

Cache Design

CS/COE 1541 (Fall 2020)
Wonsun Ahn

Evaluating Cache Design

Impact of Memory on Performance

- **CPU Cycles = CPU Execution Cycles + Memory Stall Cycles**
 - CPU Execution Cycles = cycles where CPU is doing useful work
 - Memory Stall Cycles = cycles where CPU is waiting on cache miss
 - Same memory stall cycles we measured using the PMU
- Processor design features that impact CPU execution cycles:
 - Pipelining, branch prediction, wide execution, out-of-order, ...
- Processor design features that impact Memory stall cycles:
 - Caches
 - Write buffer
 - Prefetcher (we haven't learned this yet)

Impact of Memory on Performance

- CPU Cycles = CPU Execution Cycles + Memory Stall Cycles
- This gives rise to two categories of programs
 - **CPU Bound**: Programs where CPU execution is majority of cycles
 - **Memory Bound**: Programs where Mem stalls is majority of cycles
- To improve performance of **CPU Bound** programs
 - Improve HW by getting **wider superscalar CPUs**
 - Improve SW by **optimizing the computation** in the program
- To improve performance of **Memory Bound** programs
 - Improve HW installing more CPU **caches** or faster DRAM
 - Improve SW by **optimizing memory access pattern** of program
 - We already saw how array.c is much faster than linked-list.c

How about overclocking using DVFS?

- CPU Time = CPU Cycles * Cycle Time
= (CPU Execution Cycles + Memory Stall Cycles) * Cycle Time
- What if we halved the Cycle Time using DVFS?
 - Memory Stall Cycles could increase by close to 2X!
 - Why? You may speed up CPU, but DRAM speed remains the same
 - The bus (wire) that connects CPU to DRAM is not getting any faster
 - The DRAM chip itself is not getting clocked any faster
 - So if DRAM access speed was 100 ns,
 - If CPU is clocked at 1 GHz, it takes 100 cycles to access memory
 - If CPU is clocked at 2 GHz, it takes 200 cycles to access memory
- So if a program is **Memory Bound, overclocking is mostly useless**
 - Reduction in Cycle Time canceled out by increase in Memory Stall Cycles

Oracle Cache

- CPU Cycles = CPU Execution 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
 - Hit time is incurred regardless of hit or miss (miss rate)
 - Since even after a miss, the cache must be accessed again and “hit”
 - Miss penalty is incurred only on a miss
- 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 AMAT

AMAT for Multi-level Caches

- 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):
 - $AMAT(L2) = L1 \text{ hit time} + (L1 \text{ miss rate} \times L1 \text{ miss penalty})$
 - $L1 \text{ miss penalty} = L2 \text{ hit time} + (L2 \text{ miss rate} \times \text{DRAM access time})$
 - $AMAT(L2) = L1 \text{ hit time} + L1 \text{ miss rate} \times L2 \text{ hit time}$
 $+ L1 \text{ miss rate} \times L2 \text{ miss rate} \times \text{DRAM access time}$
- Which is better? $AMAT(L1)$ or $AMAT(L2)$?
 - $AMAT(L2) - AMAT(L1) = L1 \text{ miss rate} \times L2 \text{ hit time}$
 $+ (L1 \text{ miss rate} \times L2 \text{ miss rate} - L1 \text{ miss rate}) \times \text{DRAM access time}$
 $= L1 \text{ miss rate} \times (L2 \text{ hit time} + (L2 \text{ miss rate} - 1) \times \text{DRAM access time})$

AMAT for Multi-level Caches

- Which is better? $AMAT(L1)$ or $AMAT(L2)$?
 - $AMAT(L2) - AMAT(L1)$
 $= L1 \text{ miss rate} \times (L2 \text{ hit time} + (L2 \text{ miss rate} - 1) \times \text{DRAM access time})$
- For $AMAT(L1) > AMAT(L2)$ (that is, for it to be worth it to put in an L2):
 - $L1 \text{ miss rate} \times (L2 \text{ hit time} + (L2 \text{ miss rate} - 1) \times \text{DRAM access time}) < 0$
 - $L2 \text{ hit time} + (L2 \text{ miss rate} - 1) \times \text{DRAM access time} < 0$
 - $L2 \text{ hit time} < (1 - L2 \text{ miss rate}) \times \text{DRAM access time}$
 - If $L2 \text{ hit time} = 10 \text{ cycles}$ and $\text{DRAM access time} = 100 \text{ cycles}$,
 $10 < (1 - L2 \text{ miss rate}) \times 100$
 $L2 \text{ miss rate} < 0.9$
 - **Unless L2 cache miss rate is greater than 90%** (which is horrible),
worth it to install an L2 cache (if we can keep hit time at 10 cycles)!
- But that conclusion is application dependent
 - If your program has poor locality and **L2 miss rate is above 90%**,
the **additional L2 cache will hurt performance!**

Cache Design Parameter 1: 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 and muxes 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 where it's more important not to miss stays in **L3**

What cache size(s) should I choose?

