CS161 – Design and Architecture of Computer Systems

Week 4 - Discussion

UNIVERSITY OF CALIFORNIA, RIVERSIDE

Midterm Review

Instructions	Frequency	Cycles
Arithmetic and logic	44%	2
Load	24%	4
Store	12%	1
Conditional branch	16%	4
Jump	4%	4

1. (8 points) Compute the CPI for this program on this CPU.

CPI = 0.44*2 + 0.24*4 + 0.12*1 + 0.16*4 + 0.04*4
= 0.88 + 0.96 + 0.12 + 0.64 + 0.16
= 2.76 cycles/instruction

2. (8 points) A new compiler reduces the number of Conditional Branches by half, compute the new CPI.

new CPI = (0.44*2 + 0.24*4 + 0.12*1 + 0.08*4 + 0.04*4) / 0.92
= (0.88 + 0.96 + 0.12 + 0.32 + 0.16) / 0.92
= 2.65 cycles/instruction (regrade?)

Midterm Review

3. (2 points each) Name the three types of pipeline hazards and name three solutions discussed in class.

Hazards:	Solutions
1. Structural Hazard	1. Stall
2. Data Hazard	2. Forwarding
3. Control Hazard	3. Reorder Instructions

4. (8 points) Write the MIPS code corresponding to this C code. Assume i and k correspond to $s3 and $s5 while the base of the array is at $s6.

```
While (save[i] == k)
        i += 1;
```

```
LOOP :   sll $t0, $s3, 2        # $t0 = i * 4
         add $t0, $t0, $s6      # $t0 = &save[i]
         lw $t1, 0($t0)         # $t1 = save[i]
         bne $t1, $s5, EXIT     # if (save[i] != k) goto EXIT
         addi $s3, $s3, 1       # i = i + 1
         j LOOP                 # goto LOOP
EXIT:
```

Midterm Review

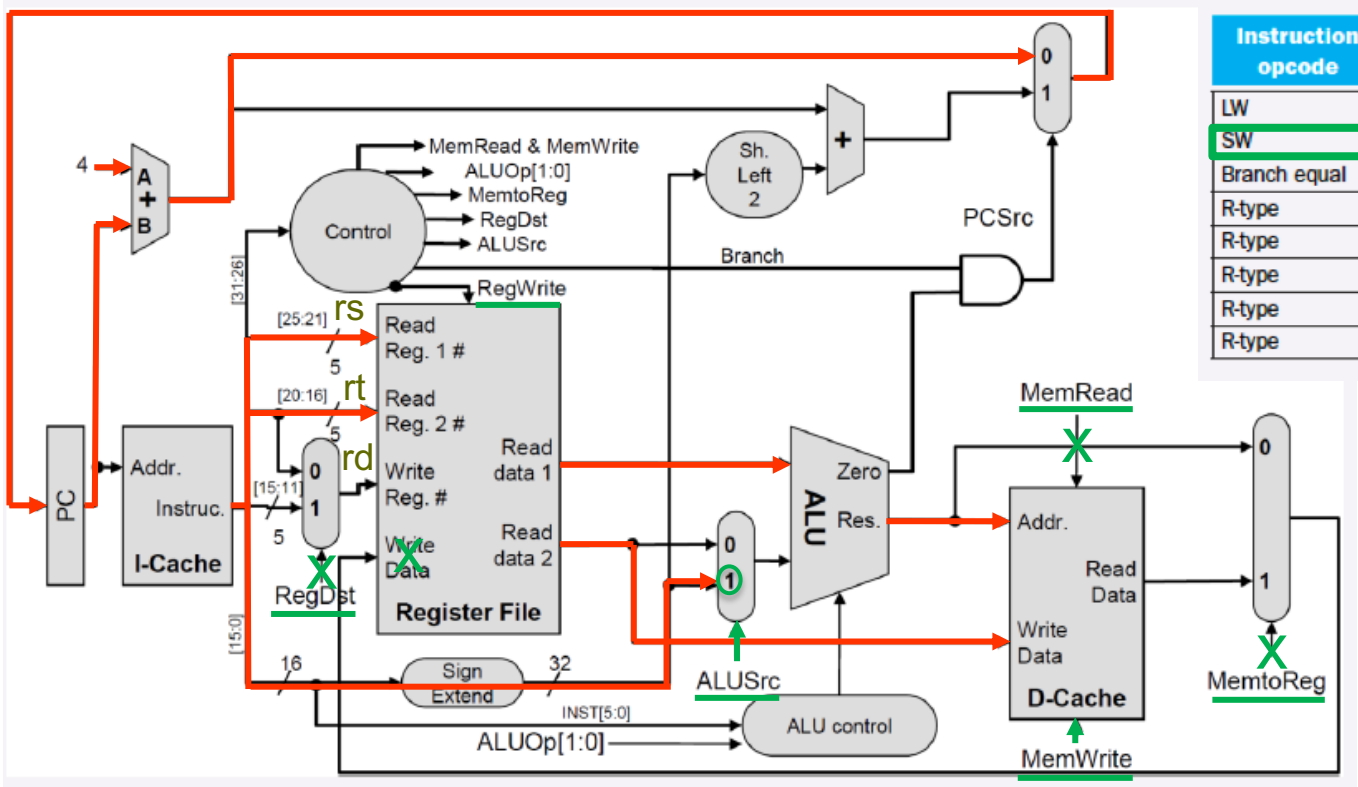
5. (8 points) Given the MIPS instruction lw $t0, 4($s0) explain how the 32 bits of the instruction word will be allocated.

```
lw rt, imm(rs)      # rt   ← Mem[imm+rs]
lw $t0, 4($s0)      # $r0  ← Mem[4+$s0]
```

[31-26]: opcode (lw)
[25-21]: register source ($s0)
[20-16]: register destination ($t0)
[15-0]: immediate(#4)

Midterm Review

6. (8 points) Given the MIPS instruction sw R0, 8(R0), determine the necessary control signals.

