

CSE 140

Computer Architecture

Lecture 7 – Memory: Cache (2)

Announcement

- ▶ Lab #2 this week
 - Due date extended (2 weeks)
- ▶ Project #1
 - Due 10/11 (Monday) at 11:59pm
 - Start as soon as possible
- ▶ Reading assignment #4
 - Chapter 5.7 – 5.8
 - Do all **Participation Activities** in each section
 - Access through CatCourses
 - Due Thursday (9/26) at 11:59pm
 - Basic idea of using cache (at CatCourses, under Files)
 - ***How L1 and L2 CPU Caches Work.pdf***

Direct-Mapped Cache Terminology

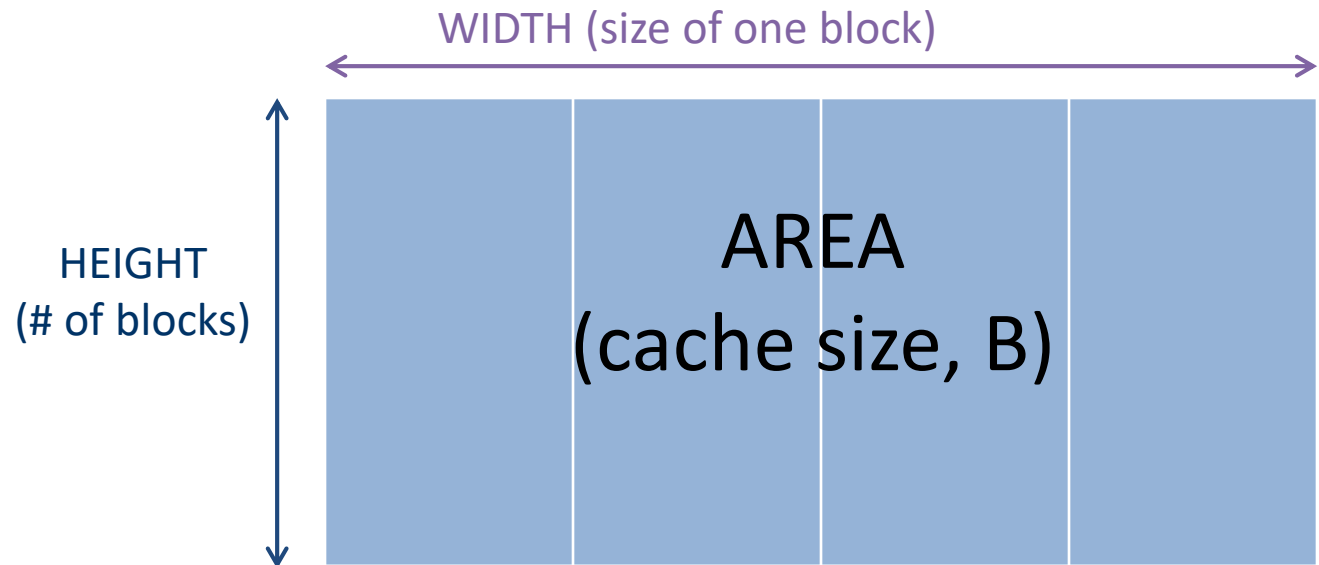
- ▶ All fields are read as unsigned integers
- ▶ **Index**
 - specifies the cache index (which “row” or block of the cache we should look in)
- ▶ **Offset**
 - once we’ve found correct block, specifies which byte within the block we want
- ▶ **Tag**
 - the remaining bits after offset and index are determined; these are used to distinguish between all the memory addresses that map to the same location



AREA (cache size, B)

= HEIGHT (# of blocks) * WIDTH (size of one block)

Tag	Index	Offset
-----	-------	--------



Direct-Mapped Cache Example (1/3)

- ▶ Suppose we have a 8B of data in a direct-mapped cache with 2 byte blocks
 - Sound familiar?
- ▶ Determine the number of bits in the tag, index and offset fields if we're using a 32-bit architecture
- ▶ **Offset**
 - need to specify correct byte within a block
 - block contains 2 bytes
 - $= 2^1$ bytes
 - need **1 bit** to specify correct byte

Direct-Mapped Cache Example (2/3)

