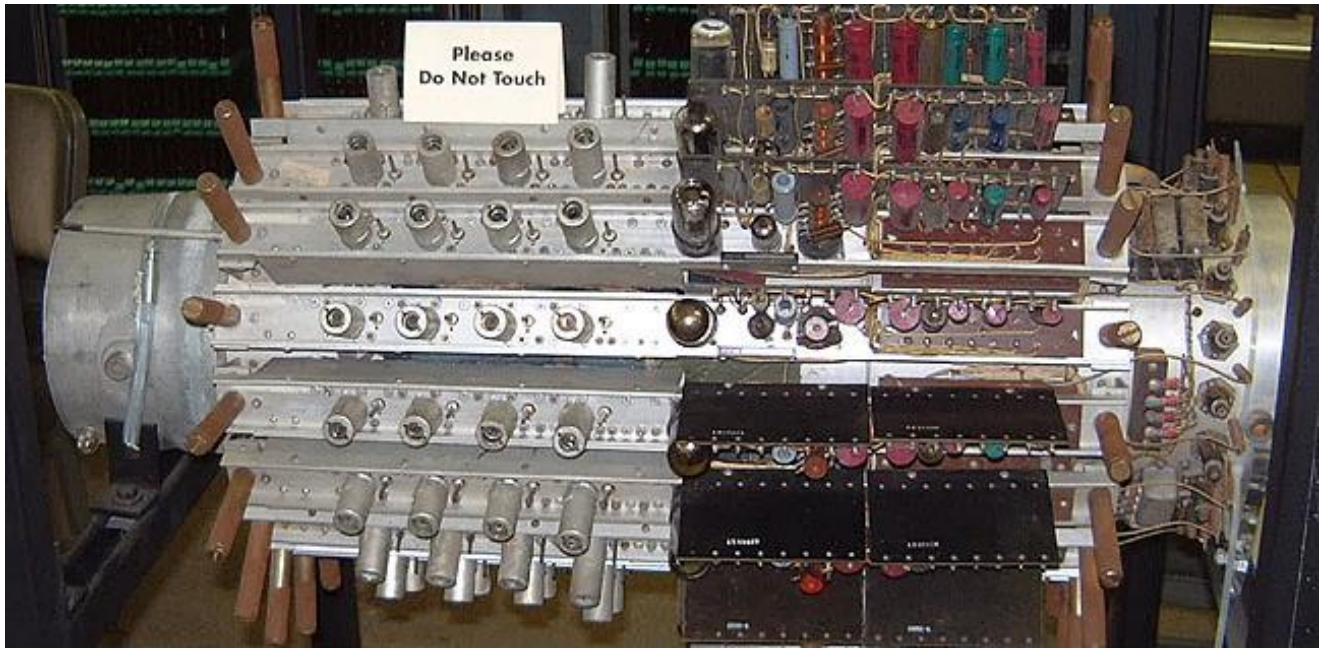
CSCI 210: Computer Architecture
Lecture 32: Caches

Stephen Checkoway
Slides from Cynthia Taylor

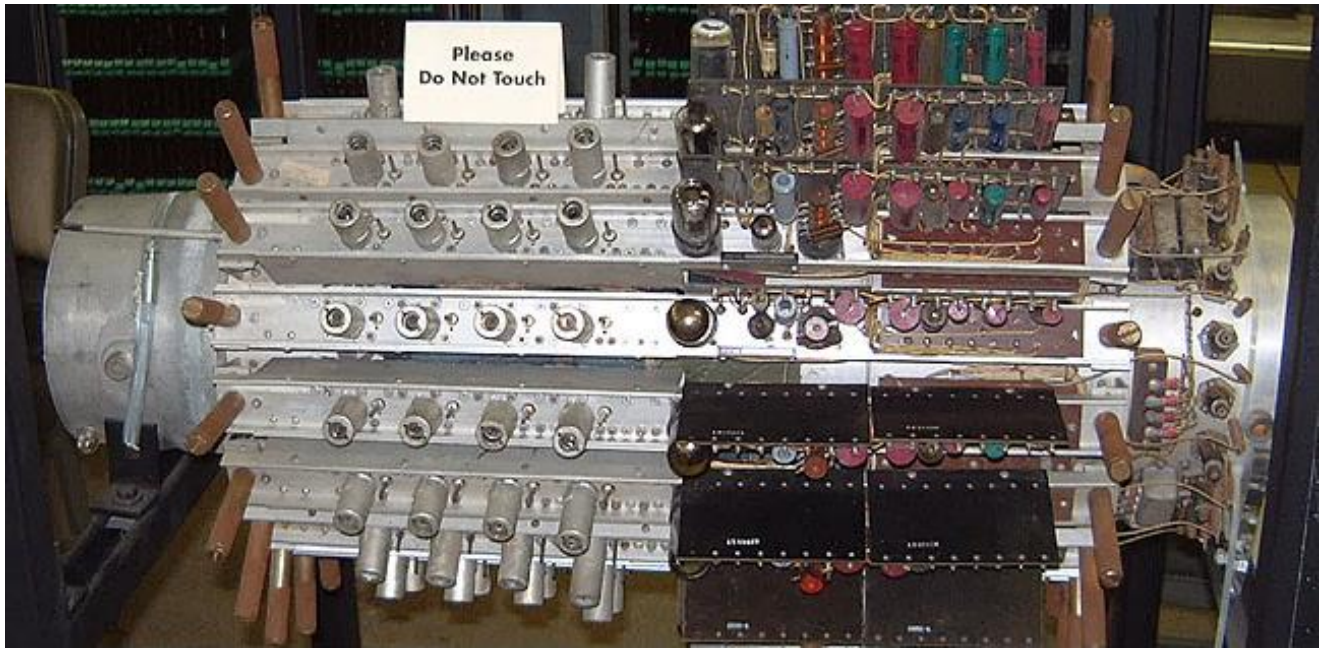
CS History: Delay Line Memory



Mercury memory of UNIVAC I (1951)

- The first computer memory systems, used in the 1940s for the EDVAC and UNIVAC
- Originally used by radar
- Stored a series of audio pulses in a liquid medium
- To save the value, repeat the pulses after they are read