- How many levels of caches? How should they 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 access pattern**.
- 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 2: Cache Block Size

Impact of Cache Block Size on AMAT

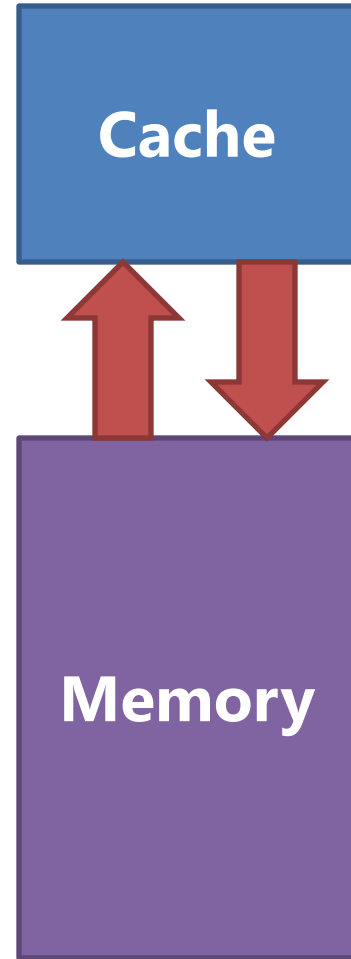
- $AMAT = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
- **Cache block** (a.k.a. **cache line**)
 - Unit of transfer for cache data (typically 32, 64, or 128 bytes)
 - If program accesses any byte in cache block, entire block is brought in
 - Each level of a multi-level cache can 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 3: 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
- Doesn't 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 cache block addresses** in memory to entries in cache
 - On a conflict:
 - Replace old cache block with more recent one
 - Or, chain multiple blocks for same hash value

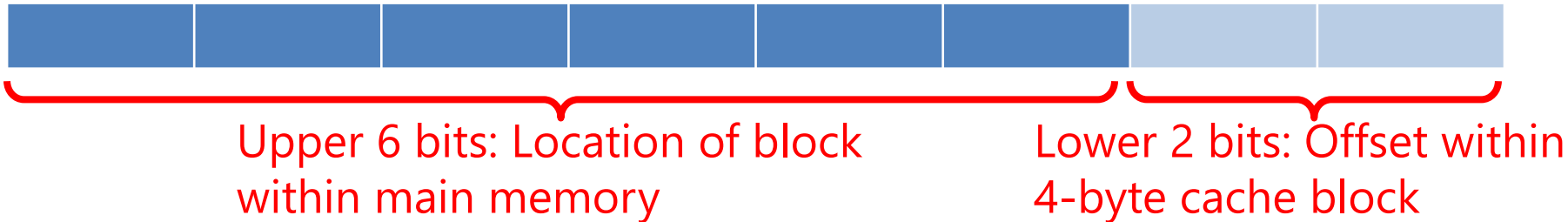
Impact of Cache Associativity on AMAT

- 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 cache block sizes
- Of course these are not typical values. Typical values are:
 - 32-bit or 64-bit memory addresses
 - 32-byte or 64-byte cache blocks sizes
 - But too many bits in addresses is going to give you a headache
- According to our assumption, here's a breakdown of address bits