- ▶ **Index:** (~index to an “array of blocks”)
 - need to specify correct number of block in cache
 - Cache contains 8 B = 2^3 bytes
 - block contains 2 B = 2^1 bytes
 - # blocks/cache
 - = $\frac{\text{bytes/cache}}{\text{bytes/block}}$
 - = $\frac{2^3 \text{ bytes/cache}}{2^1 \text{ bytes/block}}$
 - = 2^2 blocks/cache
 - need **2 bits** to specify this many blocks

Direct-Mapped Cache Example (3/3)

- ▶ **Tag:** use remaining bits as tag
 - tag length = addr length – offset – index
= 32 - 1 - 2 bits
= 29 bits
 - so tag is leftmost **29 bits** of memory address
- ▶ Why not full 32 bit address as tag?
 - Index must be same for every address within a block, so it's redundant in tag check, thus can leave off to save memory
- ▶ Each row has Valid and Dirty bit

Caching Terminology

- ▶ When reading memory, 3 things can happen:
 - cache hit:
 - cache block is valid and contains proper address, so read desired word
 - cache miss:
 - nothing in cache at appropriate block, so fetch from memory
 - cache miss, block replacement:
 - wrong data is in cache at appropriate block, so discard it and fetch desired data from memory (cache always copy)

Accessing data in a direct-mapped cache

- ▶ Ex.: 16KB of data, direct-mapped, 4 word blocks (16B)
 - Can you work out height, width, area?
- ▶ Read 4 addresses (Hex)

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014

...	...
00000010	a
00000014	b
00000018	c
0000001C	d
...	...
00000030	e
00000034	f
00000038	g
0000003C	h
...	...
00008010	i
00008014	j
00008018	k
0000801C	l
...	...

Memory

Accessing data in a direct-mapped cache

- ▶ 4 Addresses:
 - 0x00000014, 0x0000001C, 0x00000034, 0x00008014
- ▶ 4 Addresses divided (for convenience) into **Tag**, **Index**, **Byte Offset** fields

Tag	Index	Offset
000000000000000000000000	00000000001	0100
000000000000000000000000	00000000001	1100
000000000000000000000000	00000000011	0100
000000000000000000000010	00000000001	0100

16 KB Direct Mapped Cache, 16B blocks

- ▶ **Valid bit:** determines whether anything is stored in that row (start with all entries invalid)

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read 0x00000014

Tag

Index

Offset

000000000000000000000000 000000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read block 1

Tag

Index

Offset

000000000000000000000000 0000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

No valid data

Tag

Index

Offset

000000000000000000000000 0000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Load data into cache, set tag and valid

Tag

Index

Offset

00000000000000000000

00000000001

0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read from cache at offset

Tag

Index

Offset

000000000000000000000000 000000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Return word b

Read 0x0000001C

Tag

Index

Offset

000000000000000000000000 00000000001 1100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Index is valid

Tag

Index

Offset

000000000000000000000000 0000000001 1100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Tag matches

Tag

Index

Offset

00000000000000000000

00000000001

1100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read from cache at offset

Tag

Index

Offset

000000000000000000000000 000000000001 1100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Return word d

Read 0x00000034

Tag

Index

Offset

000000000000000000000000 00000000011 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read block 3

Tag

Index

Offset

000000000000000000000000 0000000011 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

No valid data

Tag

Index

Offset

000000000000000000000000 0000000011 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Load data, read cache at offset

Tag				Index	Offset	
000000000000000000000000				000000000011	0100	
Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Return word f

Read 0x00008014

Tag

Index

Offset

000000000000000000000010 000000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read block 1, valid data

Tag

Index

Offset

0000000000000000000010 0000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Tag not matched!

Tag

Index

Offset

000000000000000000010

000000000001

0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	0	d	c	b	a
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Miss: replace block 1

Tag

Index

Offset

00000000000000000010

000000000001

0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Read cache at offset

Tag

Index

Offset

0000000000000000000010 000000000001 0100

Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					
...				...		
1022	0					
1023	0					

Return word j

Do these examples yourself

- ▶ Read address 0x00000030
- ▶ Read address 0x0000001C
- ▶ Cache: Hit, Miss, or Miss with replace?
- ▶ Returned values: a, b, c, ..., k, l?