CS History: Delay Line Memory



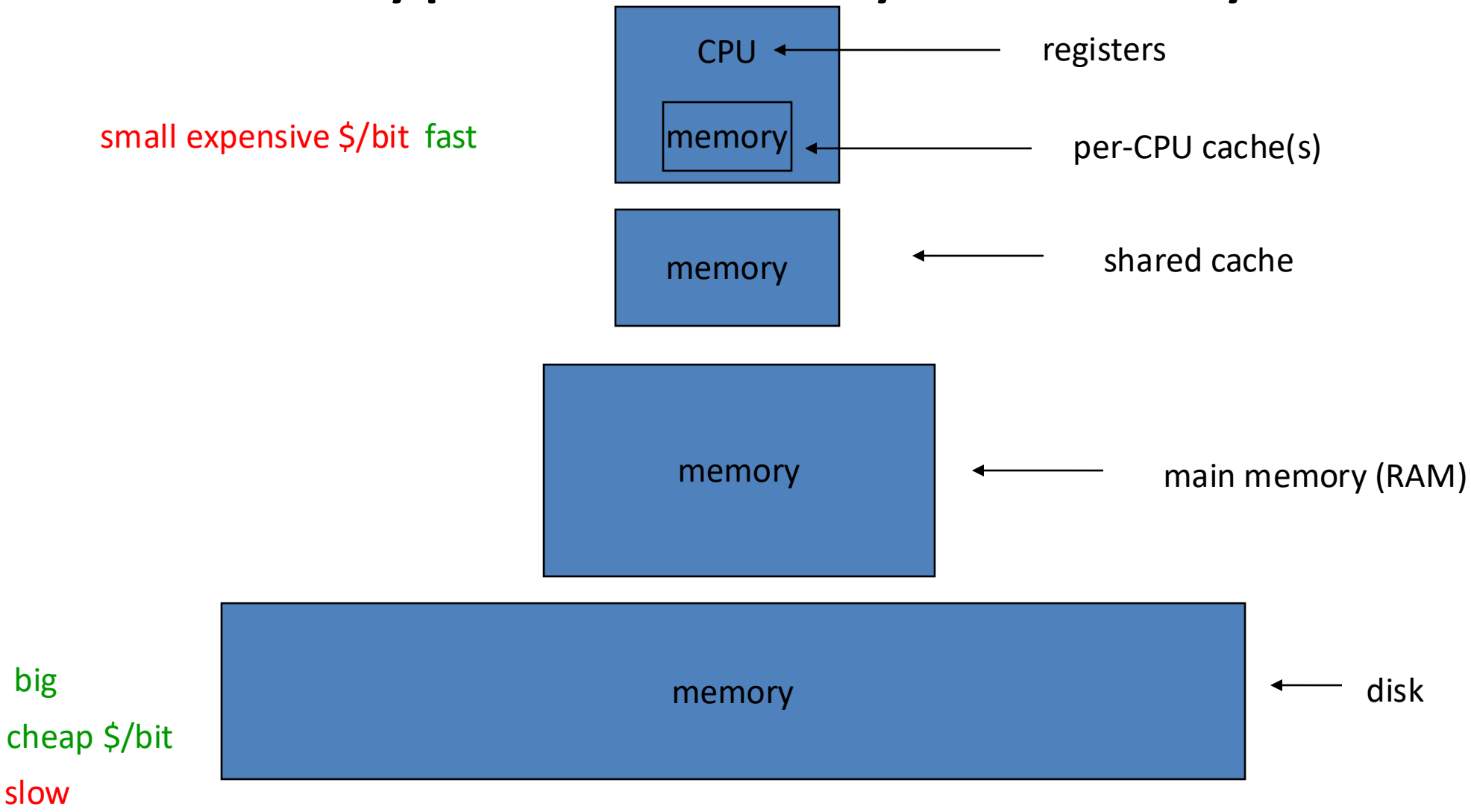
Mercury memory of UNIVAC I (1951)

- Used mercury as the storage medium
- Had to be kept at 104 degrees F
- Made a sound like a human mumbling, giving them the name “mumble-boxes”
- Alan Turing proposed using gin as the liquid medium

Memory

- So far we have only looked at the CPU/datapath
- Now we’re going to look at memory

A typical memory hierarchy



1 ns = 1 billionth of a second
1 ms = 1 thousandth of a second

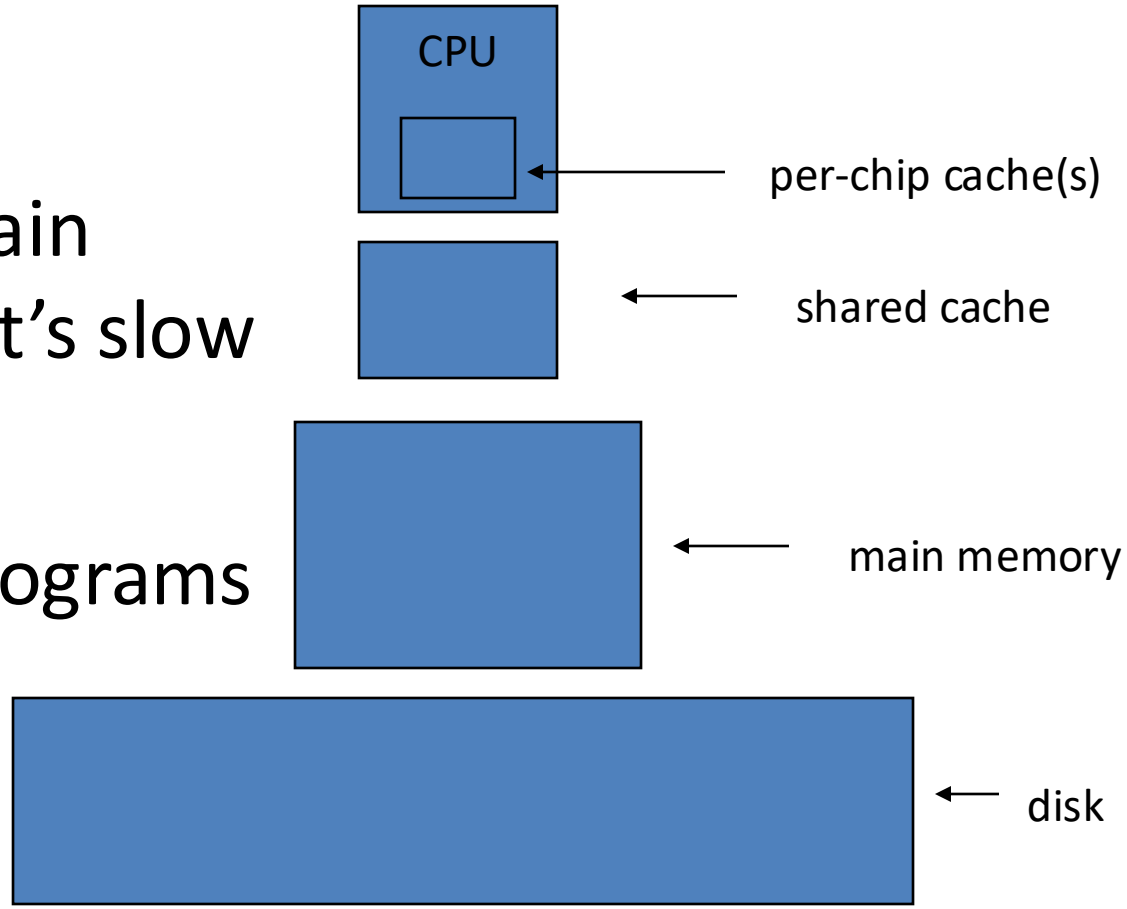
Latency

Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia

Caching

- Everything is on disk, very few things are in the registers
- Want to avoid going to main memory or disk because it's slow
- Take advantage of how programs actually access memory