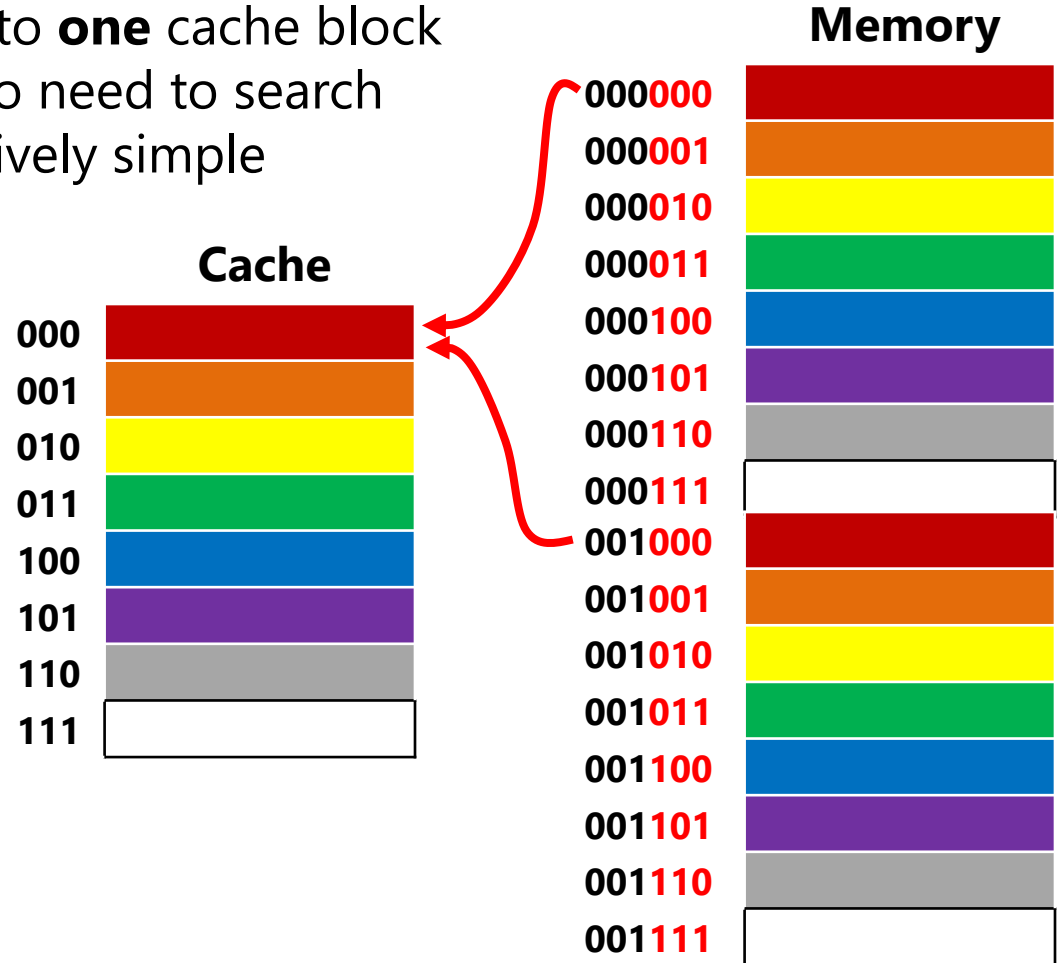
- When I refer to addresses, I will only talk about the upper 6 bits
(Cache block is the unit of transfer, so that's the only relevant part)

Hash Function

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

For this 8-entry cache, to find the cache block for a memory address, take the lowest 3 address bits.

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

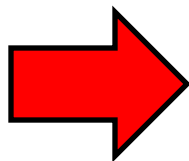


Tags

- Each cache block has a **tag** that indicates the original memory location

This seems redundant...

	Tag	Data
000	001000	DEADBEEF
001		
010		
011	110011	CAFEFACE
100	101100	B0DECA7
101		
110		
111		



For address **110011**, **011** is the **block index**, and **110** is the **tag**.

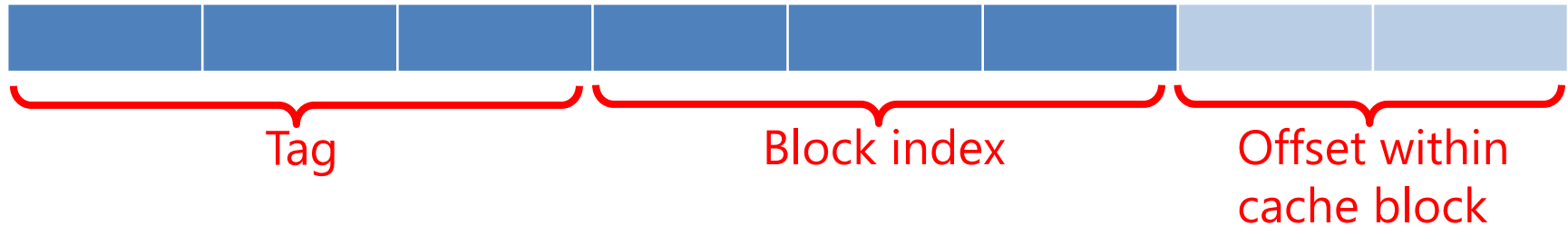
	Tag	Data
000	001	DEADBEEF
001		
010		
011	110	CAFEFACE
100	101	B0DECA7
101		
110		
111		

How do we tell what entries are empty/full?

