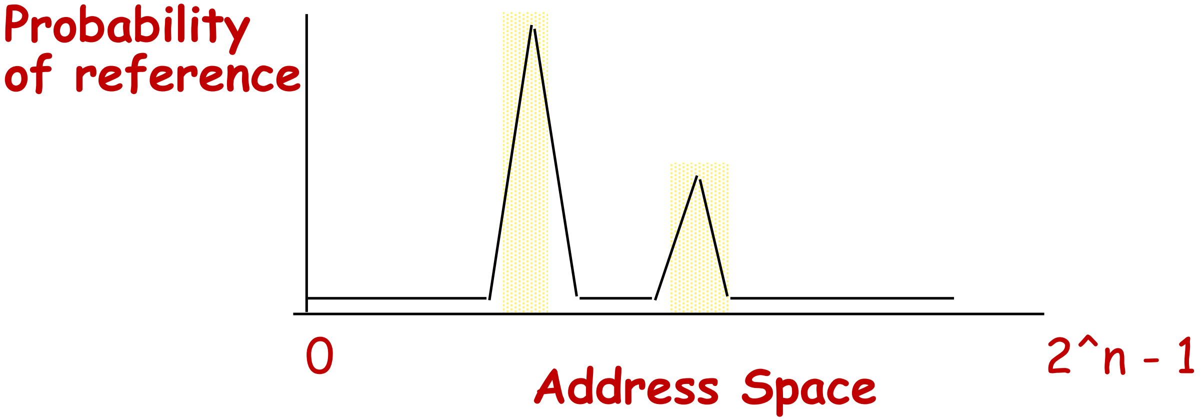


Locality in Memory (cont.)

- Example 2:
  - Data
    - Reference array elements in succession (spatial)
  - Instructions
    - Reference instructions in sequence (spatial)
    - Cycle through loop repeatedly (temporal)

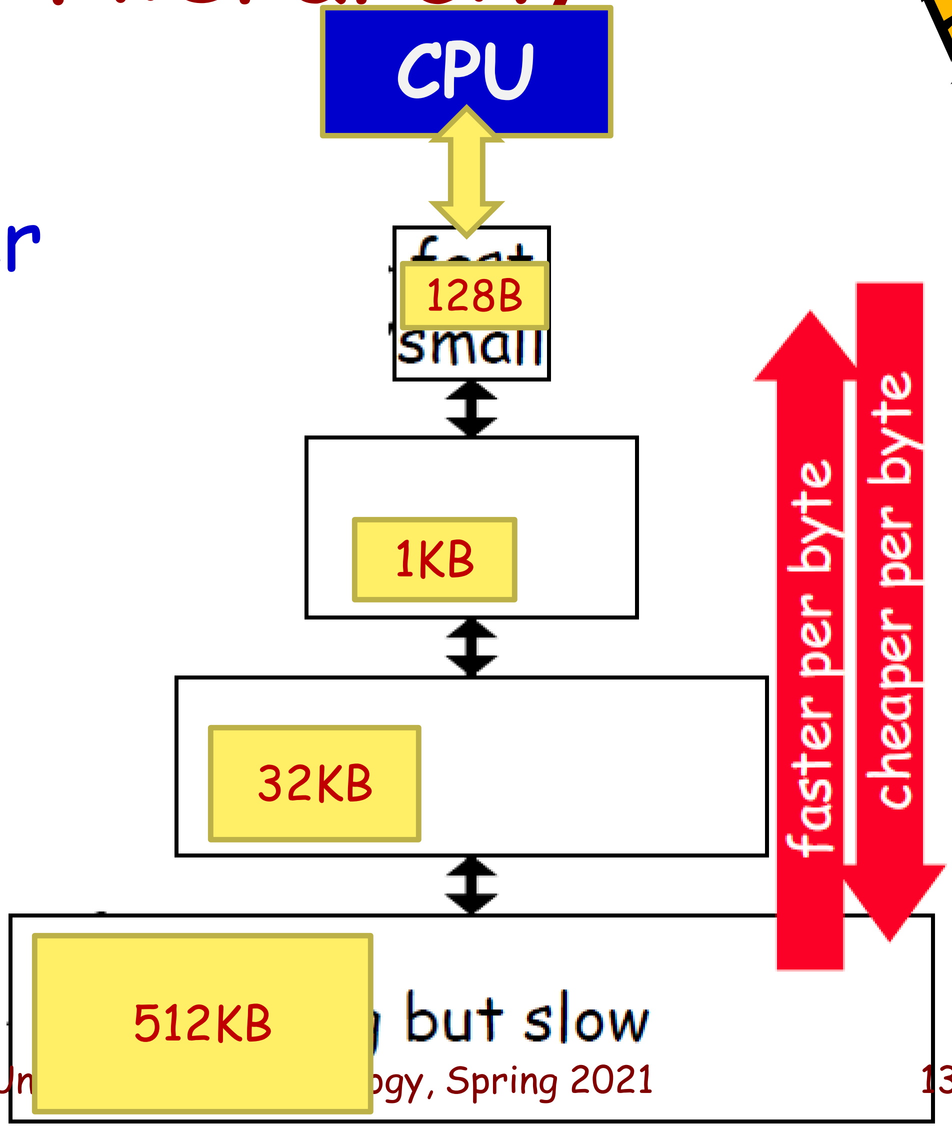
```
sum = 0;
for (i = 0; i < n; i++)
   sum += a[i];
*v = sum;
```

Probability of Reference

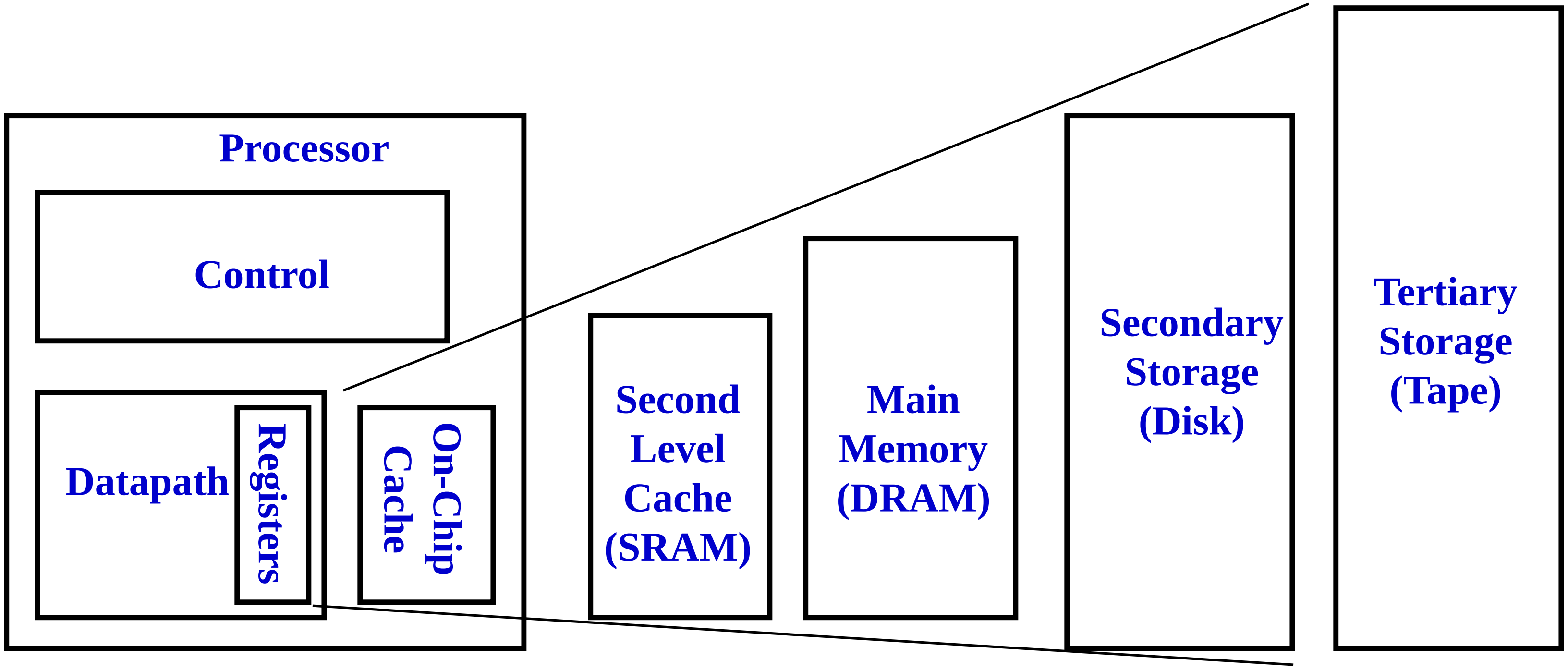


Memory Hierarchy

- Faster & Small
- Bigger & Slower



Memory Hierarchy (cont.)


Processor

Control

Datapath

Registers

On-Chip Cache

Second Level Cache (SRAM)

Main Memory (DRAM)

Secondary Storage (Disk)

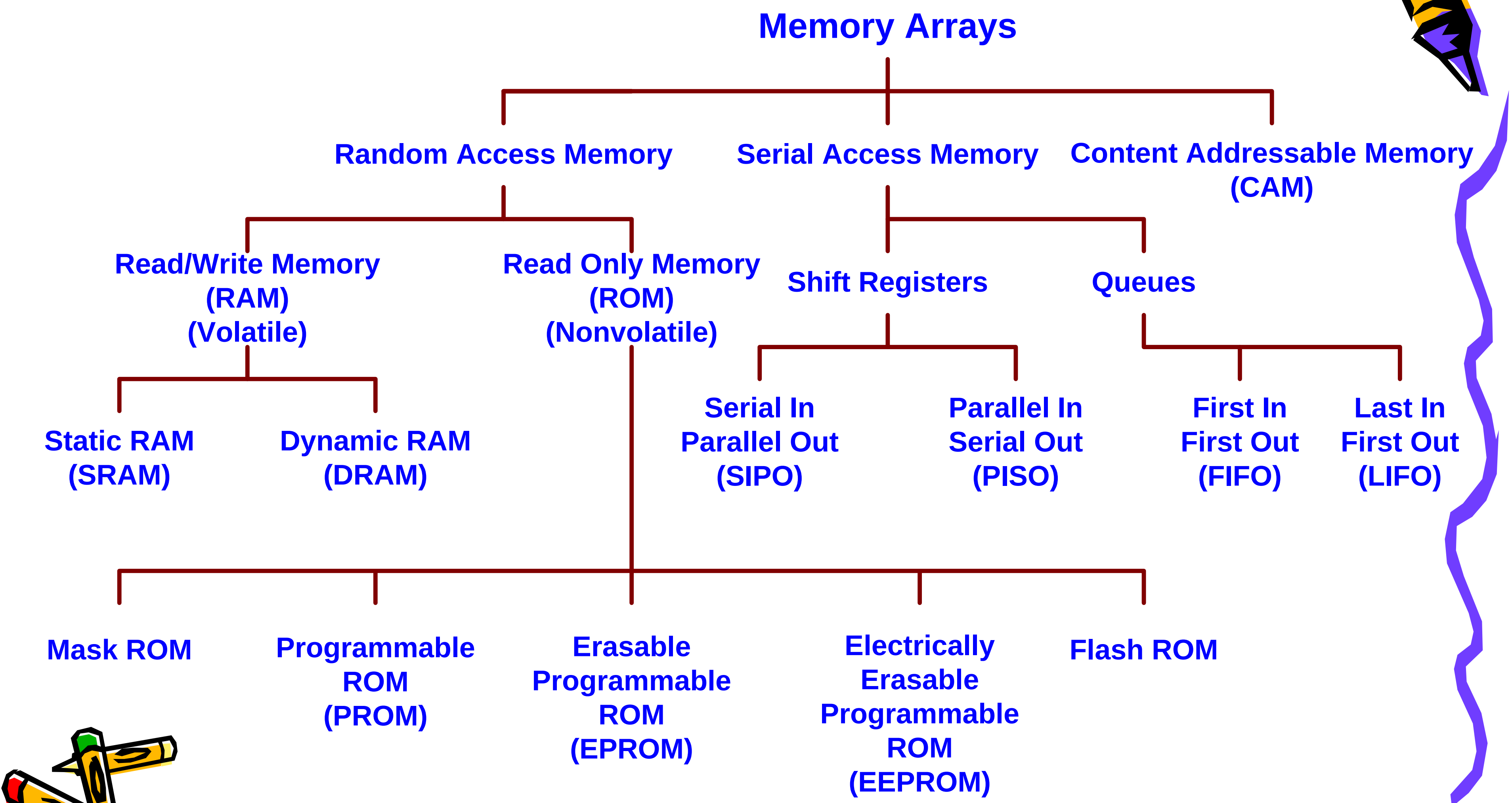
Tertiary Storage (Tape)

Speed (ns):	**1s**	**10s**	**100s**	**10,000,000s (10s ms)**	**10,000,000,000s (10s sec)**
Size (bytes):	**100s**	**Ks**	**Ms**	**Gs**	**Ts**

Technology Used in Main Memory & Cache Memory

- SRAM (Static Random Access Memory)
  - No refresh (6 transistors/bit vs. 1 transistor)
  - # of transistors per bit: DRAM < SRAM
  - Cycle time: DRAM > SRAM
- DRAM (Dynamic Random Access Memory)
  - Dynamic since needs to be refreshed periodically
  - Addresses divided into 2 halves
    - Memory as a 2D matrix
    - RAS or Row Address Strobe
    - CAS or Column Address Strobe

Memory Arrays



Memory Arrays

Random Access Memory

Serial Access Memory

Content Addressable Memory (CAM)

Read/Write Memory (RAM) (Volatile)

Read Only Memory (ROM) (Nonvolatile)

Shift Registers

Queues

Static RAM (SRAM)

Dynamic RAM (DRAM)

Serial In Parallel Out (SIPO)

Parallel In Serial Out (PISO)

First In First Out (FIFO)

Last In First Out (LIFO)

Mask ROM

Programmable ROM (PROM)

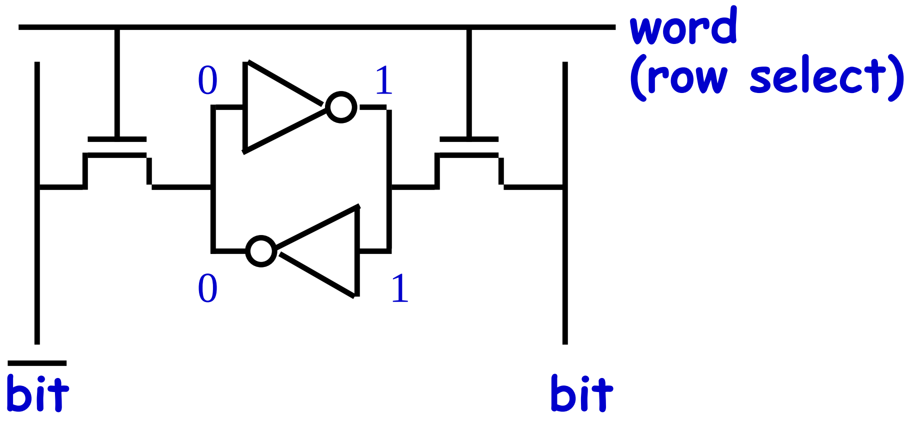
Erasable Programmable ROM (EPROM)

Electrically Erasable Programmable ROM (EEPROM)

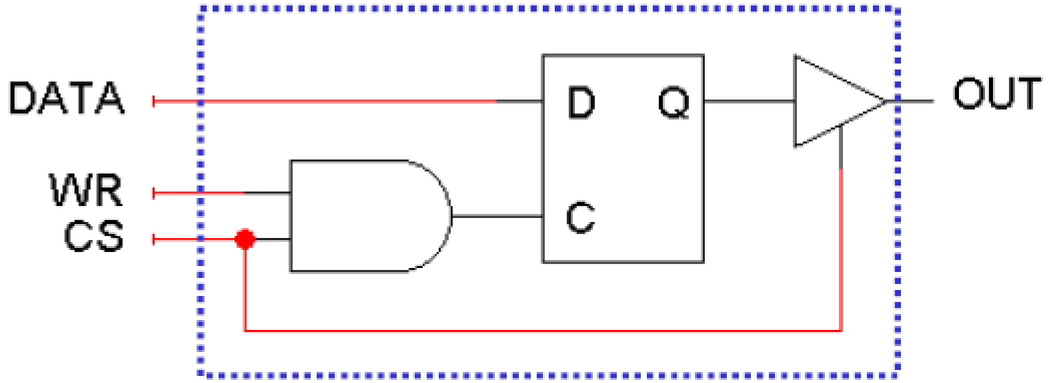
Flash ROM

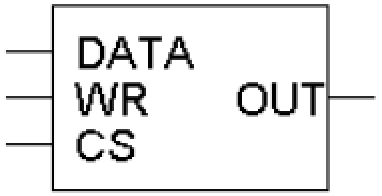
Static RAM Cell

6-Transistor SRAM Cell



How to Build Big Memory?