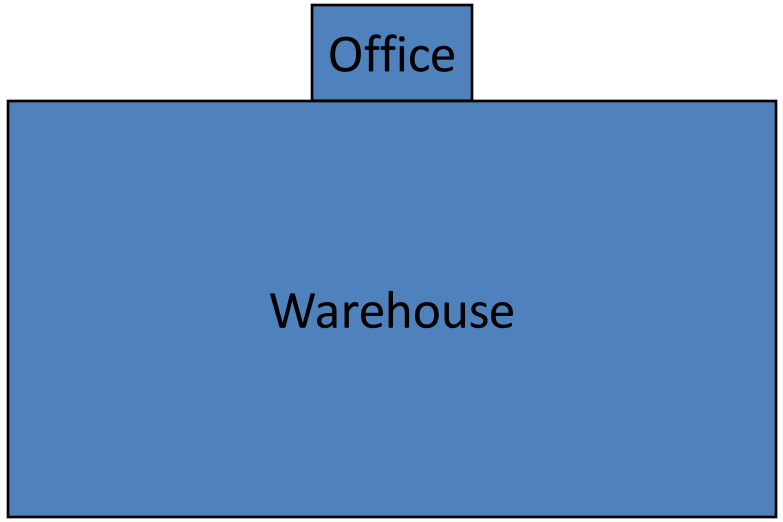


Principle of Locality

- Programs access a small proportion of their address space at any time
- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, registers spilled to the stack
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Library

- You have a huge library with EVERY book ever made.
- Getting a book from the library's warehouse takes 15 minutes.
- You can't serve enough people if every checkout takes 15 minutes.
- You have some small shelves in the front office.

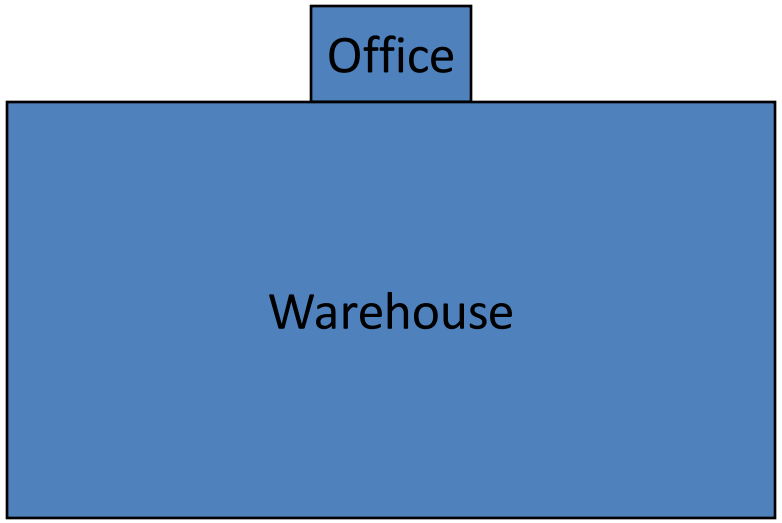


Here are some suggested improvements to the library:

1. Whenever someone checks out a book, keep other copies in the front office for a while in case someone else wants to check out the same book.
2. Watch the trends in books and attempt to guess books that will be checked out soon – put those in the front office.
3. Whenever someone checks out a book in a series, grab the other books in the series and put them in the front.
4. Buy motorcycles to ride in the warehouse to get the books faster

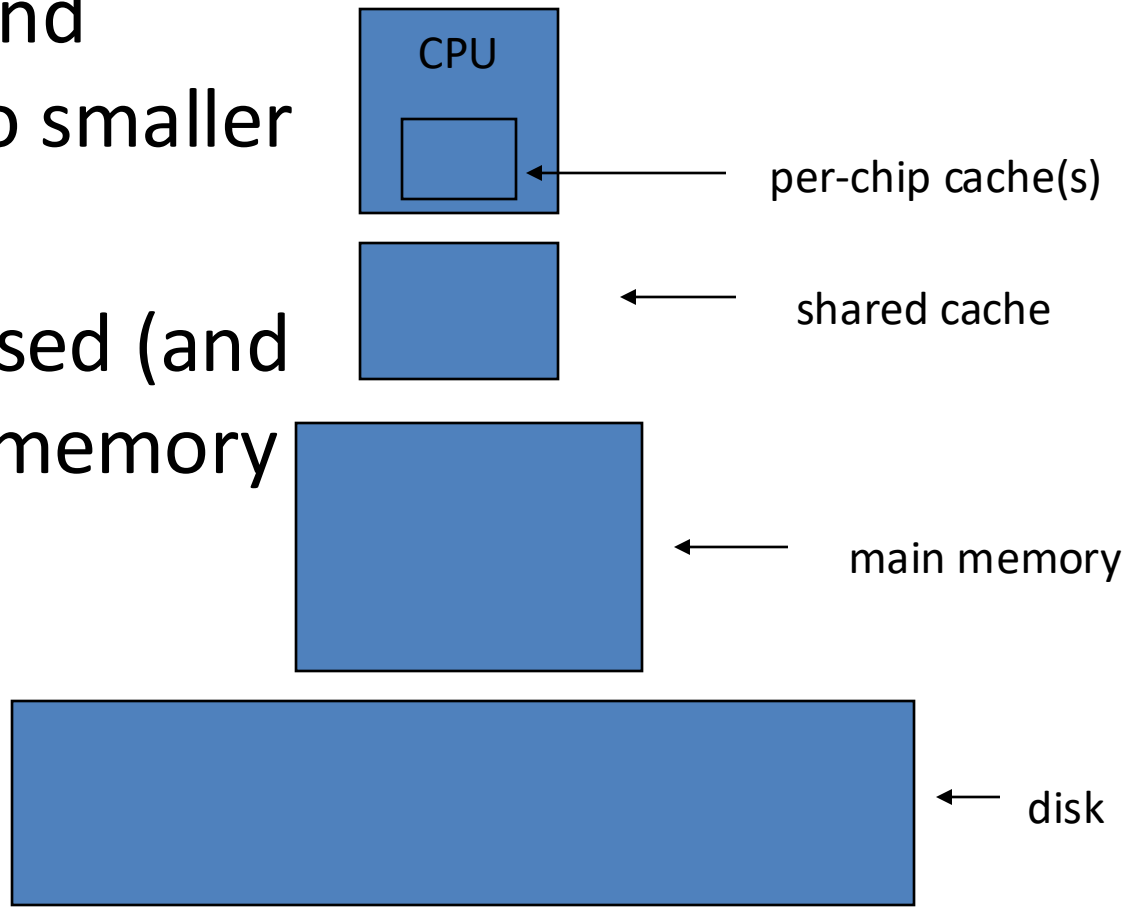
Extending the analogy to locality for caches, which pair of changes most closely matches the analogous cache locality?