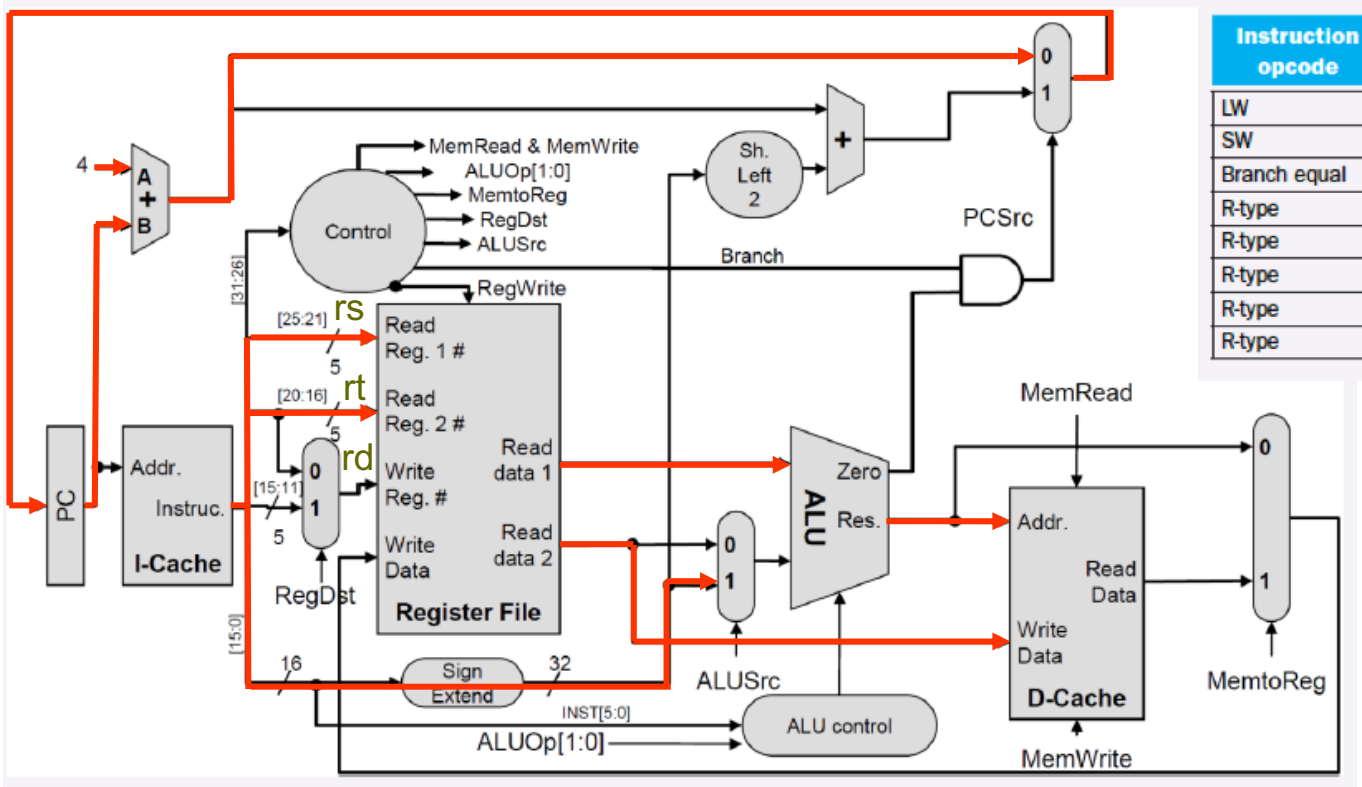


Instruction opcode	ALUOp	Instruction operation
LW	00	load word
SW	00	store word
Branch equal	01	branch equal
R-type	10	add
R-type	10	subtract
R-type	10	AND
R-type	10	OR
R-type	10	set on less than

Jump	Branch	Reg Dest	ALU Src	MemToReg	Reg Write	Mem Read	Mem Write	ALU Op[1]	ALU Op[0]
0	0	X	1	X	0	0	1	0	0

Midterm Review

7. (8 points) Trace the datapath that is active during the sw R0, 8(R0) instructi

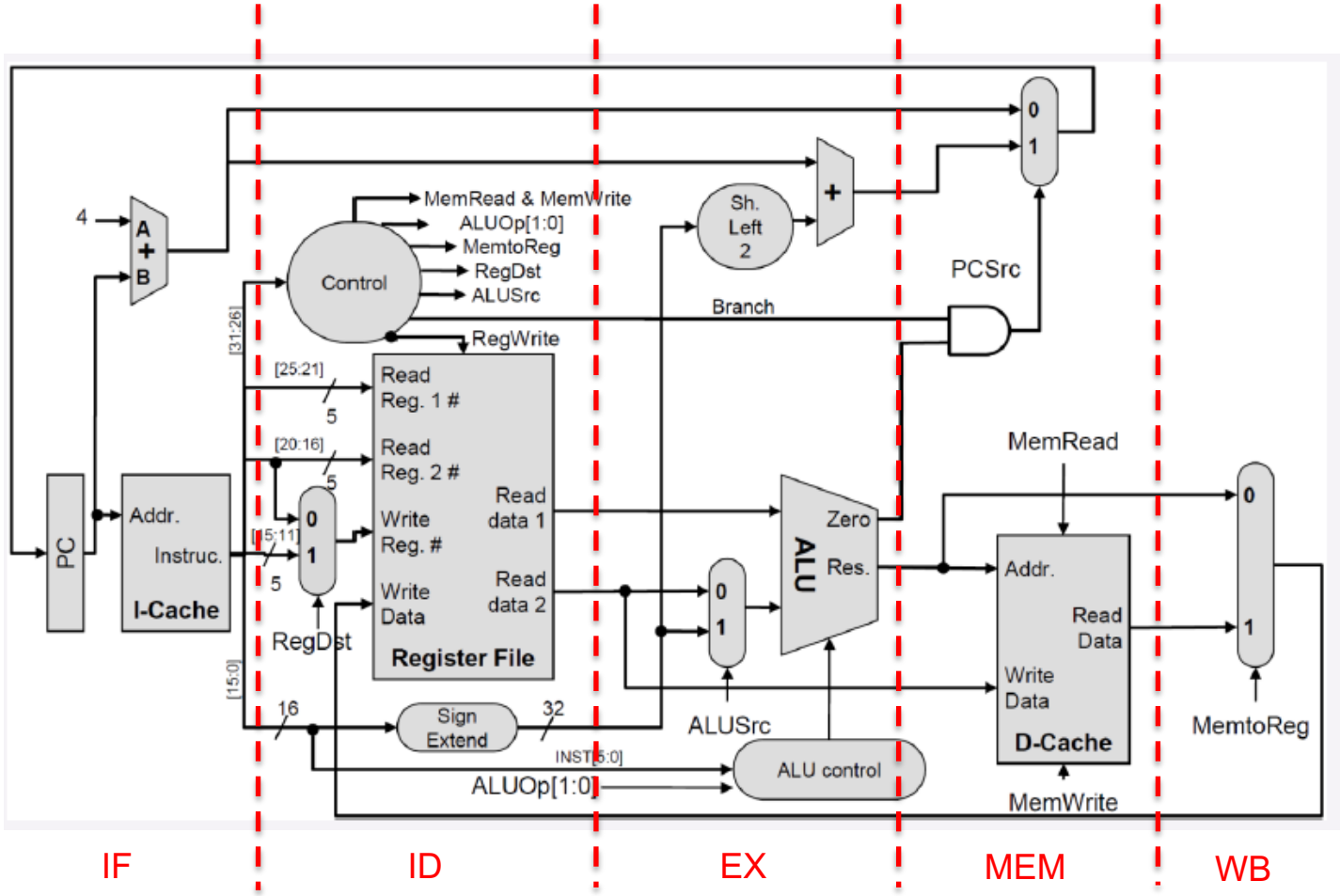


Instruction opcode	ALUOp	Instruction operation
LW	00	load word
SW	00	store word
Branch equal	01	branch equal
R-type	10	add
R-type	10	subtract
R-type	10	AND
R-type	10	OR
R-type	10	set on less than

Jump	Branch	Reg Dest	ALU Src	MemToReg	Reg Write	Mem Read	Mem Write	ALU Op[1]	ALU Op[0]
0	0	X	1	X	0	0	1	0	0

Midterm Review

8. (8 points) Separate the following datapath into stages of the pipeline discussed in class. Label each stage clearly.



Midterm Review

9. (8 points) Map the pipeline schedule for the code below. Assume there is no forwarding enabled.

With no forwarding enabled (dependencies)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Comments
`lw $t1, 0($t0)`	F	D	E	M	W															
`lw $t2, 4($t0)`		F	D	E	M	W														
`add $t3, $t1, $t2`			F	-	-	D	E	M	W											Wait on $t2, stall 2 cycles
`sw $t3, 12($t0)`						F	-	-	D	E	M	W								Wait on $t3, stall 2 cycles
`lw $t4, 8($t0)`									F	D	E	M	W							
`add $t5, $t1, $t4`										F	-	-	D	E	M	W				Wait on $t4, stall 2 cycles
`sw $t5, 16($t0)`													F	-	-	D	E	M	W	Wait on $t5, stall 2 cycles

Midterm Review

10. (8 points) Map the pipeline schedule for the code below using forwarding when necessary.

With forwarding enabled (dependencies – forwarding)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	Comments
`lw $t1, 0($t0)`	F	D	E	M	W															
`lw $t2, 4($t0)`		F	D	E	M	W														
`add $t3, $t1, $t2`			F	-	D	E	M	W												Forward $t2 from Mem/WB to Exec, Stall 1 cycle
`sw $t3, 12($t0)`					F	D	E	M	W											Forward $t3 from Exec/Mem to Exec.
`lw $t4, 8($t0)`						F	D	E	M	W										
`add $t5, $t1, $t4`							F	-	D	E	M	W								Forward $t4 from Mem/WB to Exec, Stall 1 cycle
`sw $t5, 16($t0)`									F	D	E	M	W							Forward $t5 from Exec/Mem to Exec.