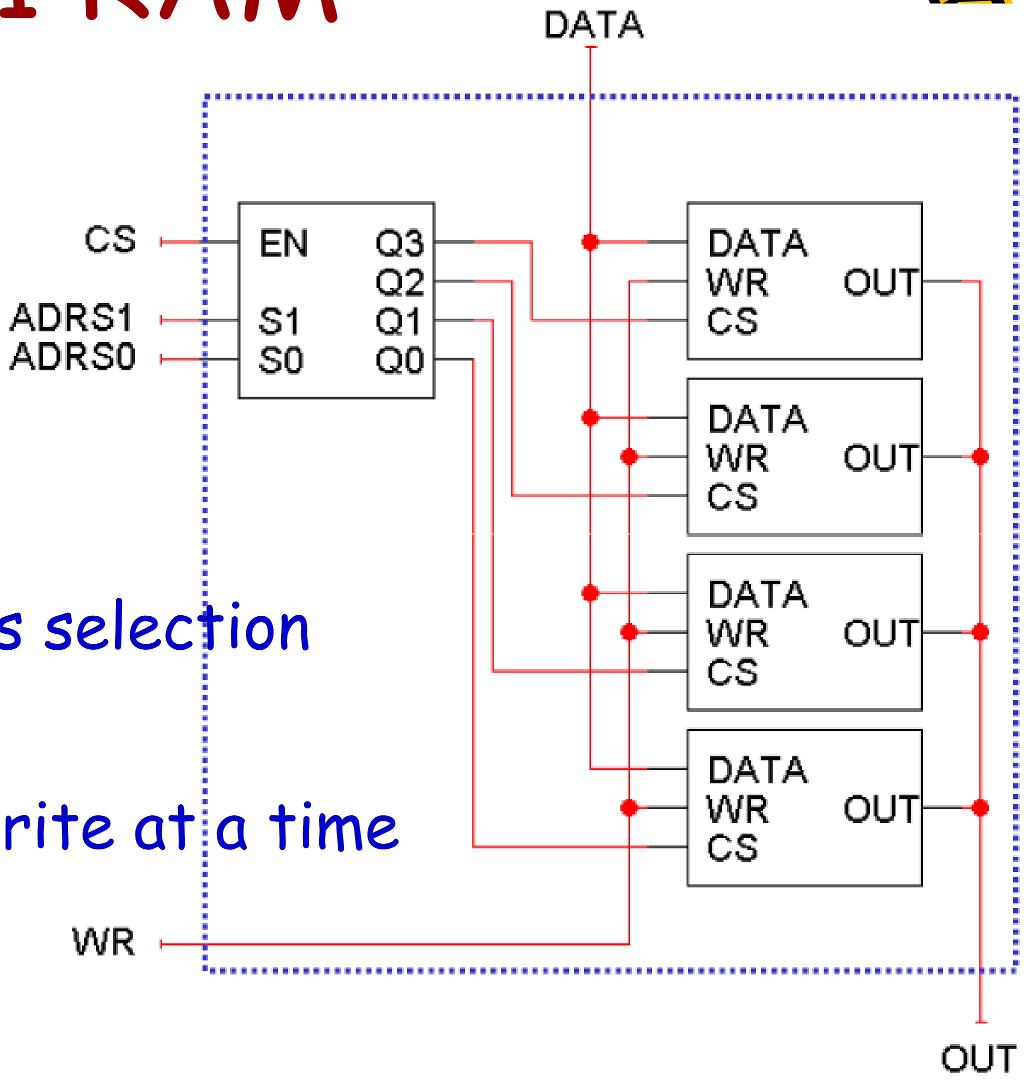




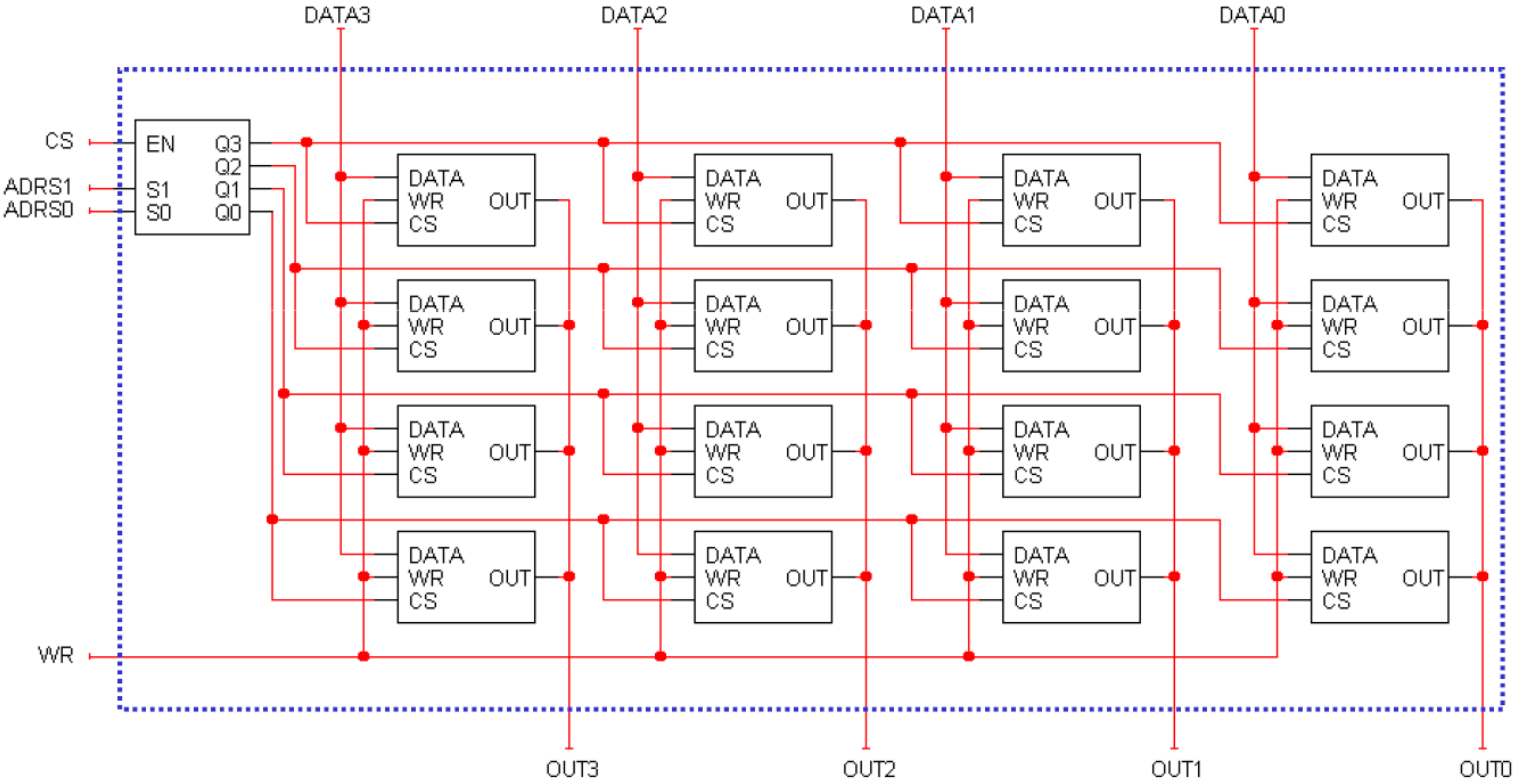
- CS / OE
  - Chip select or Output Enable
- WR
  - Write enable
- Address not required (one-bit)

4x1 RAM

- Address Lines
  - Two bits
- Data Line
  - One bit
- Decoder
  - Used for address selection
- Only One Bit
  - Can be read or write at a time

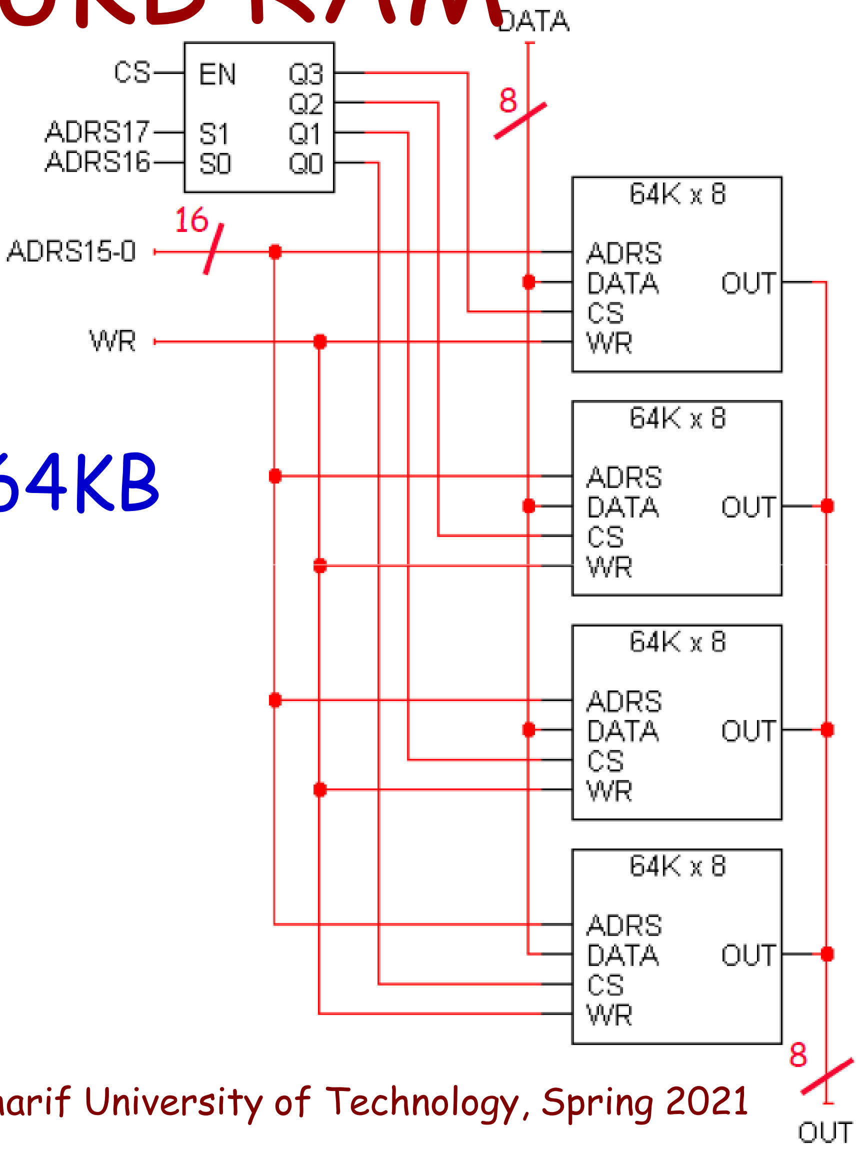


4x4 RAM

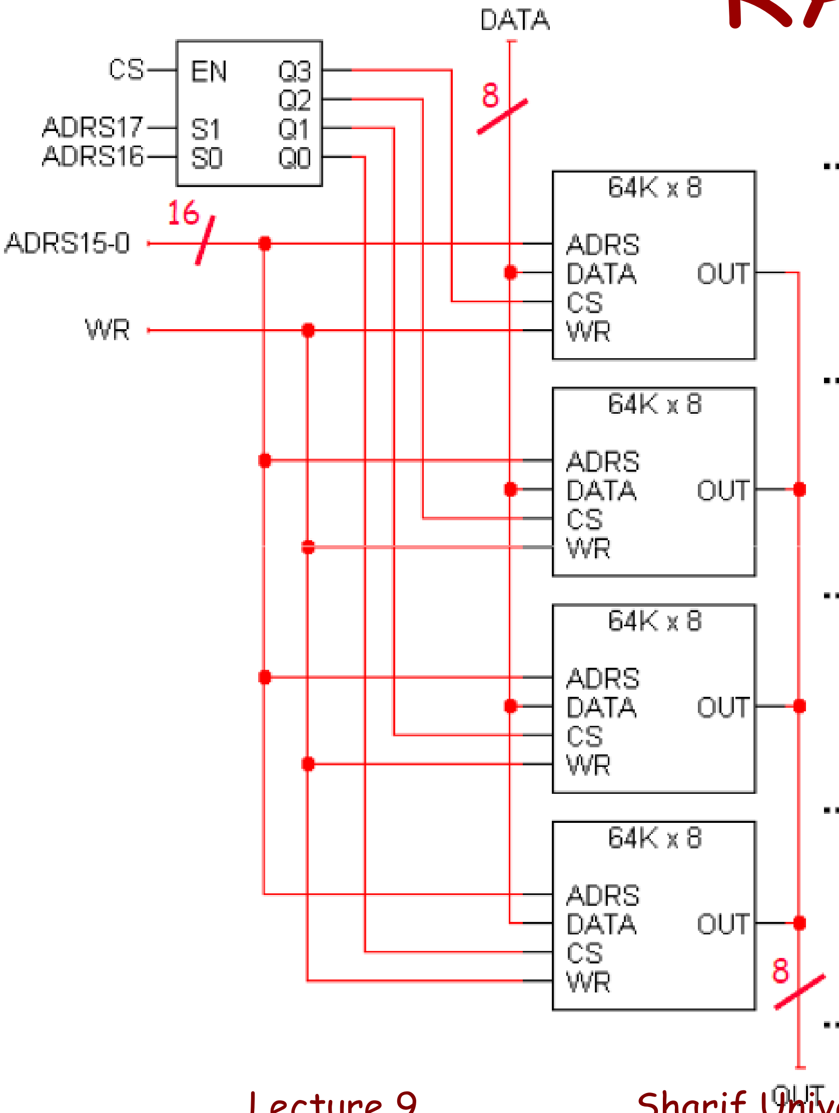

DATA3
DATA2
DATA1
DATA0
CS
ADRS1
ADRS0
EN
S1
S0
Q3
Q2
Q1
Q0
DATA
WR
CS
OUT
WR
OUT3
OUT2
OUT1
OUT0

256KB RAM

- Made of Four 64KB RAM chips



Address Ranges in 256KB RAM



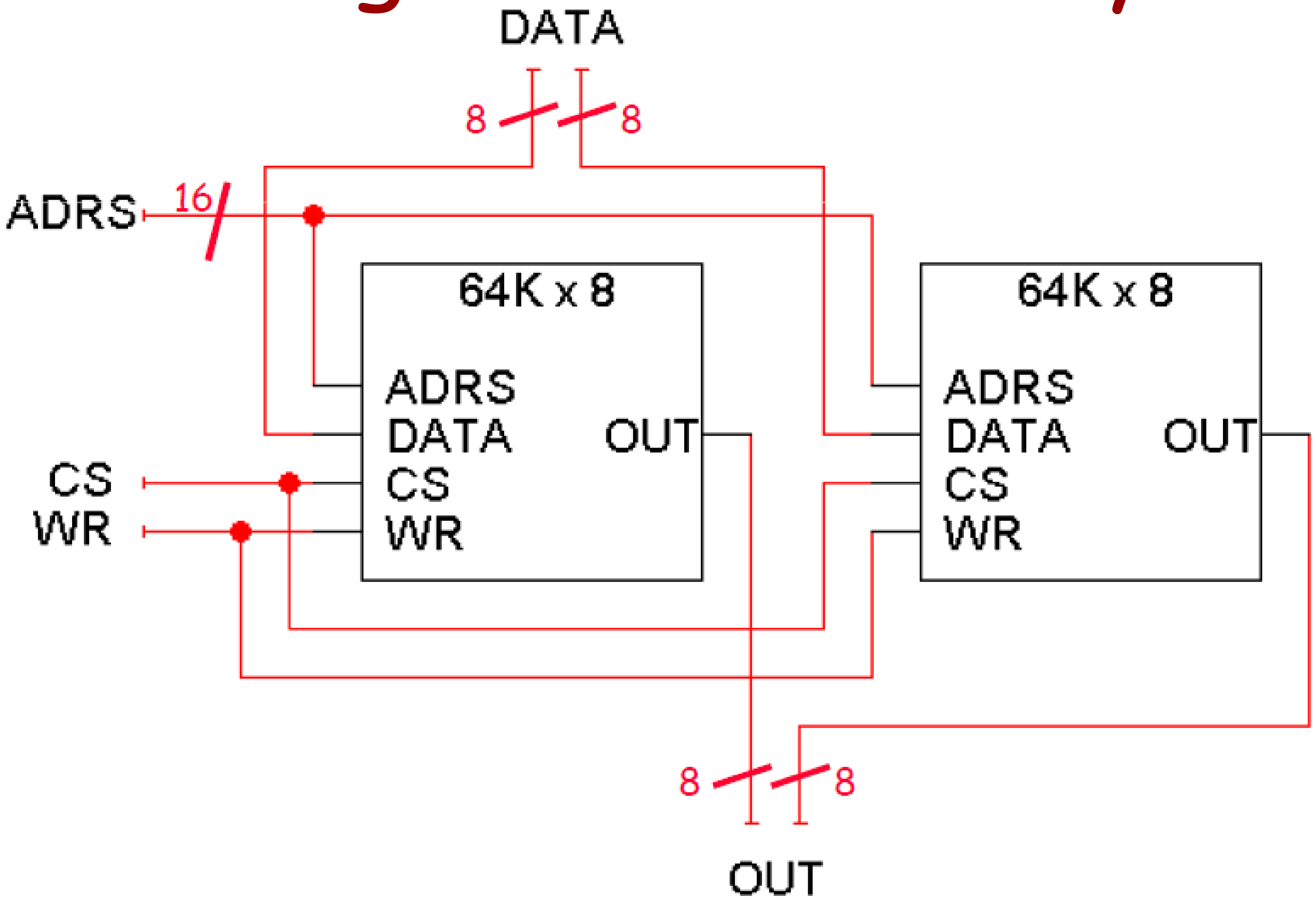
11 1111 1111 1111 1111 (0x3ffff)
to
11 0000 0000 0000 0000 (0x30000)