Selection	Spatial	Temporal
A	2	1
B	4	2
C	4	3
D	3	1
E	None of the above	



Taking Advantage of Locality

- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller main memory
- Copy more recently accessed (and nearby) items from main memory to cache

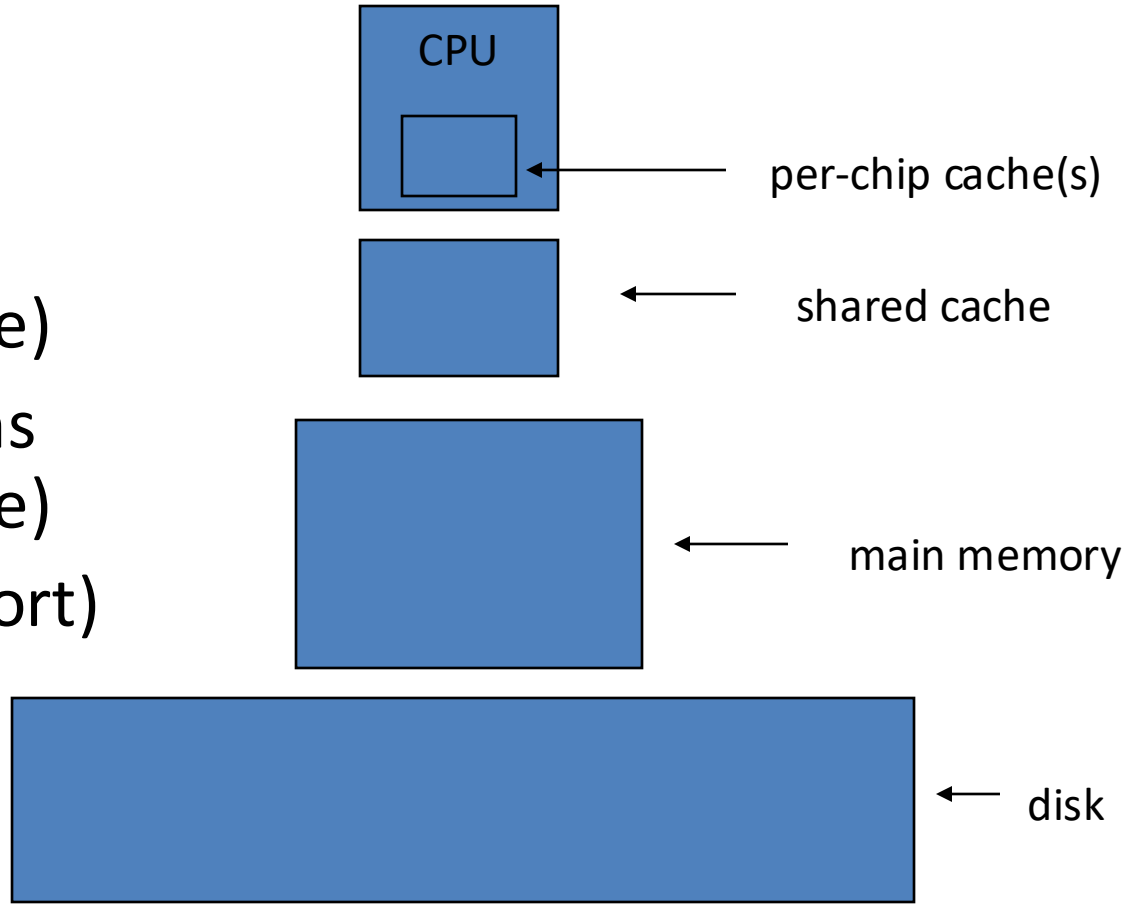


We know SRAM is very fast, expensive ($/GB), and small. We also know disks are slow, inexpensive ($/GB), and large. Which statement best describes the **main goal** of caching.

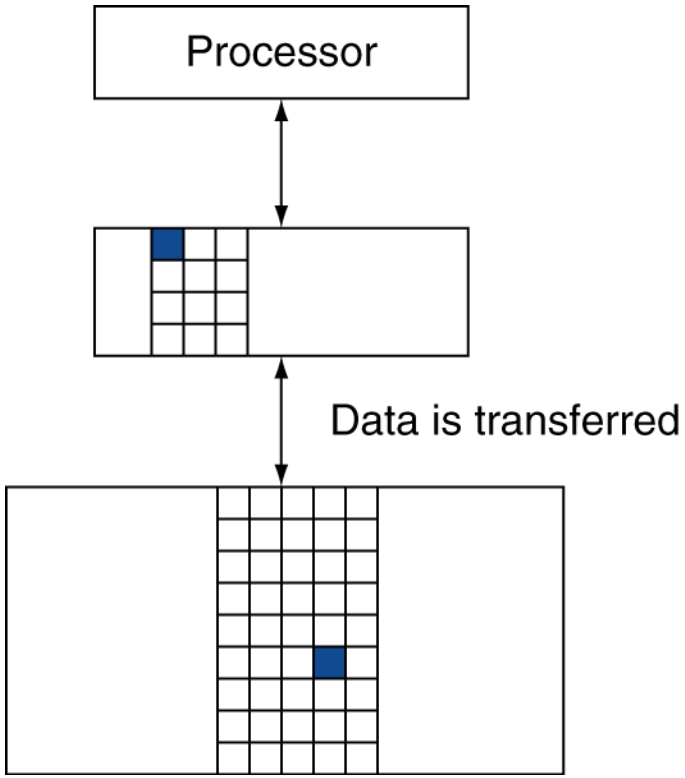
Selection	Role of caching
A	Locality allows us to keep frequently touched data in SRAM.
B	Locality allows us the illusion of memory as fast as SRAM but as large as a disk.
C	SRAM is too expensive to make large – so it must be small and caching helps use it well.
D	Disks are too slow – we have to have something faster for our processor to access.
E	None of these accurately describes the role of cache.