Just add another bit (a **valid** bit)!

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

Watching it in action

- 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	} Miss because $V == 0$
100101	miss	
100100	miss	
100101	hit	
010000	miss	} Miss due to wrong tag
000000	miss	

	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		

Steps in Hardware

1. Split the address into two parts: **block index** and **tag**
 - *a bus splitter*
 2. Use the **block index** to find the right cache entry.
 - *a mux*
 3. If the **valid** bit is 1, and the entry's **tag** matches,
 - *an equality comparator and an AND gate*
 - **It's a hit!** Read the cached data.
 4. Otherwise...
 - **It's a miss.**
 - If load miss, find something else to issue in instruction queue
 - If store miss, put in Write Buffer and retire following instructions
- Takeaway: Direct mapped cache hardware is simplistic (and **fast!**)

Conflict Misses

- Q: Which are **cold misses**, and which are not?
- A: second miss to **000000** is not a cold miss
- Then what should we call it?
 - 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

000000 miss

	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 very low 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 000000, 000100, 001000, 001100, ...
- **What would happen?**
 - They will all land on the same block index, and all conflict miss!
 - What about those other 3 blocks? What a waste!
 - What if we could just... put the blocks anywhere in the cache?

	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

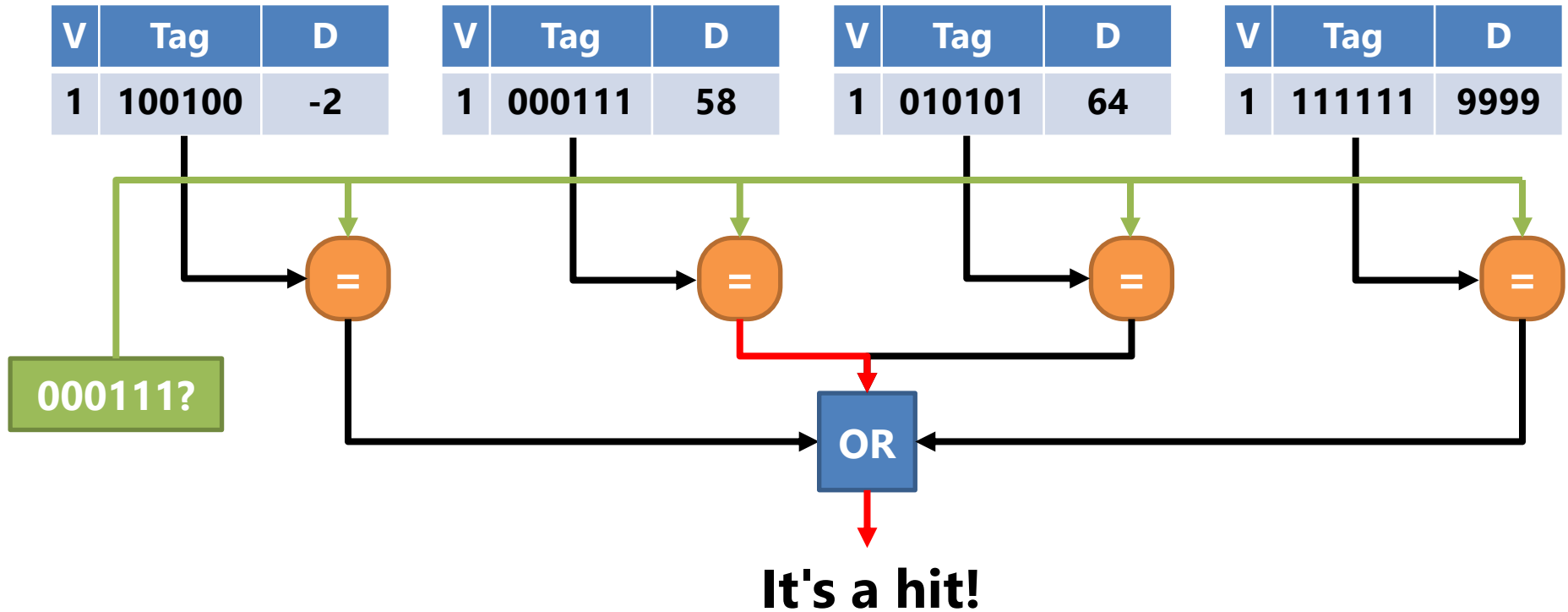
V	Tag	D
1	001000	*32

- What's the difference?
 - Now memory location can be associated with 1 of 4 cache blocks
 - Analogous to having a "chain" 4-entries long for hash collisions
 - 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!