10 1111 1111 1111 1111 (0x2ffff)
to
10 0000 0000 0000 0000 (0x20000)

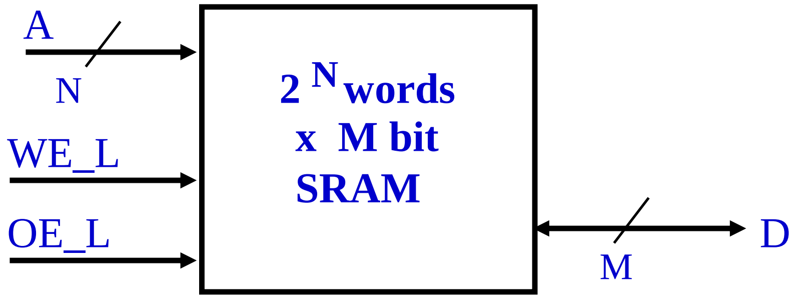
01 1111 1111 1111 1111 (0x1ffff)
to
01 0000 0000 0000 0000 (0x10000)

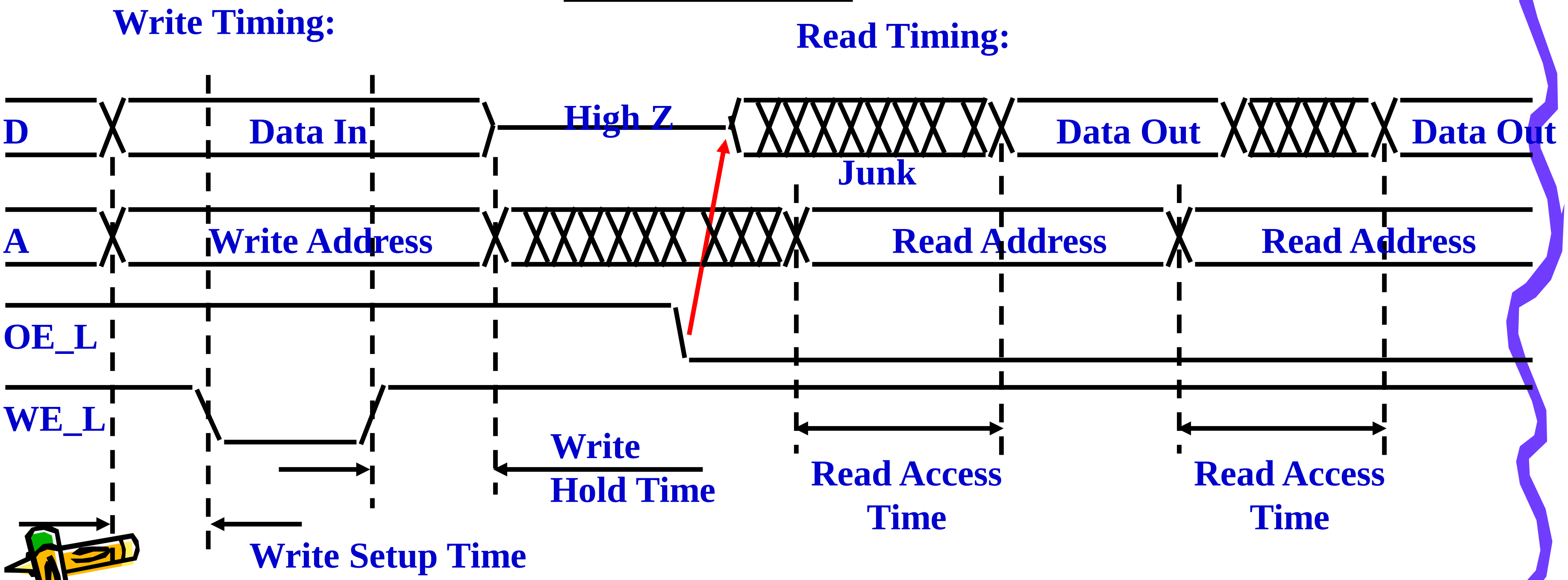
00 1111 1111 1111 1111 (0x0ffff)
to
00 0000 0000 0000 0000 (0x00000)

Making Wider Memory



Typical SRAM Timing





Slow Memory in Pipeline Datapath

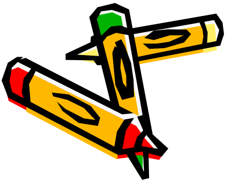
- Freeze pipeline in *Mem* stage:

IF0	ID0	EX0	Mem0	Wr0	Noop	…	Noop	Noop				
	IF1	ID1	EX1	Mem1	stall	…	stall	Mem1	Wr1			
		IF2	ID2	EX2	stall	…	stall	Ex2	Mem2	Wr2		
			IF3	ID3	stall	…	stall	ID3	Ex3	Mem3	Wr3	
				IF4	stall	…	stall	IF4	ID4	Ex4	Mem4	Wr4
									IF5	ID5	Ex5	Mem5

- Stall detected by end of Mem1 stage

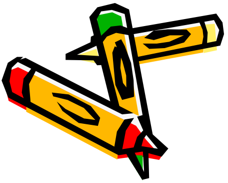
CPU Performance

- CPU Time = (CPU execution clock cycles + memory stall clock cycles) x clock cycle time
- Memory Stall Clock Cycles = (reads x read miss rate x read miss penalty + writes x write miss rate x write miss penalty)
- Memory Stall Clock Cycles = Memory accesses x Miss rate x Miss penalty



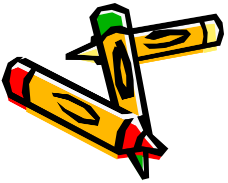
CPU Performance (cont.)

- Different Measure:
  - Average Memory Access Time (AMAT)
- Expressed in Terms of:
  - Hit time
  - Miss rate
  - Miss penalty



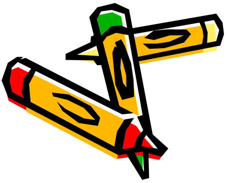
CPU Performance (cont.)

- Hit Time
  - Time required to access a level of memory hierarchy including time required to determine whether access is hit or miss
- Hit Rate (Hit Ratio)
  - Fraction of memory accesses found in a cache
  - Miss Rate = 1 – Hit Rate



CPU Performance (cont.)

- Miss Penalty:
  - Time required to fetch a block into a level of memory hierarchy from lower level:
    - Time to access block +
    - Time to transmit it to higher level +
    - Time to insert it in appropriate block
- Block
  - Minimum unit of information transferred between two levels of memory hierarchy
  - Also called line



CPU Performance (cont.)

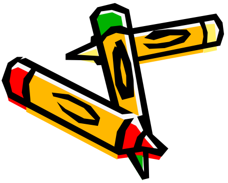
- AMAT = Hit Time + (Miss Rate x Miss Penalty)

CPU Performance (cont.)

- Example 1
  - A memory system consists of a cache and a main memory
  - Cache hit = 1 cycle
  - Cache miss = 100 cycles
  - What is average memory access time if hit rate in cache is 97%?

CPU Performance (cont.)

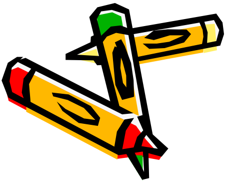
- Example 2
  - A memory system has a cache, a main memory, and a virtual memory
  - Hit rate = 98%
  - Hit rate in main memory = 99%
  - 2 cycles to access cache
  - 150 cycles to fetch a line from main mem.
  - 100,000 cycles to access virtual memory
  - What is average memory access time?



Improving Cache Performance

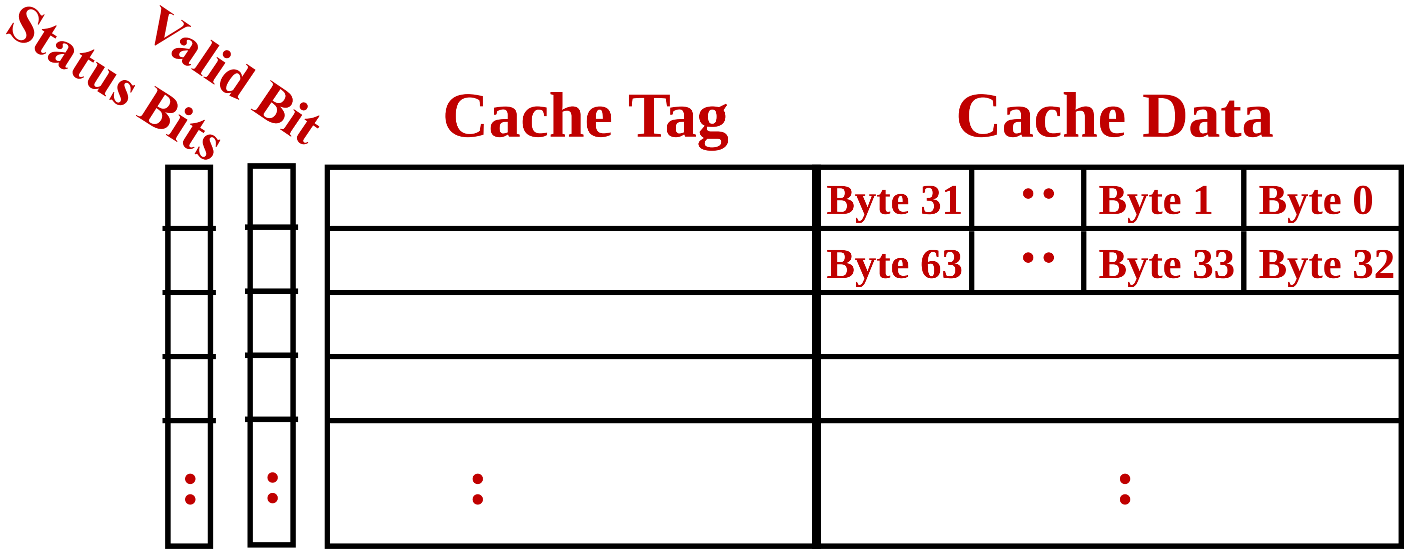
- AMAT =

  Hit Time + (Miss Rate x Miss Penalty)
- Options to Reduce AMAT
  - Reduce time to hit in cache
    - Use smaller cache size
  - Reduce miss rate
    - Increase cache size!
  - Reduce miss penalty
    - Use multi-level cache hierarchy



Cache Structure

- Data Bits
- Tag Bits
- Invalid Bit
- Status Bits (LRU bit, Dirty Bit, …)



Status Bits

Valid Bit

Cache Tag

Cache Data

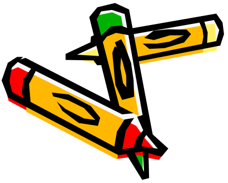
Byte 31 .. Byte 1 Byte 0

Byte 63 .. Byte 33 Byte 32

:

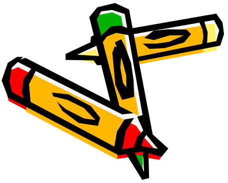
Cache Configuration

- Q1:
  - Where can a block be placed in upper level?
  - Block placement
- Q2:
  - How is a block found if it is in upper level?
  - Block identification



Cache Configuration (cont.)

- Q3:
  - Which block should be replaced on a miss?
  - Block replacement
- Q4:
  - What happens on a write?
    - How to propagate changes?
  - Write strategy

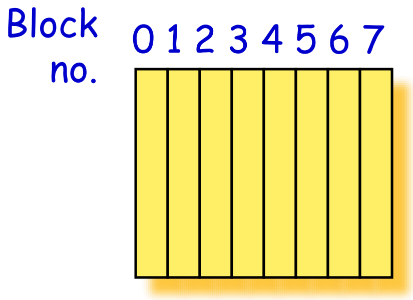


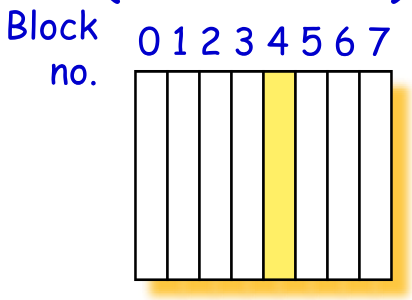
Q1: Where can a block be placed in upper level?

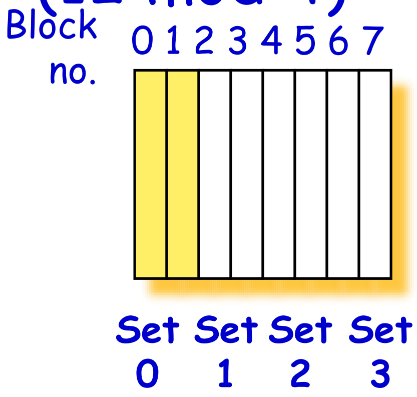
Fully associative:
block 12 can go anywhere

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

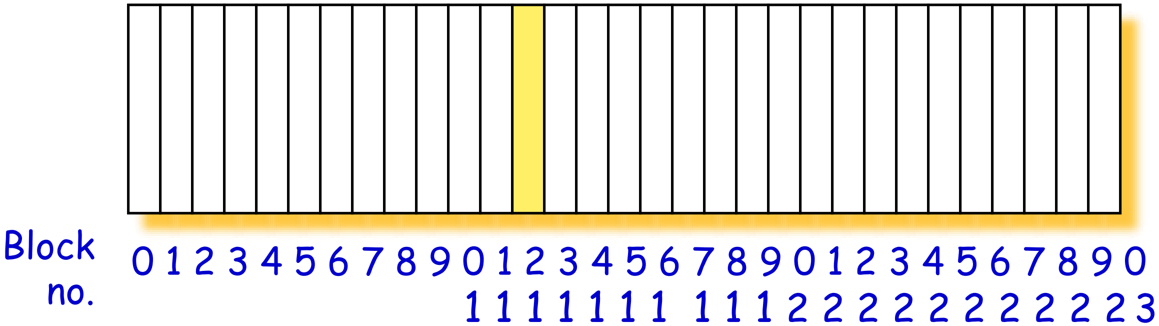
Set associative:
block 12 can go anywhere in set0 (12 mod 4)



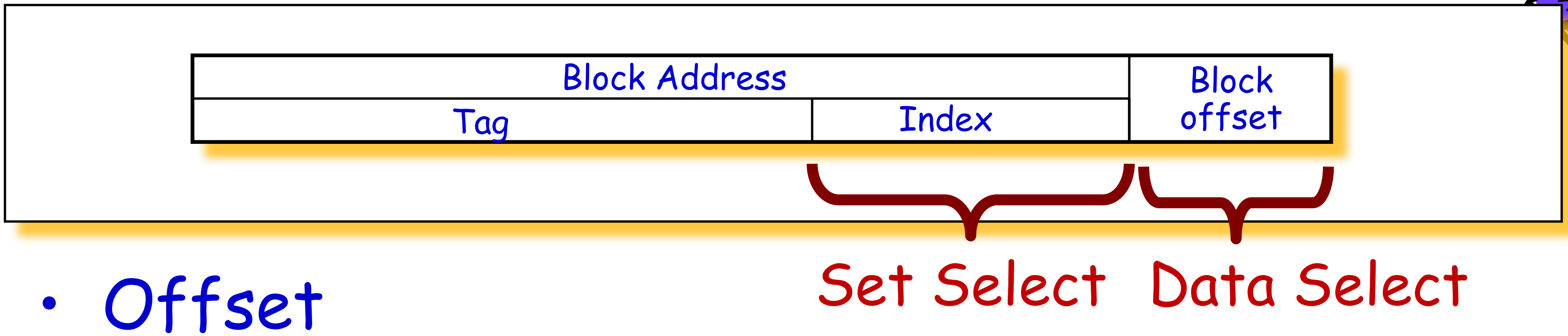




Block-frame address



Q2: How is a block found if it is in upper level?



