

Cache Design

CS 1541

Wonsun Ahn

Oracle Cache

- CPU Cycles = CPU Compute Cycles + Memory Stall Cycles
- **Oracle cache**: a cache that never misses
 - In effect, **Memory Stall Cycles == 0**
 - Impossible, since even with infinite capacity, there are still cold misses
 - But useful to set **bounds** on performance
- Real caches may approach performance of oracle caches but can't exceed
- What metric can we use to compare and evaluate real cache designs?
 - AMAT (Average Memory Access Time)

AMAT (Average Memory Access Time)

- **AMAT** (Average Memory Access Time) is defined as follows:
 - **AMAT = hit time + (miss rate × miss penalty)**
 - **Hit time:** time to get the data from cache when we hit
 - **Miss rate:** what percentage of cache accesses we miss
 - **Miss penalty:** time to get the data from lower memory when we miss
 - Shouldn't it be **hit rate × hit time**?
 - Hit time is incurred regardless of hit or miss
 - It is more aptly called access time (the time to search for the data)
- Hit time, miss rate, miss penalty are the 3 components of a cache design
 - When evaluating a cache design, we need to consider all 3
 - Cache designs trade-off one for the other
 - E.g. a large cache trade-offs longer hit time for smaller miss rate
 - Whether trade-off is beneficial depends on the resulting AMAT

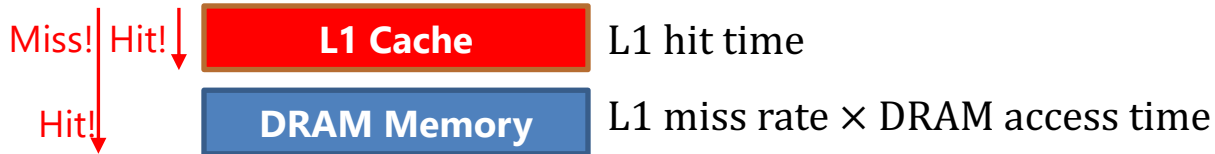
Cache Design Parameter 1: Number of Levels

AMAT of No-Cache vs. L1 Cache

- For a CPU with no caches:
 - $AMAT(\text{none}) = \text{DRAM access time}$



- For a single-level cache (L1 cache):
 - $AMAT(L1) = \text{L1 hit time} + (\text{L1 miss rate} \times \text{DRAM access time})$



AMAT of No-Cache vs. L1 Cache

- For L1 Cache to be worth it, $AMAT(\text{None}) > AMAT(\text{L1})$ needs to be true.

DRAM Memory

DRAM access time

>?

L1 Cache

L1 hit time

DRAM Memory

L1 miss rate \times DRAM access time

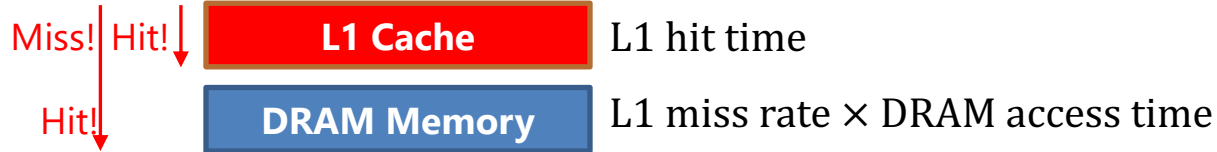
- $AMAT(\text{None}) > AMAT(\text{L1})$?
 - $\text{DRAM access time} > \text{L1 hit time} + \text{L1 miss rate} \times \text{DRAM access time}$
 - $(1 - \text{L1 miss rate}) \times \text{DRAM access time} > \text{L1 hit time}$
 - Benefit from reduced DRAM accesses > Penalty from accessing L1
- So, should we add an L1 cache or not?
 - Depends on L1 miss rate, which depends on locality present in software!

AMAT of No-Cache vs. L1 Cache

- For it to be worth it to install an L1 cache:
 $(1 - \text{L1 miss rate}) \times \text{DRAM access time} > \text{L1 hit time}$
- Let's assume L1 hit time = 1 cycle and DRAM access time = 100 cycles:
 $(1 - \text{L1 miss rate}) \times 100 > 1$
L1 miss rate < 0.99
→ If L1 miss rate can be kept below 99%, worth it to install L1 cache!
- Let's assume L1 hit time = 1 cycle and DRAM access time = 10 cycles:
 $(1 - \text{L1 miss rate}) \times 10 > 1$
L1 miss rate < 0.90
→ If L1 miss rate can be kept below 90%, worth it to install L1 cache!
- Let's assume L1 hit time = 1 cycle and DRAM access time = 1 cycle:
 $(1 - \text{L1 miss rate}) \times 1 > 1$
L1 miss rate < 0
→ There is no reason to put in an L1 whatsoever.

