

# **CSC 252/452: Computer Organization**

## **Fall 2024: Lecture 15**

Instructor: Yanan Guo

Department of Computer Science  
University of Rochester

# Data Dependencies: 3 Nop's

	1	2	3	4	5	6	7	8	9	10	11
0x000: irmovq \$10,%rdx	F	D	E	M	W						
0x00a: irmovq \$3,%rax		F	D	E	M	W					
0x014: nop			F	D	E	M	W				
0x015: nop				F	D	E	M	W			
0x016: nop					F	D	E	M	W		
0x017: addq %rdx,%rax						F	D	E	M	W	
0x019: halt							F	D	E	M	W

**addq reads the correct %rdx  
and %rax**

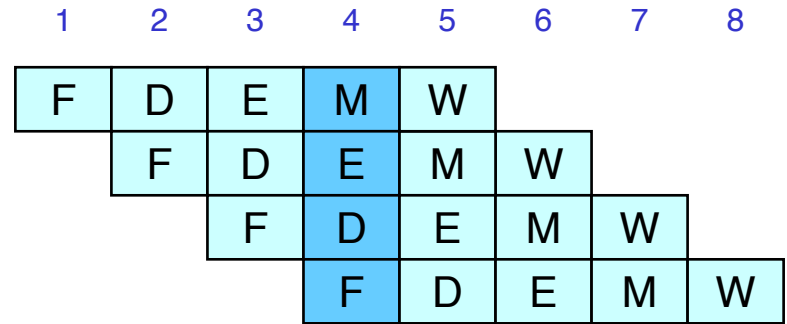
# Data Forwarding Example

0x000: `irmovq $10,%rdx`

0x00a: `irmovq $3,%rax`

0x014: `addq %rdx,%rax`

0x016: `halt`



Register `%rdx`

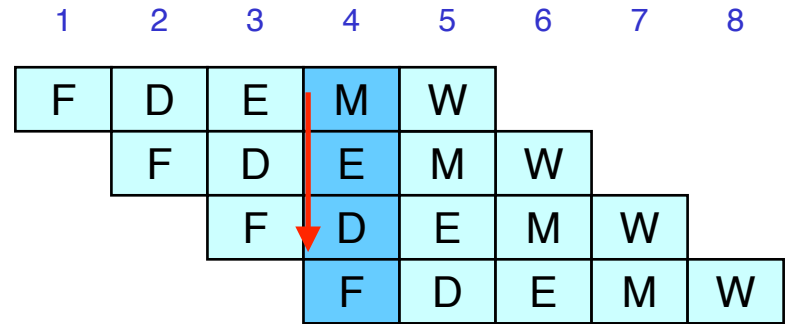
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

# Data Forwarding Example

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

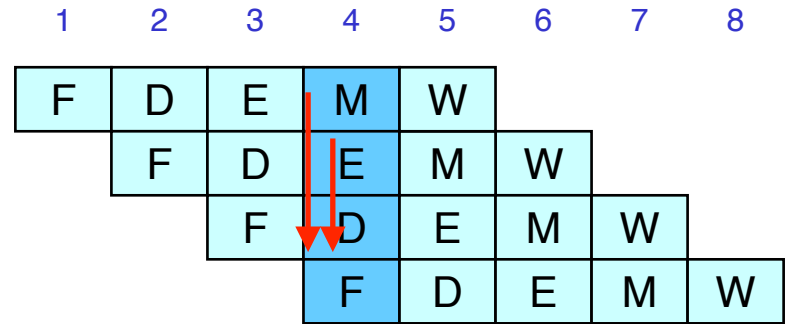
- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

# Data Forwarding Example

```
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Register `%rdx`

- Forward from the memory stage

Register `%rax`

- Forward from the execute stage

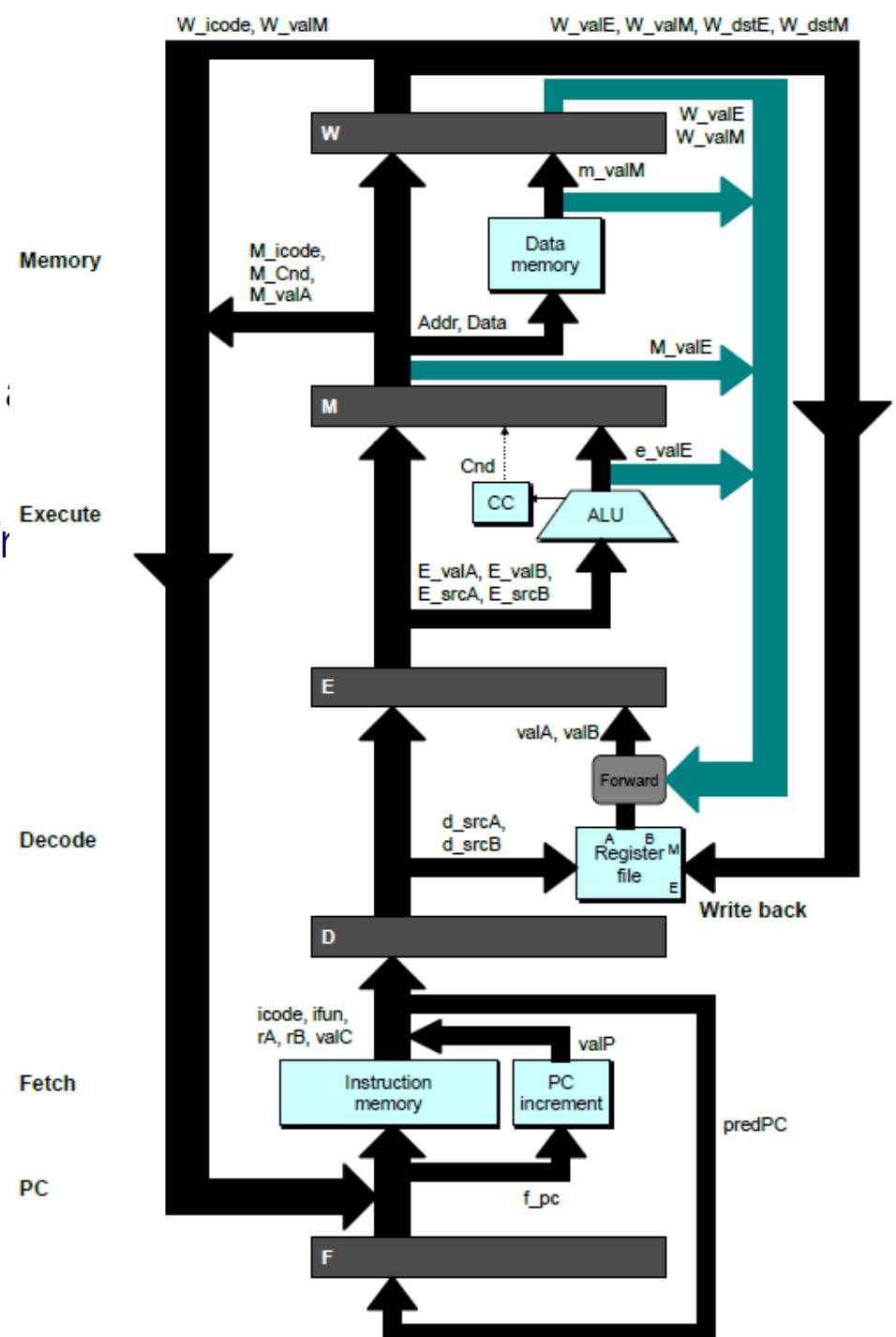
# Bypass Paths

## Decode Stage

- Forwarding logic selects valA or valB from
- Normally from register file
- Forwarding: get valA or valB from

## Forwarding Sources

- Execute: valE
- Memory: valE, valM
- Write back: valE, valM



# Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.  
Forces the pipeline to stall.

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r4 = **r3** + r6  
r7 = r5 + r1  
...



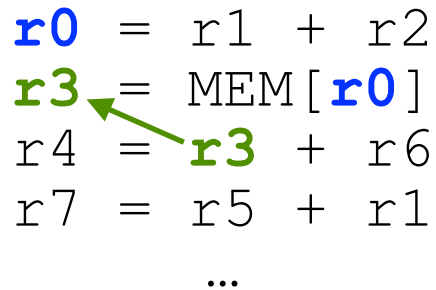
**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r7 = r5 + r1  
...  
r4 = **r3** + r6

# Out-of-order Execution

- Compiler could do this, but has limitations
- Generally done in hardware

Long-latency instruction.  
Forces the pipeline to stall.

**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r4 = **r3** + r6  
r7 = r5 + r1  
...



**r0** = r1 + r2  
**r3** = MEM[**r0**]  
r7 = r5 + r1  
...  
r4 = **r3** + r6



# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

# Out-of-order Execution

```
r0 = r1 + r2  
r3 = MEM[r0]  
r4 = r3 + r6  
r6 = r5 + r1  
...
```

Is this correct?



```
r0 = r1 + r2  
r3 = MEM[r0]  
r6 = r5 + r1  
...  
r4 = r3 + r6
```

```
r0 = r1 + r2  
r3 = MEM[r0]  
r4 = r3 + r6  
r4 = r5 + r1  
...
```

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r4 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r5 + r1
...
r4 = r3 + r6
```

# Out-of-order Execution

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r6 = r5 + r1
...
```

Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r6 = r5 + r1
...
r4 = r3 + r6
```

```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r3 + r6
r4 = r5 + r1
...
```

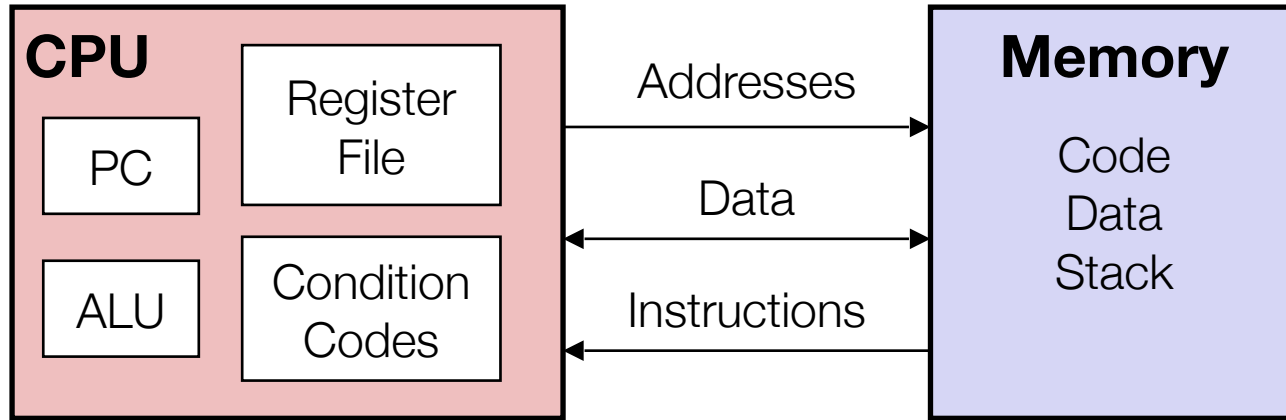
Is this correct?



```
r0 = r1 + r2
r3 = MEM[r0]
r4 = r5 + r1
...
r4 = r3 + r6
```

“**Tomasolu Algorithm**” is the algorithm that is most widely implemented in modern hardware to get out-of-order execution right.

# So far in 252...



- We have been discussing the CPU microarchitecture
  - Single Cycle, sequential implementation
  - Pipeline implementation
  - Resolving data dependency and control dependency
- What about memory?