Block Address		Block offset
Tag	Index	

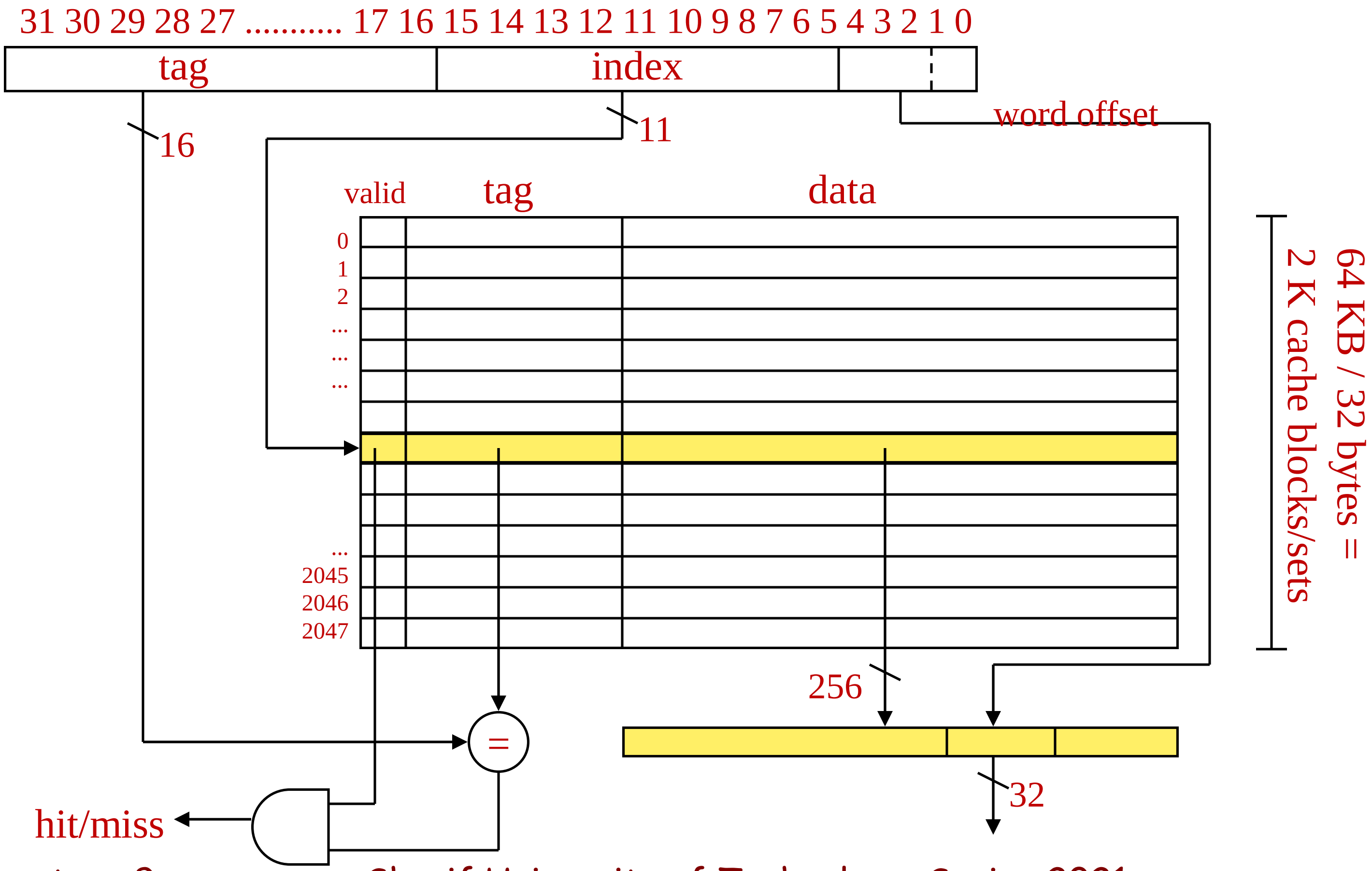
Set Select

Data Select

- Offset
  - Identifies a byte/word within a block
- Index
  - Identifies corresponding set
- Tag
  - Identifies whether associated block corresponds to a requested word or not

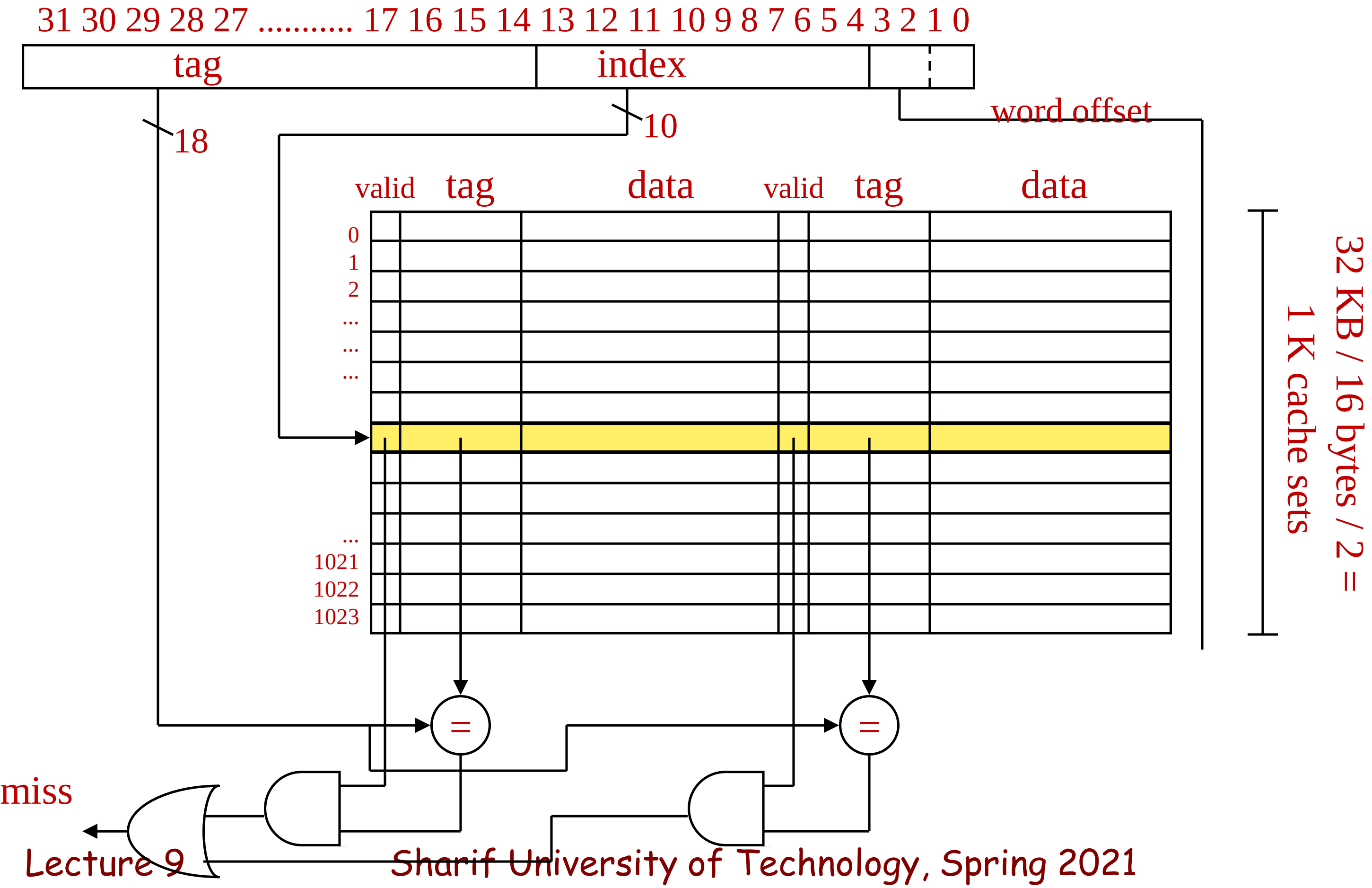
Direct-Mapped

64 KB cache, 32-byte cache block



Set-Associative

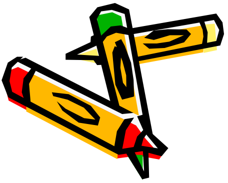
32 KB cache, 2-way,16-byte blocks



Cache Parameters

- Cache size =

  # of sets * block size * associativity
- Example 1
  - 128 blocks, 32-byte blocks, direct mapped, size = ?
- Example 2
  - 128 KB cache, 64-byte blocks, 512 sets, associativity = ?



Direct-Mapped vs. Fully-Associative

- Direct-Mapped
  - Require less area
    - Only one comparator
    - Fewer tag bits required
  - Fast hit time
    - Less bits to compare
  - Cache block is available BEFORE Hit/Miss
    - Return data to CPU in parallel with hit process
    - Possible to assume a hit and continue recover later if miss
  - Conflict misses reduce hit rate

Direct-Mapped vs. Fully-Associative (cont.)

- Fully-Associative
  - More area compared to direct-mapped
    - Need one comparator for each line in cache
  - Longer hit time
    - Too many comparison required
  - Less miss rate vs. direct-mapped
    - No conflict misses (conflict Miss = 0)
  - No cache index
    - Compare tag with all tags of all cache entries in parallel

Q3: Which block should be replaced on a miss?

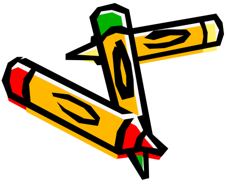
- Easy for Direct Mapped; why?
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used): in status bits
  - FIFO

Associativity:

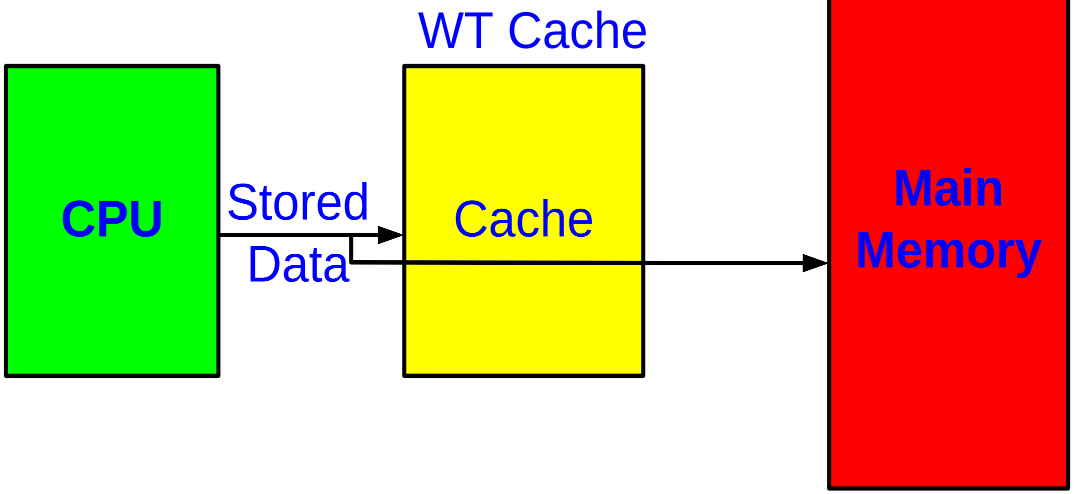
	2-way		4-way		8-way	
Size	LRU	Random	LRU	Random	LRU	Random
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Q4: What happens on a write?

- Write Through (Allocate/Non-Allocate)
  - Information written to both block in cache and to block in lower-level memory
- Write Back
  - Information written only to block in cache
  - Modified cache block written to main memory only when it is replaced
  - Inconsistent
    - Need cache coherency policy for multi-core chips
  - Is block clean or dirty? (status bits)



Write-Through (WT) vs. Write-Back (WB)



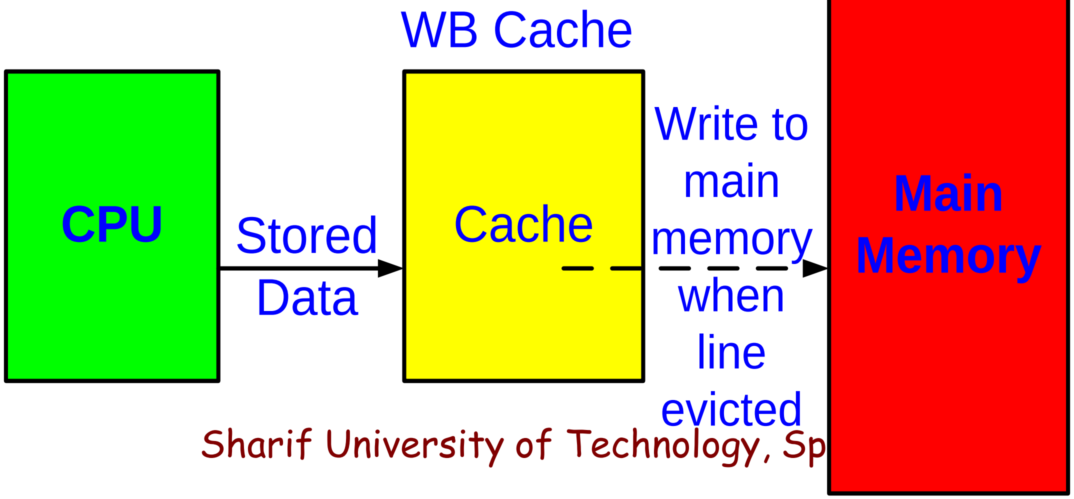
WT Cache

CPU

Stored Data

Cache

Main Memory



WB Cache

CPU

Stored Data

Cache

Write to main memory when line evicted

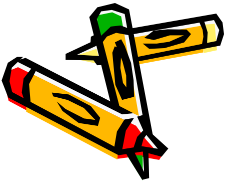
Main Memory

WT Cache

- Pros
  - Simpler to implement
  - Don't need dirty bit
  - No interface issues with I/O devices
    - Cache memory consistent with memory

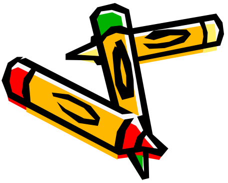
WT Cache (cont.)

- Cons
  - Less performance vs. WB cache
  - Processor held up on writes unless writes buffered
- Write Buffer
  - Stores data while waiting to be written to memory



WB Cache

- Pros
  - Tends to have better performance
    - Repeated writes not sent to DRAM
    - Processor not held up on writes
    - Combines multiple writes into one line WB
  - Virtual memory systems use write-back
    - because of huge penalty for going out to disk

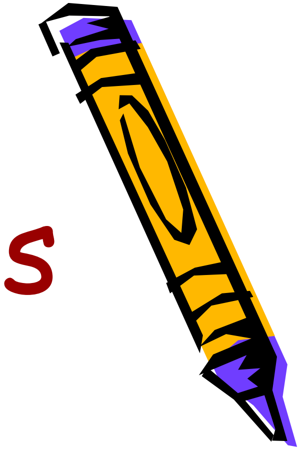


WB Cache (cont.)

- Cons
  - More complex
    - Read miss may require writeback of dirty data
  - Need to implement cache coherency
  - Typically requires two cycles on writes
    - Can't overwrite data and do tag comparison at same time as block may be dirty
    - Unless using store buffer

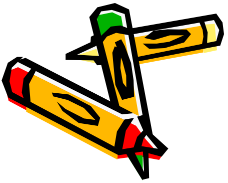
Write Policy in WT Caches

- Allocate-on-Write (Write Allocate)
  - Fetch line into cache
  - Then perform write in cache
  - Also called, fetch-on-miss, fetch-on-write
- No-Allocate-on-Write (No-Write Allocate)
  - Pass write through to main memory
  - Don't bring line into cache
  - Also called Write-Around or Read-Only



Write Policy in WT Caches

- Allocate-on-Write Pros
  - Better performance if data referenced again before it is evicted
- No-Allocate-on-Write Pros
  - Simpler write hardware
  - May be better for small caches if written data won't be read again soon

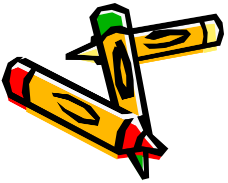


L1 Cache Configuration

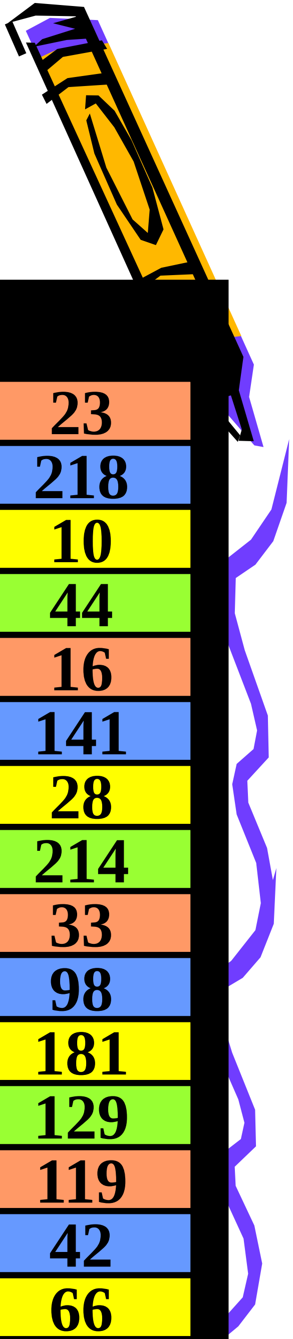
- Split Cache
  - Two independent caches
    - Instruction cache (IL1)
    - Data cache (DL1)
- Unified
  - One unified L1 cache
  - Usually better hit ratio (same size); why?
- Question:
  - Most processors use split caches; why?