CS161 – Design and Architecture of Computer Systems

Chp 5 – Memory

*some slides adapted from:
Prof Daniel Wong UCR – EE/CS)

UNIVERSITY OF CALIFORNIA, RIVERSIDE

DRAM vs SRAM

- Dynamic Random Access Memory (DRAM)
  - slowly discharges, need to refresh to preserve
  - slower → cheaper → larger (can afford more)
- Static Random Access Memory (SRAM)
  - no need to refresh (more transistors to “store”)
  - much faster than DRAM
  - faster → expensive → smaller (use less of them)

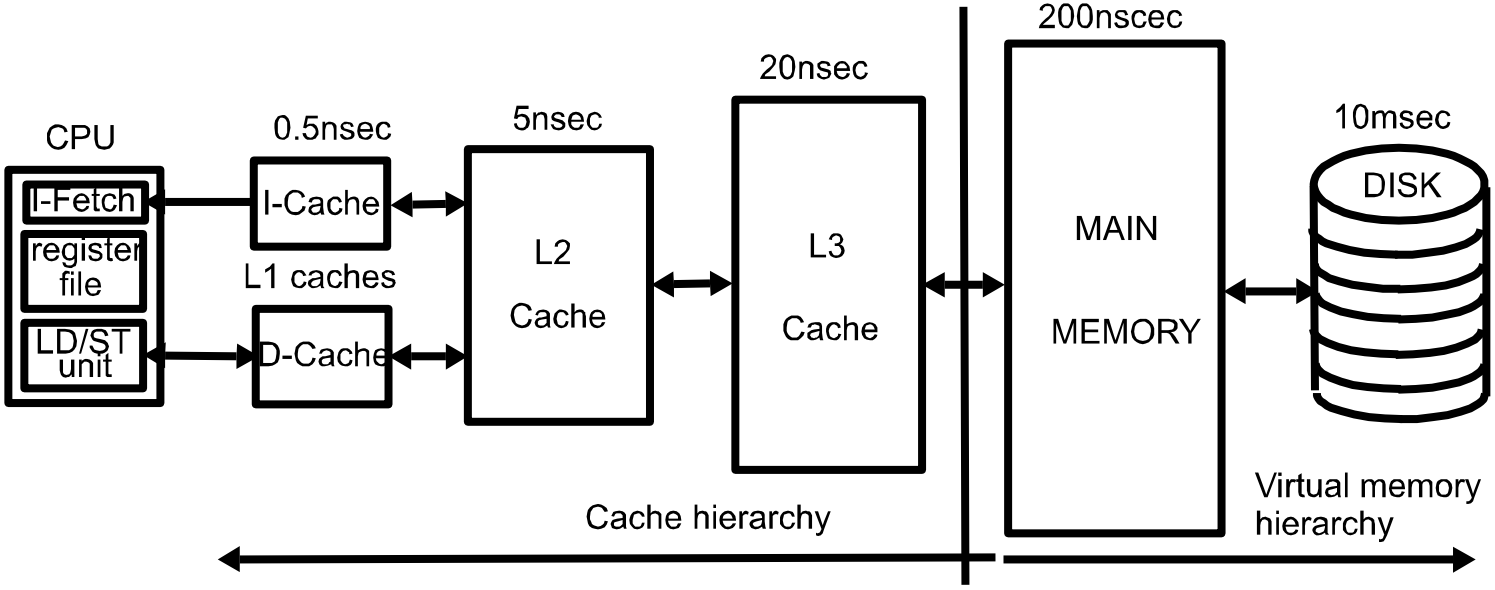
Storage	Speed	Cost	Capacity	Delay	Cost/GB
Static RAM	Fastest	Expensive	Smallest	0.5 – 2.5 ns	$1,000’s
Dynamic RAM	Slow	Cheap	Large	50 – 70 ns	$10’s
Hard disks	Slowest	Cheapest	Largest	5 – 20 ms	$0.1’s

Latency vs Bandwidth

- Latency
  - time it takes to get something
  - i.e., time it takes for water to get through the hose

- Bandwidth
  - how much you can get per second
  - i.e., how much water flowing through hose / sec

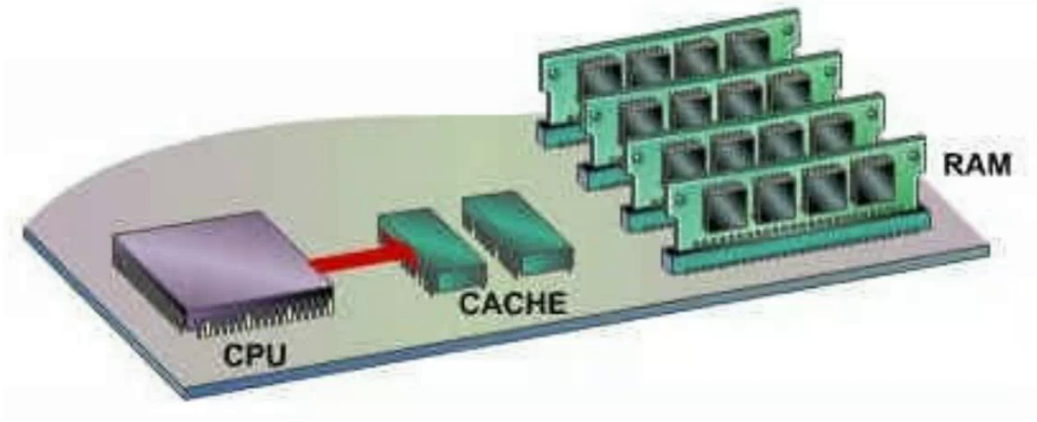
Memory Hierarchy



- small & fast memory → close to CPU
- large & slow memory → further away

Cache

- Cache
  - small, fast, expensive memory
  - between CPU and main memory
  - keeps copy of most frequently used data

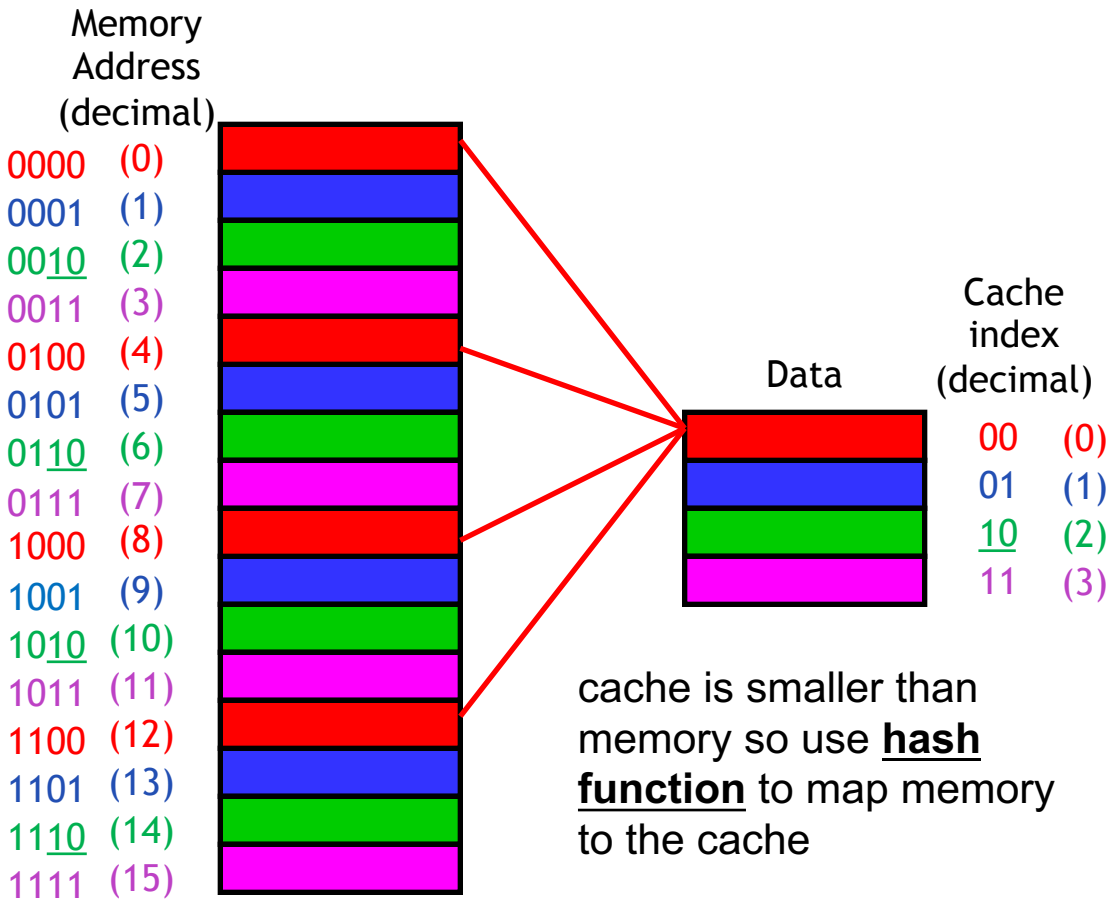


Locality

- Temporal Locality (time)
  - likely to access same data repeatedly over time
  - i.e., for loops