AMAT of No-Cache vs. L1 Cache

- For a single-level cache (L1 cache):
 - $AMAT(L1) = L1 \text{ hit time} + (L1 \text{ miss rate} \times \text{DRAM access time})$



- For a multi-level cache (L1, L2 caches):
 - $AMAT(L2) = L1 \text{ hit time} + (L1 \text{ miss rate} \times L1 \text{ miss penalty})$
 - $L1 \text{ miss penalty} = L2 \text{ hit time} + (L2 \text{ miss rate} \times \text{DRAM access time})$
 - $AMAT(L2) = L1 \text{ hit time} + L1 \text{ miss rate} \times L2 \text{ hit time}$
+ $L1 \text{ miss rate} \times L2 \text{ miss rate} \times \text{DRAM access time}$



AMAT for Multi-level Caches

- For L2 Cache to be worth it, $AMAT(L1) > AMAT(L2)$ needs to be true.

L1 Cache

L1 hit time

DRAM Memory

L1 miss rate \times DRAM access time

>?

L1 Cache

L1 hit time

L2 Cache

L1 miss rate \times L2 hit time

DRAM Memory

L1 miss rate \times L2 miss rate \times DRAM access time

- $AMAT(L1) - AMAT(L2)$
 $= (L1 \text{ miss rate} - L1 \text{ miss rate} \times L2 \text{ miss rate}) \times \text{DRAM access time}$
 $\quad - L1 \text{ miss rate} \times L2 \text{ hit time}$
 $= L1 \text{ miss rate} \times ((1 - L2 \text{ miss rate}) \times \text{DRAM access time} - L2 \text{ hit time}) > 0$
 $\rightarrow (1 - L2 \text{ miss rate}) \times \text{DRAM access time} > L2 \text{ hit time}$
 $\rightarrow \text{Benefit from reduced DRAM accesses} > \text{Penalty from accessing L2}$

Cache Design Parameter 2: Cache Size

Impact of Cache Size (a.k.a. Capacity) on AMAT

- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Larger caches are **good** for **miss rates**
 - More capacity means you can keep around cache blocks for longer
 - Means you can leverage more of the pre-existing **temporal locality**
 - If entire working set can fit into the cache, no capacity misses!
- But larger caches are **bad** for **hit times**
 - Longer wires and larger decoders mean longer access time
- Exactly why there are multiple levels of caches
 - **Frequently** accessed data where hit time is important stays in **L1** cache
 - **Rarely** accessed data which is part of a larger working set stays in **L3**

What cache size(s) should I choose?

- How should each cache level be sized?
- That depends on the application
 - Working set sizes of the application at various levels. E.g.:
 - Small set of data accessed very frequently (typically stack variables)
 - Medium set of data accessed often (currently accessed data structure)
 - Large set of data accessed rarely (rest of program data)
 - **Ideally**, cache levels and sizes would **reflect working set sizes**.
- Simulate multiple cache levels and sizes and choose one with lowest AMAT
 - Simulate on the applications that you care about
 - In the end, it must be a **compromise** (giving best average AMAT)

Cache Design Parameter 3: Cache Block Size

Impact of Cache Block Size on AMAT

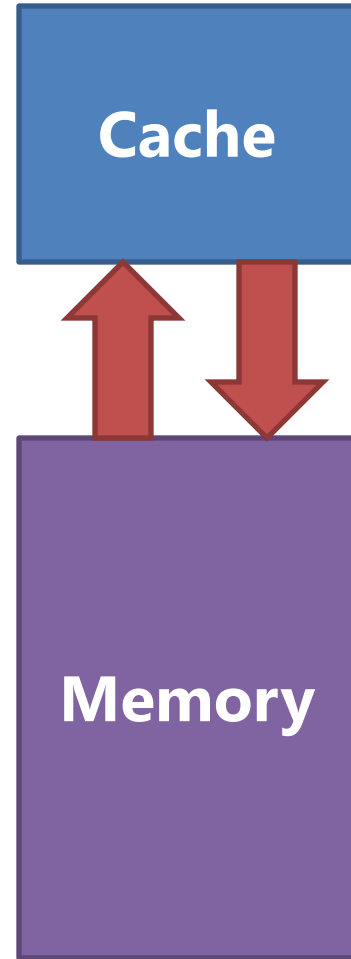
- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- **Cache block** (a.k.a. **cache line**)
 - Unit of transfer for cache data (typically 32 or 64 bytes)
 - If program accesses any byte in cache block, entire block is brought in
 - Each level of a multi-level cache can have a different cache block size
- Impact of larger cache block size on **miss rate**
 - Maybe **smaller miss rate** due to **better** leveraging of **spatial locality**
 - Maybe **bigger miss rate** due to **worse** leveraging of **temporal locality**
(Bringing in more data at a time may push out other useful data)
- Impact of larger cache block size on **miss penalty**
 - With a limited bus width, may take multiple transfers for a large block
 - E.g. DDR 4 DRAM bus width is 8 bytes, so 8 transfers for 64-byte block
 - Could lead to **increase in miss penalty**