Cost of Associativity

Silicon is pretty cheap, right?

- The whole concept of associative caches is that **you look at all the cache blocks at once to see if you hit**. How would we do that in hardware?

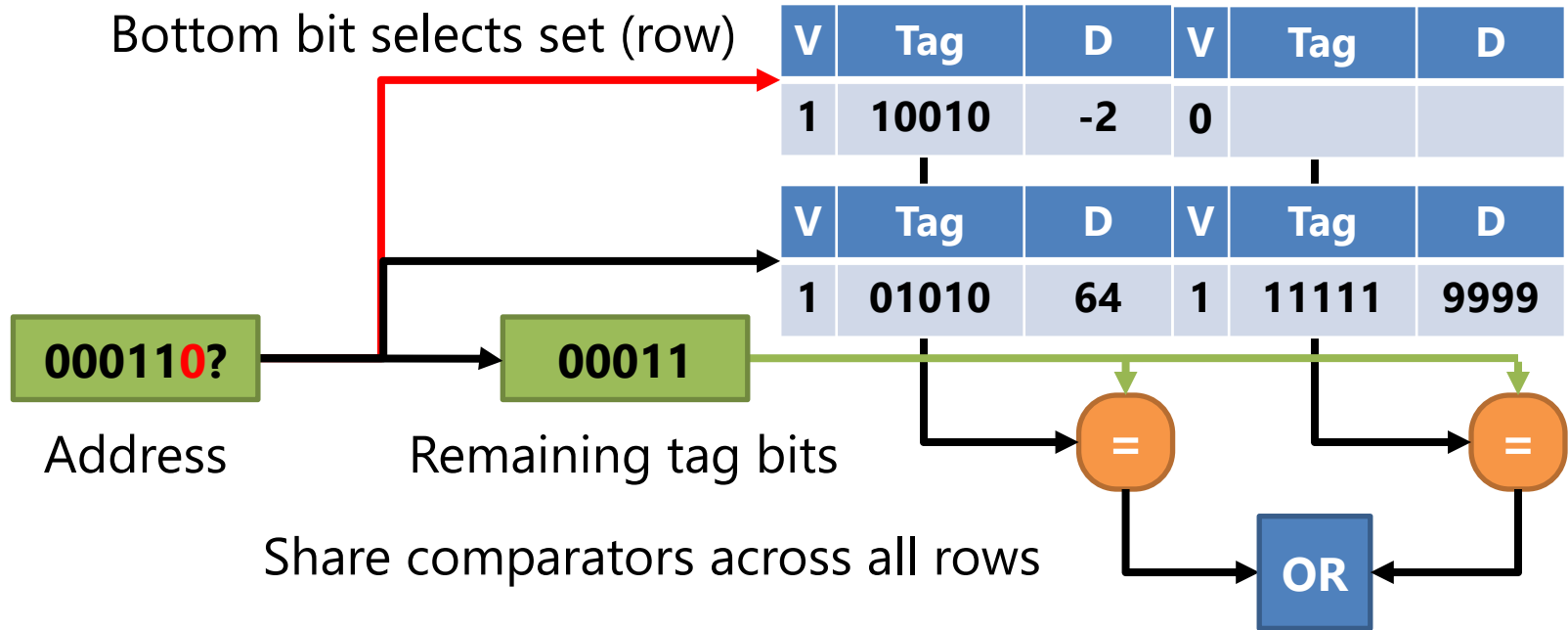


...but not THAT cheap

- Let's do some back-of-the-envelope calculations:
 - Each comparator has to compare <tag bits> bits
 - With 32-bit addresses, that might be something like 30 bits.
 - So, each comparator needs 60 wires coming into it.
 - Each comparator has to have 30 XOR gates and 29 OR gates.
 - We have 4 comparators. So we need **240 wires, 120 XOR gates, and 116 OR gates**, for **just a four-entry** fully associative cache.
 - We also need MUXes/buses to get the data out of the blocks.
 - What if we had a 64-bit machine with 32768 cache blocks?
- All those transistors could have been used for more cache blocks
 - Also, that complex circuitry increases cache hit time

Set-associative caches usually hit the sweet spot

- Full associativity is impractical for all but the tiniest caches.
- Instead we usually stick to 2-, 4-, or maybe 8- way set-associativity, and **have multiple sets or rows of associative caches**