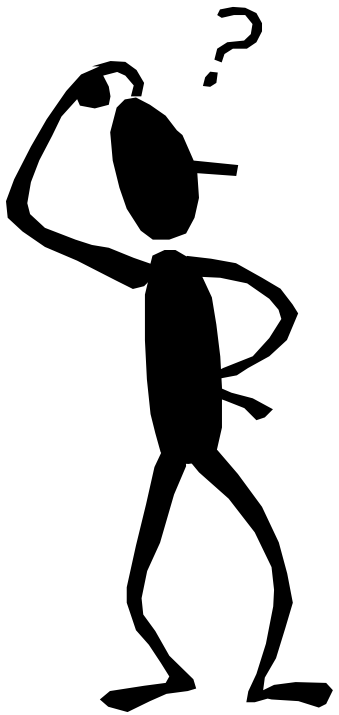
- Spatial Locality (space/area)
  - likely to access data close together in memory
  - i.e., array element accesses / instructions

Simple Cache Design



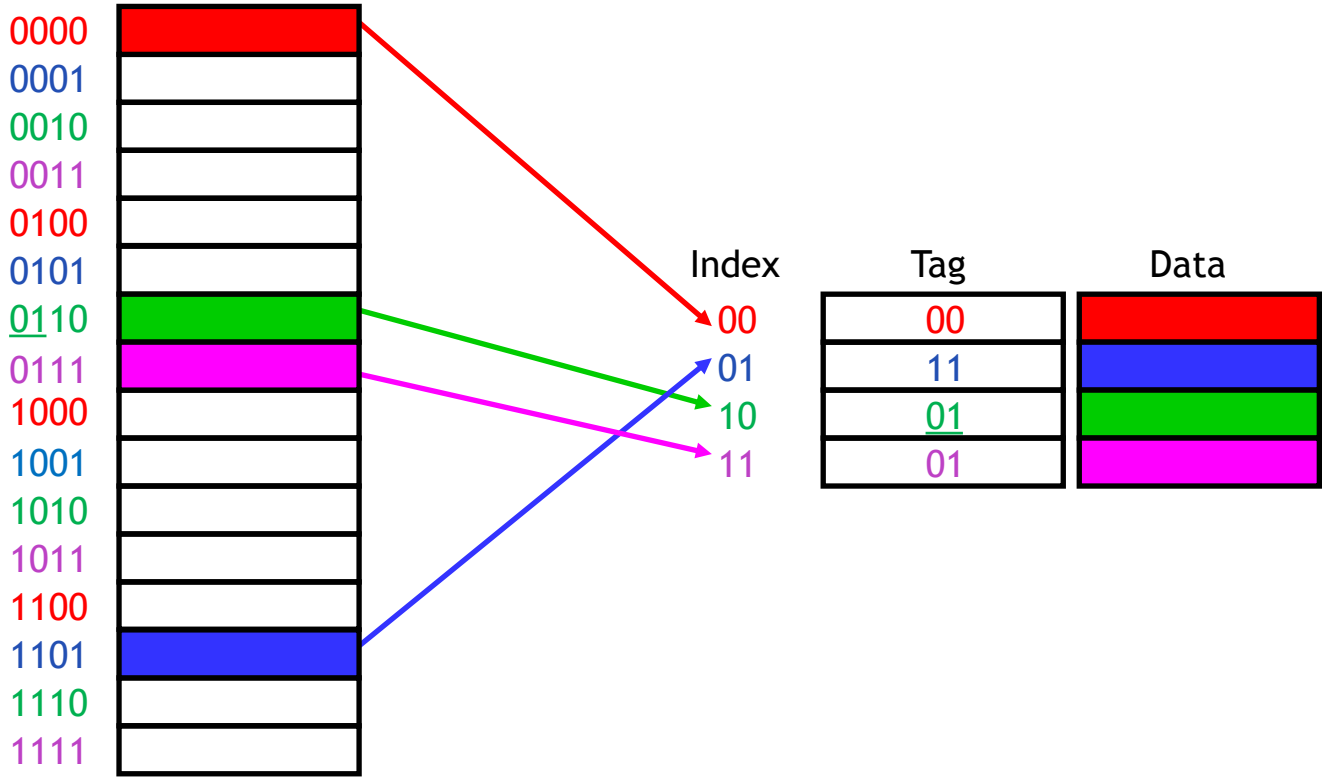
cache is smaller than memory so use **hash function** to map memory to the cache

but how do we know which memory address is in the cache?



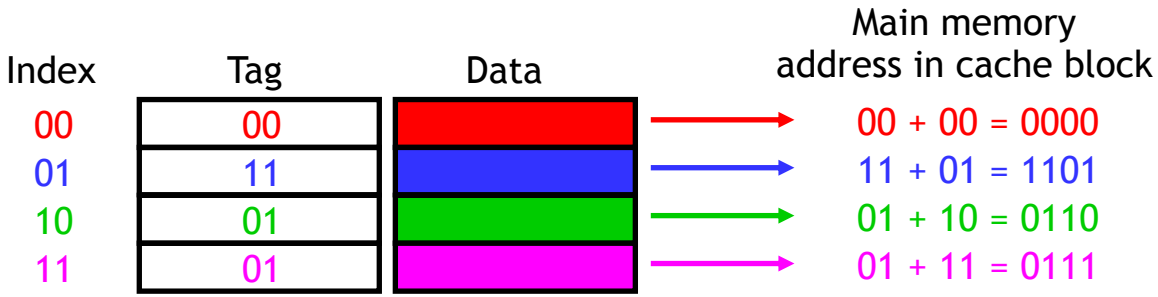
Adding Tags

- Need to add tags to cache to:
  - distinguish between different memory locations that map to same cache block



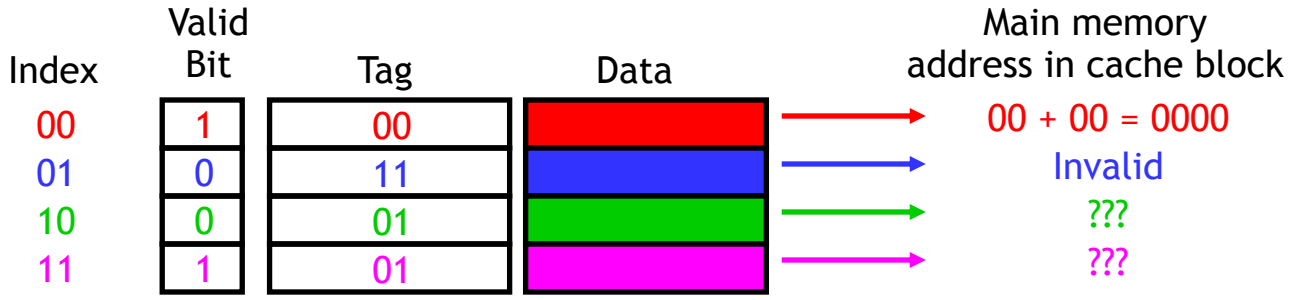
Putting it Together

- Concatenating:
  - cache block tags
  - block indices
- Gives us address in main memory



Valid Bit

- At start → cache is empty (no valid data)
- When system is initialized, all valid bits are set to 0
- When data loaded into a cache block:
  - corresponding valid bit is set to 1