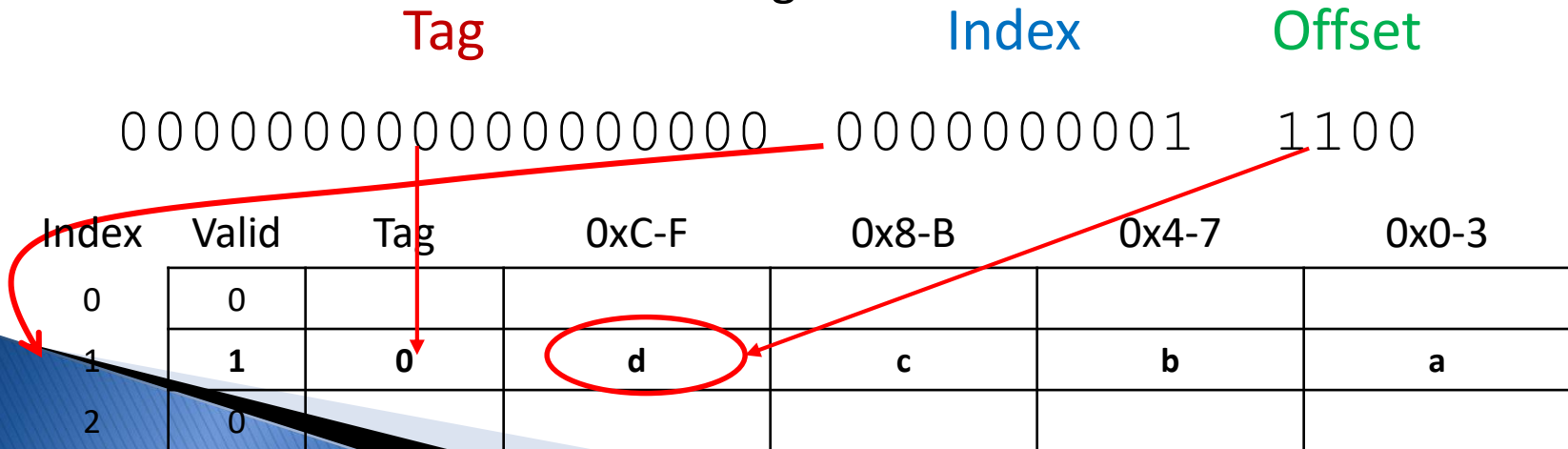
Index	Valid	Tag	0xC-F	0x8-B	0x4-7	0x0-3
0	0					
1	1	2	l	k	j	i
2	0					
3	1	0	h	g	f	e
4	0					
5	0					
6	0					
7	0					

Answers

- ▶ 0x00000030: hit
 - Index = 3, Tag matches
 - Offset = 0, returned value = e
- ▶ 0x0000001c: miss
 - Index = 1, Tag mismatch, so replace from memory
 - Offset = 0xc, value = d

Summary: Direct-Mapped Cache

- ▶ Mechanism for transparent movement of data among levels of a storage hierarchy
 - set of address/value bindings
 - address \Rightarrow index to set of candidates
 - compare desired address with tag
 - service hit or miss
 - load new block and binding on miss



What to do on a write hit?

▶ Write-through

- update the word in cache block and corresponding word in memory
 - PRO: read misses cannot result in writes
 - CON: processor held up on writes unless writes buffered

▶ Write-back

- update word in cache block
- allow memory word to be “stale”
- add ‘dirty’ bit to each block indicating that memory needs to be updated when block is replaced
- OS flushes cache before I/O...
 - PRO: repeated writes not sent to DRAM processor not held up on writes
 - CON: More complex Read miss may require write-back of dirty data

Block Size Tradeoff (1/3)

- ▶ Benefits of Larger Block Size
 - **Spatial Locality:** if we access a given word, we're likely to access other nearby words soon (reduce miss rate)
 - Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well
 - Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

▶ Drawbacks of Larger Block Size

- Larger block size means **larger miss penalty**
 - on a miss, takes longer time to load a new block from next level
- If block size is too big relative to cache size, then there are too few blocks
 - Result: miss rate goes up

▶ In general, minimize

Average Memory Access Time (AMAT)

$$= \text{Hit Time} + \text{Miss Penalty} \times \text{Miss Rate}$$

Block Size Tradeoff (3/3)

▶ Hit Time

- time to find and retrieve data from current level cache

▶ Miss Penalty

- average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)