Memory Access

- Use main memory addresses
- When looking for data, check
  - 1. caches (happens automatically in hardware)
  - 2. main memory (happens automatically in hardware)
  - 3. disk (requires OS support)



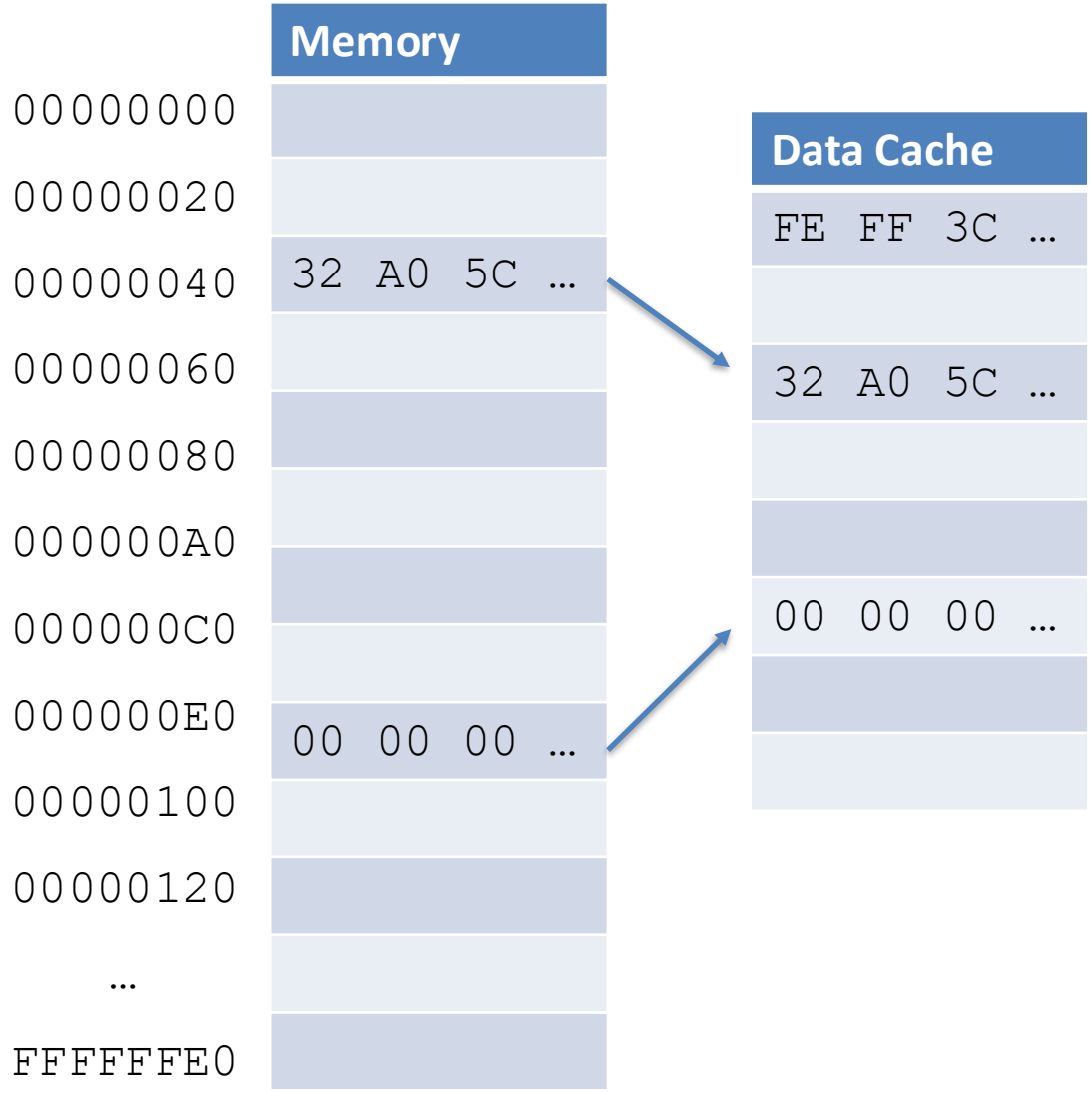
Memory Hierarchy Terms



- Block: unit of copying
  - May be multiple words
  - On x86-64, a block is 64 bytes
- Cache Hit: data in the cache
  - Hit ratio: hits/accesses
- Cache Miss: data not in the cache
  - Time taken: miss penalty
  - Miss ratio: misses/accesses
    = 1 – hit ratio

High-level cache strategy

- Divide all of memory into consecutive blocks
- Copy data (memory ↔ cache) one block (e.g., 64 bytes) at a time
- To access data, check if it exists in the cache before checking memory



Memory addresses, block addresses, offsets

- Imagine we have blocks of size 32 bytes (not bits!)
- Every byte of memory can be specified by giving
  - A (32 – 5)-bit block address (in purple)
  - A 5-bit offset into the block (in green)
- To read a byte of memory
  - find the appropriate 32-byte block in either cache or memory using the block address
  - Use the offset to select the appropriate byte from the block

With a block size of 64 bytes, how many bits is the block address? How many bits is the offset? (Assume 32-bit addresses.)

A. Block address size is 32 – 4 = 28 bits; offset size is 4 bits
B. Block address size is 32 – 5 = 27 bits; offset size is 5 bits
C. Block address size is 32 – 6 = 26 bits; offset size is 6 bits
D. Block address size is 32 – 5 = 27 bits; offset size is 4 bits
E. Block address size is 32 – 5 = 27 bits; offset size is 6 bits