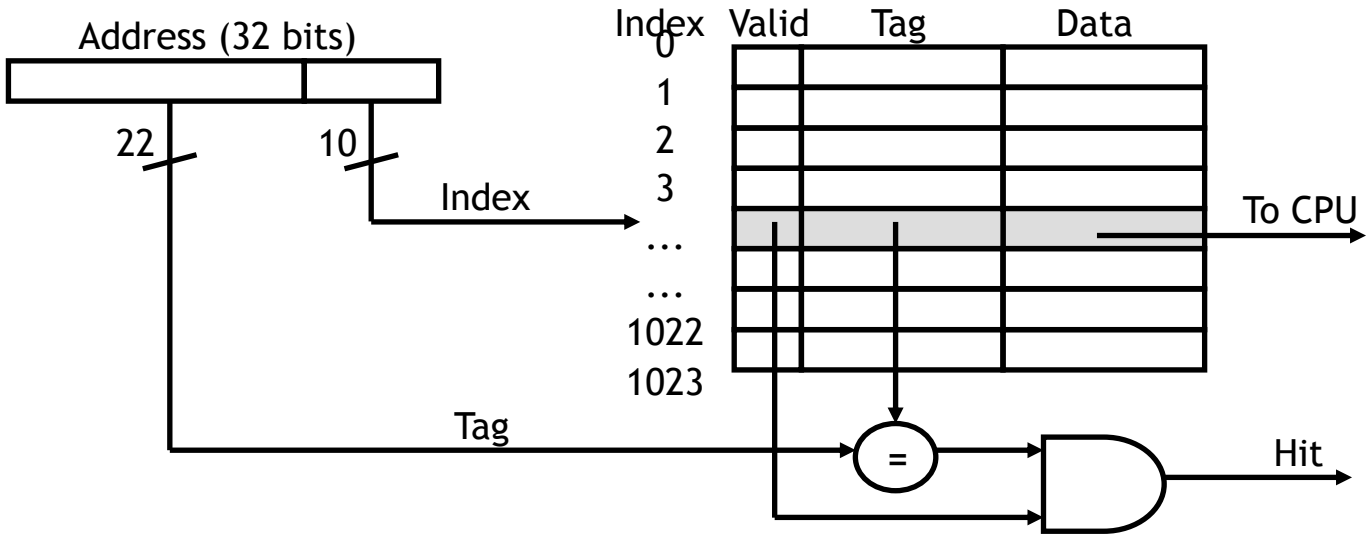


- So cache has bits to help find data within cache and verify validity

Reading From Cache (cache hit)

- Lowest *k* bits of the block address will index a block in the cache
  - block is valid (valid bit set)
  - tag matches upper (*m* − *k*) bits of *m*-bit address

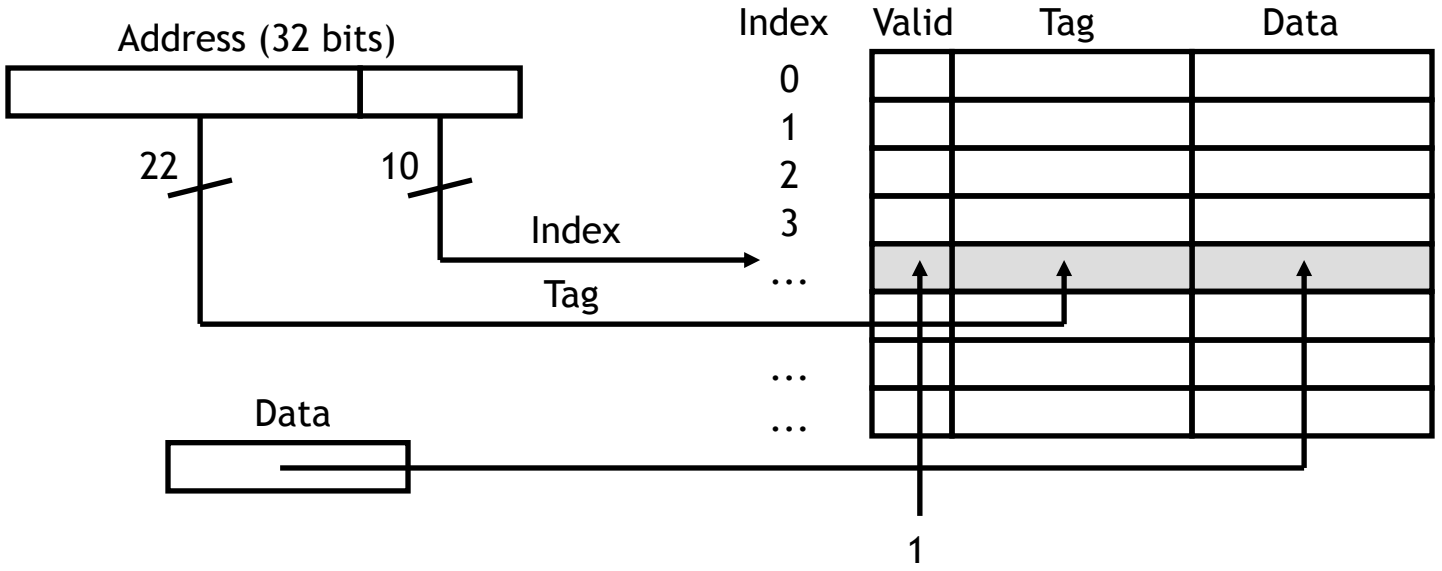
→ data sent to CPU



32-bit memory address and a 2^{10}-byte cache (2^{10}=1024)

Loading Block Into Cache

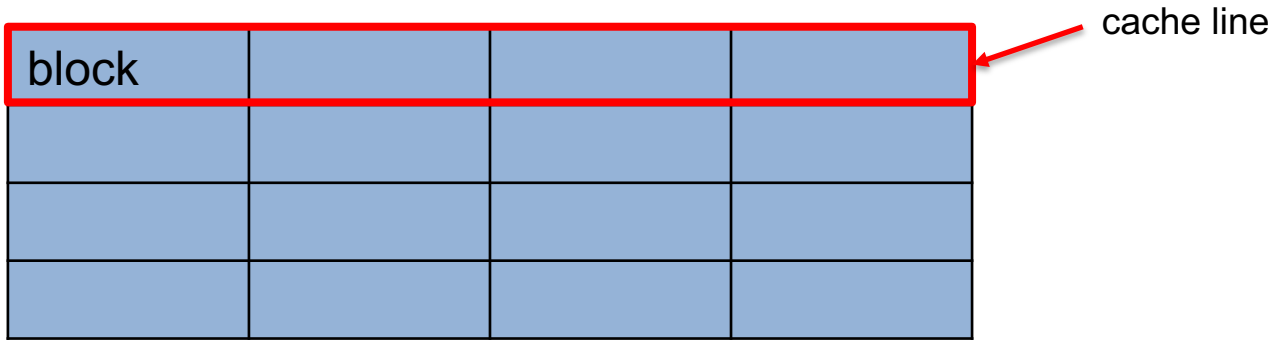
- After data read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest *k* bits of the block address specify a cache block
  - The upper (*m* – *k*) address bits are stored in the block's tag field
  - The data from main memory is stored in the block's data field
  - The valid bit is set to 1