▶ Hit Rate

- % of requests that are found in current level cache

▶ Miss Rate

- $1 - \text{Hit Rate}$

Extreme Example: One Big Block

Valid Bit

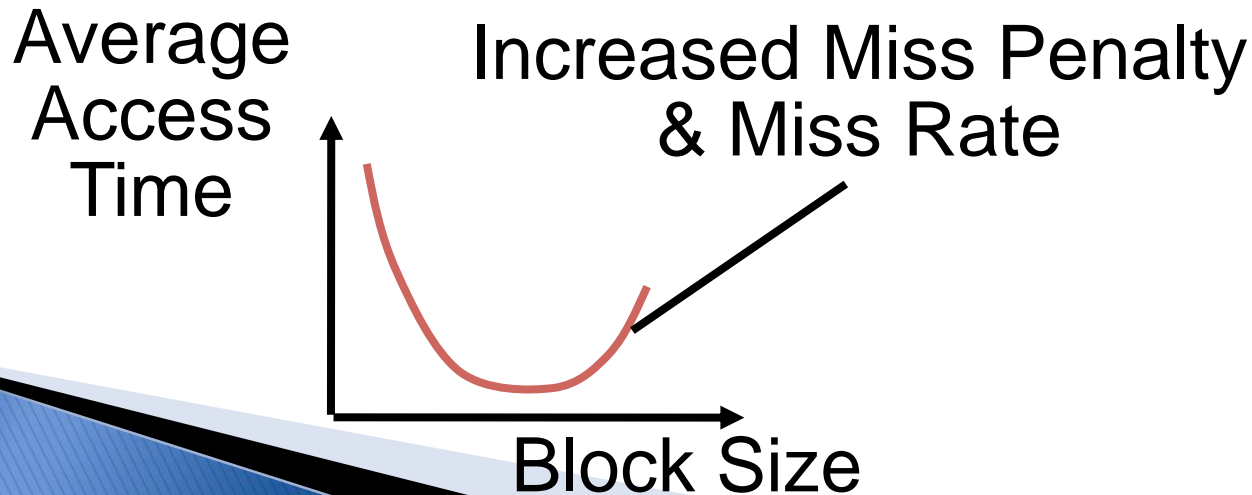
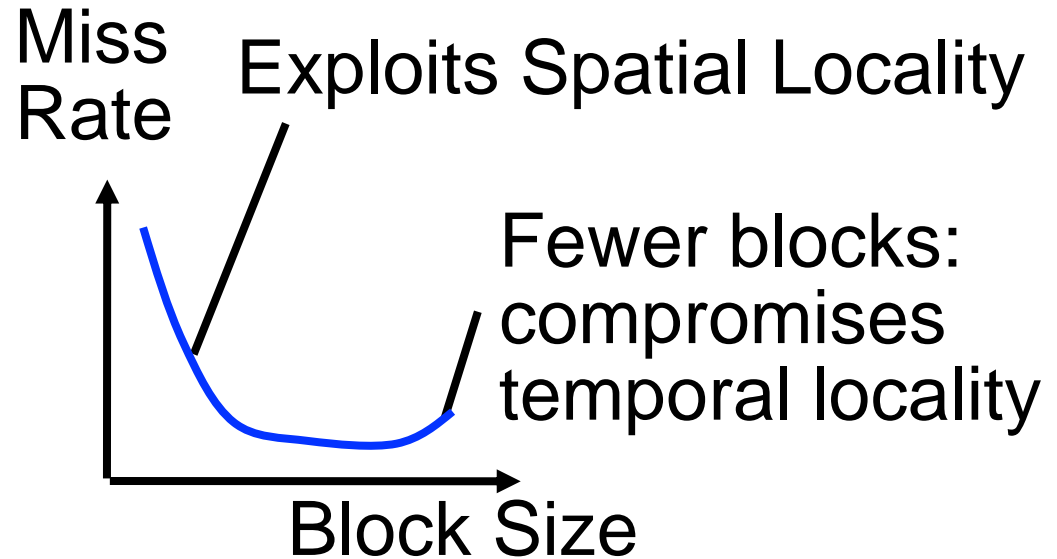
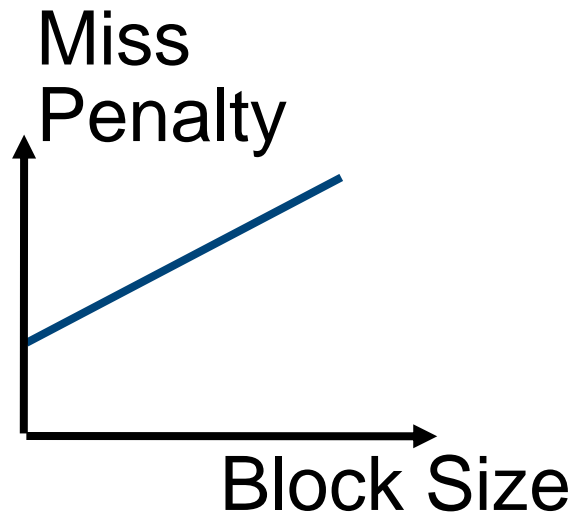
Tag