Cache Block Size and Miss Penalty

- On a miss, the data must come from lower memory
- Besides memory access time, there's transfer time
- **What things impact how long that takes?**
 - The size of the cache block (**words/block**)
 - The width of the memory bus (**words/cycle**)
 - The speed of the memory bus (**cycles/second**)
- So the transfer time will be:

$$\frac{\text{seconds}}{\text{block}} = \frac{1}{\frac{\text{cycles}}{\text{second}} \times \frac{\text{words}}{\text{cycle}}} \times \frac{\text{words}}{\text{block}}$$

bus speed **bus width** **block size**



What cache block size should I choose?

- Again, that depends on the application
 - How much spatial and temporal locality the application has
- Simulate multiple cache block sizes and choose one with lowest AMAT
 - Simulate on benchmarks that you care about and choose best average
 - You may have to simulate different combinations for multi-level caches

Cache Design Parameter 4: Cache Associativity

Mapping blocks from memory to caches

- Cache size is much smaller compared to the entire memory space
 - Must map all the blocks in memory to limited CPU cache
- Does this sound familiar? Remember branch prediction?
 - Had similar problem of mapping PCs to a limited BHT
 - What did we do then?
 - We hashed PC to an entry in the BHT
 - On a hash conflict, we replaced old entry with more recent one
- We will use a similar idea with caches
 - **Hash memory addresses** to entries in cache
 - On a conflict:
 - **Replace** old cache block with more recent one
 - Or, **chain** multiple cache blocks on to same hash entry

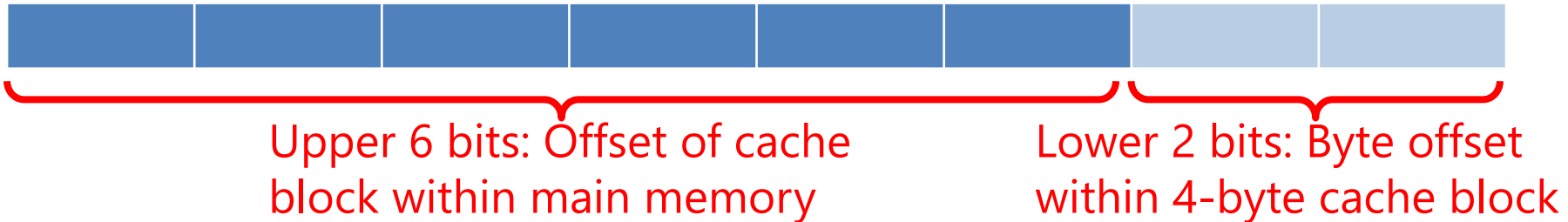
Impact of Cache Associativity on AMAT

- Depending on hash function and chaining, a cache is either:
 - **Direct-mapped** (no chaining allowed)
 - **Set-associative** (some chaining allowed)
 - **Fully-associative** (limitless chaining allowed)
- Impact of more associativity on **miss rate**
 - **Smaller miss rate** due to less misses due to hash conflicts
 - Misses due to hash conflicts are called **conflict misses**
 - A third category of misses besides cold and capacity misses
- Impact of more associativity on **hit time**
 - **Longer hit time** due to need to search through long chain

Direct-mapped Caches

Assumptions

- Let's assume for the sake of concise explanations
 - 8-bit memory addresses
 - 4-byte (one word) cache block sizes
- Of course these are not typical values. Typical values are:
 - 32-bit or 64-bit memory addresses (32-bit or 64-bit CPU)
 - 32-byte or 64-byte cache blocks sizes (for spatial locality)
 - But too many bits in addresses are going to give you a headache
- According to our assumption, here's a breakdown of address bits



- When I refer to addresses, I will sometimes omit the lower 2 bits
(When we talk about cache block transfer, that part is irrelevant)