Cache Sets and Ways

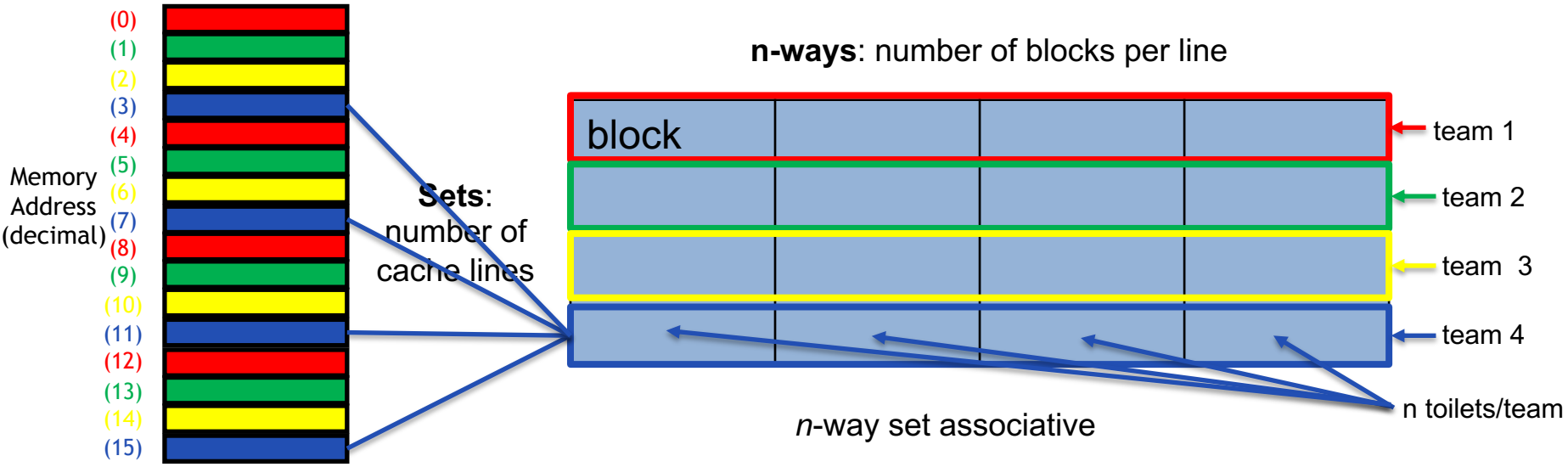
n-ways: number of blocks per line

Sets: number of cache lines



n-way set associative

Cache Sets and Ways: Example



- Example: Summer Camp!
  - Students go to summer camp (each student is a memory location)
  - Campers are grouped by teams and given a different color shirt (colors above) (each shirt color is different collection of memory locations)
  - Each team is assigned to a different bathroom facility (each set is different team's bathroom, each team hashed to cache-line/bathroom)
  - Each bathroom has same number individual toilet stalls (n = # toilets/team)

Direct-mapped Cache

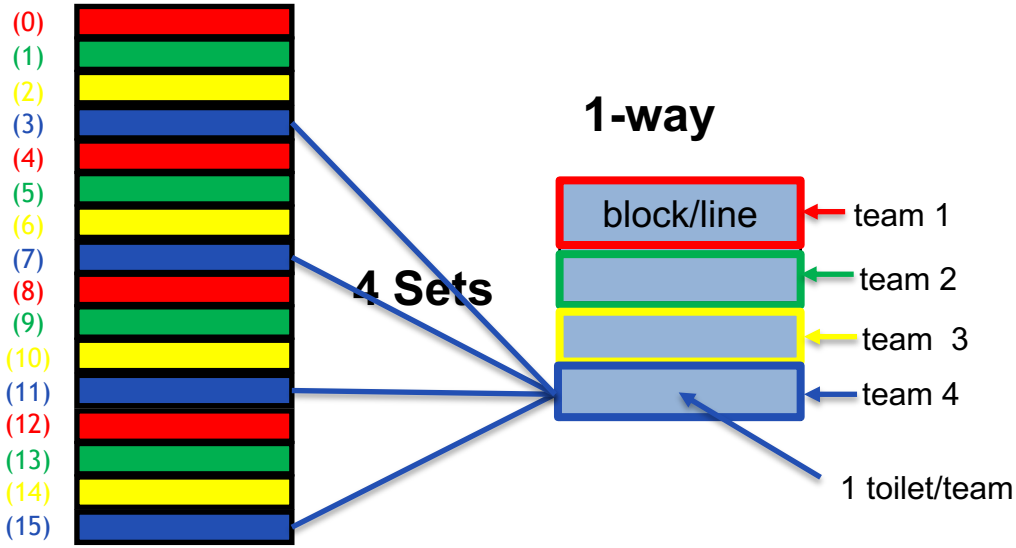
Direct mapped cache

- Each block maps to only one cache line
- aka "***1*-way set associative**"

Example:
- Cache size = 16 blocks
- 1-way
- 16-sets

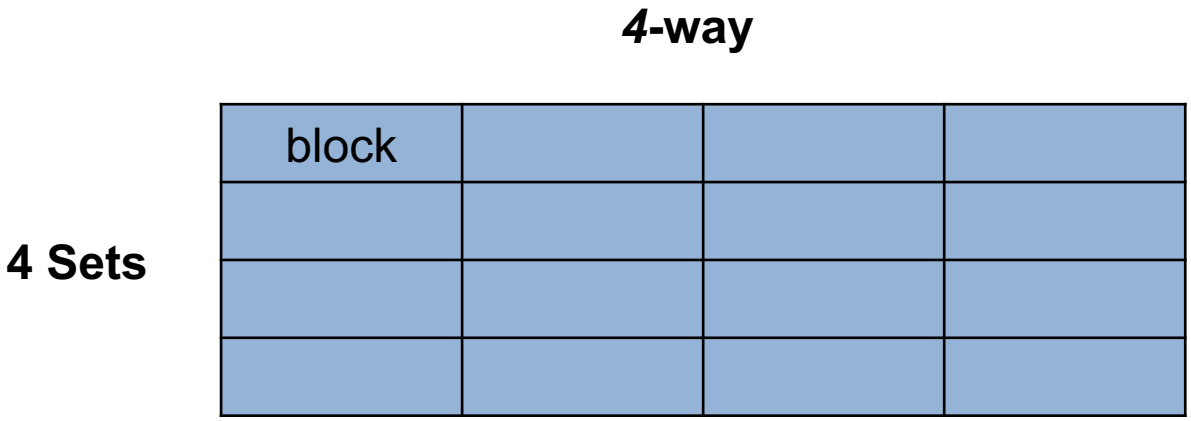
1-way set associative

Direct-mapped Cache: Example



- Example: Summer Camp!
  - only one toilet per team → a lot of “conflicts”

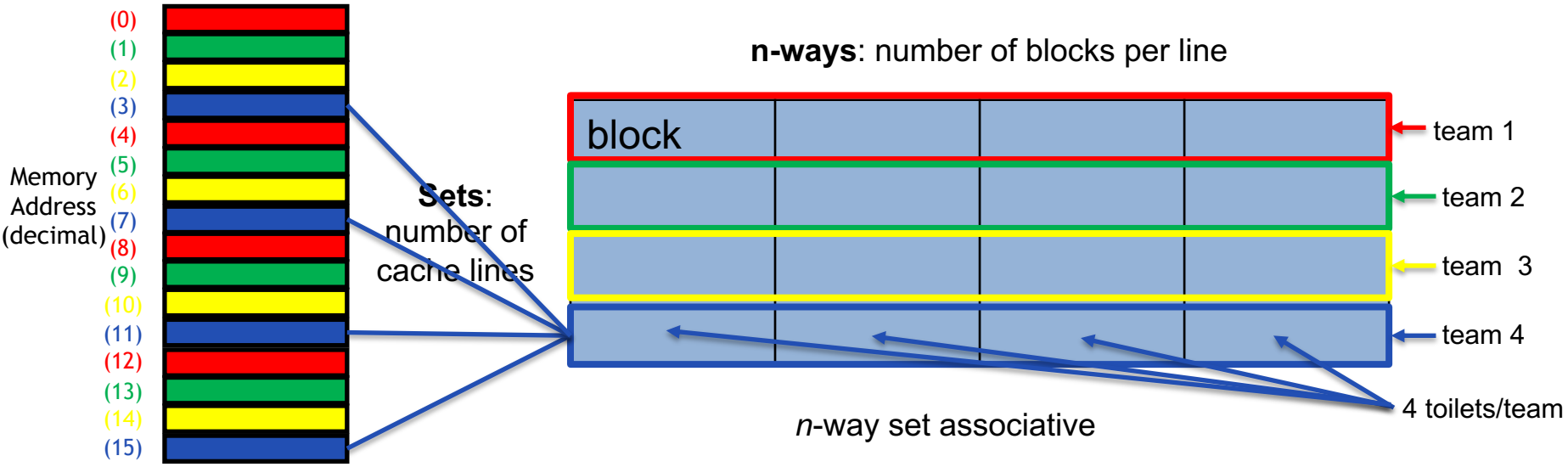
Set Associative Cache



Example:
- Cache size = 16 blocks
- 4-ways
- 4-sets

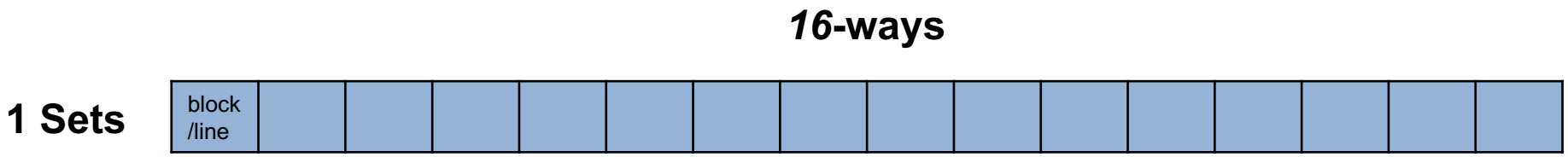
4-way set associative

Cache Sets and Ways: Example



- Example: Summer Camp!
  - four toilets per team → less "conflicts"