Cache Data



- ▶ Cache Size = 4 bytes Block Size = 4 bytes
 - Only **ONE** entry (row) in the cache!
- ▶ If item accessed, likely accessed again soon
 - But unlikely will be accessed again immediately!
- ▶ The next access will likely to be a miss again
 - Continually loading data into the cache but discard data (force out) before use it again
 - Nightmare for cache designer: **Ping Pong Effect**

Block Size Tradeoff Conclusions



Types of Cache Misses (1/2)

- ▶ “Three Cs” Model of Misses
- ▶ 1st C: **Compulsory Misses**
 - occur when a program is first started
 - cache does not contain any of that program’s data yet, so misses are bound to occur (valid bit = 0)
 - can’t be avoided easily, so won’t focus on these in this course

Types of Cache Misses (2/2)

▶ 2nd C: Conflict Misses

- miss that occurs because two distinct memory addresses map to the same cache location
- two blocks (which happen to be mapped to the same location) can keep overwriting each other
- big problem in direct-mapped caches
- how do we lessen the effect of these?

▶ Dealing with Conflict Misses

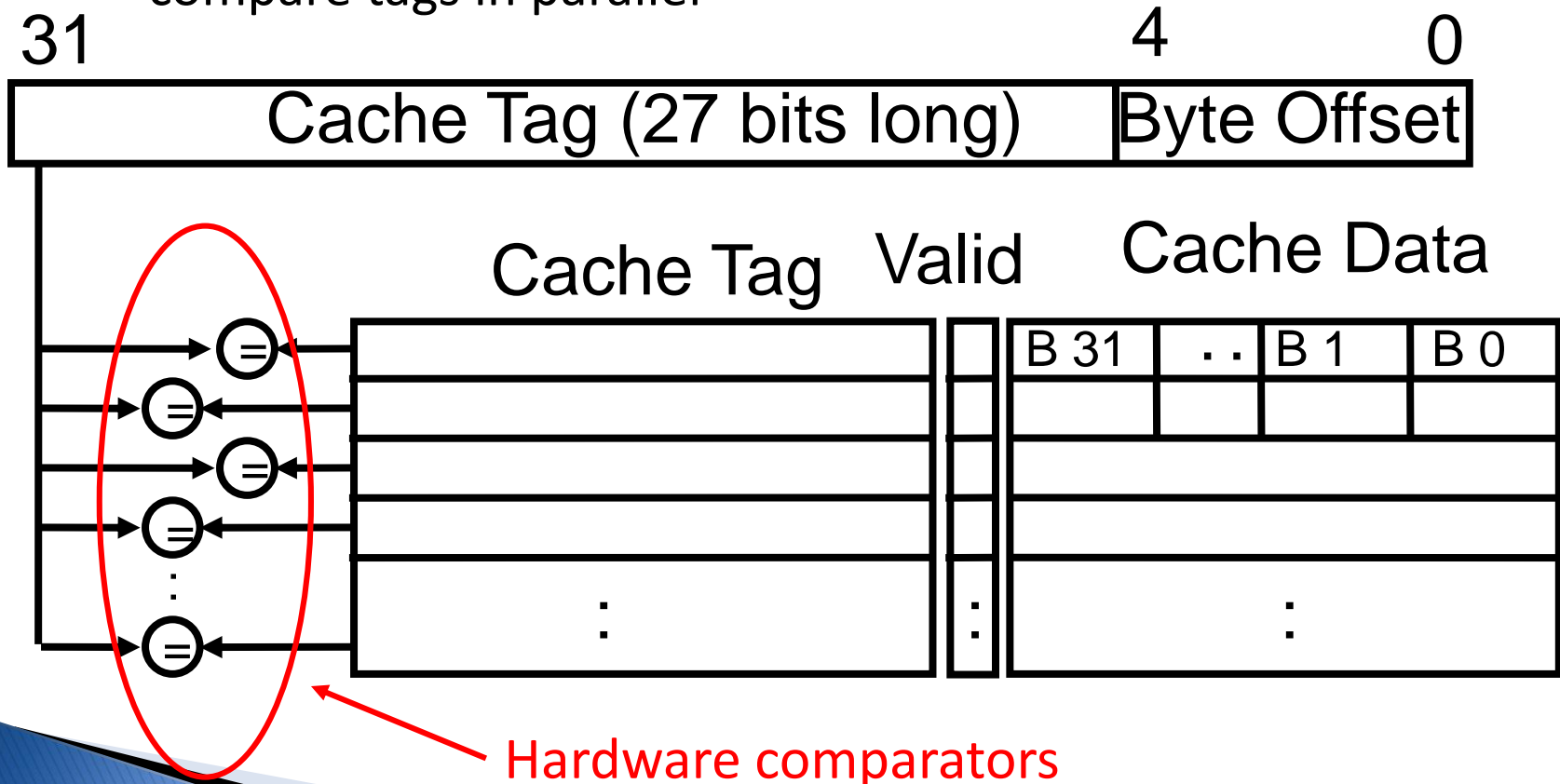
- Solution 1: Make the cache size bigger
 - Fails at some point
- Solution 2: Multiple distinct blocks can fit in the same cache Index?