Direct-mapped Cache Hash Function

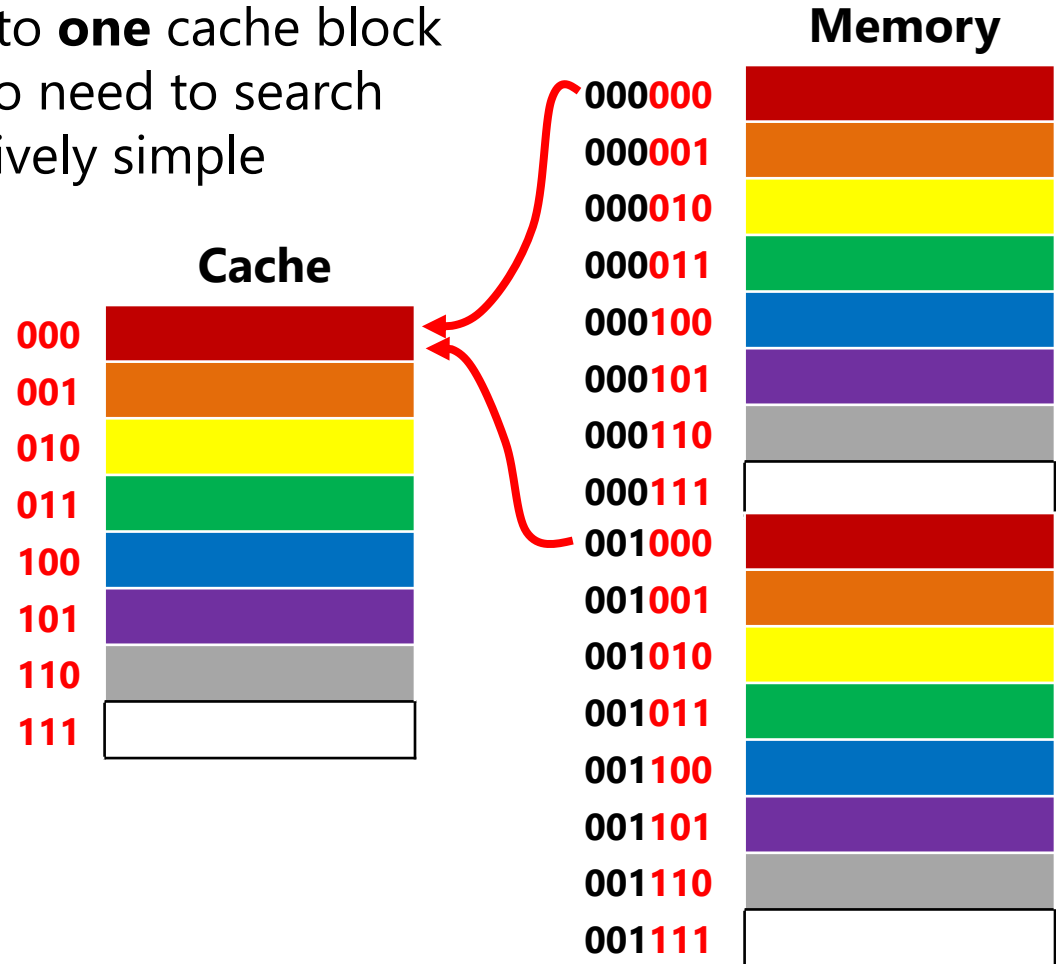
- Each memory address maps to **one** cache block
 - No chaining allowed so no need to search
 - Implementing this is relatively simple

Hash function:

For this 8-entry cache, to find **cache block index**, take the lowest 3 cache block offset bits in address.

But if our program accesses **001000**, then **000000**, how do we tell them apart?

Tags!



Tags help differentiate between conflicting blocks

Tag: part of address excluding cache block index

- On allocation of **001000**: **tag** = **001**

Cache		
	Tag	Data
000	001	
001		
010		
011		
100		
101		
110		
111		

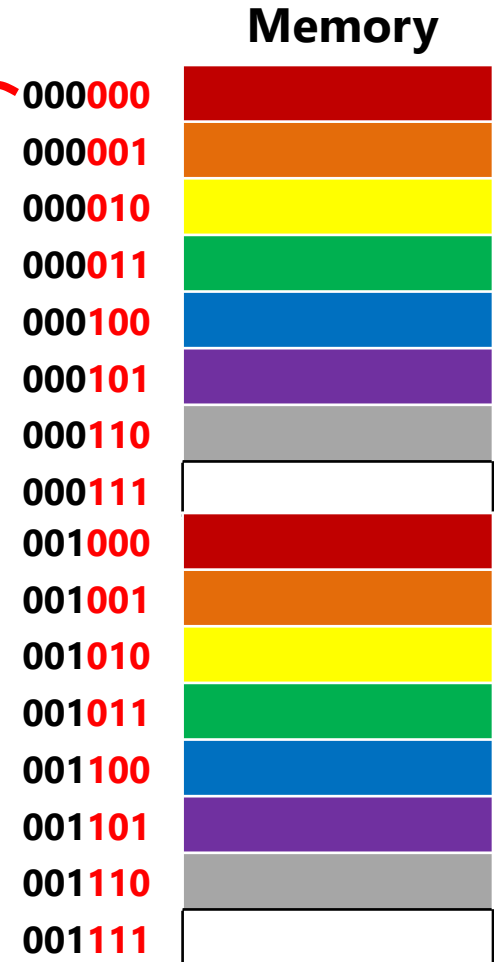
Memory	
000000	
000001	
000010	
000011	
000100	
000101	
000110	
000111	
001000	
001001	
001010	
001011	
001100	
001101	
001110	
001111	

