CSCI 210: Computer Architecture
Lecture 33: Caches II

Stephen Checkoway
Slides from Cynthia Taylor

Announcements

- Ice cream social: Rice 100B at 4:30 today!

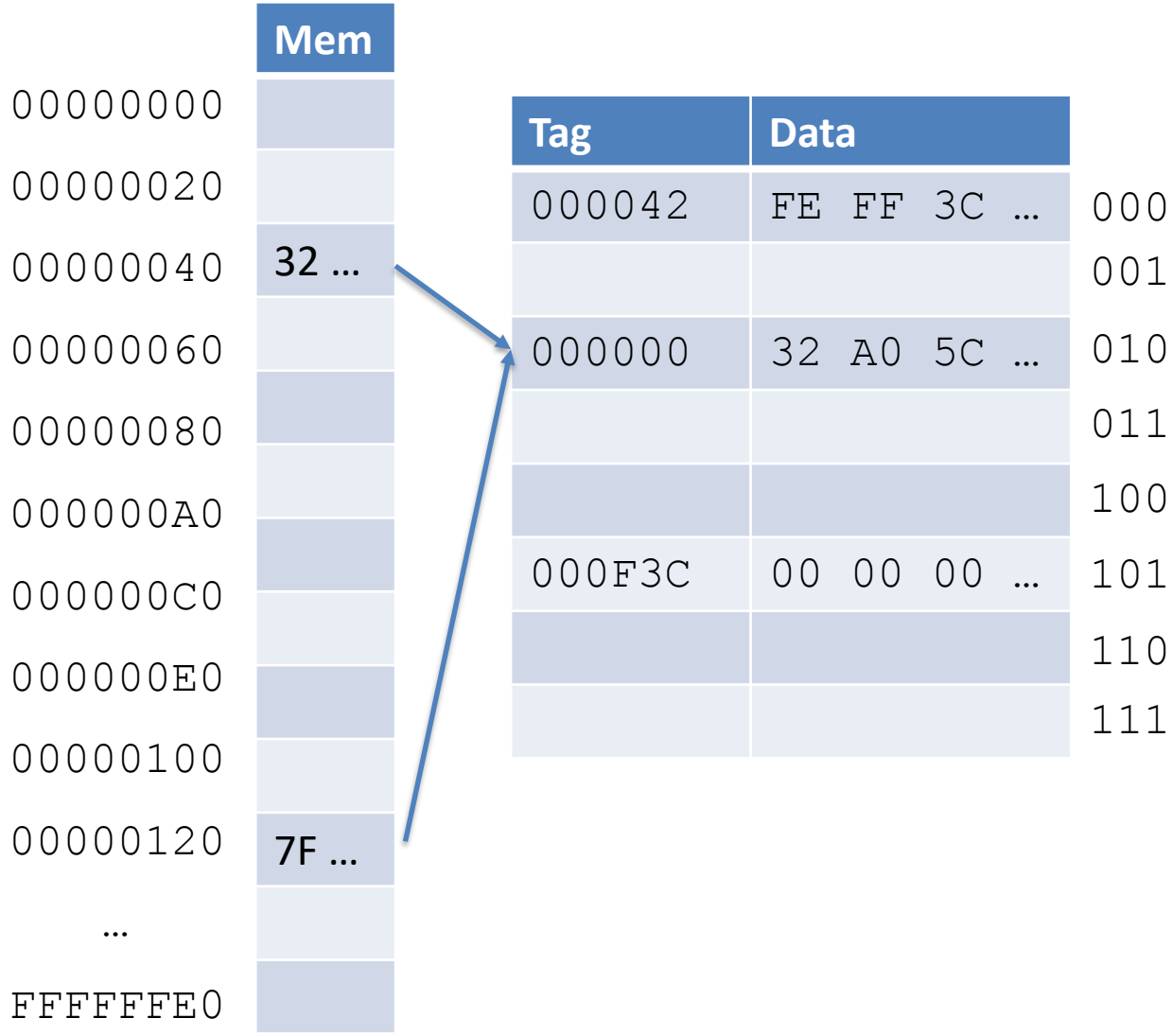
CS History: Jerry Lawson

Museum of Play / Estate of Jerry Lawson

- Born in Brooklyn in 1940
- “Father of the video game cartridge”
- Worked at Fairchild Semiconductor
- Developed the swappable video game cartridge
- Member of the Homebrew Computer Club
  - Interviewed Steve Wozniak for a position at Fairchild and didn’t hire him

High-level cache strategy

- Divide all of memory into consecutive blocks
- Copy data (memory ↔ cache) one block at a time
- Cache lookup:
  - Get the index of the block in the cache from the address
  - Compare the tag from the address with the tag in the cache



How do we know if it’s in the cache?