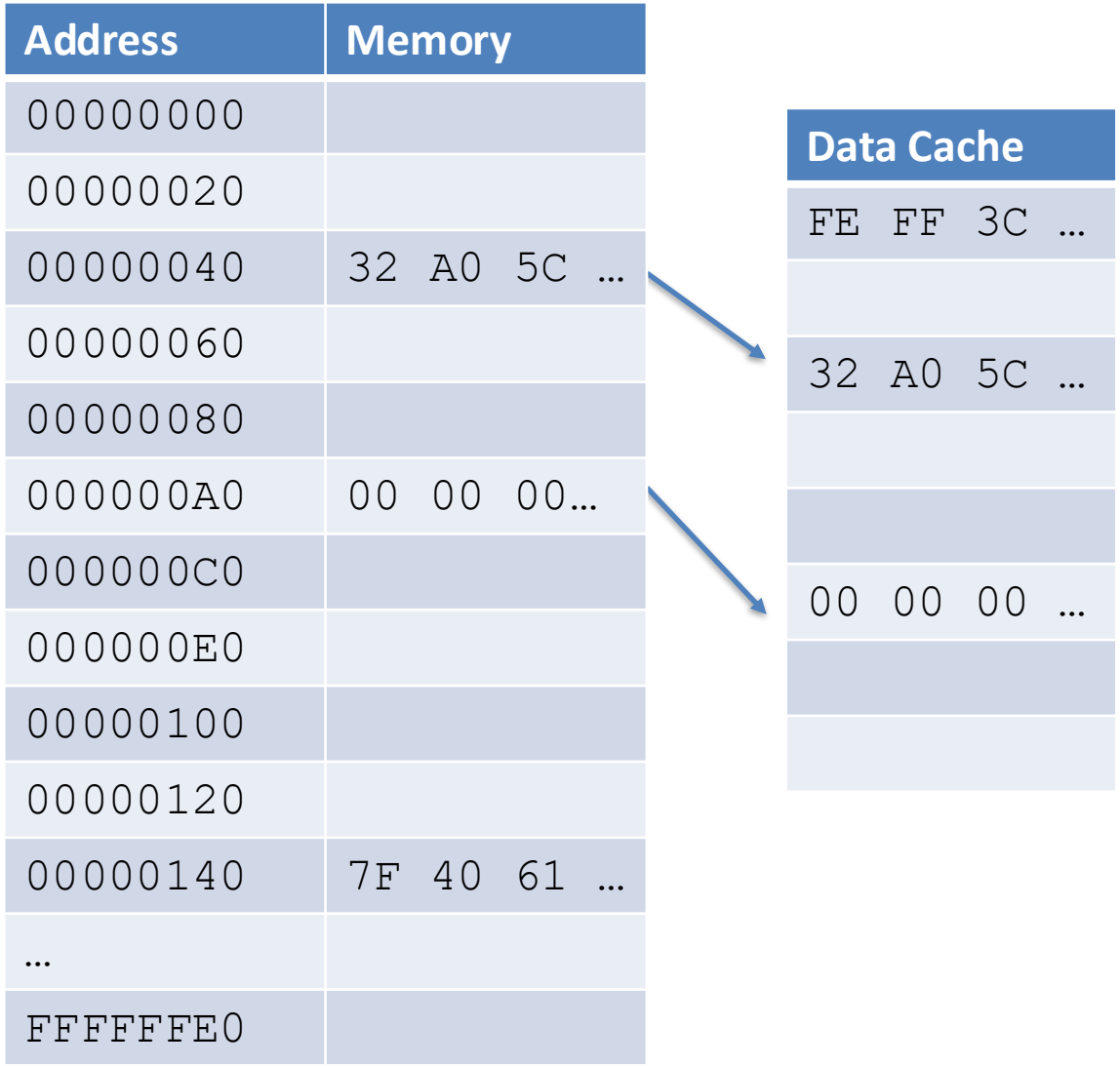
Number of offset bits

- Block sizes are powers of 2

- For a block size of 2^m bytes, the number of offset bits is *m*
  - 16-byte block size: 4 offset bits
  - 32-byte block size: 5 offset bits
  - 64-byte block size: 6 offset bits

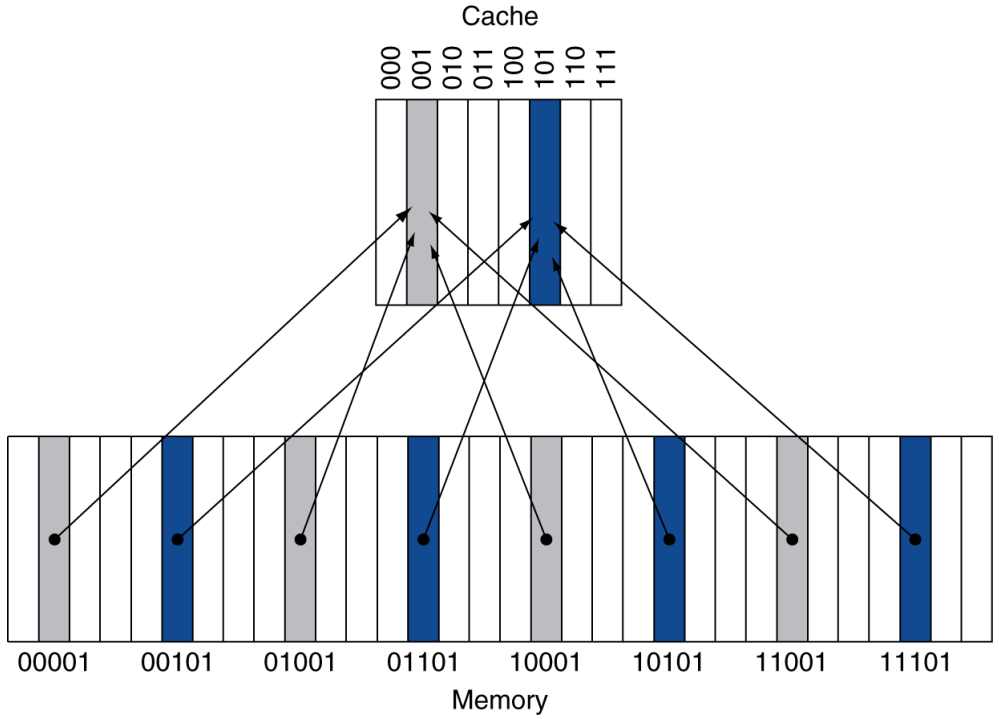
Where is a block of memory stored in cache?

- Given a memory address, we can divide it into a block address and an offset
- Where in cache is the block stored?
- Basic problem: Cache is smaller than main memory