Tags help differentiate between conflicting blocks

Tag: part of address excluding cache block index

- On allocation of **001000**: **tag = 001**
- On allocation of **000000**: **tag = 000**

Cache		
	Tag	Data
000	000	
001		
010		
011		
100		
101		
110		
111		



Valid bit indicates that block contains valid data

Valid bit: indicates that the block is valid

- Set to 0 initially when cache block is empty
- Set to 1 when a cache block is allocated

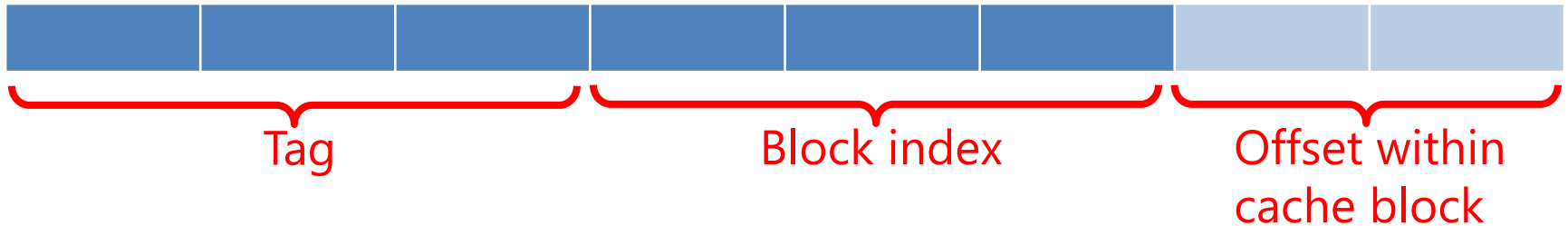
Cache			
	V	Tag	Data
000	1	000	
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Memory	
000000	
000001	
000010	
000011	
000100	
000101	
000110	
000111	
001000	
001001	
001010	
001011	
001100	
001101	
001110	
001111	

- Cache hit: $V == 1 \ \&\&$
 $\text{CacheBlock.Tag} == \text{MemoryBlock.Tag}$

Quiz: Address Bits Breakdown

- Now with the following parameters:
 - 8-bit memory addresses
 - 4-byte cache block sizes
 - 8-block cache
- How would we breakdown the memory address bits?



- First, the correct cache block is accessed using the **block index**
- Then, the **tag** is compared to the cache block tag
- If matched, **offset** is used to access specific byte within block

Example: A Direct-mapped Cache

- When the program first starts, we **set all the valid bits to 0**.
 - Signals all cache lines are empty
- Now let's try a sequence of reads...
do these **hit** or **miss**? How do the cache contents change?

000000 **miss**
100101 **miss**
100100 **miss**
100101 **hit**
010000 **miss** ← Cold miss
000000 **miss** ← Capacity miss?

} Cold misses

	V	Tag	Data
000	1	010	something
001	0		
010	0		
011	0		
100	1	100	something
101	1	100	something
110	0		
111	0		

Conflict Misses

- What should we call 2nd miss on **000000**?
 - Awkward to call it a capacity miss (It's not like capacity was lacking)
 - Let's call it a **conflict miss**

000000 **miss**
 100101 **miss**
 100100 **miss**
 100101 **hit**
 010000 **miss** ← Cold miss
 000000 **miss** ← Capacity miss?

} Cold misses

	V	Tag	Data
000	1	010	something
001	0		
010	0		
011	0		
100	1	100	something
101	1	100	something
110	0		
111	0		

Types of Cache Misses (Revised)

- Besides cold misses and capacity misses, there are conflict misses
- **Cold miss** (a.k.a. **compulsory miss**)
 - Miss suffered when data is accessed for the **first time** by program
- **Capacity miss**
 - Miss on a **repeat access** suffered due to a lack of **capacity**
 - When the program's **working set is larger than can fit in the cache**
- **Conflict miss**
 - Miss on a **repeat access** suffered due to a lack of **associativity**
 - **Associativity**: degree of freedom in associating cache block with an index
 - Direct mapped caches have no associativity
 - Since cache blocks are directly mapped to a particular block index

Associative caches

Flexible block placement

- Direct-mapped caches can have lots of **conflicts**
 - Multiple memory locations "fight" for the same cache line
- Suppose we had a 4-block direct-mapped cache
 - As before, 4-byte per cache block
 - Memory addresses are 8 bits.
- The following locations are accessed in a loop:
 - 0, 16, 32, 48, 0, 16, 32, 48...
 - or 00000000, 00010000, 00100000, 00110000, ...
- **What would happen?**
 - They will all land on the same block index, and all conflict miss!
 - Those other 3 blocks are not even getting used!
 - What if we used the space to chain conflicting blocks?