- What if there is no data in a location?
  - Valid bit: 1 = present, 0 = not present
  - Initially 0

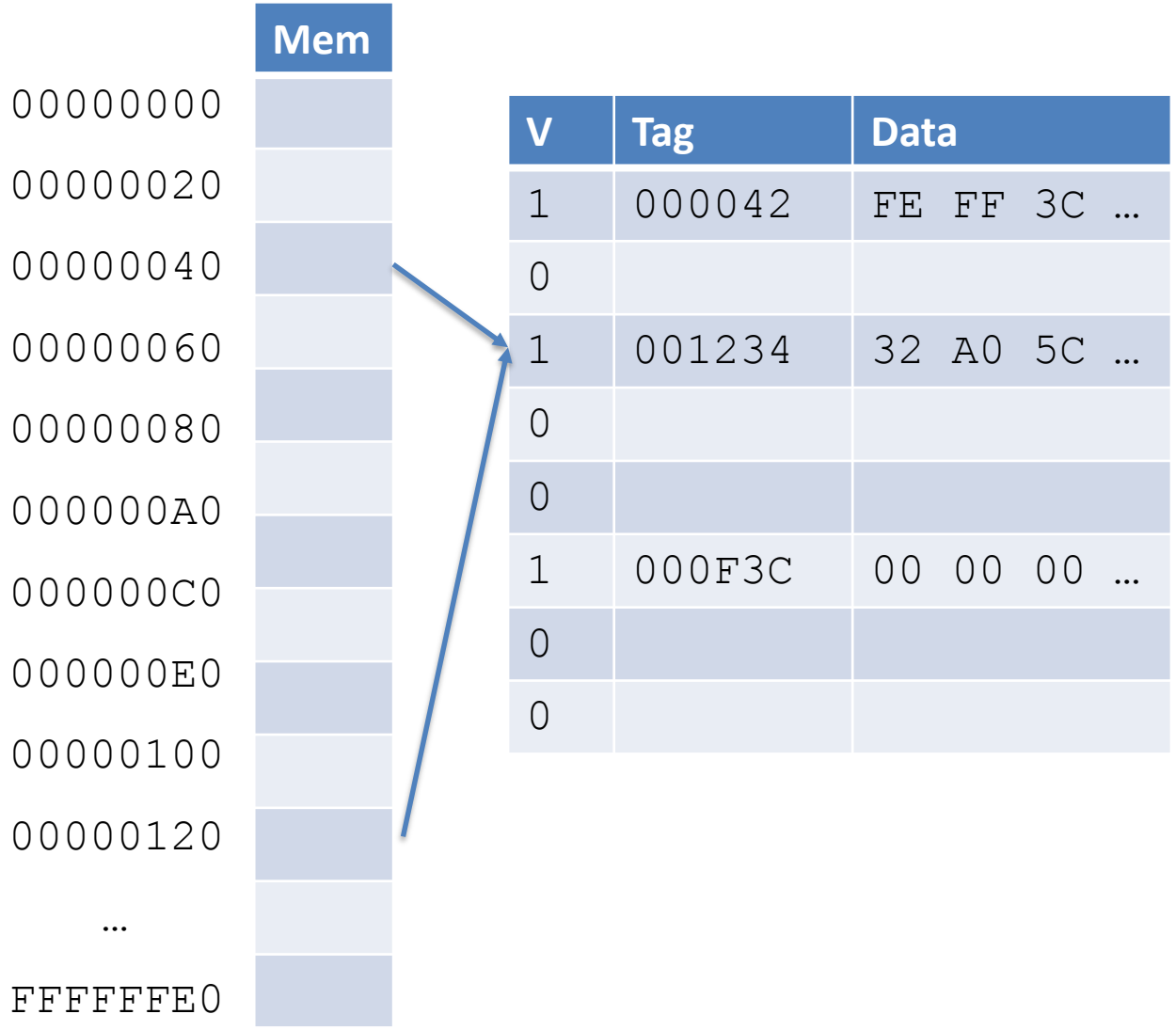
Direct-mapped cache layout

- Valid stores 1 if data is present in cache
- Tag stores high-order bits of address
- Data stores all of the data for the block (e.g., 32 bytes)

Valid	Tag	Data
1	000042	FE FF 3C 7F …
0		
1	001234	32 A0 5C 21 …
0		
0		
1	000F3C	00 00 00 00 …
0		
0		

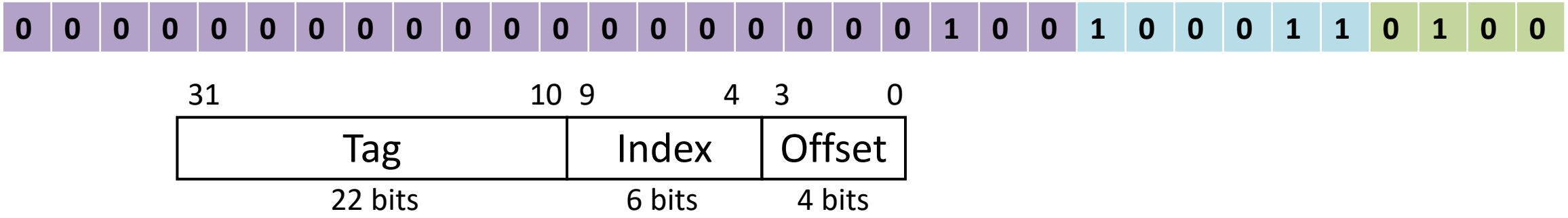
High-level cache strategy

- Divide all of memory into consecutive blocks
- Copy data (memory ↔ cache) one block at a time
- Cache lookup:
  - Get the index of the block in the cache from the address
  - Check the valid bit; compare the tag to the address