Fully Associative Cache



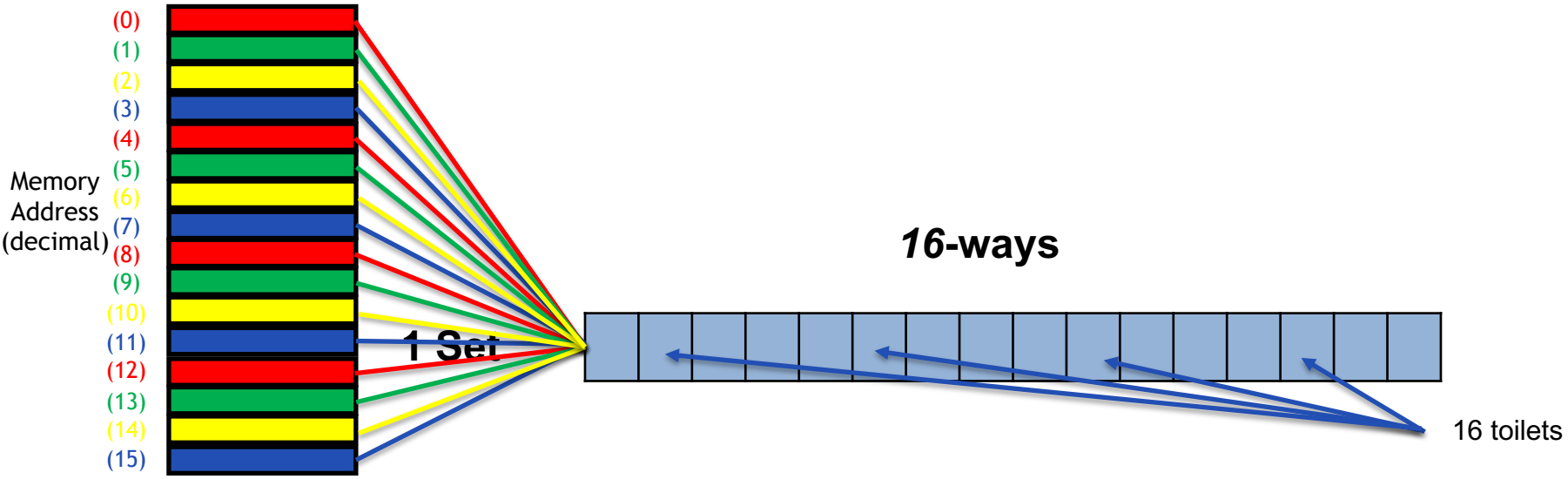
Fully Associative

- Each block can be mapped to any cache line
- aka "***m*-way set associative**"
  (where m = number of cache blocks)

Example:
- Cache size = 16 blocks
- 16-way
- 1-set

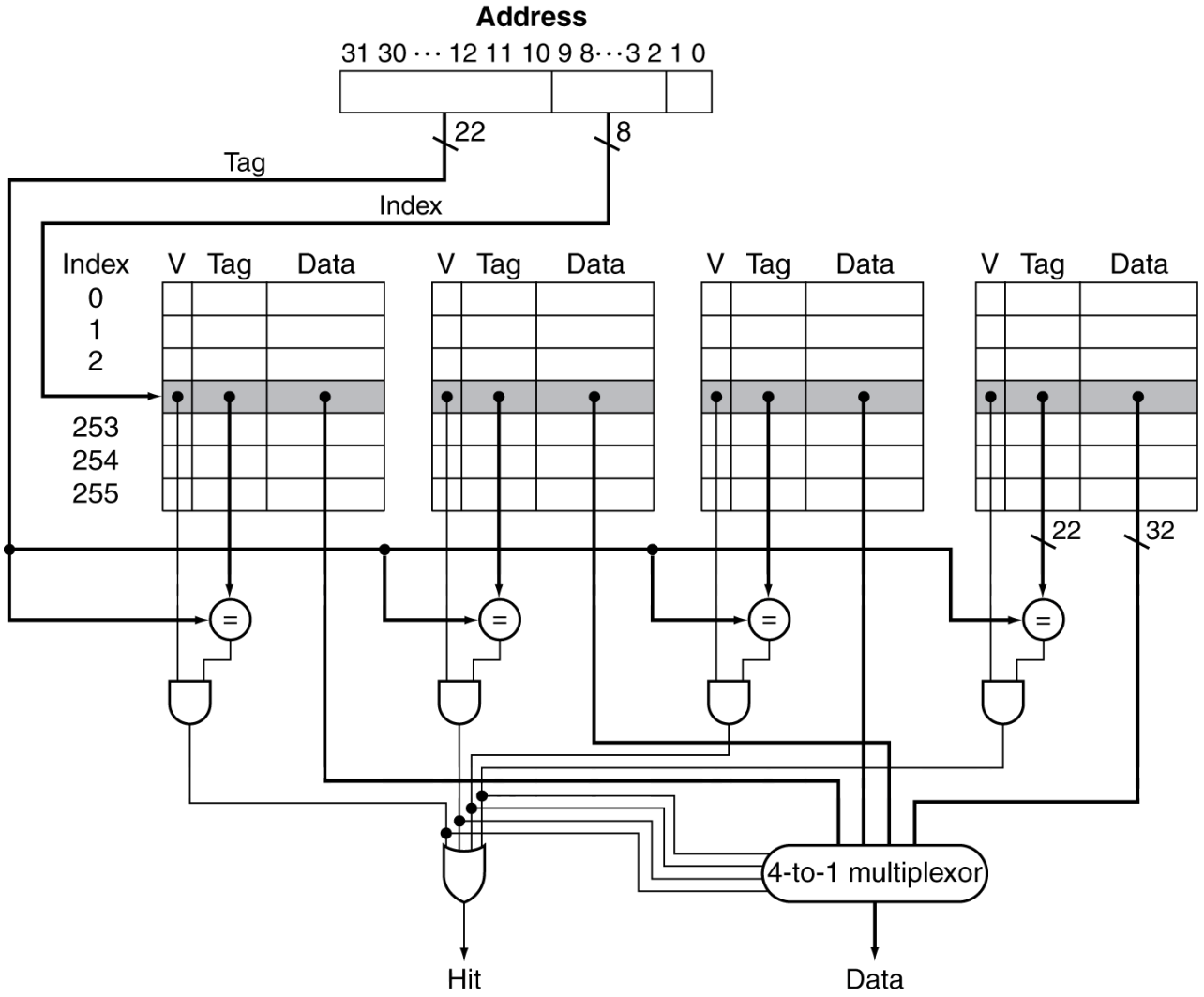
16-way set associative
(fully associative)

Fully Associative Cache: Example



- Example: Summer Camp!
  - only one bathroom, but a more toilets in the bathroom

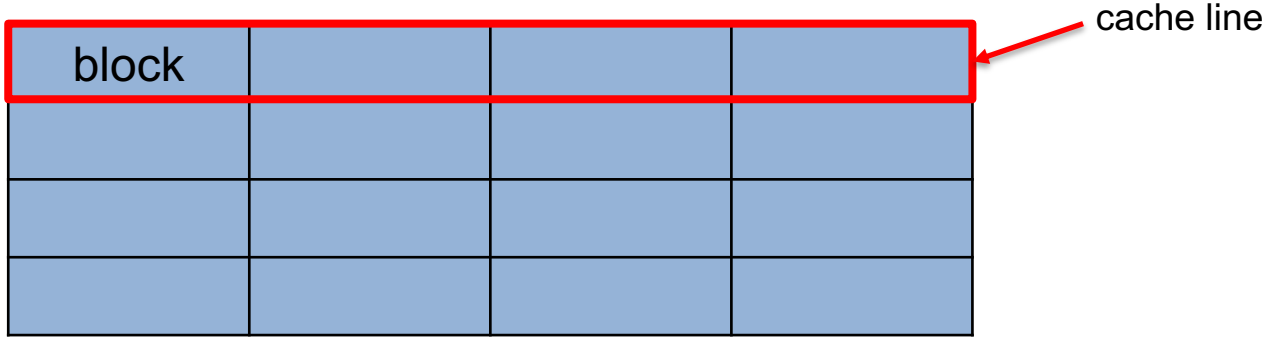
Set Associative Cache Organization



Cache Addressing

***n*-Ways**: Block can go anywhere in cache line

s-Sets: Block mapped by address (index)