	V	Tag	Data
00	1	0011	
01	0		
10	0		
11	0		

Full associativity

- Let's make our 4-block cache **4-way set-associative**.

V	Tag	D
1	000000	*0

V	Tag	D
1	001100	*48

V	Tag	D
1	000100	*16

V	Tag	D
1	001000	*32

- What's the difference?
 - Now a hashed location can be associated with **any** of the 4 blocks
 - Analogous to having a hash conflict chain 4-entries long
 - The 4 cache blocks are said to be part of a cache **set**
 - When set size == cache size, it is said to be **fully associative**
- Let's do that sequence of reads again: 0, 16, 32, 48, 0, 16, 32, 48...
- Notice tag is now bigger, since there are no block index bits
 - Or **set index** bits in this context (just one set, so none needed)
- Now cache holds the entire **working set**: no more misses!

Example: A 2-way Set-Associative Cache

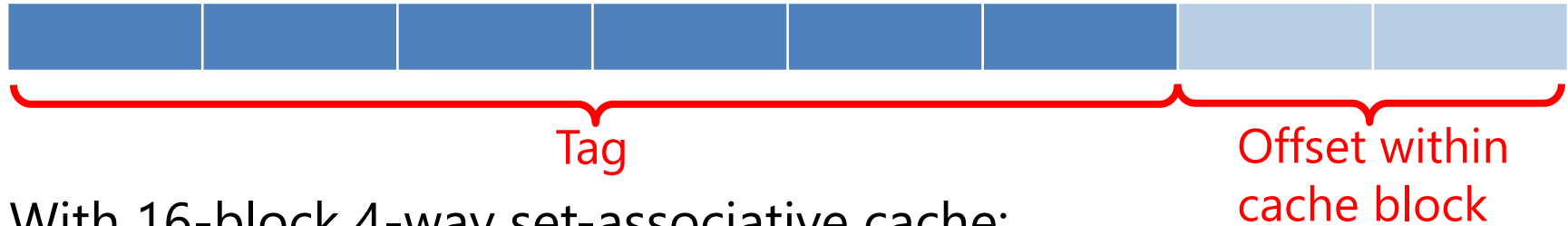
- 16-block 2-way set-associative cache
- Let's try the same stream of accesses as direct-mapped cache
- Yay! 2nd access to **000000** is no longer a conflict miss!

000000 miss
100101 miss
100100 miss
100101 hit
010000 miss
000000 hit

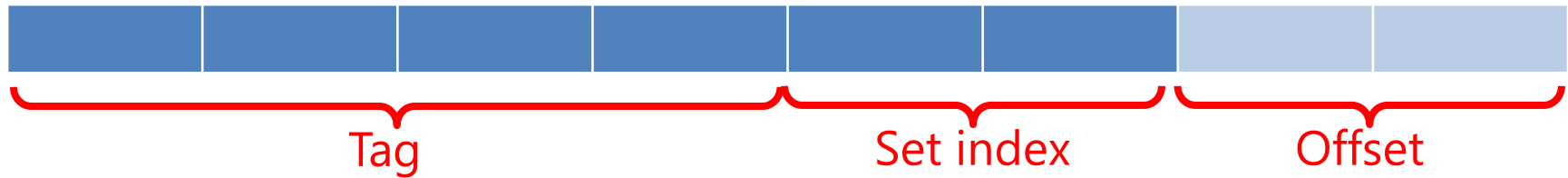
Set	V	Tag	Data	V	Tag	Data
000	1	000	something	1	010	something
001	0			0		
010	0			0		
011	0			0		
100	1	100	something	0		
101	1	100	something	0		
110	0			0		
111	0			0		

Address Bits Breakdown

- A fully associative cache (doesn't matter how many blocks):