Practice

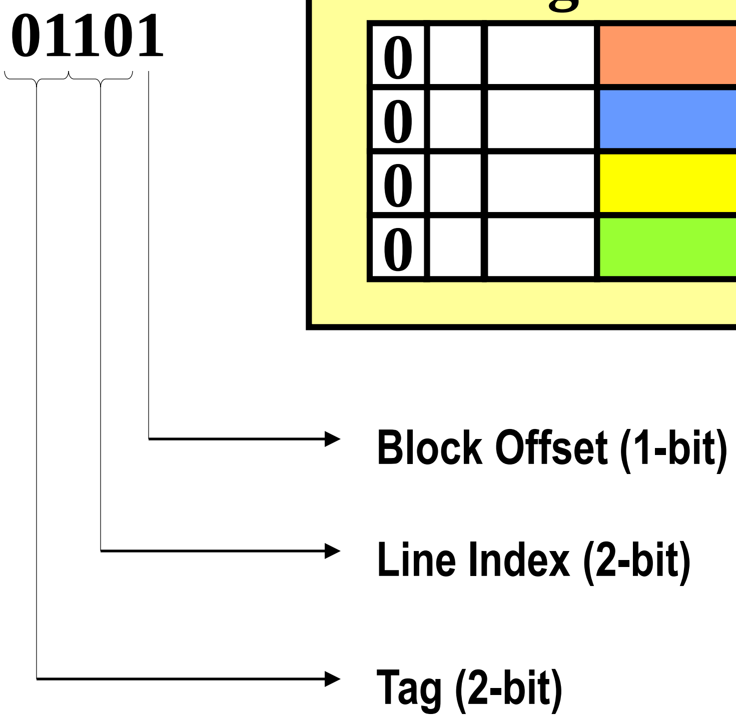
- Consider a 16KB Cache
  - 4-way, 32-bit address, byte-addressable memory, 32-byte cache blocks
- Q1:
  - How many tag bits?
  - Total tag bits in cache?
- Q2:
  - Where to find word with address = 0x200356A4?



Direct-Mapped



Address



01101

Block Offset (1-bit)

Line Index (2-bit)

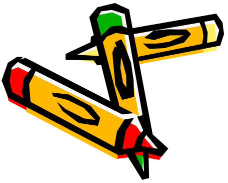
Tag (2-bit)

Cache

V	d	tag	data	
0			🟧	🟧
0			🟦	🟦
0			🟨	🟨
0			🟩	🟩

Memory

00000	78	23
00010	29	218
00100	120	10
00110	123	44
01000	71	16
01010	150	141
01100	162	28
01110	173	214
10000	18	33
10010	21	98
10100	33	181
10110	28	129
11000	19	119
11010	200	42
11100	210	66
11110	225	74



2-Way Set Associative

Address

01101

Block Offset (unchanged)

1-bit Set Index

Larger (3-bit) Tag

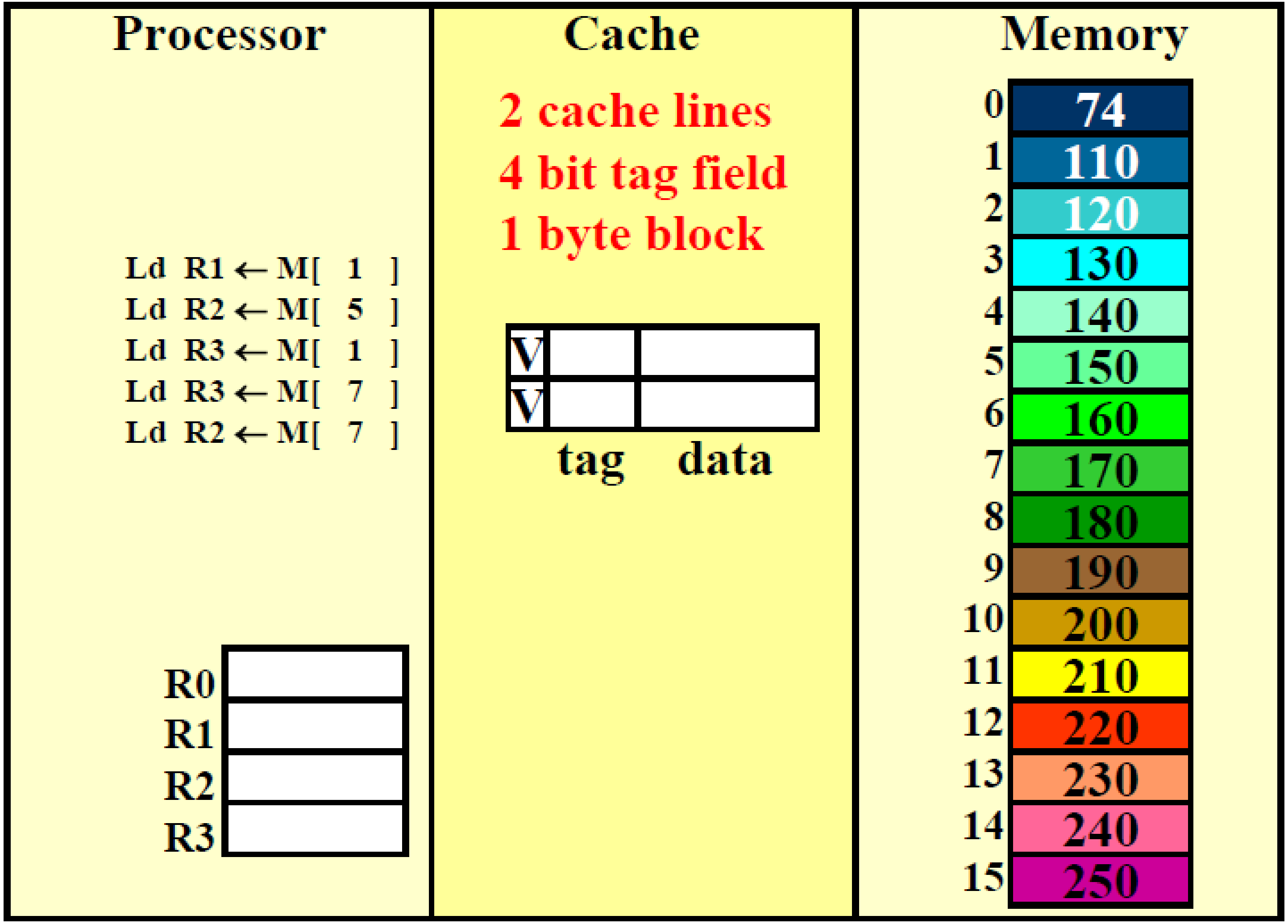
Cache

V	d	tag	data	
0			🟧	🟧
0			🟧	🟧
0			🟦	🟦
0			🟦	🟦

Memory

00000	78	23
00010	29	218
00100	120	10
00110	123	44
01000	71	16
01010	150	141
01100	162	28
01110	173	214
10000	18	33
10010	21	98
10100	33	181
10110	28	129
11000	19	119
11010	200	42
11100	210	66
11110	225	74

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

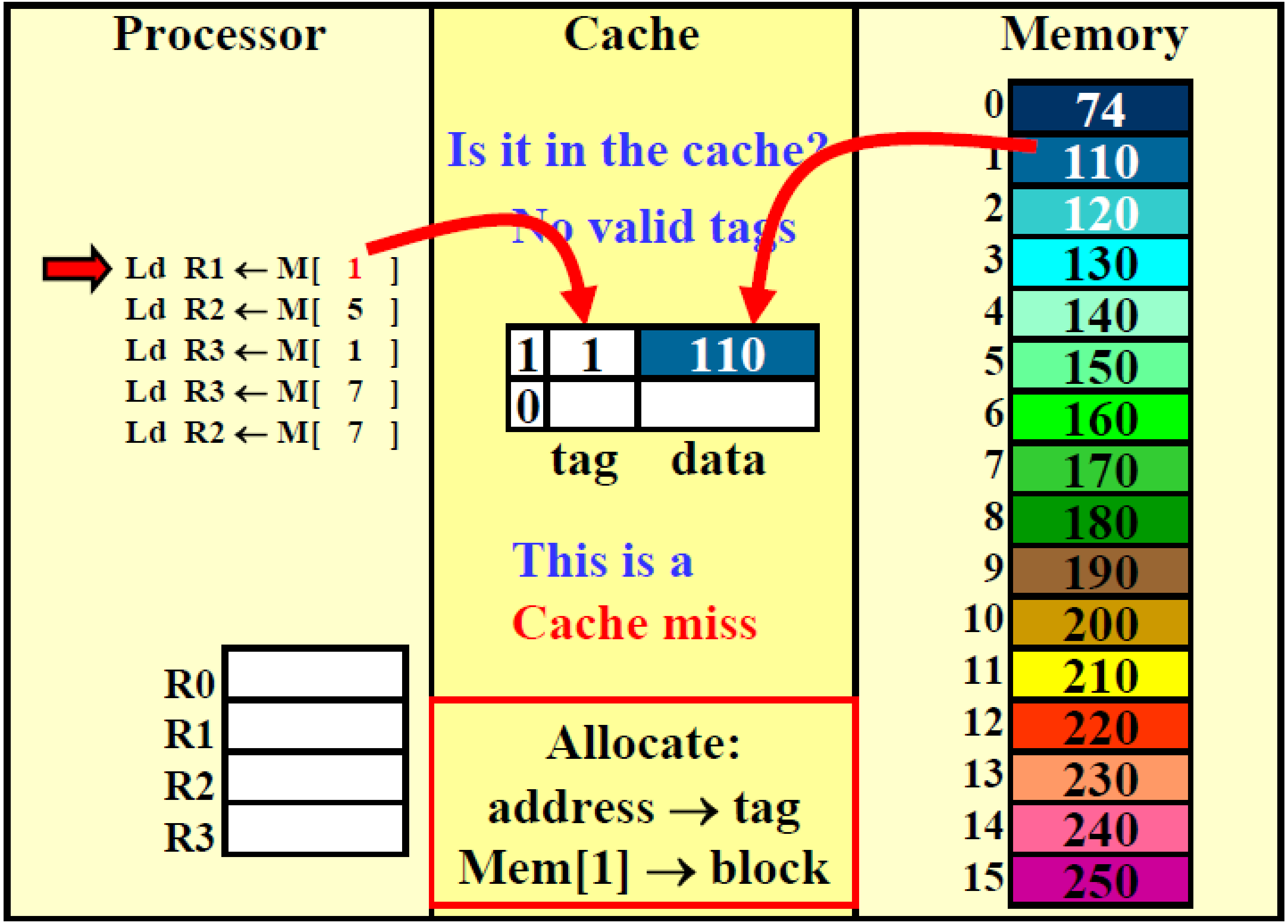
Cache

2 cache lines
4 bit tag field
1 byte block

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	
R2	
R3	

Cache

Is it in the cache?