Direct-mapped cache

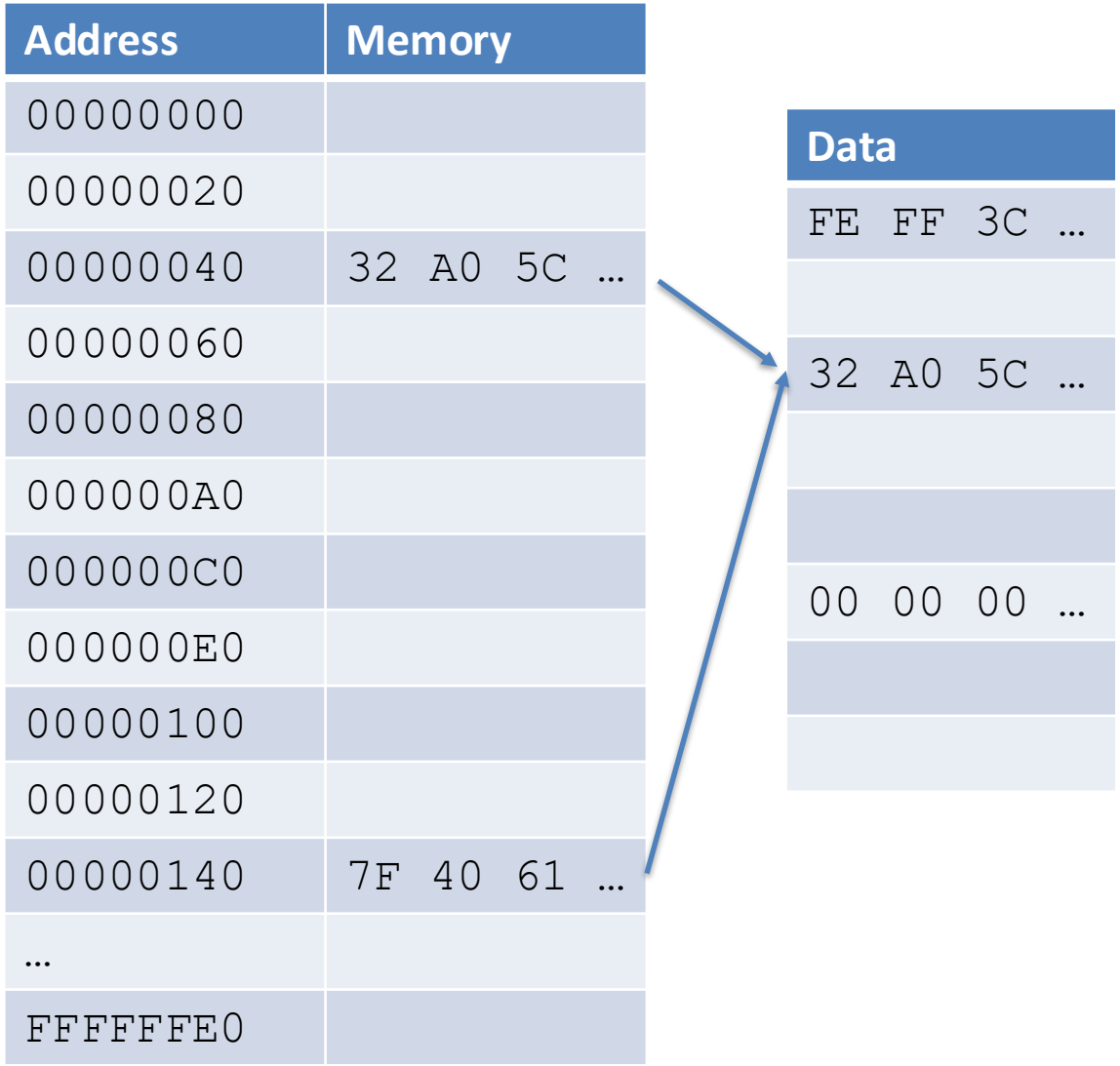
- Block location in cache determined by block address
- Direct mapped: only one possible location for a given block address
  - Index = (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Direct-mapped cache is essentially an array of blocks
- Use low-order address bits of **block address** to index it

Problem: Collisions

- Many block addresses map to the same cache location
- How do we know which particular block is stored in a cache location?
  - Store block address as well as the data
  - Actually, only need the high-order bits
  - Called the **tag**



Memory addresses, block addresses, offsets

- Block size of 32 bytes (not bits!)
- 8-block cache (this is purely an example!)
- Each address
  - A (32 – 5)-bit block address (in purple and blue)
  - A 5-bit offset into the block (in green)
- Block address can be divided into
  - A (32 – 3 – 5)-bit **tag** (purple)
  - A 3-bit cache **index** (blue)

Cache layout (so far)

- Tag stores high-order bits of address
- Data stores all of the data for the block (e.g., 32 bytes)

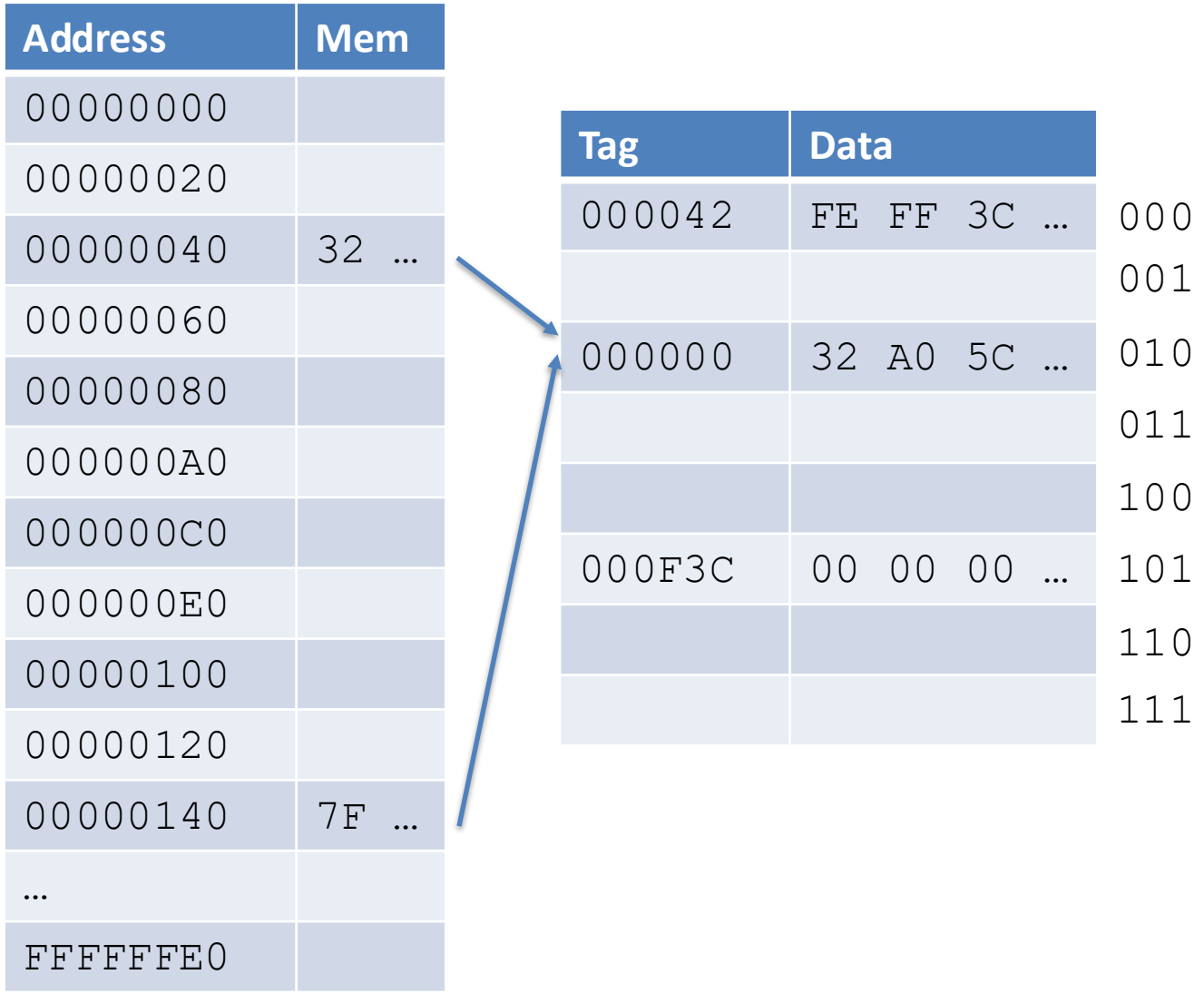
Tag	Data
000042	FE FF 3C 7F …
001234	32 A0 5C 21 …
000F3C	00 00 00 00 …

If we have a block size of 64-bytes and our cache holds 256 entries how large are the tag, index, and offset?