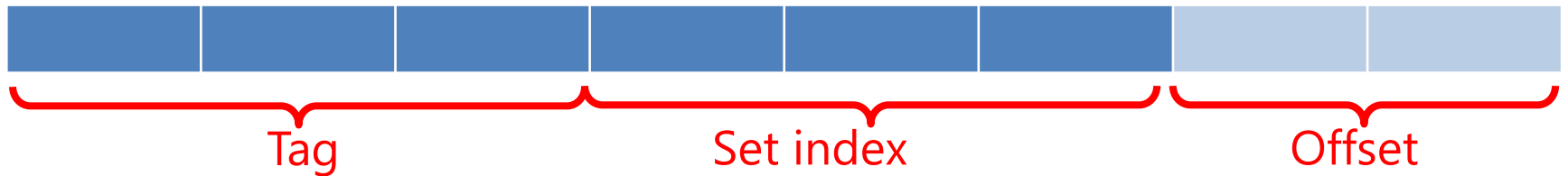


- With 16-block 4-way set-associative cache:



- $16 / 4 = 4$ sets in cache. So, 2 bits required for set index.

- With 64-block 8-way set-associative cache:



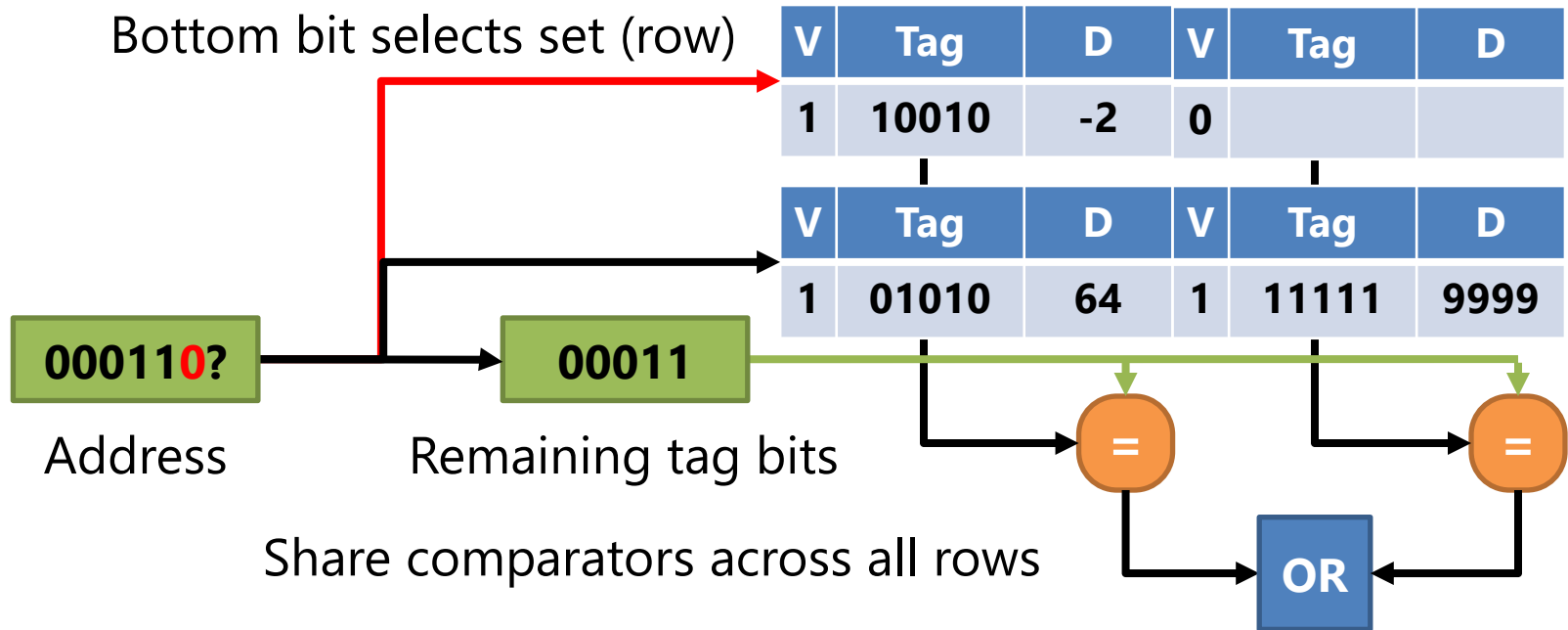
- $64 / 8 = 8$ sets in cache. So, 3 bits required for set index.

Want More Examples?