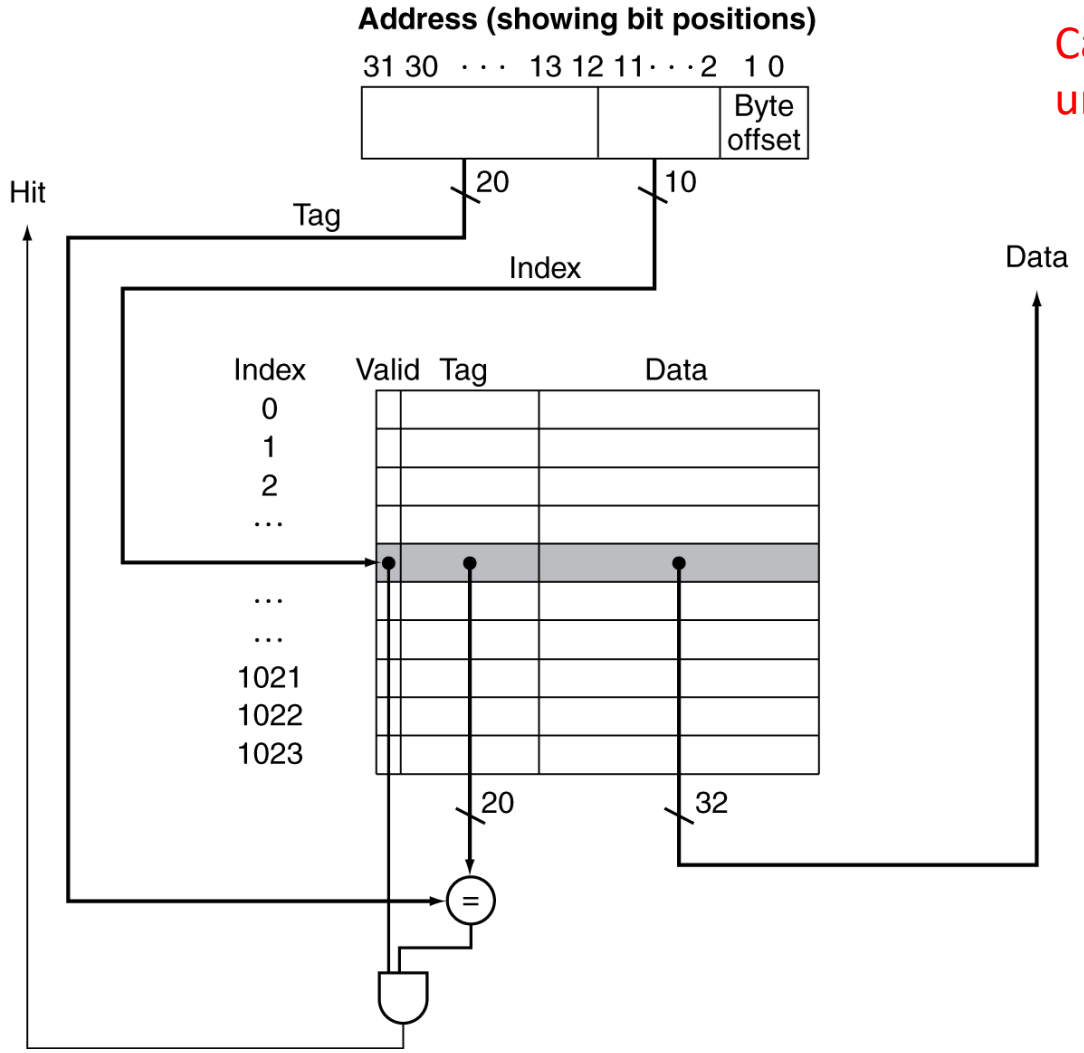


Example

- 64 blocks, 16 bytes/block
  - To what cache index does address 0x1234 map?
- Block address = $\lfloor 0x1234/16 \rfloor$ = 0x123
- Index = 0x123 modulo 64 = 0x23
- No actual math required: just select appropriate bits from address!



Memory access



Caution: This diagram shows an unusually small block size of 4 bytes!

Direct Mapped Cache

data	byte addresses	A	B	C	D
x	00 00 01 00	M	M	M	M
y	00 00 10 00	M	M	M	M
z	00 00 11 00	M	M	M	M
x	00 00 01 00	H	H	H	H
y	00 00 10 00	H	H	H	H
w	00 01 01 00	M	M	M	M
x	00 00 01 00	M	M	H	H
y	00 00 10 00	H	H	H	H
w	00 01 01 00	H	M	H	H
u	00 01 10 00	M	M	M	M
z	00 00 11 00	H	H	M	H
y	00 00 10 00	H	M	H	H
x	00 00 01 00	H	M	M	M

E None are correct

Four blocks, each block holds four bytes

How do we know how big a block in cache is?

A. Each block in the cache stores its size
B. The length of the tag in the cache determines the block size
C. The most significant bits of the address determine the block size
D. The least significant bits of the address determine the block size
E. For any given cache, the block size is constant

CACHE REPLACEMENT POLICIES

Cache Size vs Memory Size

- USB-C Charge Cable (2 m)

Configure to Order

Configure your MacBook Pro with these options, only at apple.com:

- 2.4GHz 8-core Intel Core i9, Turbo Boost up to 5.0GHz, with 16MB shared L3 cache
- 32GB of 2400MHz DDR4 memory

Memory is 2048 times bigger than cache

Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access

Cache replacement policy

- On a hit, return the requested data
- On a miss, load block from lower level in the memory hierarchy and write in cache; return the requested data
- Policy: Where in cache should the block be written? (With direct-mapped caches, there's only one possible location: block_address % number_of_blocks_in_cache)

Cache policy for stores

- Policy choice for a hit: Where do we write the data?
  - Write-back: Write to cache only
  - Write-through: Write to cache and also to the next lowest level of the memory hierarchy
- Policy choice for a miss
  - Write-allocate: Bring the block into cache and then do the write-hit policy
  - Write-around: Write only to memory