# Ideal Memory

- Low access time (latency)
- High capacity
- Low cost
- High bandwidth (to support multiple accesses in parallel)

# The Problem

- Ideal memory's requirements oppose each other



# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape

# The Problem

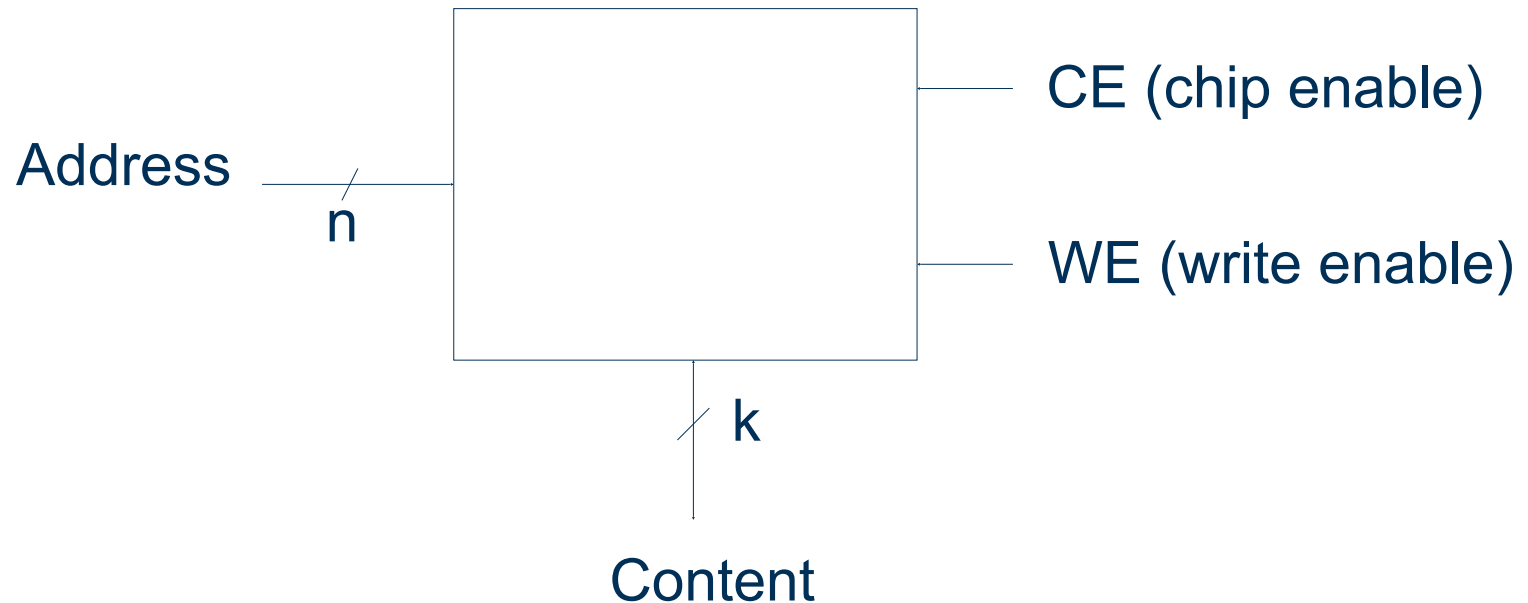
- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive

# The Problem

- Ideal memory's requirements oppose each other
- Bigger is slower
  - Bigger → Takes longer to determine the location
- Faster is more expensive
  - Memory technology: Flip-flop vs. SRAM vs. DRAM vs. Disk vs. Tape
- Higher bandwidth is more expensive
  - Need more ports, higher frequency, or faster technology

# Memory Technology: RAM

- Random access memory
- Random access means you can supply an arbitrary address to the memory and get a value back



# Latch vs. DRAM vs. SRAM

- DFF (Data Flip-Flop)
  - Fastest
  - Low density (27 transistors per bit)
  - High cost
- SRAM (Static RAM)
  - Faster access (no capacitor)
  - Lower density (6 transistors per bit; there are designs w/ fewer Ts)
  - Higher cost
  - Lower power consumption compared to DRAM
  - Manufacturing compatible with logic process (no capacitor)
- DRAM (Dynamic RAM)
  - Slower access (capacitor)
  - Higher density (1 transistor + 1 capacitor per bit)
  - Lower cost
  - Higher power consumption compared to SRAM
  - Manufacturing requires putting capacitor and logic together



# Non-volatile Memories

# Non-volatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.

# Non-volatile Memories

- DFF, DRAM and SRAM are volatile memories
  - Lose information if powered off.
- Nonvolatile memories retain value even if powered off
  - Flash (~ 5 years)
  - Hard Disk (~ 5 years)
  - Tape (~ 15-30 years)

# Summary of Trade-Offs

- Faster is more expensive (dollars and chip area)
  - SRAM, < 10\$ per Megabyte
  - DRAM, < 1\$ per Megabyte
  - Hard Disk < 1\$ per Gigabyte

# Summary of Trade-Offs

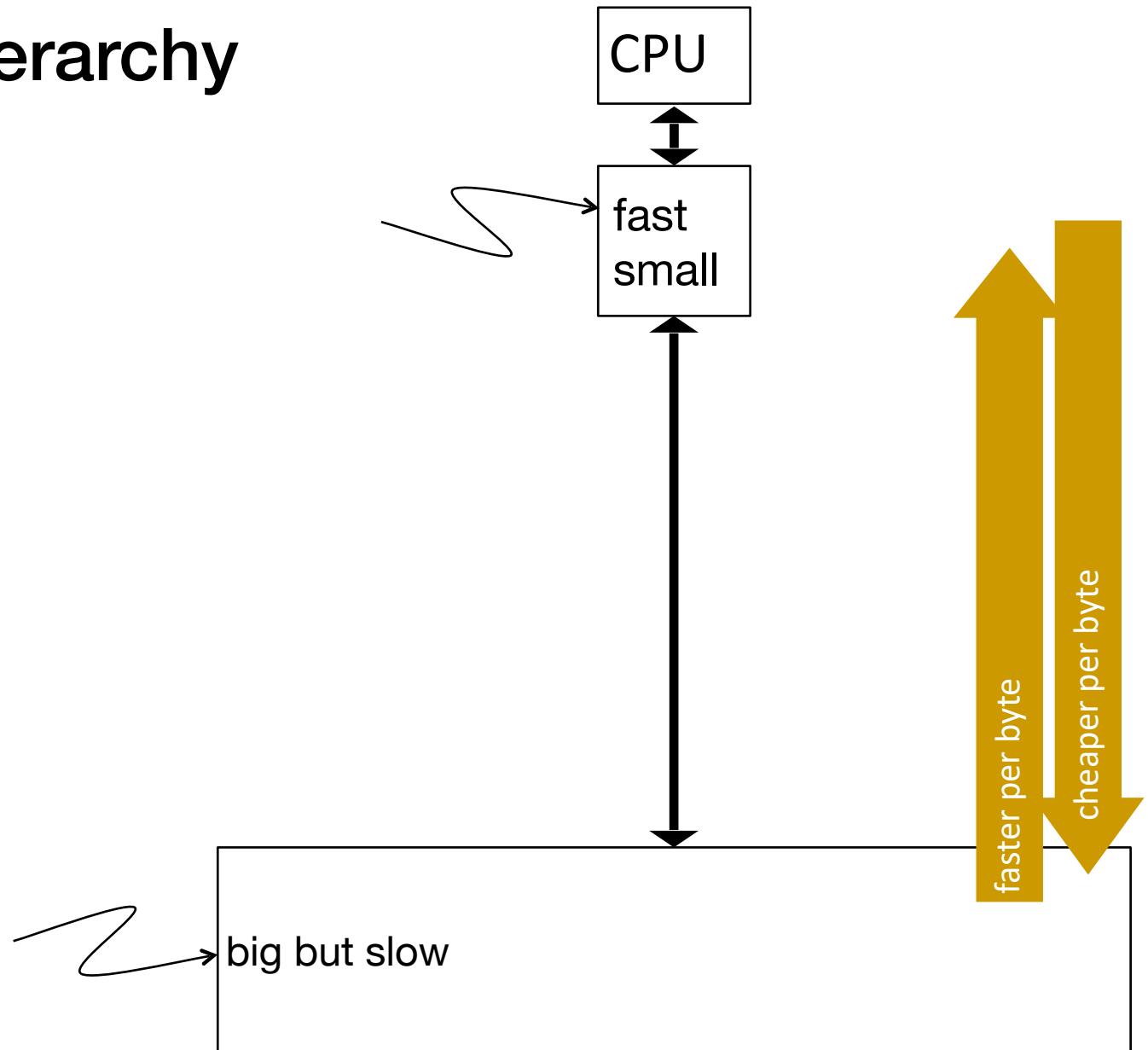
- Faster is more expensive (dollars and chip area)
  - SRAM, < 10\$ per Megabyte
  - DRAM, < 1\$ per Megabyte
  - Hard Disk < 1\$ per Gigabyte
- Larger capacity is slower
  - Flip-flops/Small SRAM, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec
- Other technologies have their place as well
  - PC-RAM, MRAM, RRAM

# We want both fast and large Memory

- But we cannot achieve both with a single level of memory
- Idea: Memory Hierarchy
  - Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor)
  - Key: manage the data such that most of the data the processor needs in the near future is kept in the fast(er) level(s)