- Try out the Cache Visualizer on the course github:
 - https://github.com/wonsunahn/CS1541_Spring2023/tree/main/repositories/cache_demo
 - Courtesy of Jarrett Billingsley
- Visualizes cache organization for various parameters
 - Cache block size
 - Number of blocks in cache (capacity)
 - Cache associativity

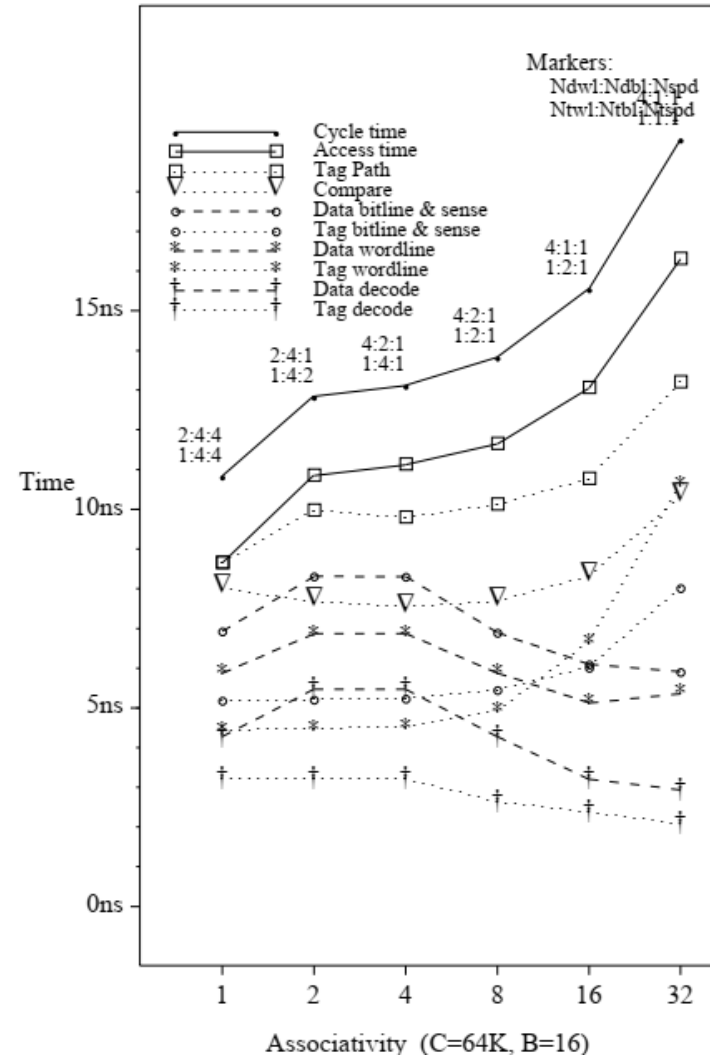
Associativity is Costly

- Associativity requires complex circuitry and may **increase hit time**
- Full associativity is only used for very small caches
 - And where a cache miss is extremely costly
- Usually caches are 2-, 4-, or maybe 8- way set-associative



Access/cycle time as a function of associativity

Thozyoor, Shyamkumar & Muralimanohar,
Naveen & Ahn, Jung Ho & Jouppi, Norman.
(2008). CACTI 5.1.



Cache Design Parameter 5: Cache Replacement Policy

Cache Replacement

- If we have a cache miss and no empty blocks, what then?

V	Tag	D
1	000000	*0

V	Tag	D
1	001100	*48

V	Tag	D
1	000001	*4

V	Tag	D
1	001000	*32

- Let's read memory address 4 (**00000100**).
 - Uh oh. That's a miss. Where do we put it?
- With associative caches, you must have a **replacement scheme**.
 - Which block to evict (kick out) when you're out of empty slots?
- The simplest replacement scheme is **random**.
 - Just pick one. Doesn't matter which.
- What would make more sense?
 - How about taking **temporal locality** into account?

LRU (Least-Recently-Used) Replacement

- When you need to evict a block, kick out the oldest one.

V	Tag	D
1	000001	*4

4 reads old

V	Tag	D
1	001100	*48

1 read old

V	Tag	D
1	000100	*16

3 reads old

V	Tag	D
1	001000	*32

2 reads old

- Our read history looked like 0, 16, 32, 48. How old are the blocks?
- Now we want to read address 4. Which block should we replace?
- But now we must maintain the age of the blocks
 - Easy to say. How do we keep track of this in hardware?
- Have a saturating counter for each cache block indicating age
 - When accessing a set, increment counter for each block in set
 - On a cache hit, reset counter to 0 (most recently used)

Impact of LRU on AMAT

- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- Impact of LRU on **miss rate**
 - **Smaller miss rate** due to **better** leveraging of **temporal locality**
(Recently used cache lines more likely to be used again)
- Saturating counter for LRU uses bits and **adds to amount of metadata**
 - Cache **tag**, the **valid bit**, the **saturating counter** are all metadata
 - Every bit you spend on metadata is a bit you don't spend on real data
 - Spending many bits on counter may reduce capacity for real data
 - This may lead to a **larger miss rate**, if LRU is not very effective