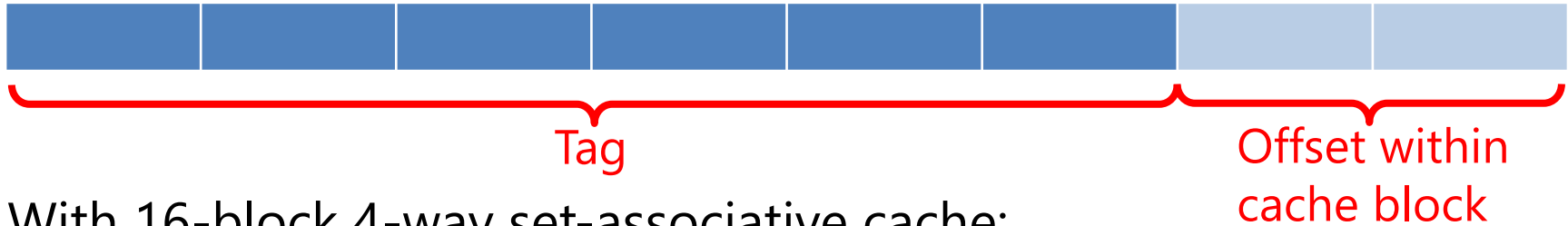


Set-associative caches usually hit the sweet spot

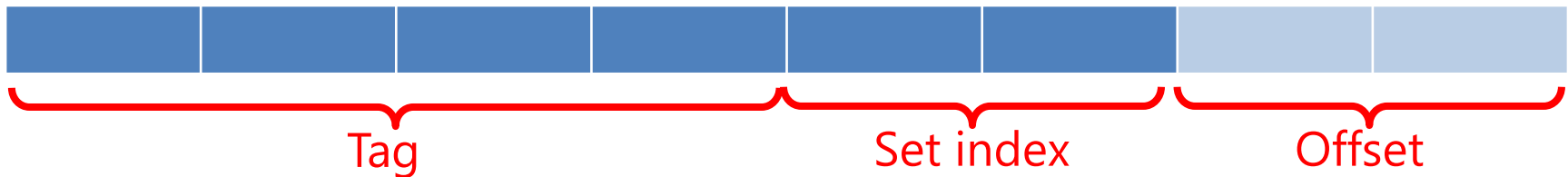
- Going from direct-mapped to 2-way set associative:
 - Can give a big improvement in miss rate
- But increasing associativity gives diminishing returns
 - Beyond 4-way and 8-way, conflict misses are rare

Address Bits Breakdown

- With 4-block fully associative cache:

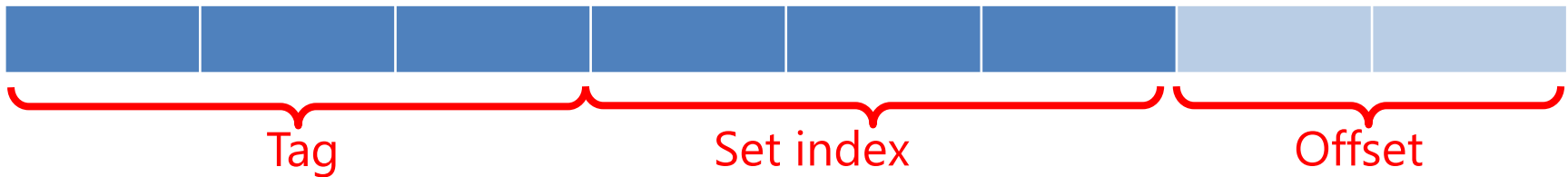


- With 16-block 4-way set-associative cache:



- $16 / 4 = 4$ 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_Fall2020/tree/master/resources/cache_demo
 - Courtesy of Jarrett Billingsley
- Visualizes cache organization for various parameters
 - Cache block size
 - Number of blocks in cache (capacity)
 - Cache associativity

Cache Design Parameter 4: 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.
 - Uh oh. That's a miss. Where do we put it?
- With associative caches, you have to have a **replacement scheme**.
 - Which block to evict (kick out) when you're out of empty slots?
- The simplest (and pretty effective) 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 (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)
- Possible slight **increase** in **miss penalty**
 - On a miss, a cache line needs to be replaced to make room for new line
 - Complex replacement policies can delay the replacement
- Saturating counter for LRU uses bits and adds to amount of meta-data
 - Cache tag, the valid bit, the saturating counter are all meta-data
 - Every bit you spend on meta-data is a bit you don't spend on real data