No valid tags

	tag	data
1	1	110
0		

This is a

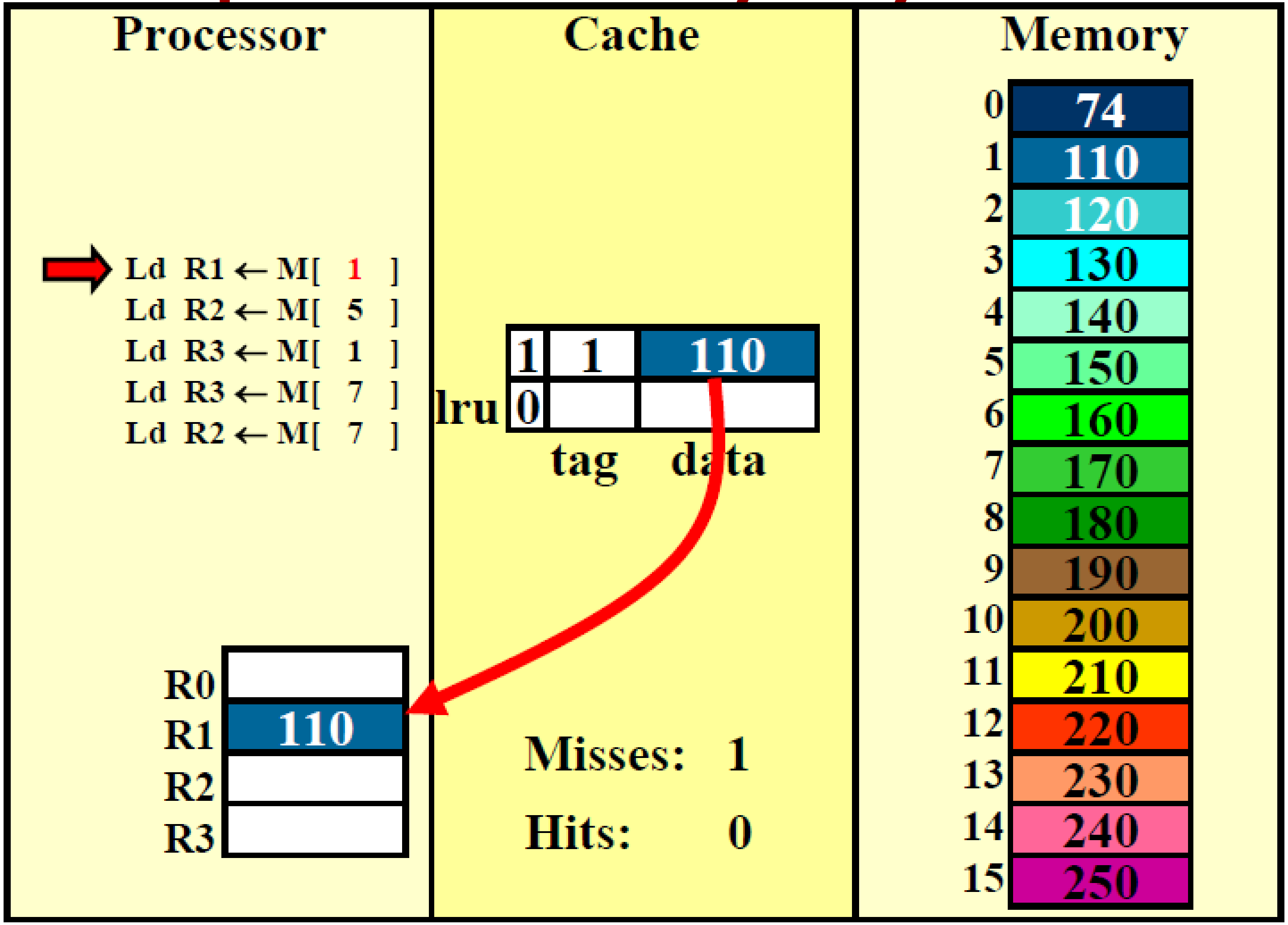
Cache miss

Allocate:
address → tag
Mem[1] → block

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Cache

		tag	data
	1	1	110
lru	0		

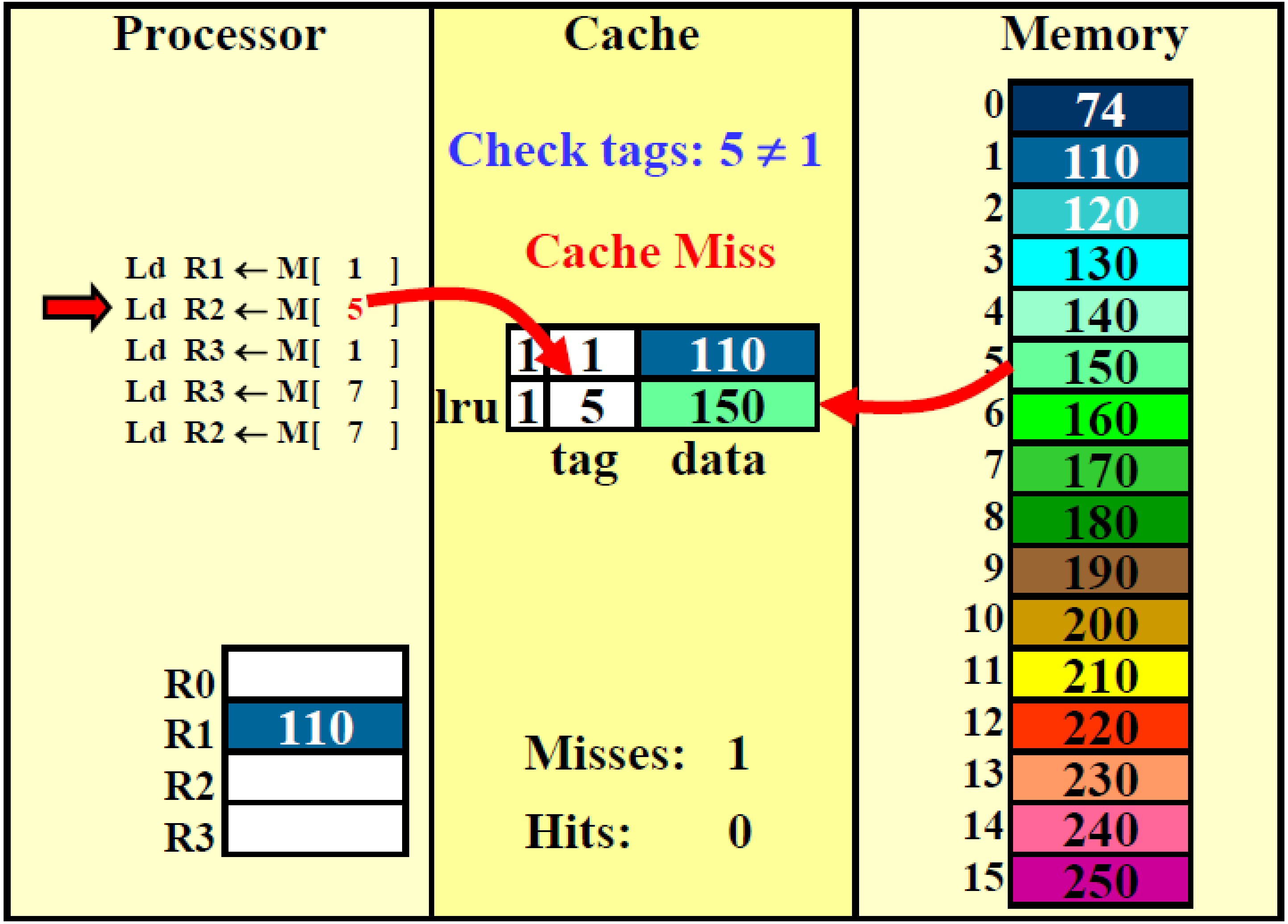
Misses: 1

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	
R3	

Cache

Check tags: 5 ≠ 1

Cache Miss

		tag	data
	1	1	110
lru	1	5	150

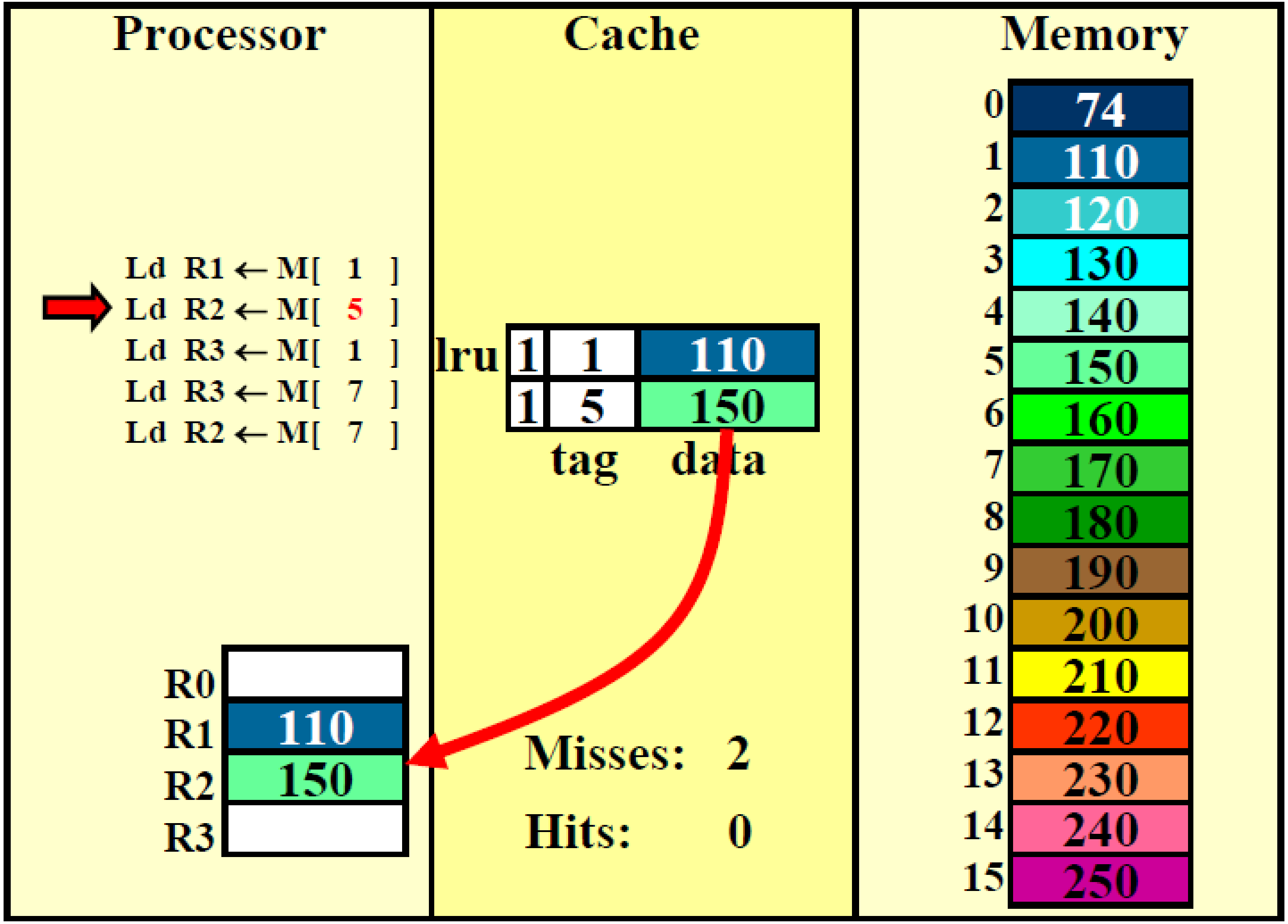
Misses: 1

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	150
R3	

Cache

lru	tag	data
1	1	110
1	5	150

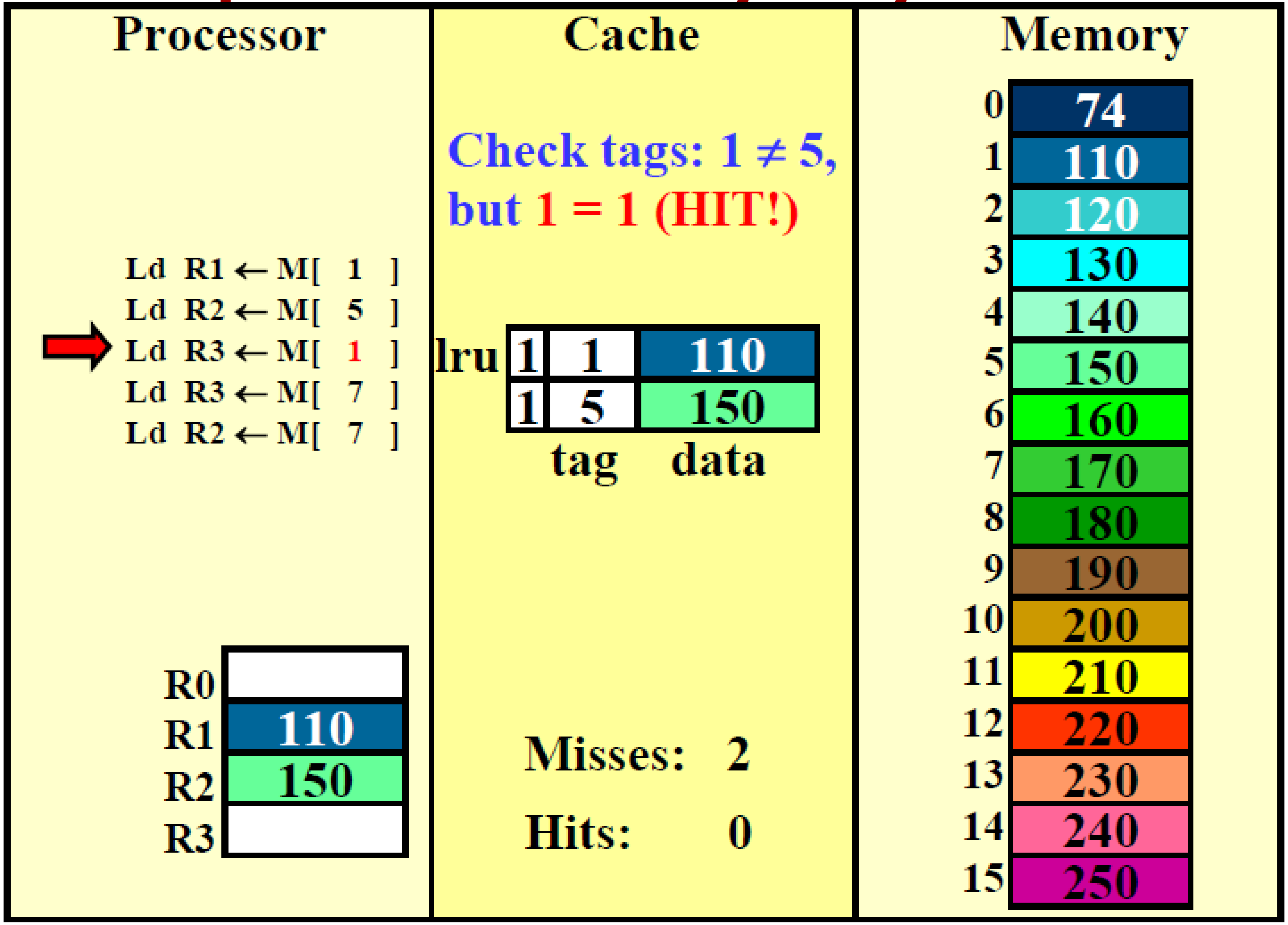
Misses: 2

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

```
Ld R1 ← M[ 1 ]
Ld R2 ← M[ 5 ]
Ld R3 ← M[ 1 ]
Ld R3 ← M[ 7 ]
Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	150
R3	

Cache

Check tags: $1 \neq 5$, but $1 = 1$ (HIT!)

lru	tag	data
1	1	110
1	5	150

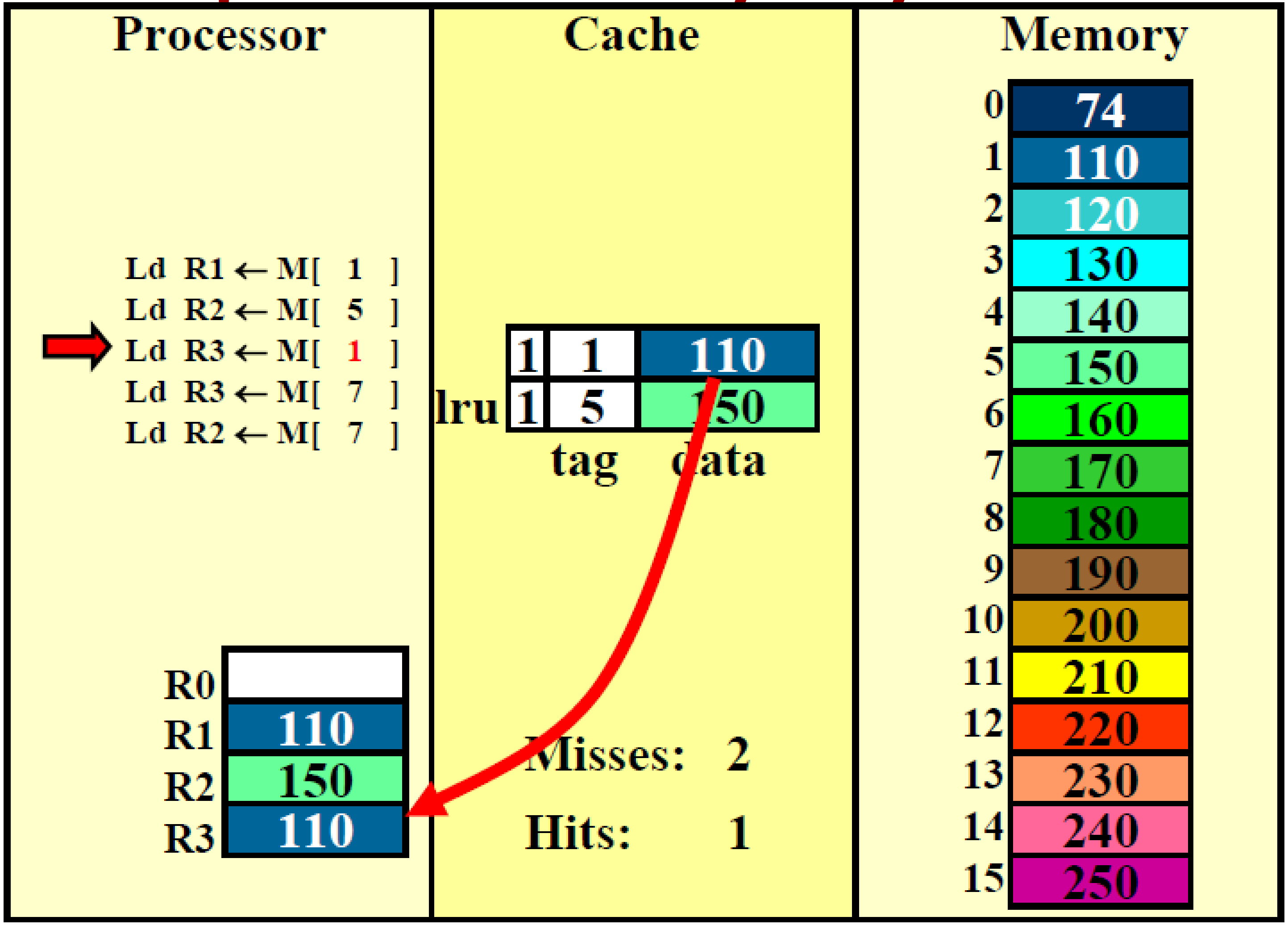
Misses: 2

Hits: 0

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

```
  Ld R1 ← M[ 1 ]
  Ld R2 ← M[ 5 ]
➡ Ld R3 ← M[ 1 ]
  Ld R3 ← M[ 7 ]
  Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	150
R3	110