Address

Tag bits (remainder) t = a-*i*-*b*	Index bits (sets) i = $\log_2 s$	Offset bits (block) b = $\log_2 bl$

m = # blocks = $\frac{cache\ size\ (c)}{block\ size\ (bl)}$

n = # ways

s = # sets = $\frac{\#\ blocks\ (m)}{\#\ ways\ (n)}$

bl = block size in bytes

c = cache size = *s* * *n* * *b*

a = # address bits

i = # index bits = $log_2 s$

b = # block bits = $log_2 bl$

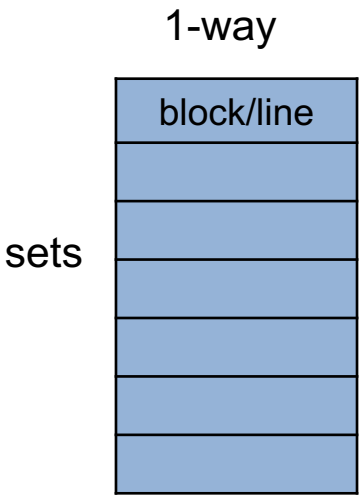
t = # tag bits = a - i - b

Cache Addressing: Example 1

Ex. 64KB cache, direct mapped, 16 B block

Address	Tag bits (remainder) t = a-*i*-*b*	Index bits (sets) i = $\log_2 s$	Offset bits (block) b = $\log_2 bl$
	16	12	4

$$m = \#\ blocks = \frac{cache\ size}{block\ size} = \frac{64\ KB}{16\ B} = 4K = 2^2 2^{10} = 2^{12}$$



$$s = \#\ sets = \frac{\#\ blocks\ (m)}{\#\ ways\ (n)} = \frac{2^{12}}{1} = 2^{12}$$

$$i = \#\ index\ bits = log_2 s = log_2 2^{12} = 12$$

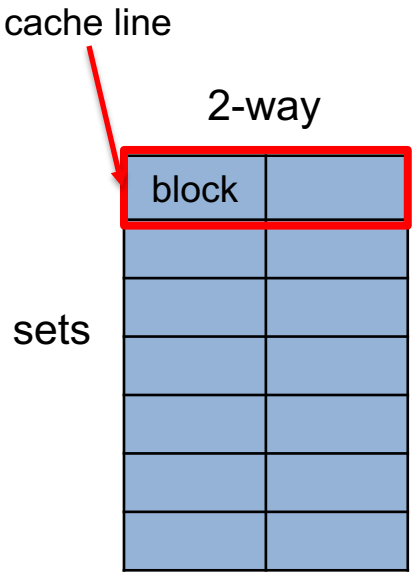
$$b = \#\ block\ bits = log_2 bl = log_2 16 = log_2 2^4 = 4$$

$$t = \#\ tag\ bits = a - i - b = 32 - 12 - 4 = 16$$

Cache Addressing : Example 2

Ex. 64KB cache, 2-way associative, 16 B block

Address	Tag bits (remainder) t = a-*i*-*b*	Index bits (sets) i = $\log_2 s$	Offset bits (block) b = $\log_2 bl$
	17	11	4



$$m = \#\ blocks = \frac{cache\ size}{block\ size} = \frac{64\ KB}{16\ B} = 4K = 2^2 2^{10} = 2^{12}$$

$$s = \#\ sets = \frac{\#\ blocks\ (m)}{\#\ ways\ (n)} = \frac{2^{12}}{2} = \frac{2^{12}}{2^1} = 2^{11}$$

$$i = \#\ index\ bits = log_2 s = log_2 2^{11} = 11$$

$$b = \#\ block\ bits = log_2 bl = log_2 16 = log_2 2^4 = 4$$

$$t = \#\ tag\ bits = a - i - b = 32 - 11 - 4 = 17$$

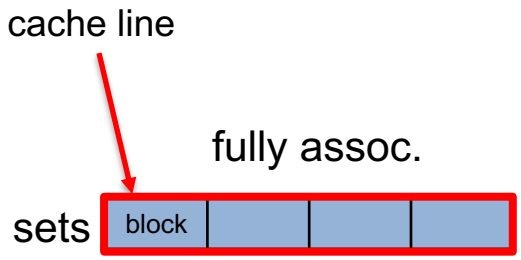
Cache Addressing : Example 3

Ex. 64KB cache, fully associative, 16 B block

Address

Tag bits (remainder) t = a-*i*-*b*	Index bits (sets) i = $\log_2 s$	Offset bits (block) b = $\log_2 bl$
28	0	4

$$m = \#\ blocks = \frac{cache\ size}{block\ size} = \frac{64\ KB}{16\ B} = 4K = 2^2 2^{10} = 2^{12}$$



$$s = \#\ sets = ? \quad \rightarrow 1$$

$$i = \#\ index\ bits = log_2 s = log_2 1 = 0$$

$$b = \#\ block\ bits = log_2 bl = log_2 16 = log_2 2^4 = 4$$

$$t = \#\ tag\ bits = a - i - b = 32 - 0 - 4 = 28$$

Measuring Cache Performance

$$memory\ stall\ cycles = \frac{memory\ accesses}{program} * miss\ rate * miss\ penalty$$

- When do we access memory?
  - Instruction Fetch (I$)
  - Data (D$)
    - Load
    - Store

$$memory\ stall\ cycles\ (I\$) = 1.00 * miss\ rate * miss\ penalty$$

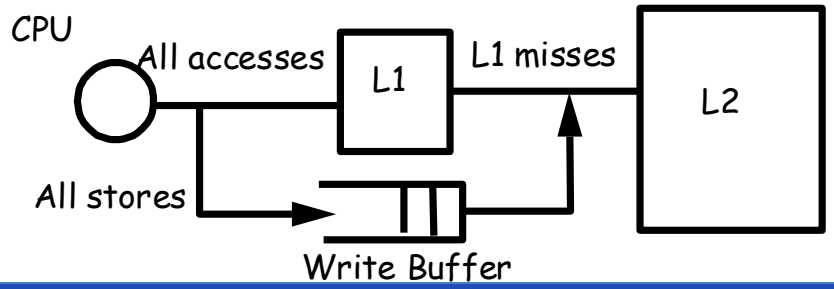
$$memory\ stall\ cycles\ (D\$) = \frac{num\ loads/stores}{num\ instructions} * miss\ rate * miss\ penalty$$

Cache Performance: Example

- Example:
  - Given:
    - I-cache miss rate = 2%
    - D-cache miss rate = 4%
    - Miss penalty = 100 cycles
    - Base (“ideal) CPI = 2
    - Load & stores are 36% of instructions
  - Miss cycles (penalty) per instruction
    - I-cache: $0.02 \times 100 = 2$
    - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
  - Actual CPI = $2 + 2 + 1.44 = 5.44$
  - Speedup:
    - Ideal CPU is 5.44/2 =2.72 times faster

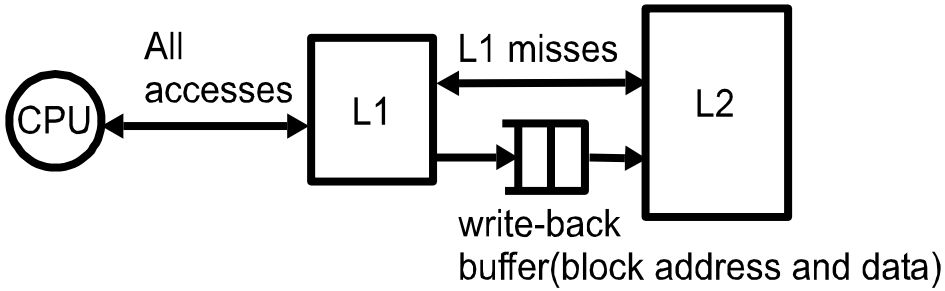
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1 × 100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full



Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first



Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don't fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block