Fully Associative Cache (1/3)

- ▶ Memory address fields:
 - Tag: same as before
 - Offset: same as before
 - Index: non-existent
- ▶ What does this mean?
 - no “rows”: any block can go anywhere in the cache
 - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- ▶ Fully Associative Cache (e.g., 32 B block)
 - compare tags in parallel



Fully Associative Cache (3/3)

- ▶ Benefit of Fully Assoc. Cache
 - No Conflict Misses (since data can go anywhere)
- ▶ Drawbacks of Fully Assoc. Cache
 - Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: **infeasibly high cost**

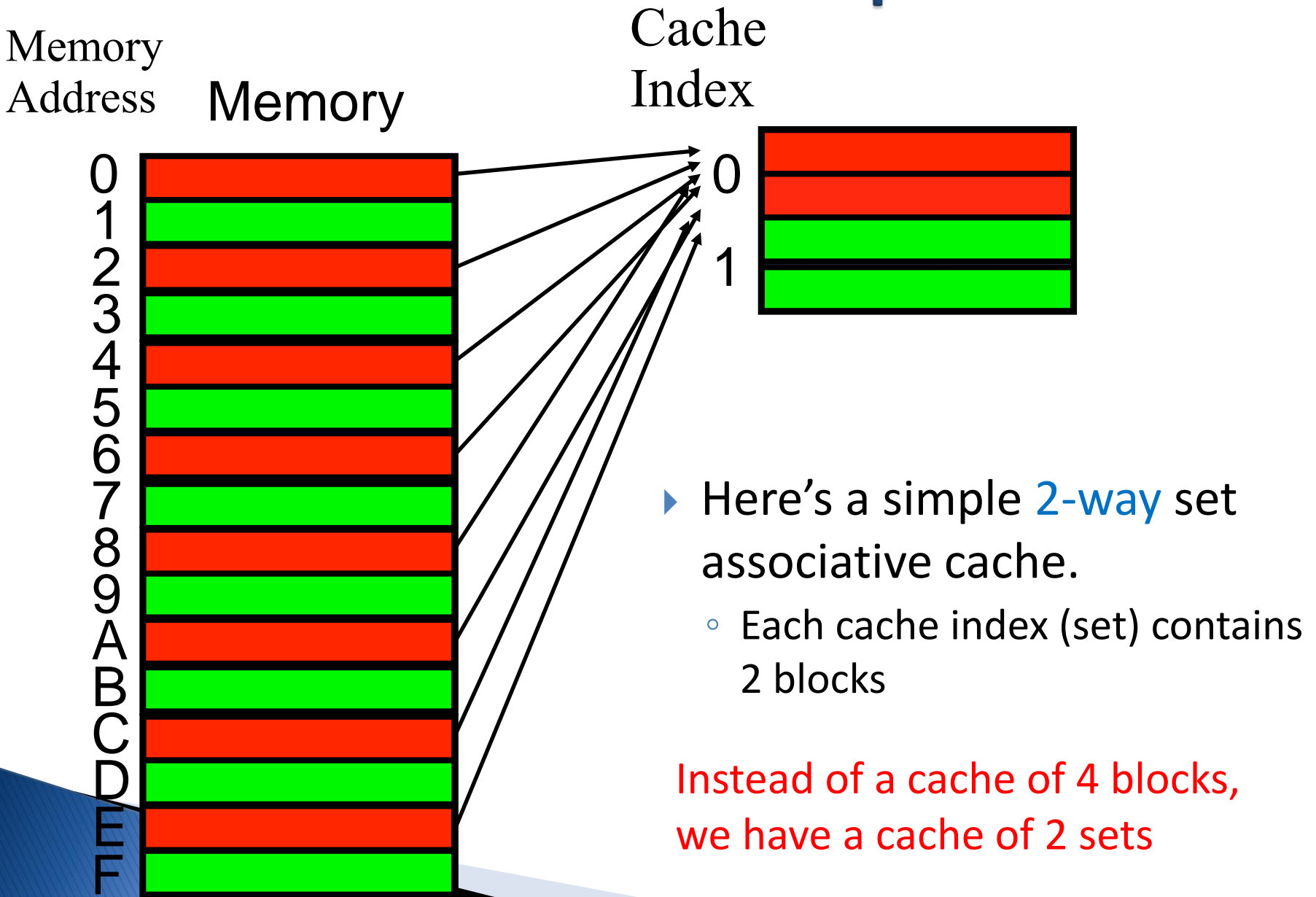
Final Type of Cache Miss

- ▶ 3rd C: Capacity Misses
 - miss that occurs because the cache has a limited size
 - miss that would not occur if we increase the size of the cache
 - sketchy definition, so just get the general idea
- ▶ This is the primary type of miss for Fully Associative caches.
 - Cache size < memory size

N-Way Set Associative Cache (1/3)

- ▶ A “hybrid” model
- ▶ Memory address fields:
 - **Tag**: same as before
 - **Offset**: same as before