Cache

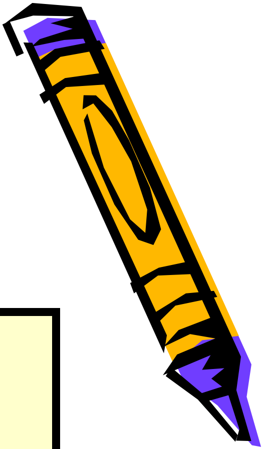
		tag	data
	1	1	110
lru	1	5	150

Misses: 2

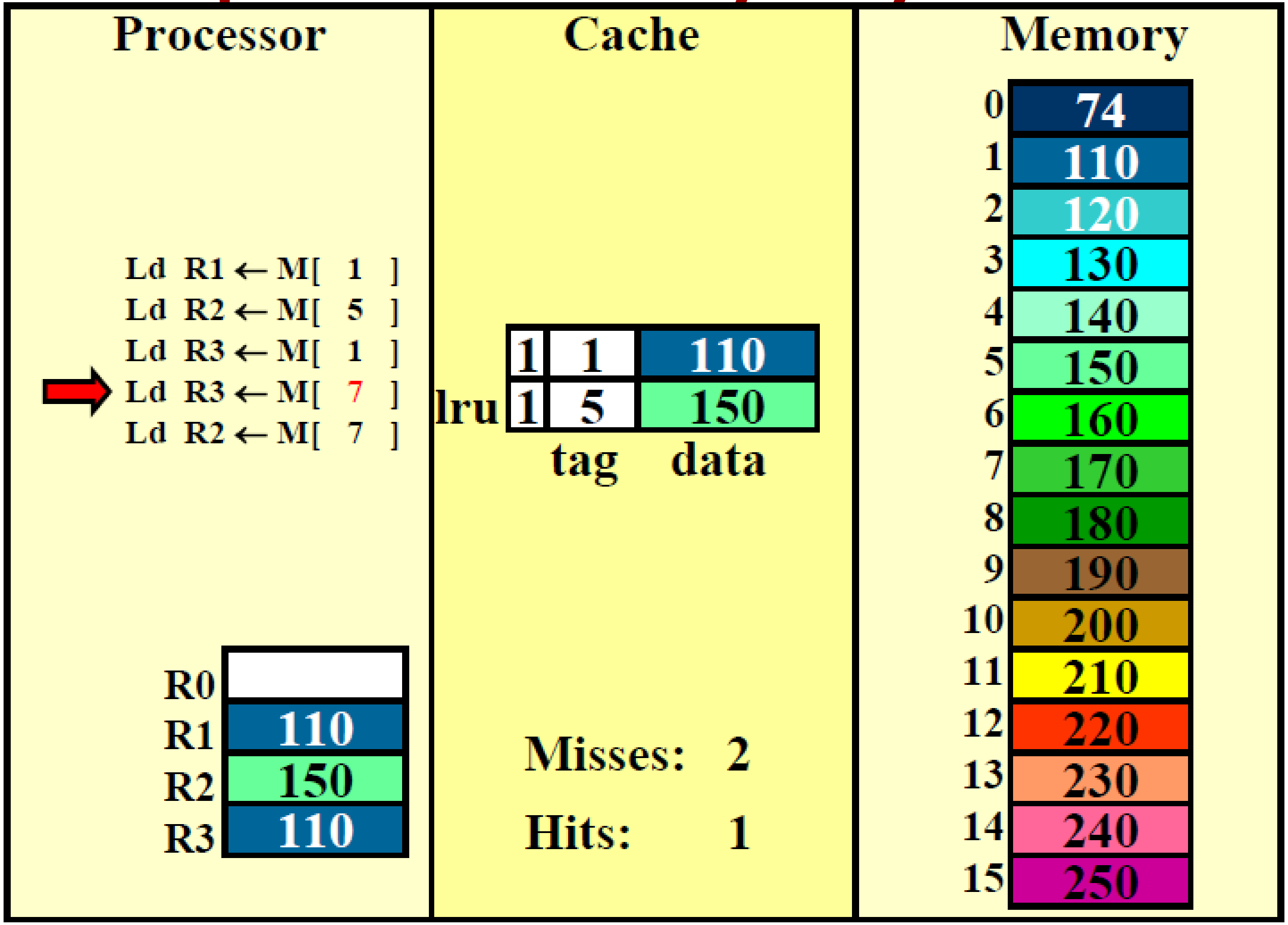
Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250



Simple Memory System



Processor

```
   Ld R1 ← M[ 1 ]
   Ld R2 ← M[ 5 ]
   Ld R3 ← M[ 1 ]
➡ Ld R3 ← M[ 7 ]
   Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	150
R3	110

Cache

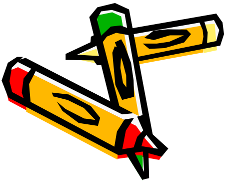
		tag	data
	1	1	110
lru	1	5	150

Misses: 2

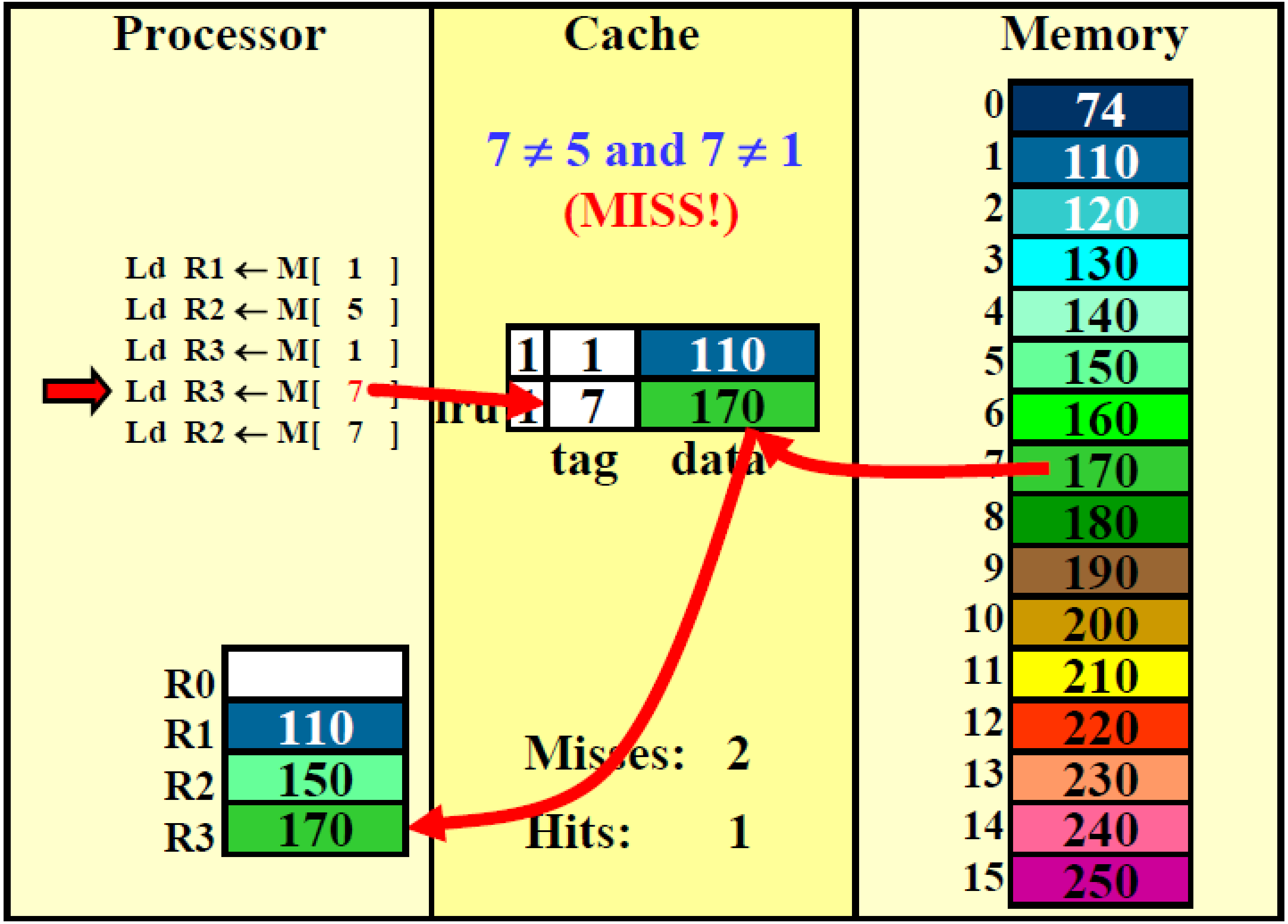
Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250



Simple Memory System



Processor

Cache

Memory

7 ≠ 5 and 7 ≠ 1

(MISS!)

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

		tag	data
	1	1	110
lru	1	7	170

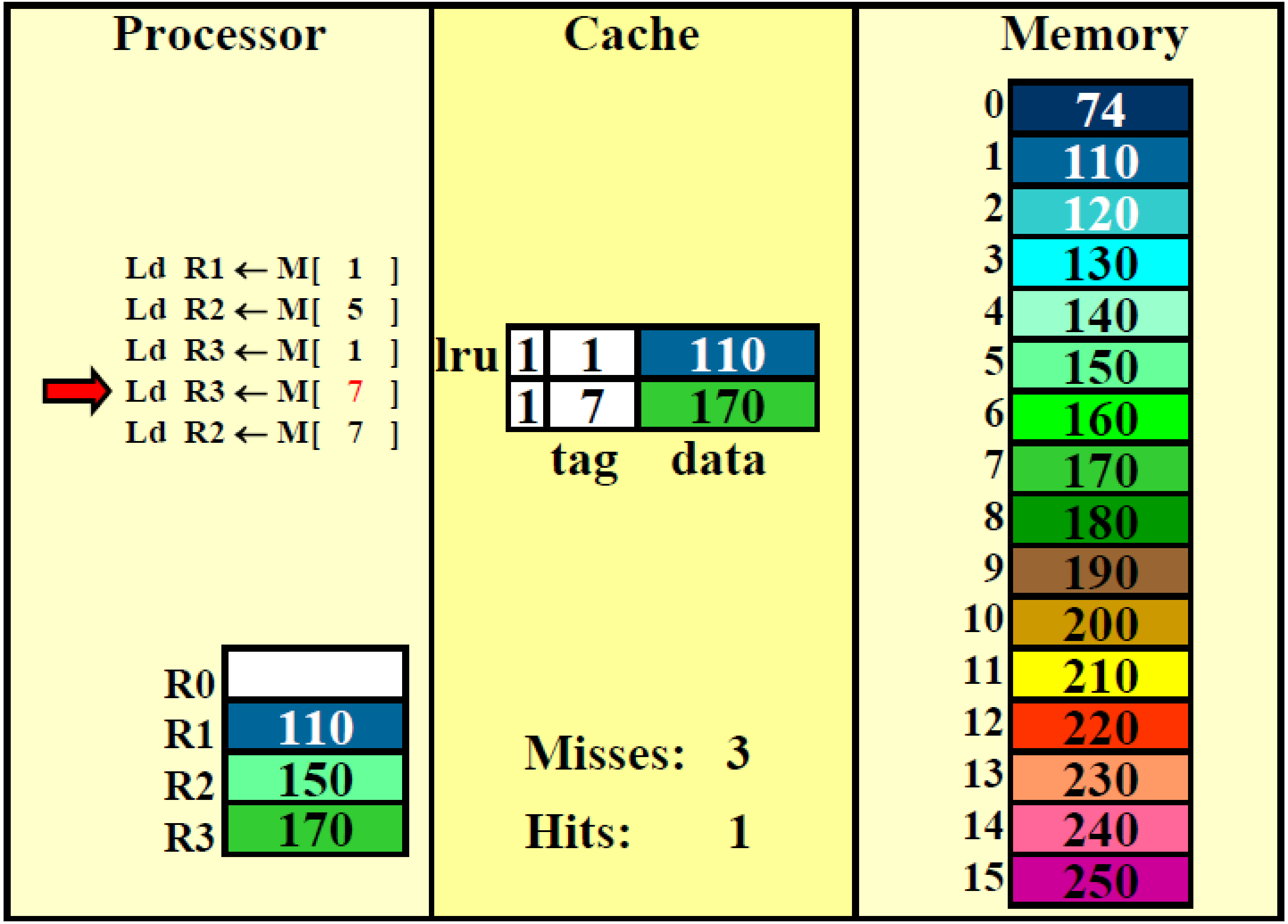
Misses: 2

Hits: 1

R0	
R1	110
R2	150
R3	170

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