	Tag size (bits)	Index size (bits)	Offset size (bits)
A	32 – 3 – 8	3	8
B	32 – 3 – 6	3	6
C	32 – 6 – 8	6	8
D	32 – 8 – 6	8	6
E	32 – 8 – 8	8	8

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

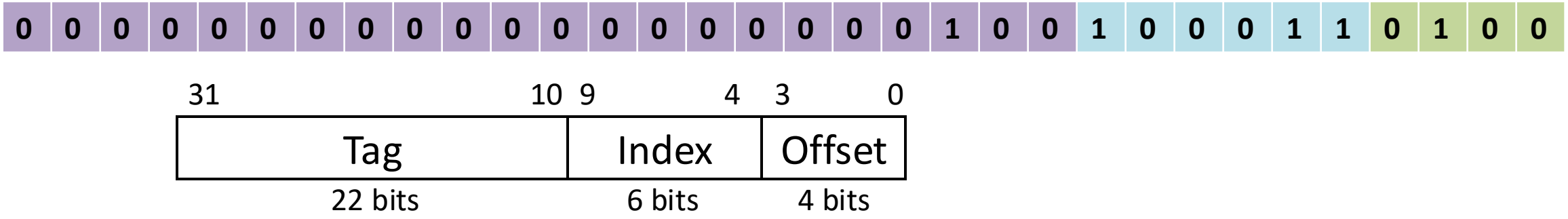
- Divide all of memory into consecutive blocks
- Copy data (memory ↔ cache) one block at a time
- Cache lookup:
  - Get the index of the block in the cache from the address
  - Check the valid bit; compare the tag to the address

Address	Mem
00000000	
00000020	
00000040	32 …
00000060	
00000080	
000000A0	
000000C0	
000000E0	
00000100	
00000120	
00000140	7F …
…	
FFFFFFE0	

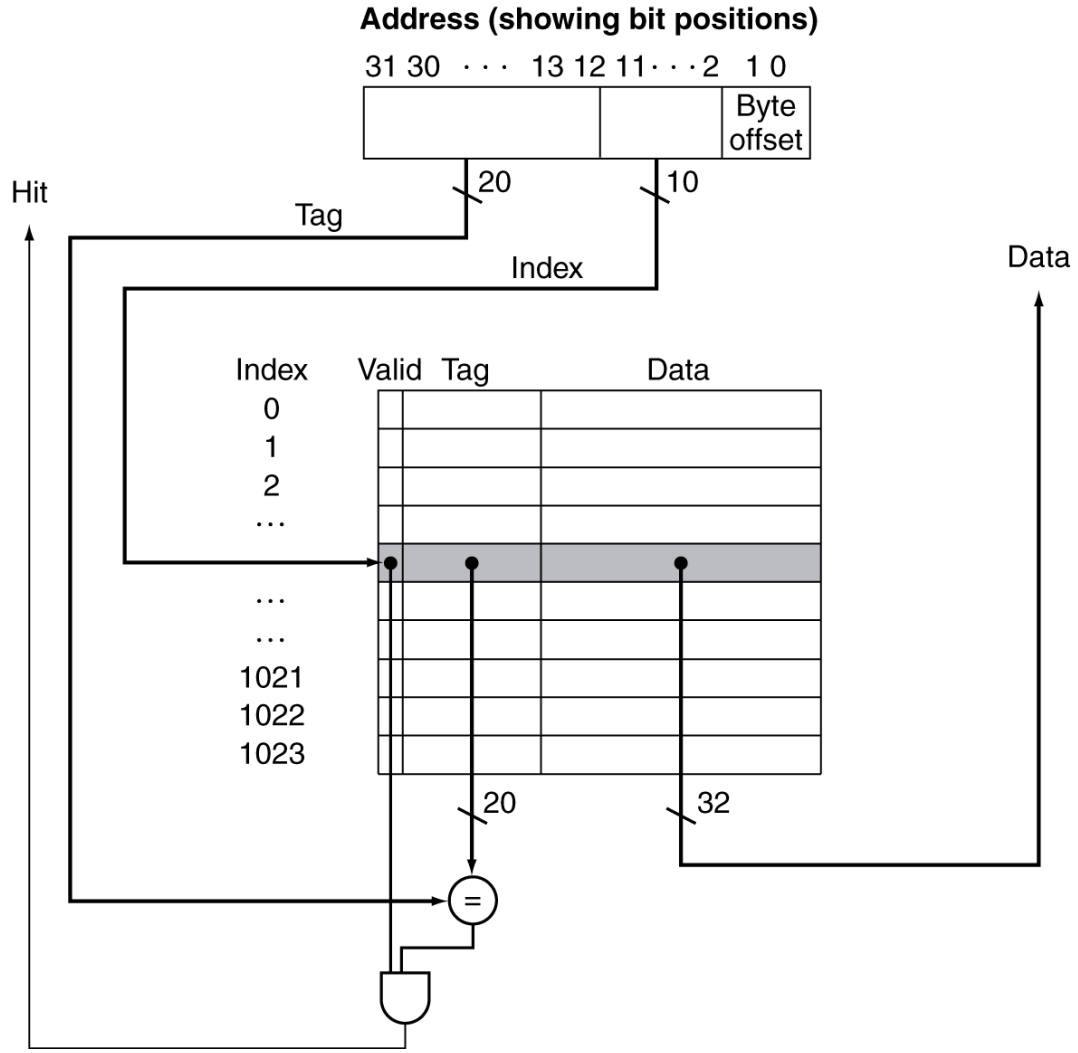
V	Tag	Data
1	000042	FE FF 3C …
0		
1	000000	32 A0 5C …
0		
0		
1	000F3C	00 00 00 …
0		
0		

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; different example!



Cache behavior

- Lookup block in cache by index bits from the address
- If the the block's valid bit is set and the tag matches the tag bits from the address (a hit!), return the data
- If the block's valid bit is not set or the tag doesn't match the tag bits from the address (a miss!)
  - Copy the data from main memory into the cache block
  - Set the block's valid bit to 1 and tag to the tag bits of the address
  - Return the data

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	H
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 specific block in the 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

Reading

- Next lecture: More Caches!
  - Section 6.2