 - **Index**: points us to the correct “group” (called a **set** this case)
- ▶ So what’s the difference?
 - each set contains multiple blocks
 - once we’ve found correct set, we must compare with all tags in that set to find our data (with hardware comparators)

Associative Cache Example

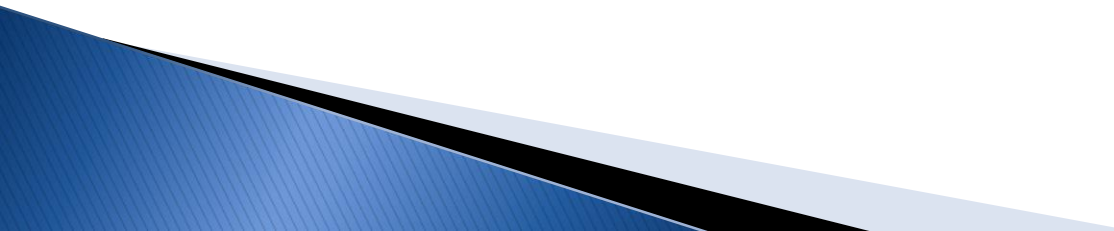


N-Way Set Associative Cache (2/3)

▶ Basic Idea

- cache is direct-mapped w/respect to sets
- each set is fully associative with N blocks in it

▶ Given memory address:

- Find correct set using Index value.
 - Compare Tag with all Tag values in the determined set.
 - If a match occurs, hit!, otherwise a miss.
 - Finally, use the offset field as usual to find the desired data within the block.
- 

N-Way Set Associative Cache (3/3)

- ▶ What's so great about this?
 - even a 2-way set assoc. cache avoids a lot of conflict misses
 - hardware cost isn't that bad: only need N comparators
- ▶ In fact, for a cache with M blocks,
 - it's **Direct-Mapped** if it's 1-way set assoc.
 - it's **Fully Assoc** if it's M-way set assoc.
 - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit