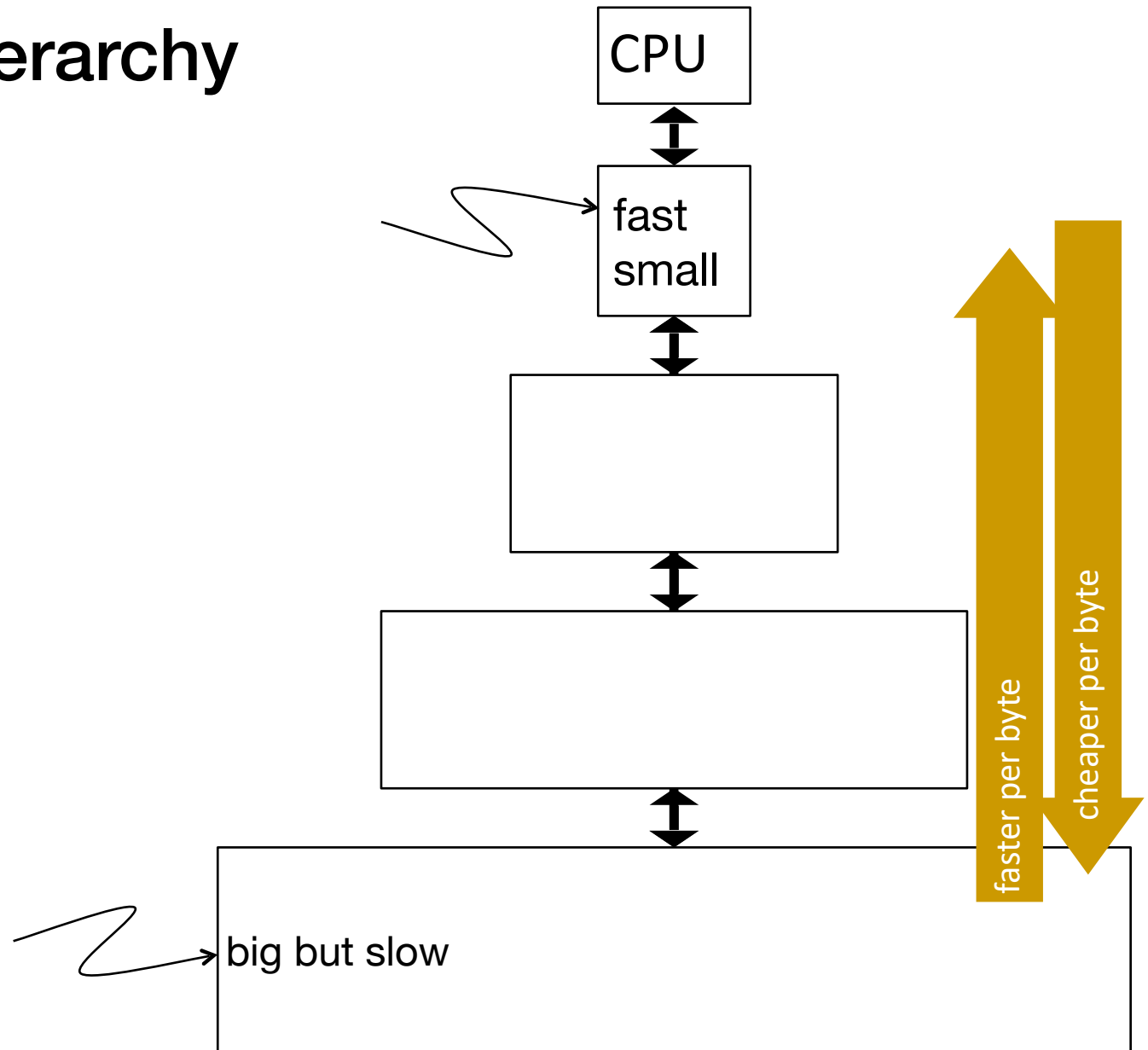
# Memory Hierarchy

backup  
everything  
here



# Memory Hierarchy

backup  
everything  
here

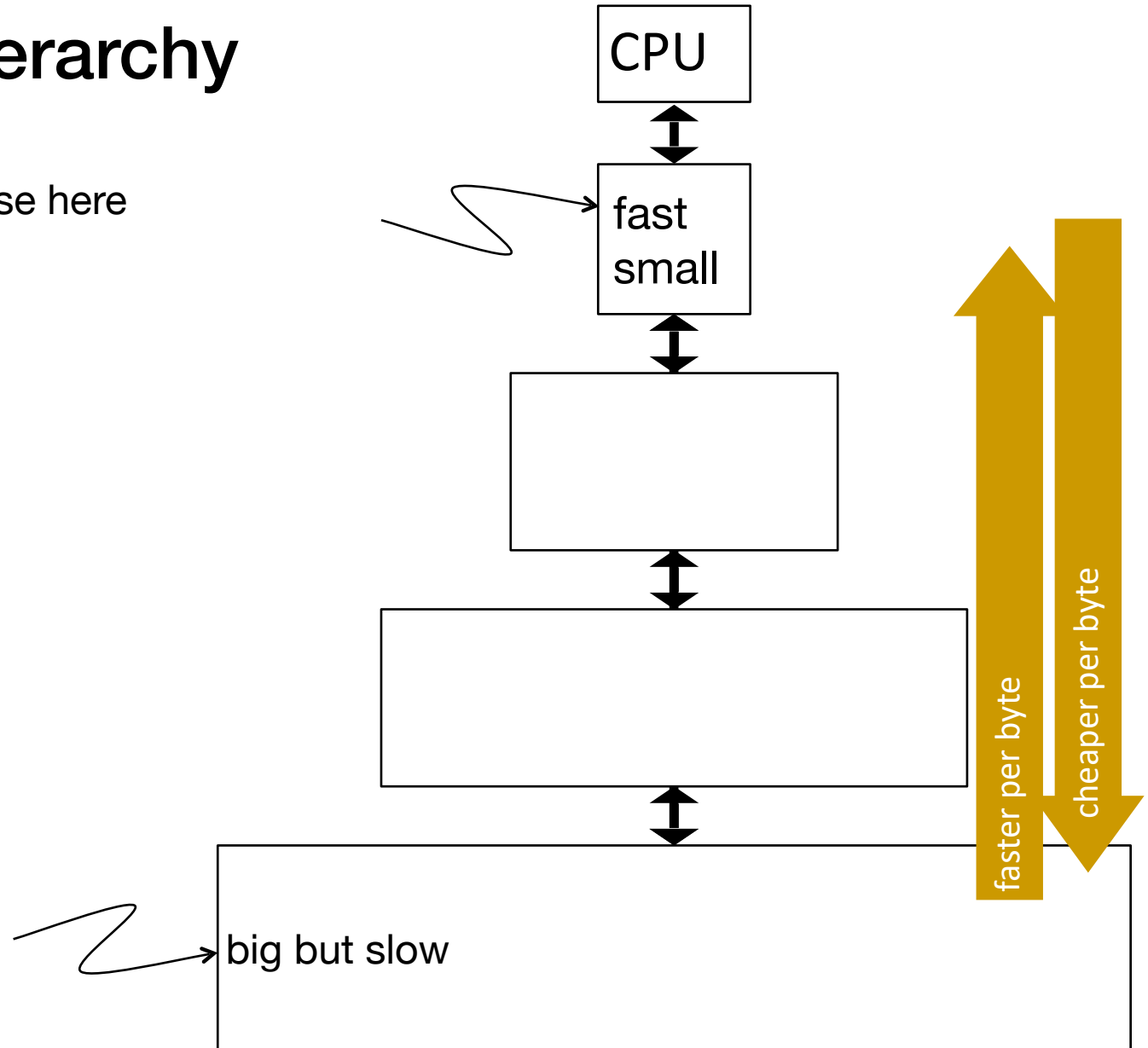




# Memory Hierarchy

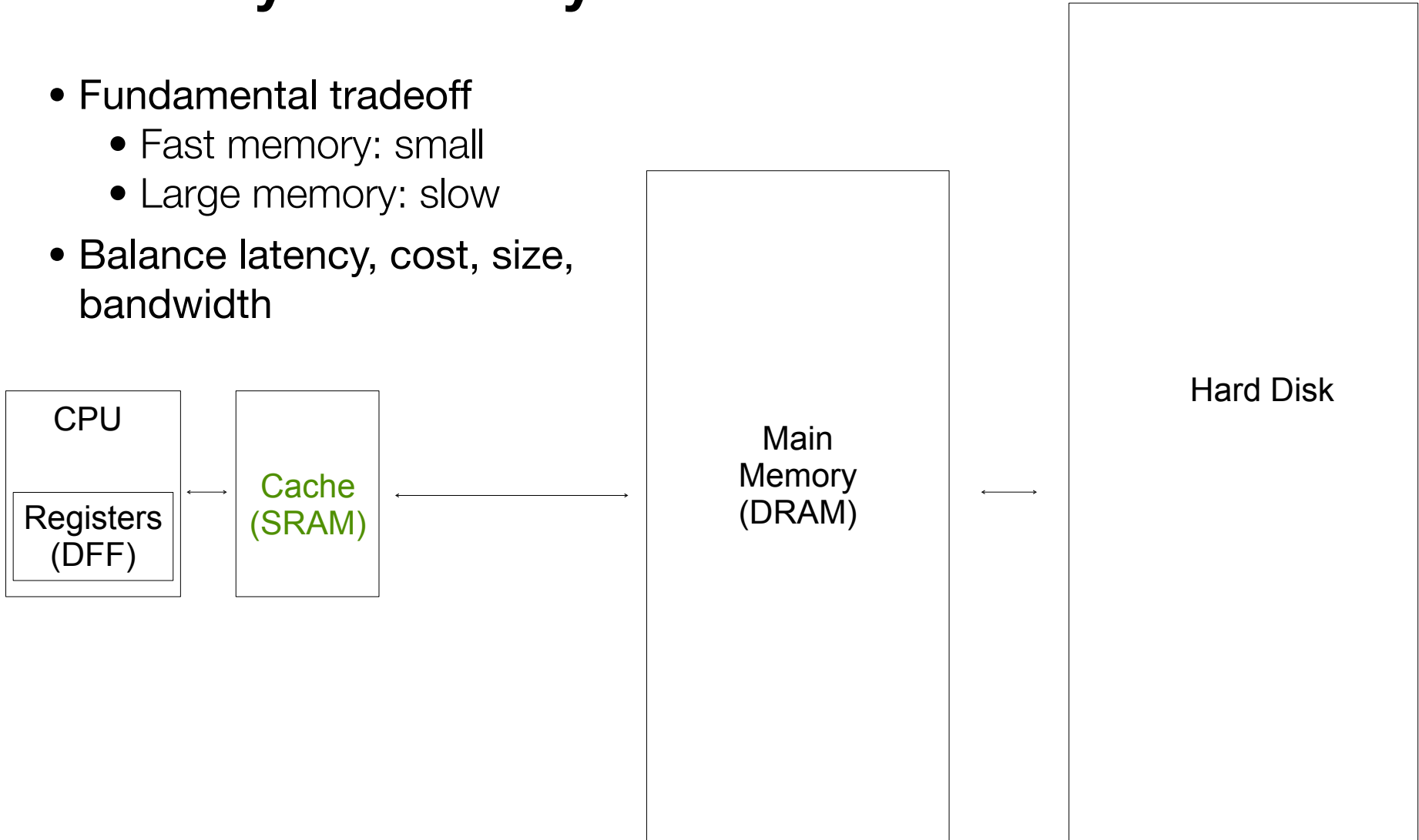
move what you use here

backup  
everything  
here



# Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Balance latency, cost, size, bandwidth



# A Modern Memory Hierarchy

Register File (DFF)  
32 words, sub-nsec

---

L1 cache (SRAM)  
~32 KB, ~nsec

L2 cache (SRAM)  
512 KB ~ 1MB, many nsec

L3 cache (SRAM)

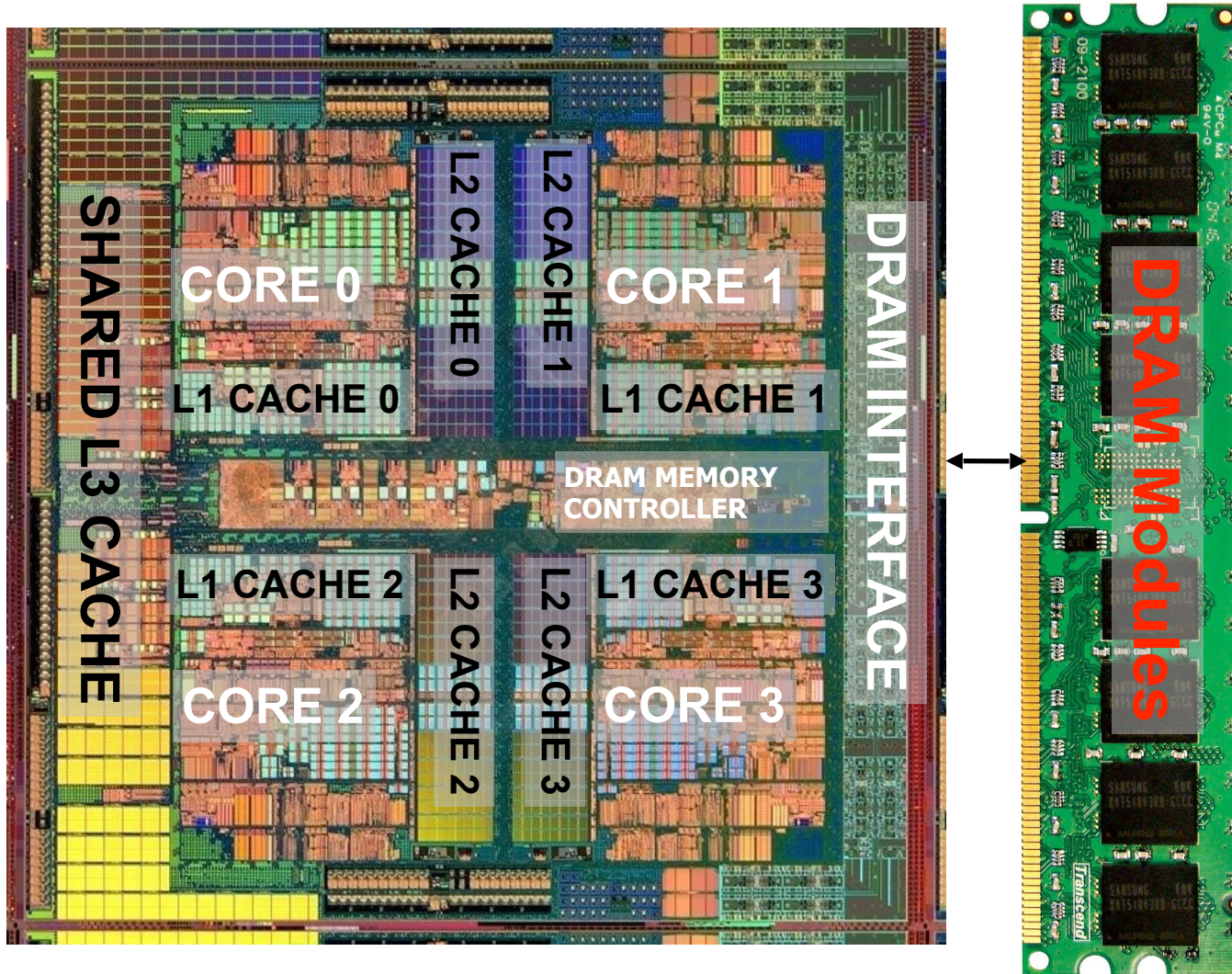
.....

Main memory (DRAM),  
GB, ~100 nsec

---

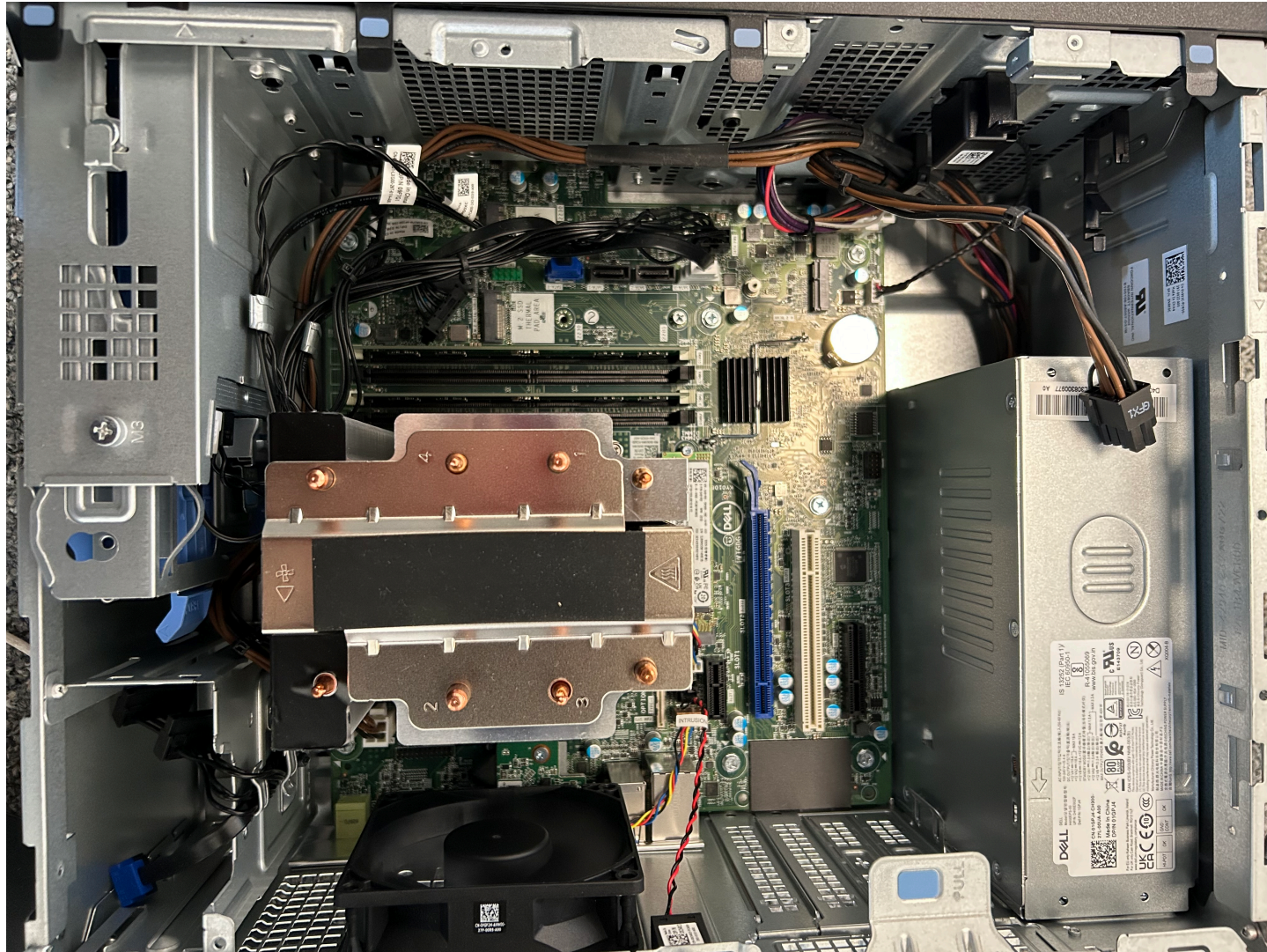
Hard Disk  
100 GB, ~10 msec

# Memory in a Modern System



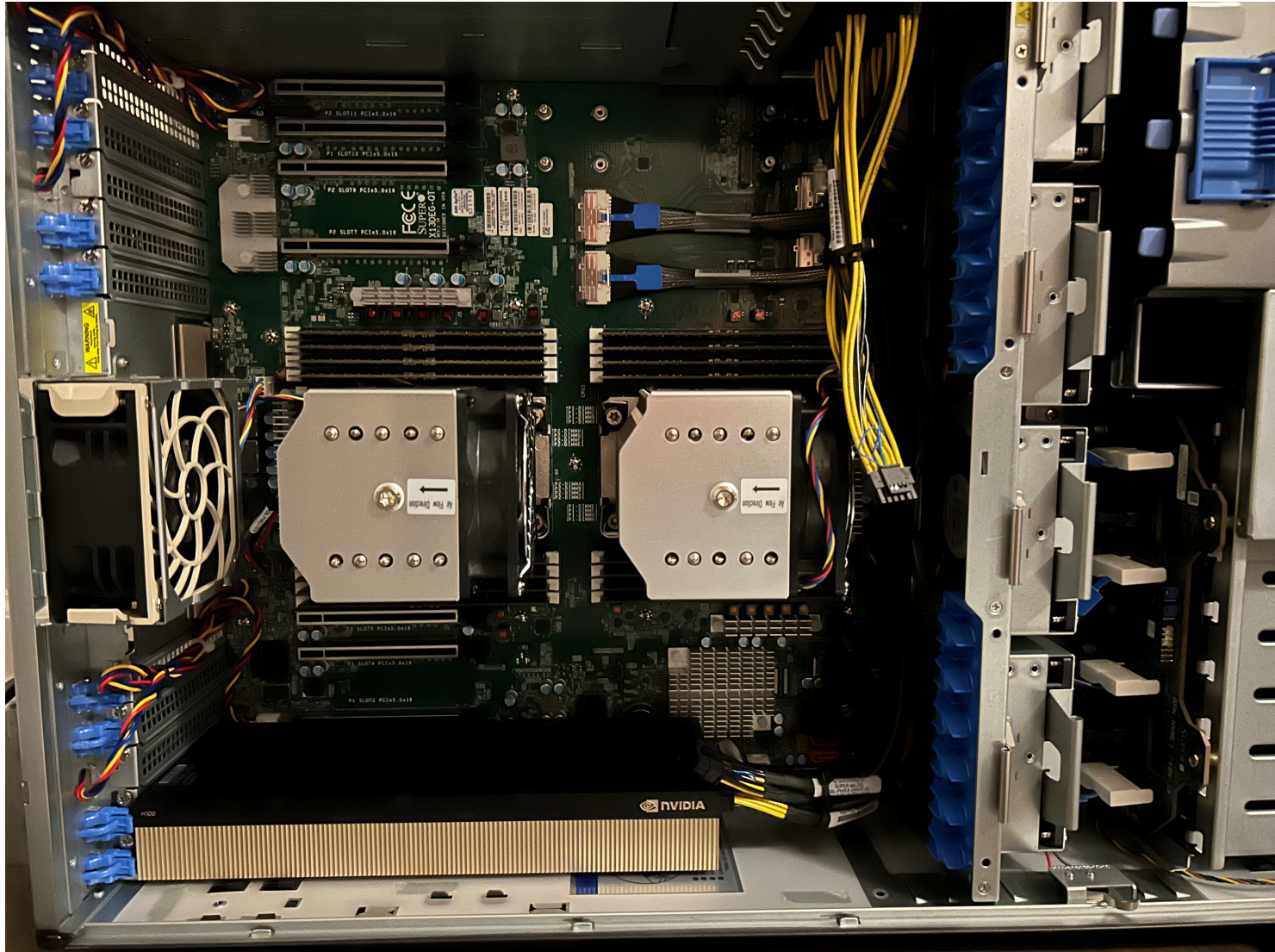


# My Desktop

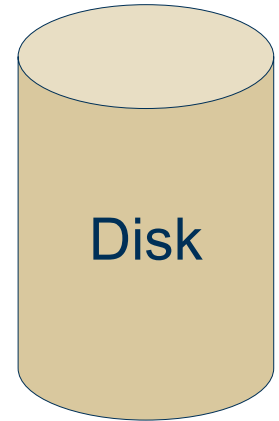
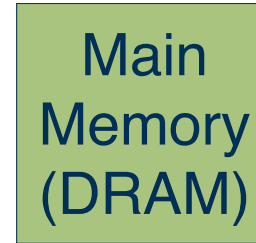
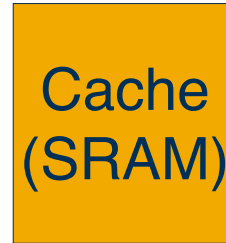
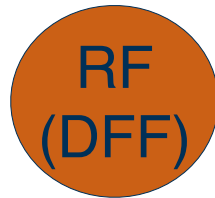




# My Server



# How Things Have Progressed



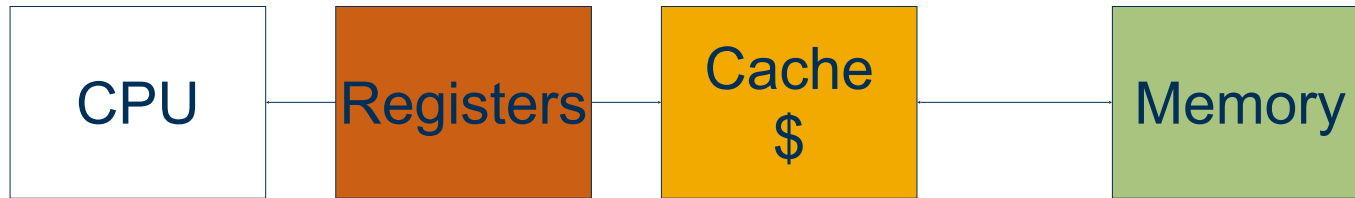
<b>1995 low-mid range</b> <small>Hennessy &amp; Patterson, Computer Arch., 1996</small>	<b>200B</b> <b>5ns</b>	<b>64KB</b> <b>10ns</b>	<b>32MB</b> <b>100ns</b>	<b>2GB</b> <b>5ms</b>
<b>2009 low-mid range</b> <small><a href="http://www.dell.com">www.dell.com</a>, \$449 including 17" LCD flat panel</small>	<b>~200B</b> <b>0.33ns</b>	<b>8MB</b> <b>0.33ns</b>	<b>4GB</b> <b>&lt;100ns</b>	<b>750GB</b> <b>4ms</b>
<b>2015 mid range</b>	<b>~200B</b> <b>0.33ns</b>	<b>8MB</b> <b>0.33ns</b>	<b>16GB</b> <b>&lt;100ns</b>	<b>256GB</b> <b>10us</b>

# How to Make Effective Use of the Hierarchy

- Fundamental question: how do we know what data to put in the fast and small memory?
- Answer: ensure most of the data the processor needs **in the near future** is kept in the fast(er) level(s)
- How do we know what data will be needed in the future?
  - Do we know before the program runs?
    - If so, programmers or compiler can place the right data at the right place
  - Do we know only after the program runs?
    - If so, only the hardware can effectively place the data




# How to Make Effective Use of the Hierarchy



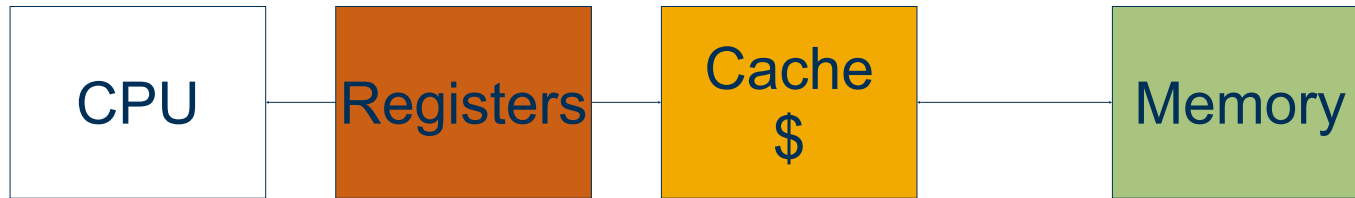
- Modern computers provide both ways
- Register file: programmers explicitly move data from the main memory (slow but big DRAM) to registers (small, very fast)
  - `movq (%rdi), %rdx`
- **Cache**, on the other hand, is automatically managed by hardware
  - Sits between registers and main memory, “invisible” to programmers
  - The hardware automatically figures out what data will be used in the near future, and place in the cache.
  - How does the hardware know that??

# Register VS Cache

```
long a = 10;            movq $10, %rax  
long b = 20;      movq $10, 4(%rbx)
```

- From the programmer's perspective, data is **either** in register or memory.
  - One or the other, not both
- If the data is in memory, the hardware may keep a copy of this data in cache to speed up access to it.

# How to Make Effective Use of the Hierarchy



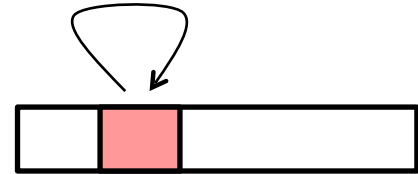
- Modern computers provide both ways
- Register file: programmers explicitly move data from the main memory (slow but big DRAM) to registers (small, very fast)
  - `movq (%rdi), %rdx`
- **Cache**, on the other hand, is automatically managed by hardware
  - Sits between registers and main memory, “invisible” to programmers
  - The hardware automatically figures out what data will be used in the near future, and place in the cache.
  - How does the hardware know that??

# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.

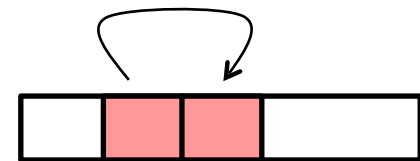
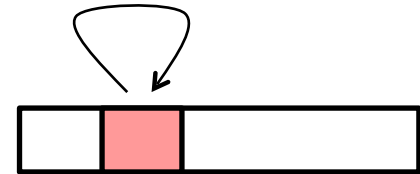
# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future



# Locality: An Empirical Observation

- **Principle of Locality:** Programs tend to use the same data over and over again, and tend to access data next to each other.
- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future
- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time



# Locality Example

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

- Data references
  - **Spatial** Locality: Reference array elements in succession (stride-1 reference pattern)
  - **Temporal** Locality: Reference variable sum each iteration.
- Instruction references
  - **Spatial** Locality: Reference instructions in sequence.
  - **Temporal** Locality: Cycle through loop repeatedly.

# Use Locality to Manage Memory Hierarchy

```
sum = 0;  
for (i = 0; i < n; i++)  
    sum += a[i];  
return sum;
```

- Exploiting temporal locality:
  - If a piece of data is recently accessed, very likely it will be needed again, so move it to cache.
- Exploiting spatial locality:
  - When moving a piece of data from the memory to the cache, move its adjacent data to the cache as well.



# The Bookshelf Analogy

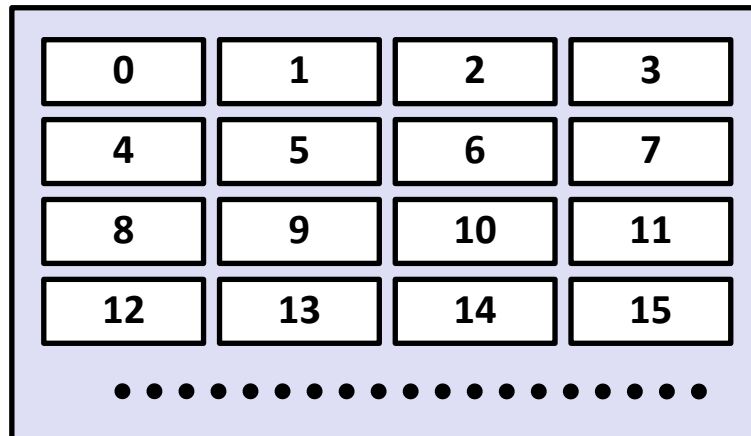
- Book in your hand
  - Desk
  - Bookshelf
  - Boxes at home
  - Library
- 
- Recently-used books tend to stay on desk, because you will likely use it again.
    - Comp Org. books
    - Books for other courses

# Cache Illustrations

CPU



Memory  
(big but slow)

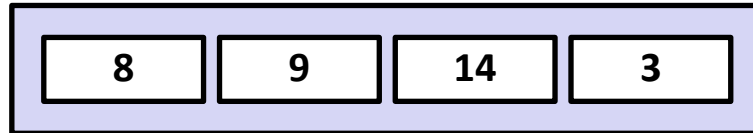


# Cache Illustrations

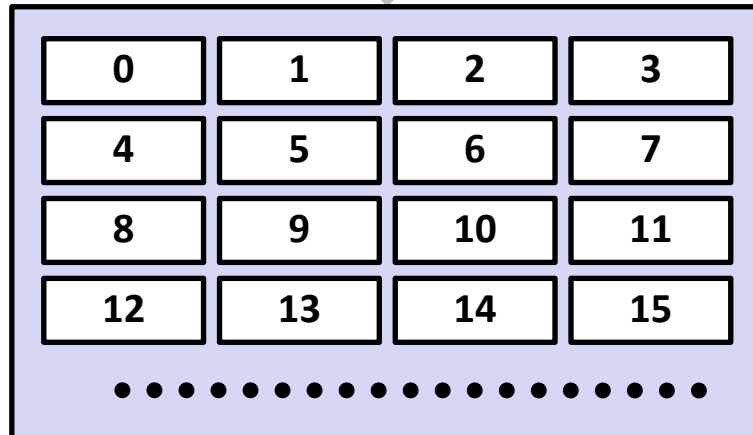
CPU



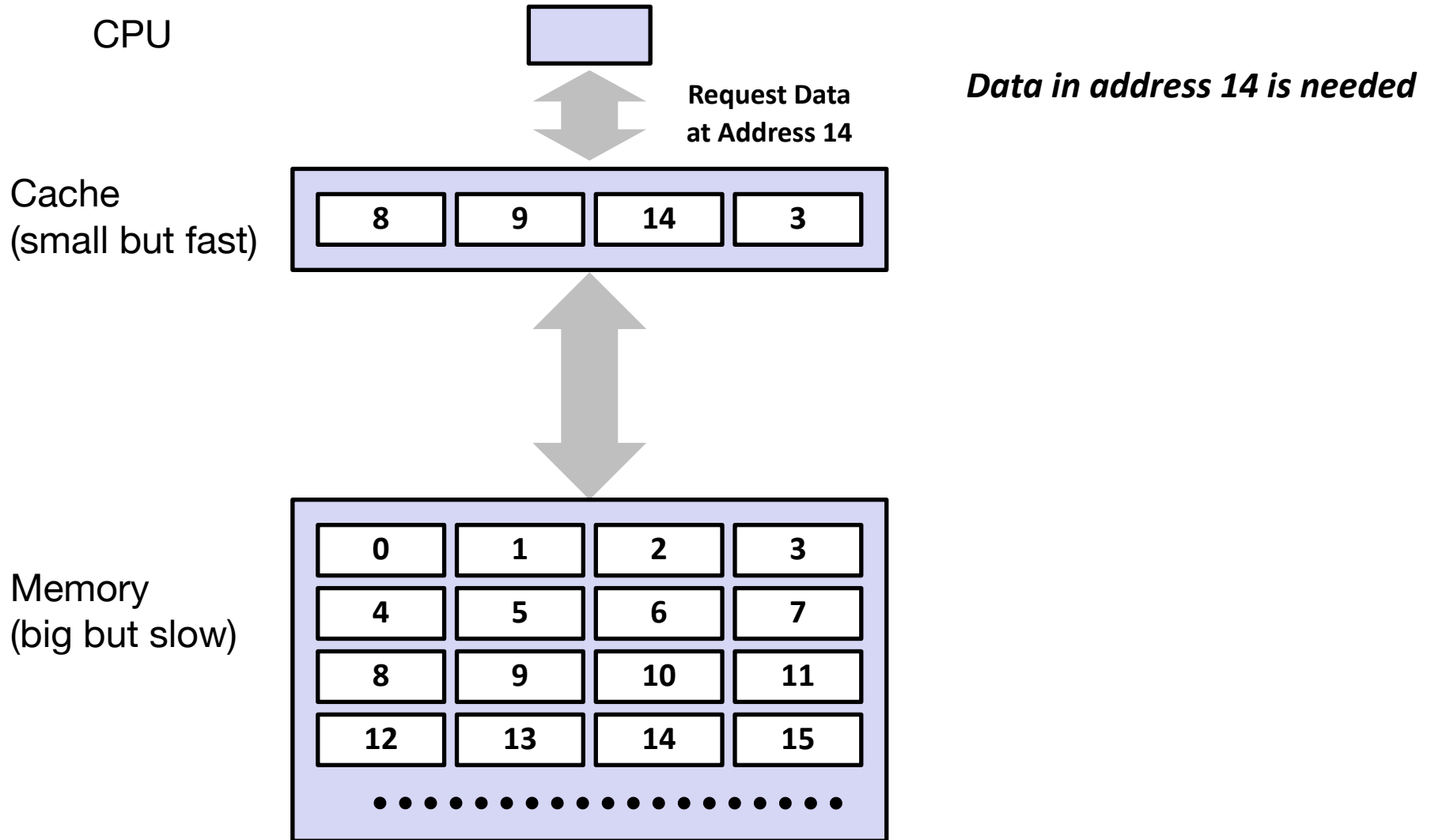
Cache  
(small but fast)



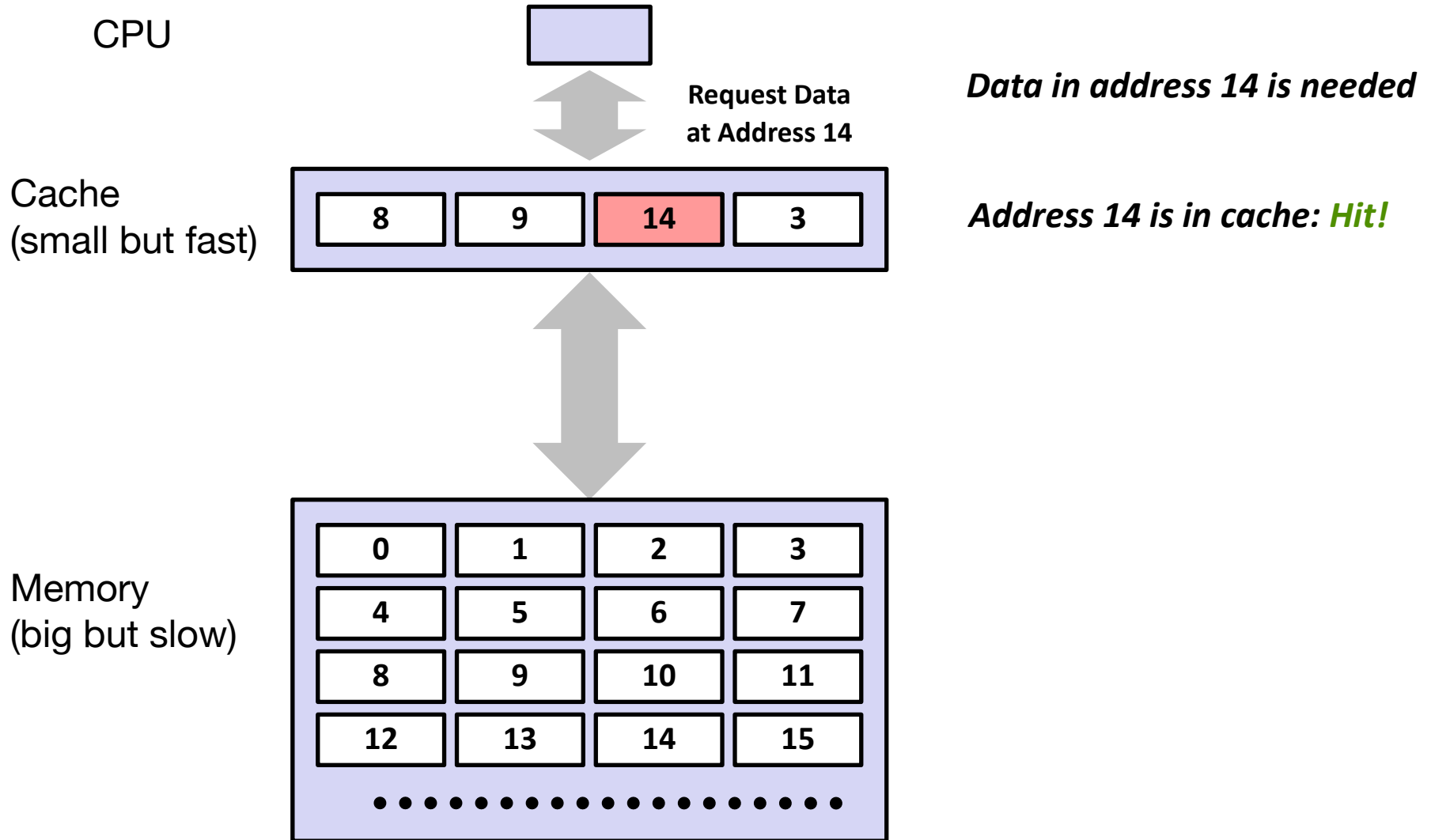
Memory  
(big but slow)



# Cache Illustrations



# Cache Illustrations

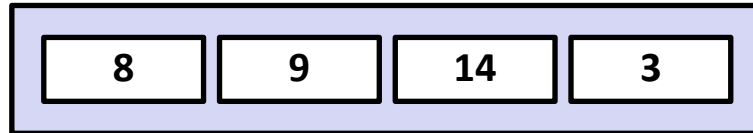


# Cache Illustrations

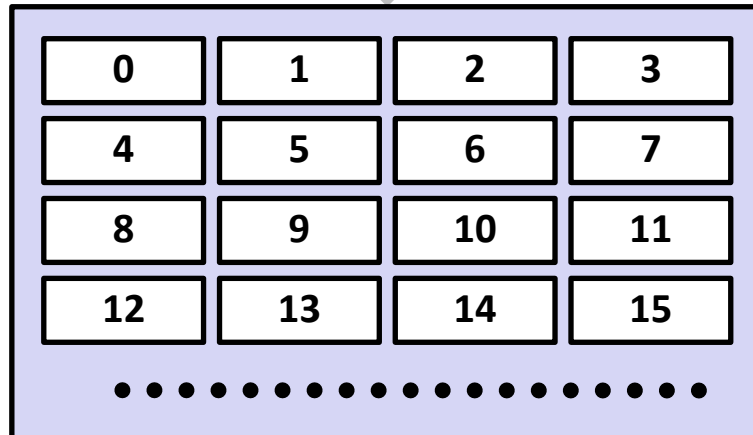
CPU



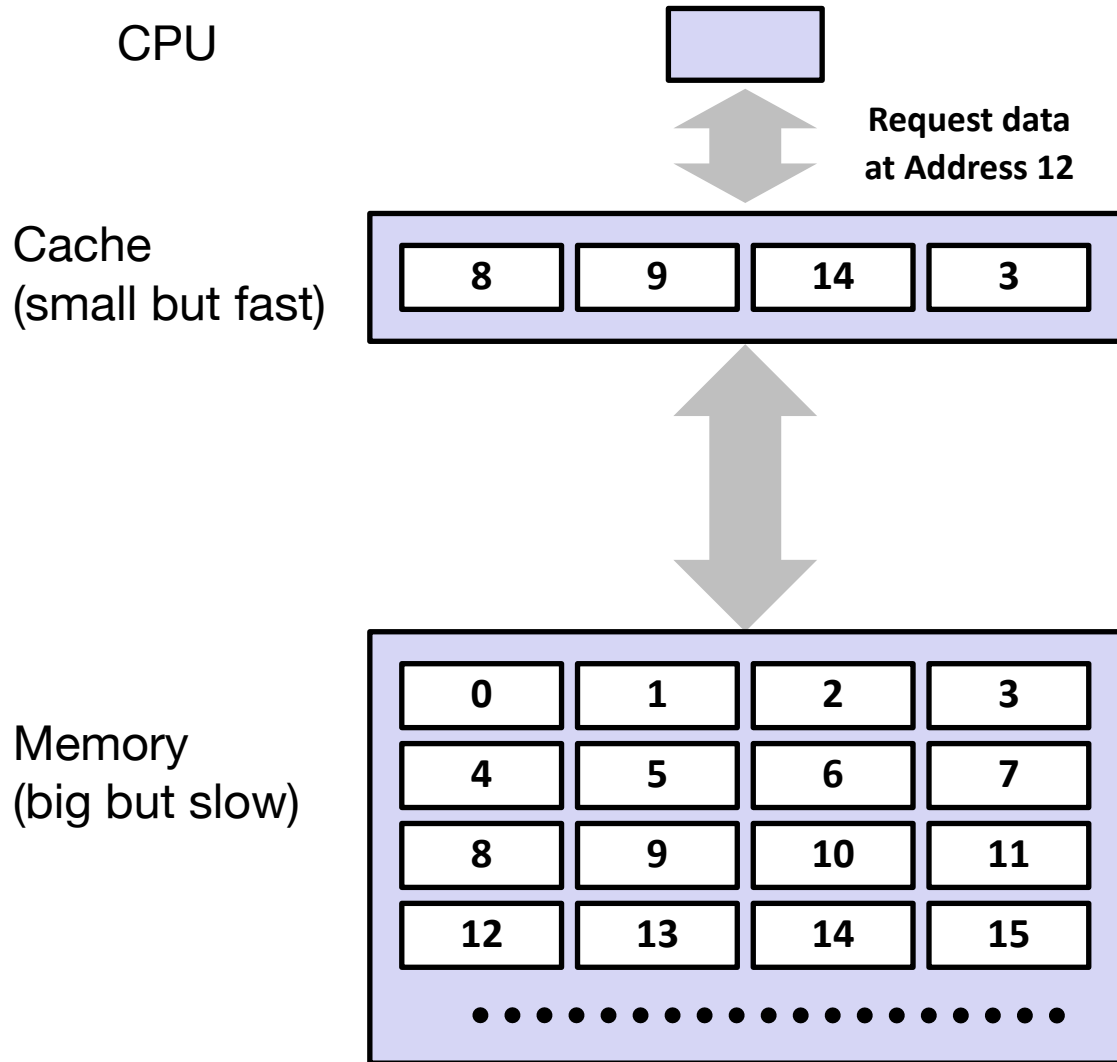
Cache  
(small but fast)



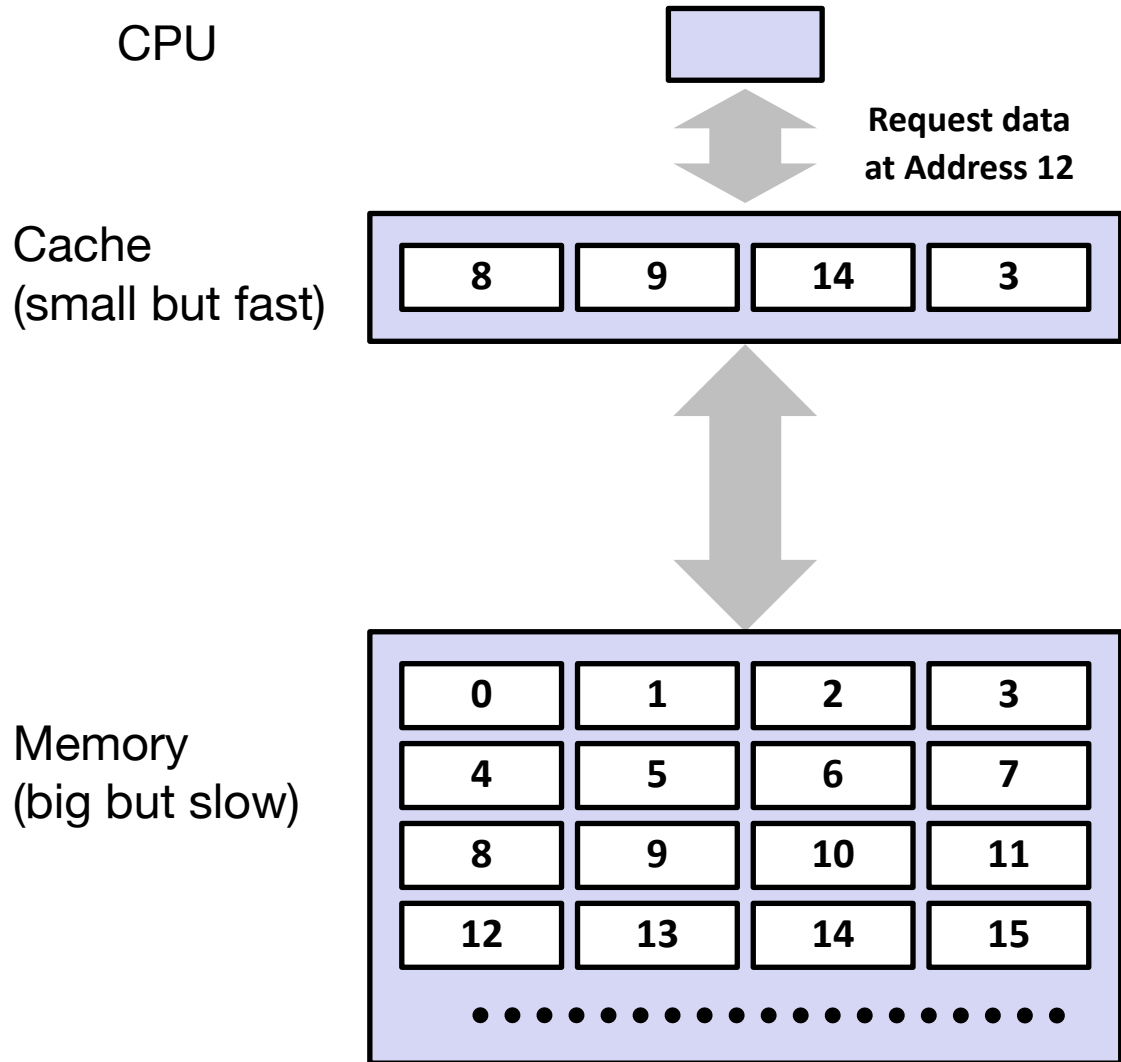
Memory  
(big but slow)



# Cache Illustrations



# Cache Illustrations



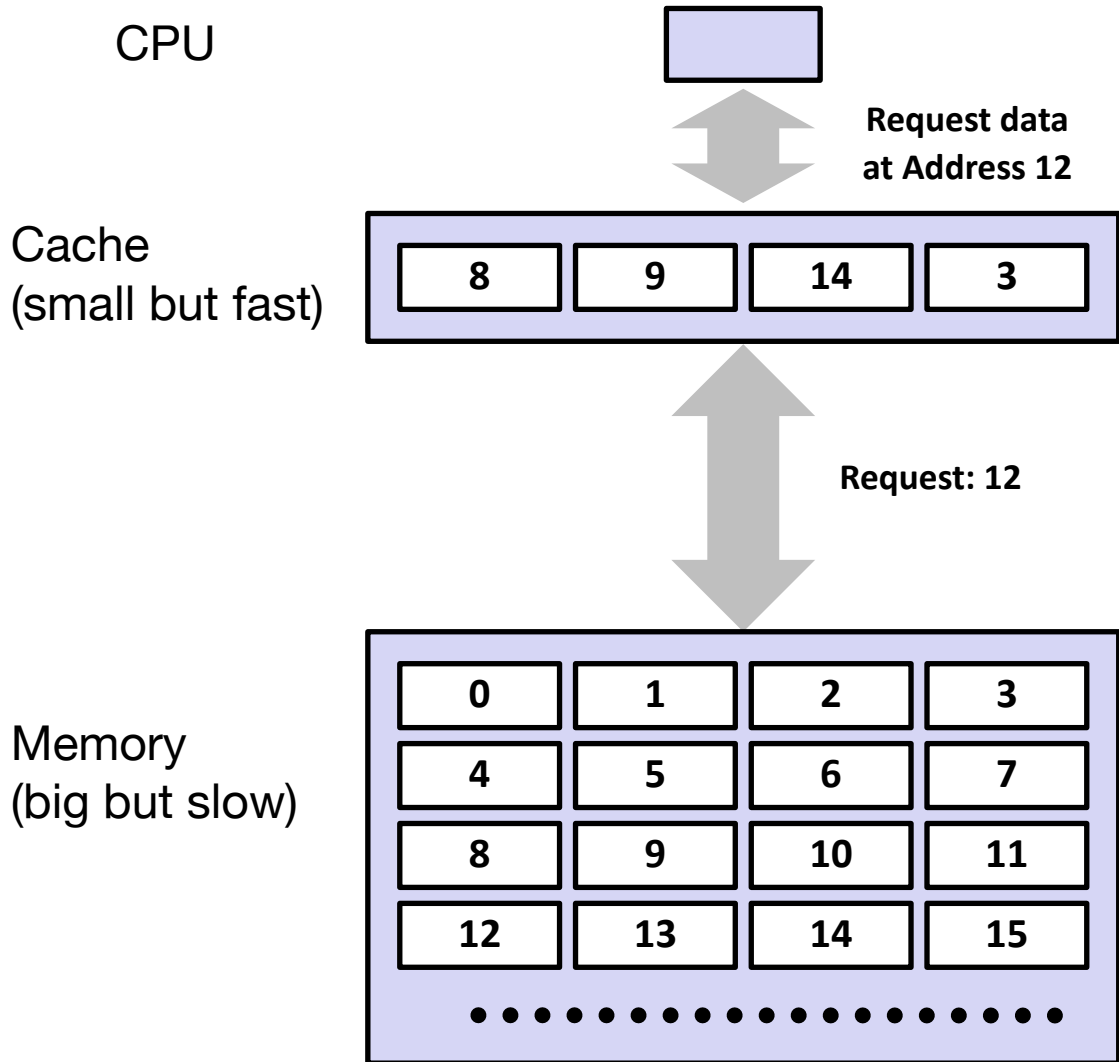
*Data in address 12 is needed*

*Address 12 is not in cache:*

***Miss!***



# Cache Illustrations

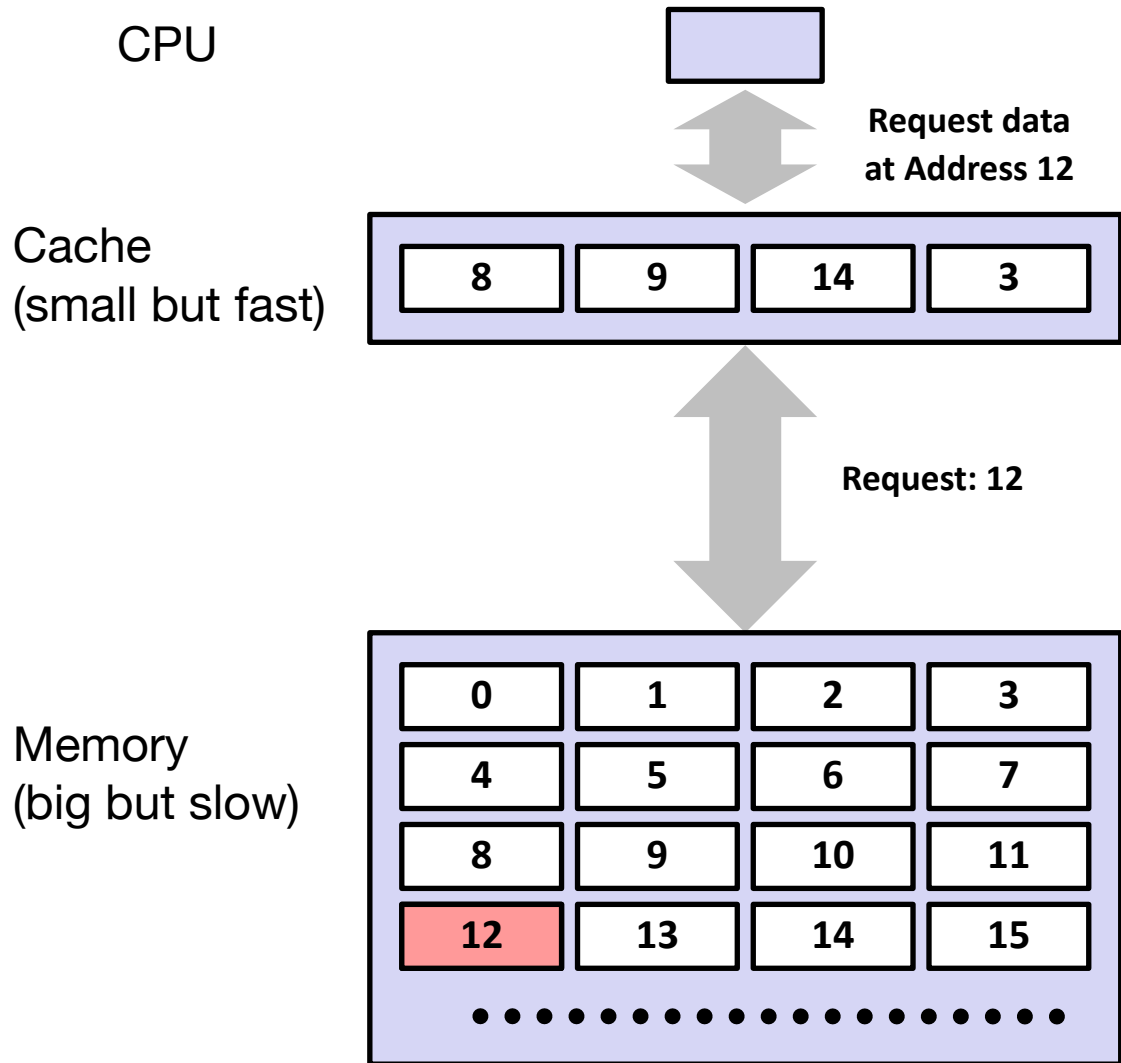


*Data in address 12 is needed*

*Address 12 is not in cache:  
**Miss!***

*Address 12 is fetched from  
memory*

# Cache Illustrations

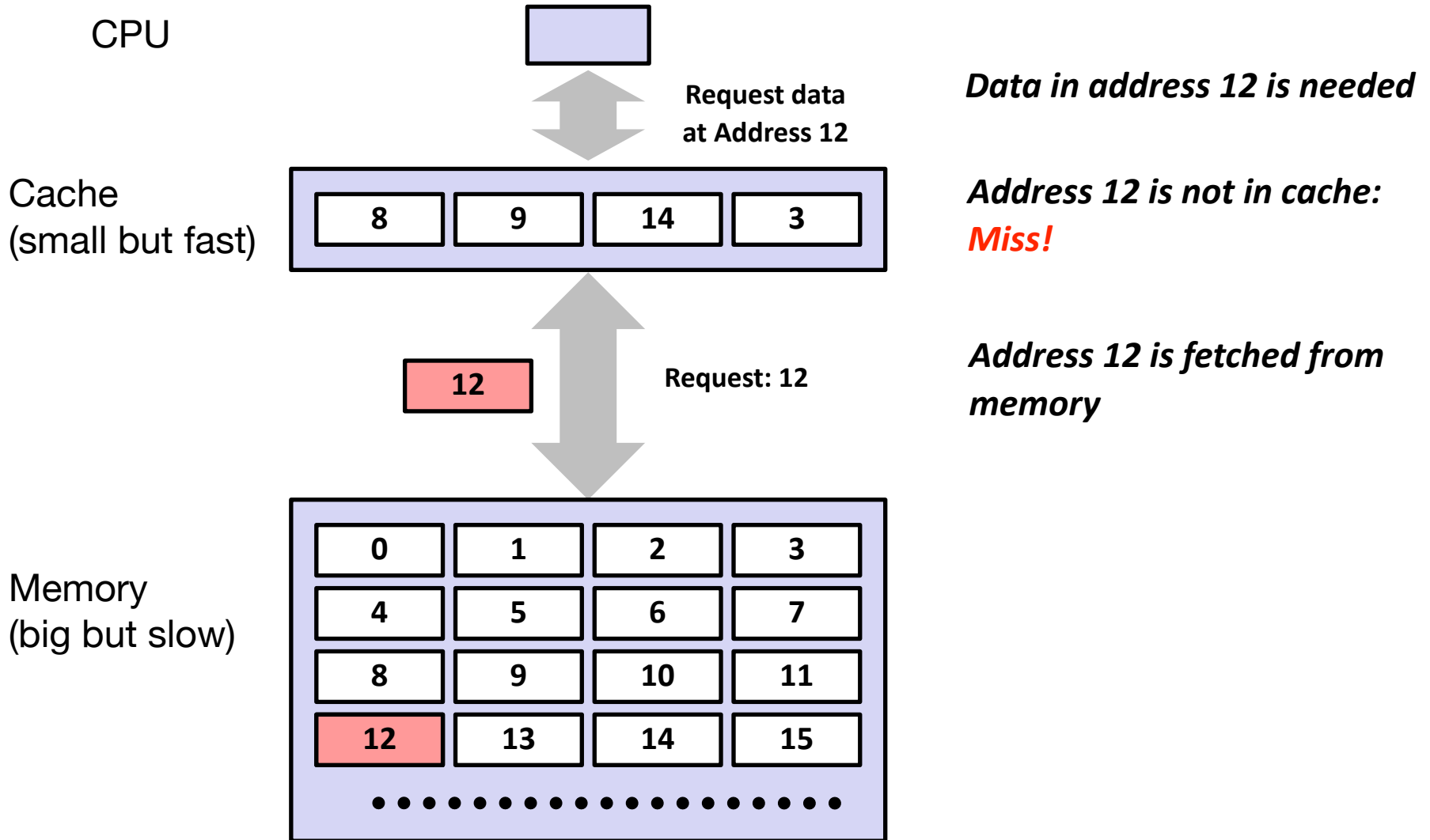


*Data in address 12 is needed*

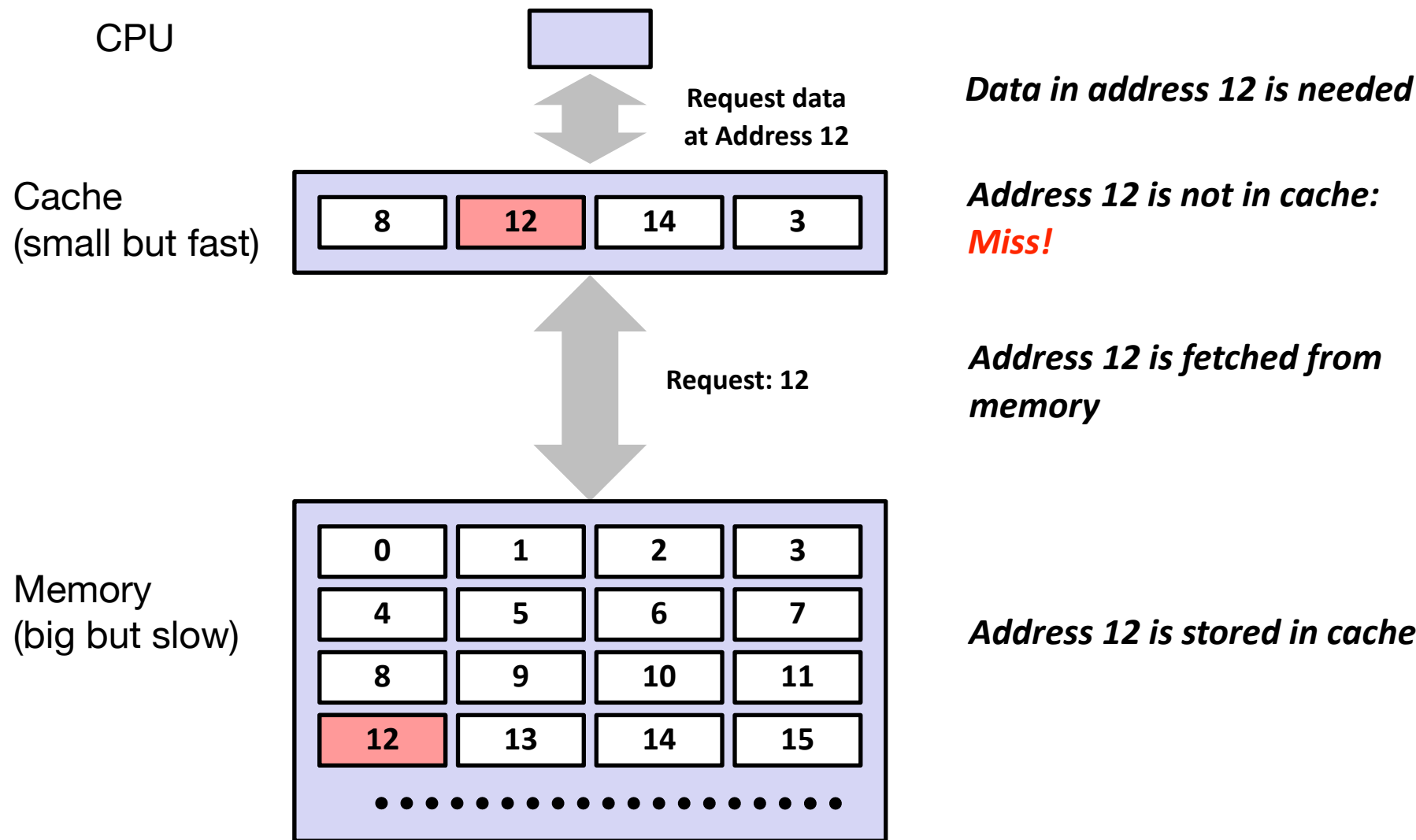
*Address 12 is not in cache:  
**Miss!***

*Address 12 is fetched from  
memory*

# Cache Illustrations



# Cache Illustrations



# Cache Hit Rate

- Cache hit is when you find the data in the cache
- Hit rate indicates the effectiveness of the cache

$$\text{Hit Rate} = \frac{\# \text{ Hits}}{\# \text{ Accesses}}$$

# Two Fundamental Issues in Cache Management

- Finding the data in the cache
  - Given an address, how do we decide whether it's in the cache or not?
- Kicking data out of the cache
  - Cache is small than memory, so when there's no place left in the cache, we need to kick something out before we can put new data into it, but who to kick out?