```
   Ld R1 ← M[ 1 ]
   Ld R2 ← M[ 5 ]
   Ld R3 ← M[ 1 ]
⇒  Ld R3 ← M[ 7 ]
   Ld R2 ← M[ 7 ]
```

R0	
R1	110
R2	150
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

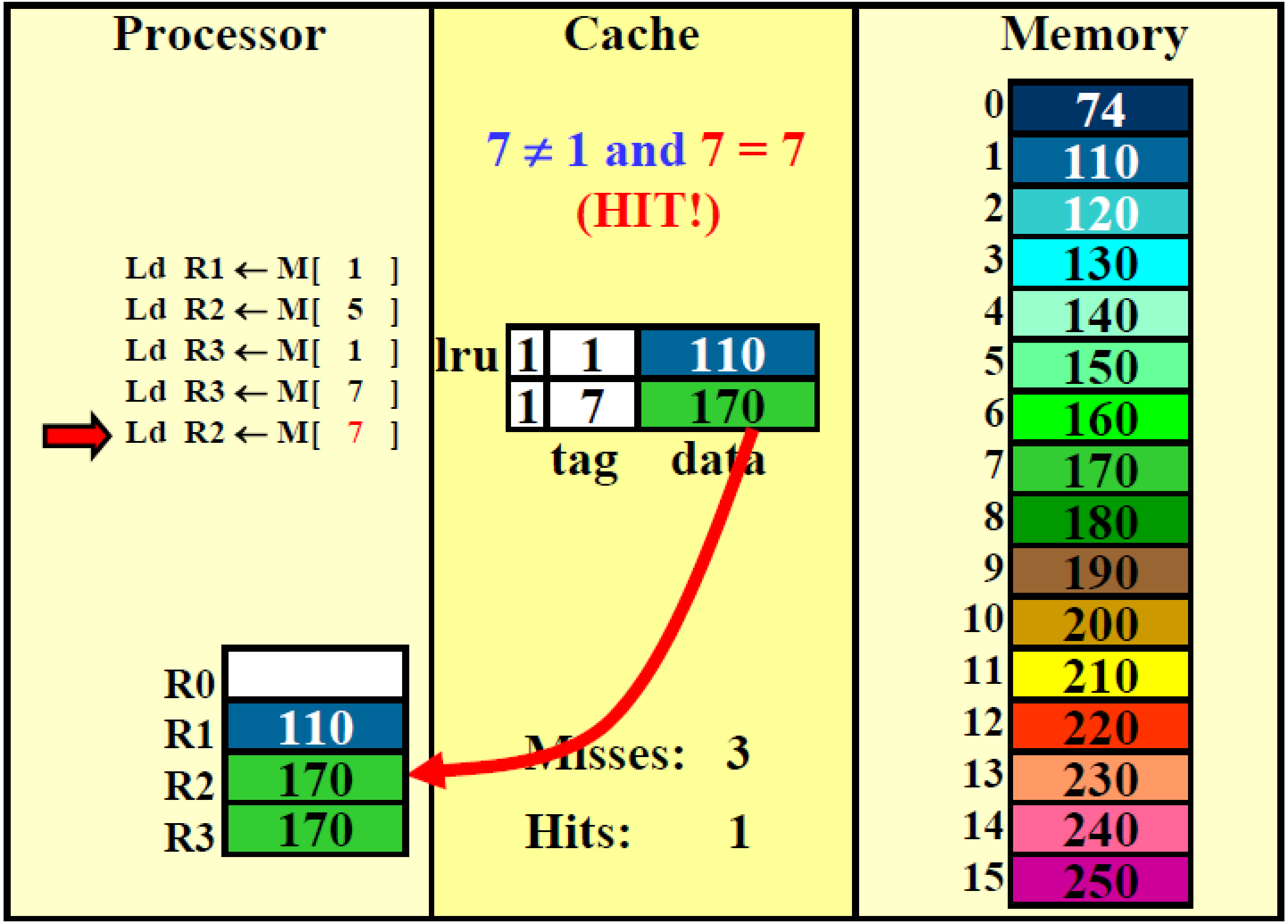
Misses: 3

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

7 ≠ 1 and 7 = 7
(HIT!)

lru	tag	data
1	1	110
1	7	170

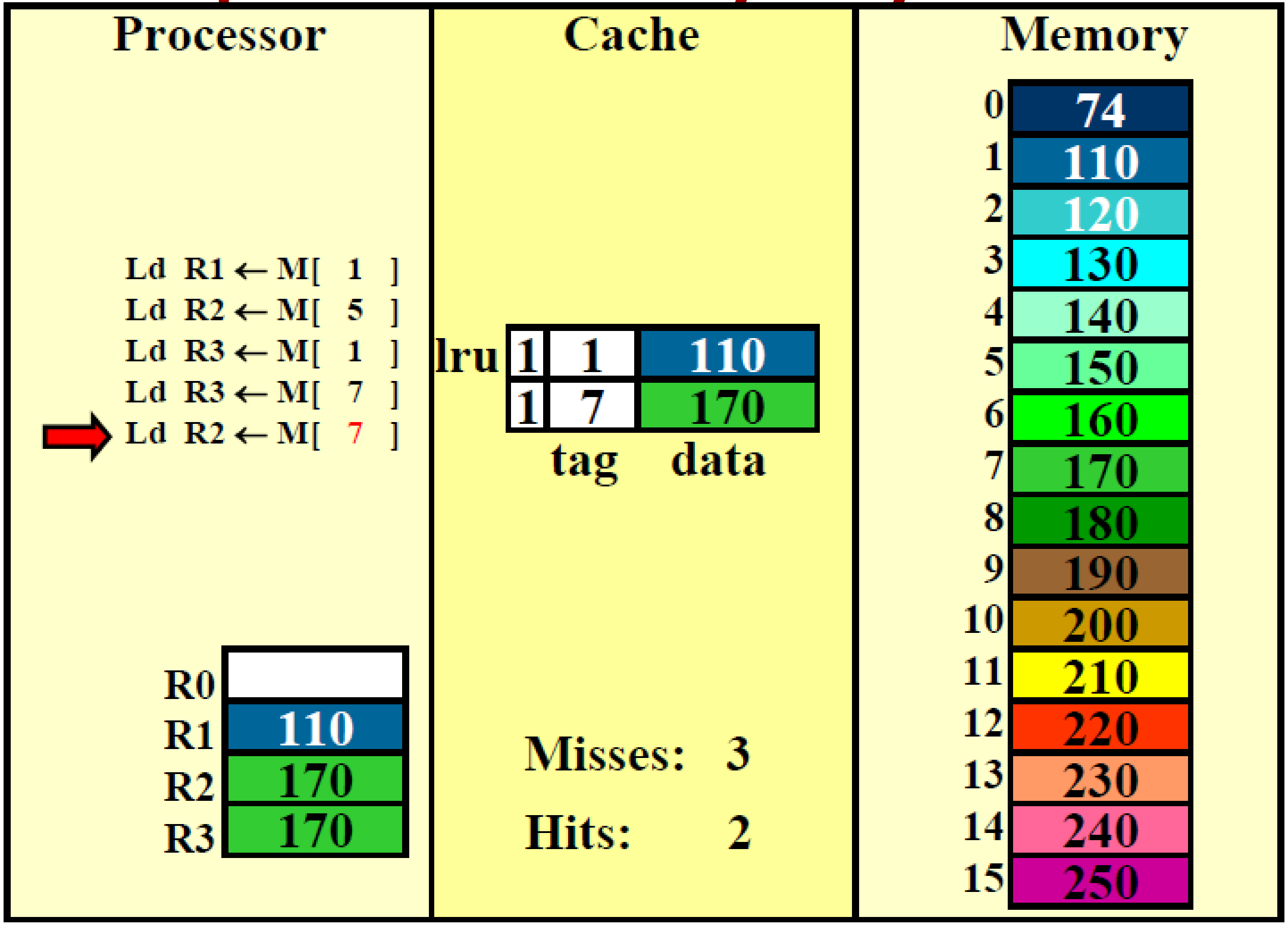
Misses: 3

Hits: 1

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Simple Memory System



Processor

Ld R1 ← M[1]
Ld R2 ← M[5]
Ld R3 ← M[1]
Ld R3 ← M[7]
Ld R2 ← M[7]

R0	
R1	110
R2	170
R3	170

Cache

lru	tag	data
1	1	110
1	7	170

Misses: 3

Hits: 2

Memory

0	74
1	110
2	120
3	130
4	140
5	150
6	160
7	170
8	180
9	190
10	200
11	210
12	220
13	230
14	240
15	250

Reminder: Improving Cache Performance

- AMAT =

  Hit Time + (Miss Rate x Miss Penalty)
- Options to Reduce AMAT
  - Reduce time to hit in cache
    - Use smaller cache size
  - Reduce miss rate
    - Increase cache size
  - Reduce miss penalty
    - Use multi-level cache hierarchy

Cache Hit Time

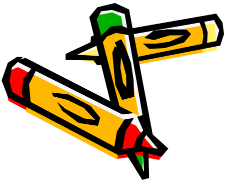
- Impact on Cycle Time
  - Directly tied to clock rate
  - Increases with cache size
  - Increases with associativity

Sources of Cache Misses

- 3Cs
  - Compulsory
  - Capacity
  - Conflict
- Another source of cache miss
  - Coherence

Sources of Cache Misses

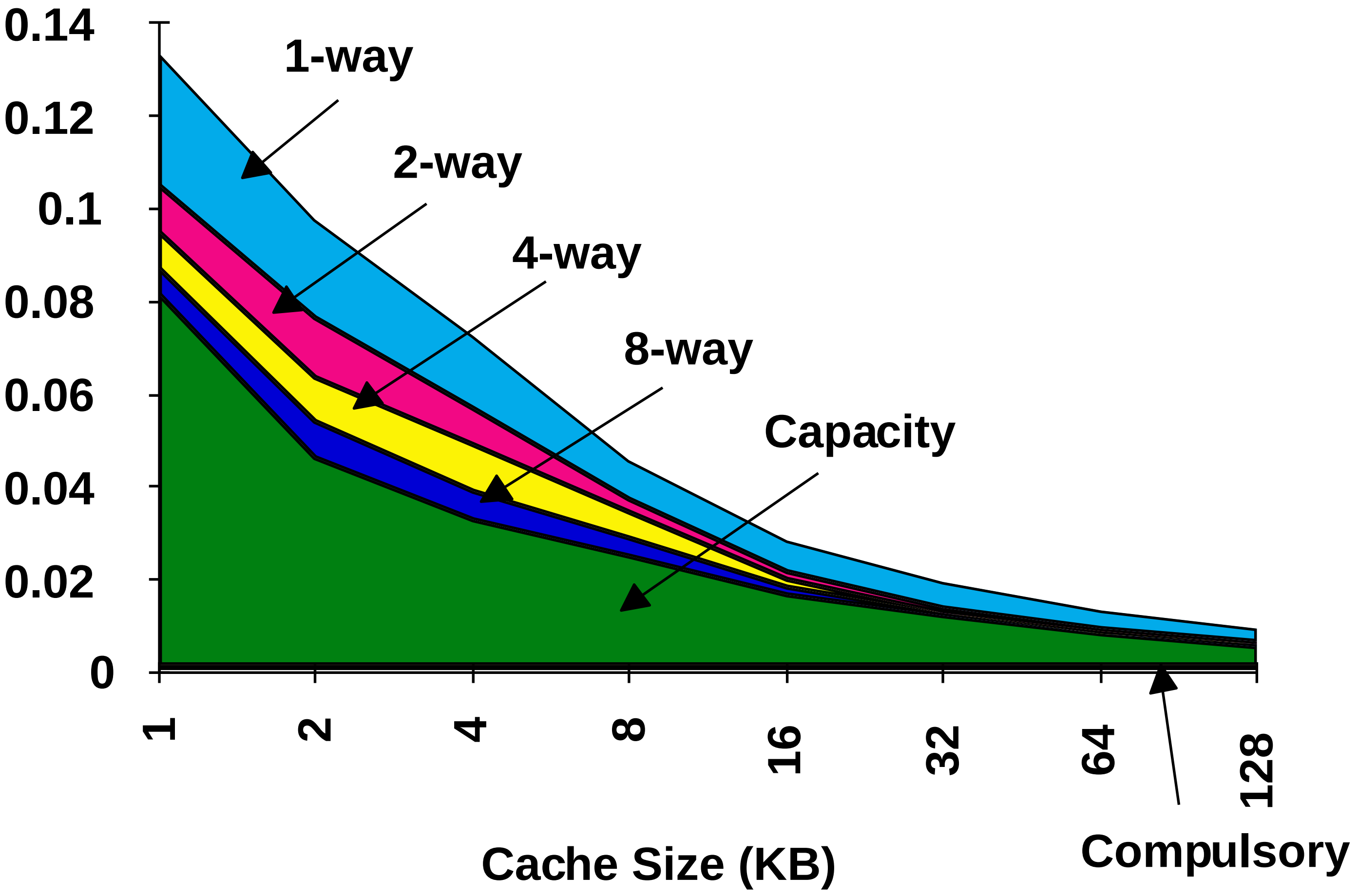
- Compulsory
  - Cold start or process migration
  - First access to a block
  - Compulsory misses are insignificant
    - When running "billions" of instruction
- Capacity
  - Cache cannot contain all blocks accessed by program
  - Solution: increase cache size



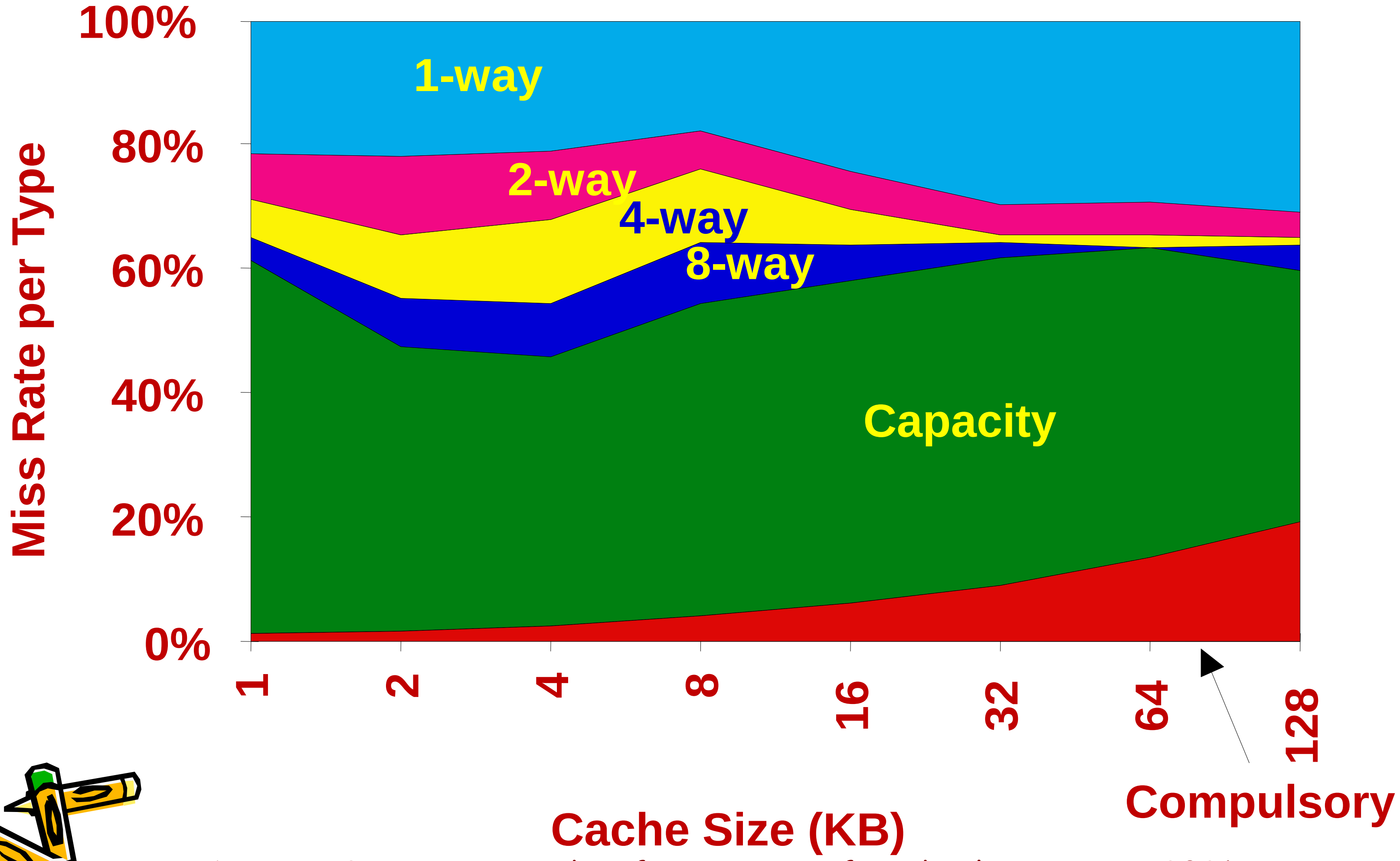
Sources of Cache Misses

- Conflict (collision)
  - Multiple memory locations mapped to same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity
- Coherence (Invalidation)
  - Other processes (e.g., I/O or a core in a CMP) updates memory

3Cs Absolute Miss Rate

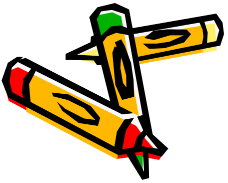


3Cs Relative Miss Rate

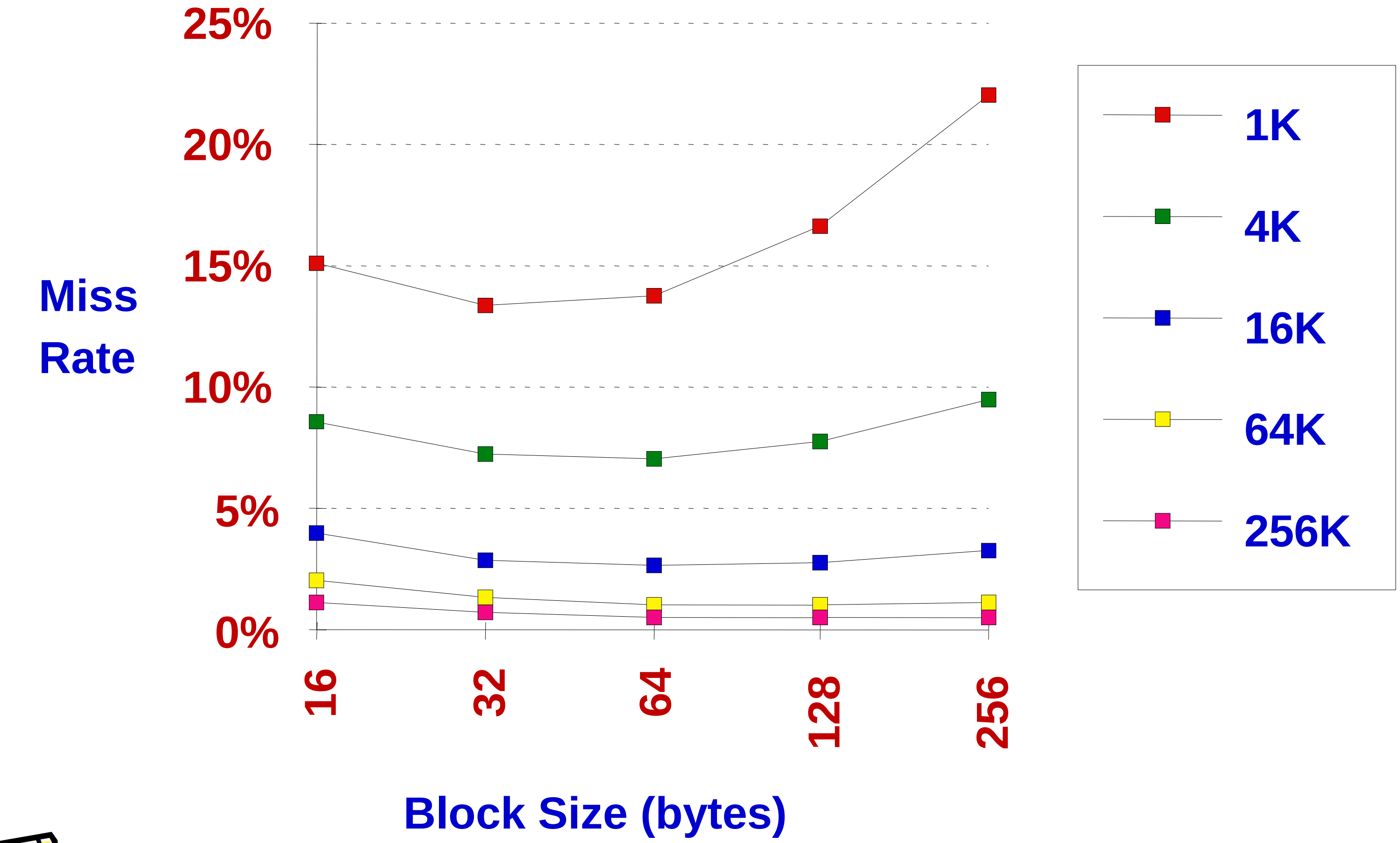


Reducing Miss Rate

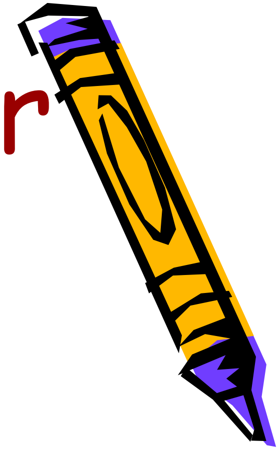
- Larger Block Size
- Higher Associativity
- Prefetching
- Complier Optimization



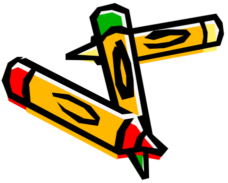
Reducing Misses via Larger Block Size



Reducing Misses via Higher Associativity

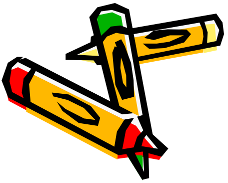


- 2:1 Cache Rule:
  - Miss Rate DM cache size N = Miss Rate 2-way cache size N/2
- Watch Out
  - Execution time is only final measure!
  - AMAT not always improved by more associativity!



Reducing Misses by Prefetching

- Instruction Prefetching
- Data Prefetching
- HW vs. SW Prefetching



Reducing Miss Penalty

- Faster RAM Technologies
  - Use of faster SRAMs and DRAMs
- More Hierarchy Levels
  - 1-level → 2-level → 3-level
- Read Priority over Write on Miss
  - Reads on critical path