Store-hit policy: write-through

- Update cache block AND memory
- 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

Store-hit policy: write-back

- Only update the block in cache
  - Keep track of whether each block is “dirty” (i.e., it has a different value than in memory)
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
- Faster than write-through, but more complex

V	D	Tag	Data
1	0	000042	FE FF 3C …
0			
1	1	001234	65 82 5C …
0			
0			
1	0	000F3C	00 00 00 …
0			
0			

```
sw    $t0, 8($sp)
lw    $t1, 0($a3)
```

If the addresses map to the same cache block (but point to different regions of memory), the lw will evict the first block from cache and replace it with the second. What will the value of the valid and dirty bits be after the block is replaced?

A. Valid = 0, Dirty = 0
B. Valid = 0, Dirty = 1
C. Valid = 1, Dirty = 0
D. Valid = 1, Dirty = 1
E. This cannot happen

V	D	Tag	Data
1	0	000042	FE FF 3C …
0			
1	1	001234	65 82 5C …
0			
0			
1	0	000F3C	00 00 00 …
0			
0			

```
sw    $t0, 8($sp)
lw    $t1, 12($sp)
```

If both addresses are to words in the same block in memory, what will the values of the valid and dirty bits be?

A. Valid = 0, Dirty = 0
B. Valid = 0, Dirty = 1
C. Valid = 1, Dirty = 0
D. Valid = 1, Dirty = 1
E. The addresses are different so they cannot be in the same block

V	D	Tag	Data
1	0	000042	FE FF 3C …
0			
1	1	001234	65 82 5C …
0			
0			
1	0	000F3C	00 00 00 …
0			
0			

Store-miss policy: write-around

- Only write the data to memory
- Good for initialization where lots of memory is written at once but won’t be read again soon

Store-miss policy: write-allocate

- Read a block from memory (just like a load miss)
- Perform the write according to the store-hit policy (i.e., write in cache or write in both cache and memory)

- Good for when data is likely to be read shortly after being written (temporal locality)

```
lw    $t0, 8($sp)
sw    $t1, 0($a3)
```

Given a write-allocate/write-back policy, if the two addresses point to different regions of memory but map to the same cache block, what will the value of the valid and dirty bits be?

A. Valid = 0, Dirty = 0
B. Valid = 0, Dirty = 1
C. Valid = 1, Dirty = 0
D. Valid = 1, Dirty = 1
E. This cannot happen

V	D	Tag	Data
1	0	000042	FE FF 3C …
0			
1	1	001234	65 82 5C …
0			
0			
1	0	000F3C	00 00 00 …
0			
0			

Write-allocate: create a block in cache on store miss
Write-back: Update memory only when the block is evicted

```
lw    $t0, 8($sp)
sw    $t1, 0($a3)
```

Given a write-around policy, if the two addresses point to different regions of memory but map to the same cache block, what will the value of the valid and dirty bits be?

A. Valid = 0, Dirty = 0
B. Valid = 0, Dirty = 1
C. Valid = 1, Dirty = 0
D. Valid = 1, Dirty = 1
E. This cannot happen

V	D	Tag	Data
1	0	000042	FE FF 3C …
0			
1	1	001234	65 82 5C …
0			
0			
1	0	000F3C	00 00 00 …
0			
0			

Write-around: On a store miss, only update memory

Store Policies

- Given either high store locality or low store locality, which policies might you expect to find?
- Write-allocate: create block in cache. Write-around: don't create block. Write-through: update cache + memory. Write-back: update cache only.

Selection	High Locality		Low Locality	
	Miss Policy	Hit Policy	Miss Policy	Hit Policy
A	Write-allocate	Write-through	Write-around	Write-back
B	Write-around	Write-through	Write-allocate	Write-back
C	Write-allocate	Write-back	Write-around	Write-through
D	Write-around	Write-back	Write-allocate	Write-through
E	None of the above			

Common policy choices

- Write-back + write-allocate
  - Dirty blocks are written to memory only when replaced
  - Stores bring block into cache
  - Subsequent loads/stores will cause cache hits (unless the block is evicted)
- Write-through + write-around
  - Writes always go to memory
  - Cache is mostly for loads