# A Simple Cache

## Cache

	Content	Valid?
00		
01		
10		
11		

## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations

# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**



# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.

# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B

# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line

# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line
- Cache is smaller than memory (obviously)

# A Simple Cache

Cache

	Content	Valid?
00		
01		
10		
11		

Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- 16 memory locations
- 4 cache locations
  - Also called **cache-line**
  - Every location has a valid bit, indicating whether that location contains valid data; 0 initially.
- For now, assume cache location size == memory location size == 1 B
- Assume each memory location can only reside in one cache-line
- Cache is smaller than memory (obviously)
  - Thus, not all memory locations can be cached at the same time

# Cache Placement

## Cache

00		
01		
10		
11		

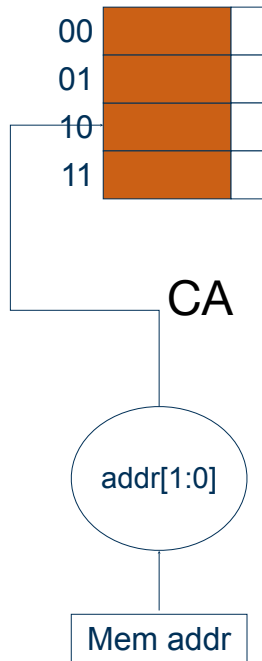
## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

- Given a memory addr, say 0x0001, we want to put the data there into the cache; where does the data go?

# Function to Address Cache

## Cache



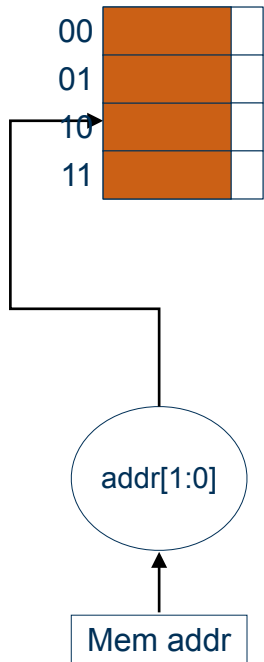
## Memory

0000	
0001	
0010	
0011	
0100	
0101	
0110	
0111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	

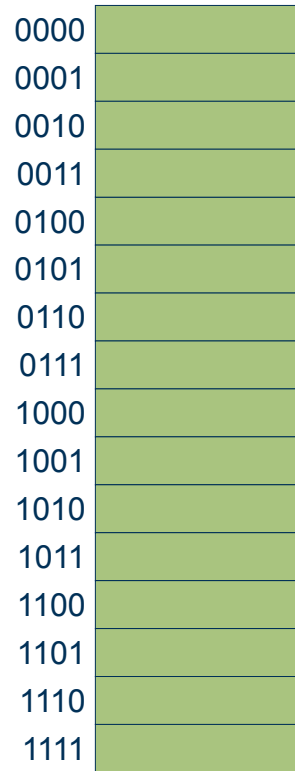
- Simplest way is to take a subset of address bits
- Direct-Mapped Cache
  - $CA = ADDR[1], ADDR[0]$

# Direct-Mapped Cache

Cache



Memory



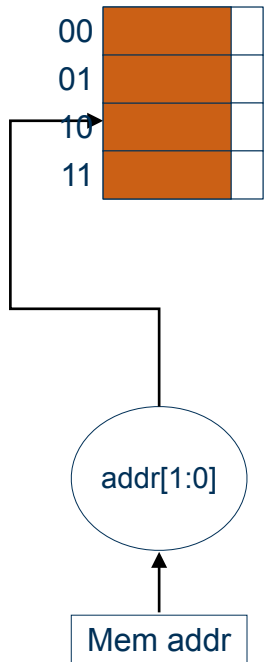
- Direct-Mapped Cache

- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

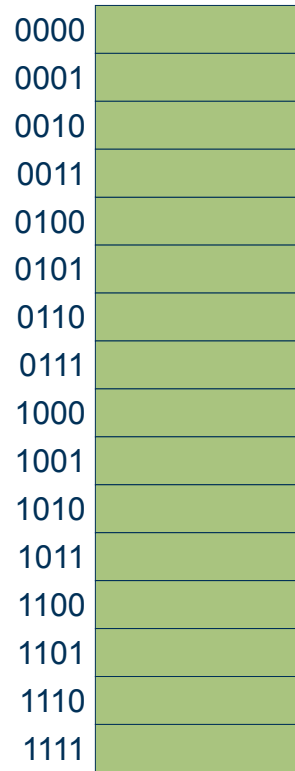


# Direct-Mapped Cache

Cache



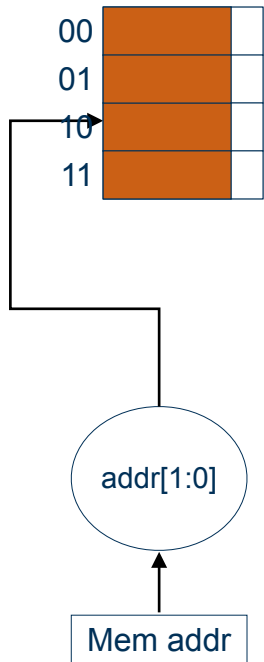
Memory



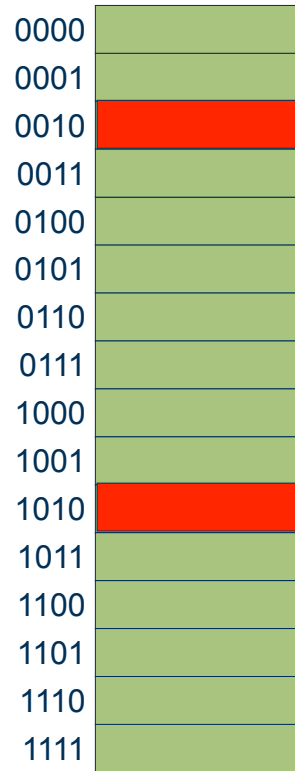
- Direct-Mapped Cache
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location

# Direct-Mapped Cache

Cache

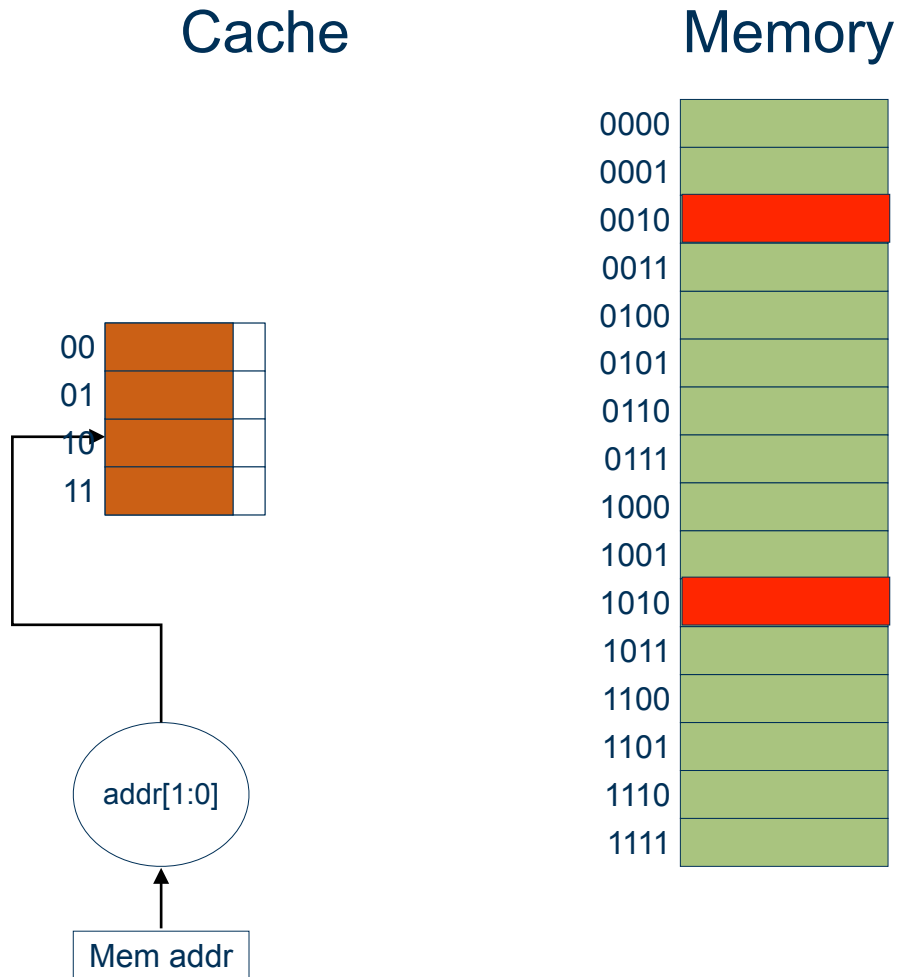


Memory



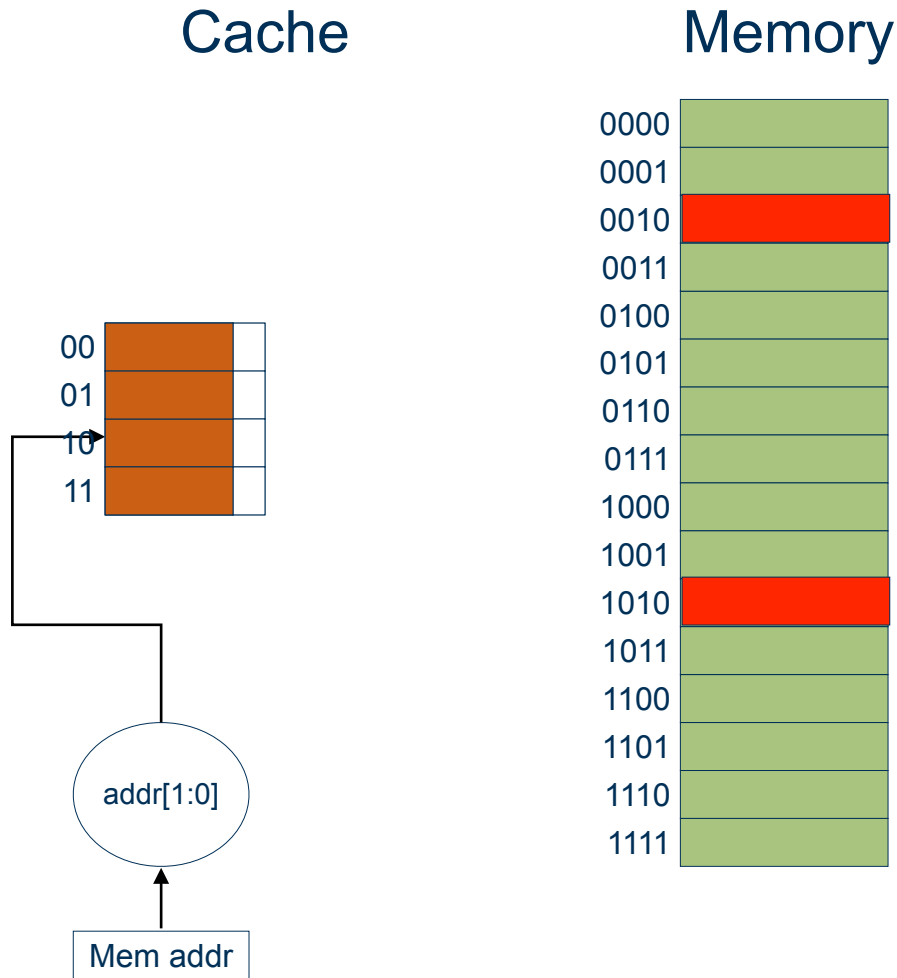
- Direct-Mapped Cache
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010

# Direct-Mapped Cache



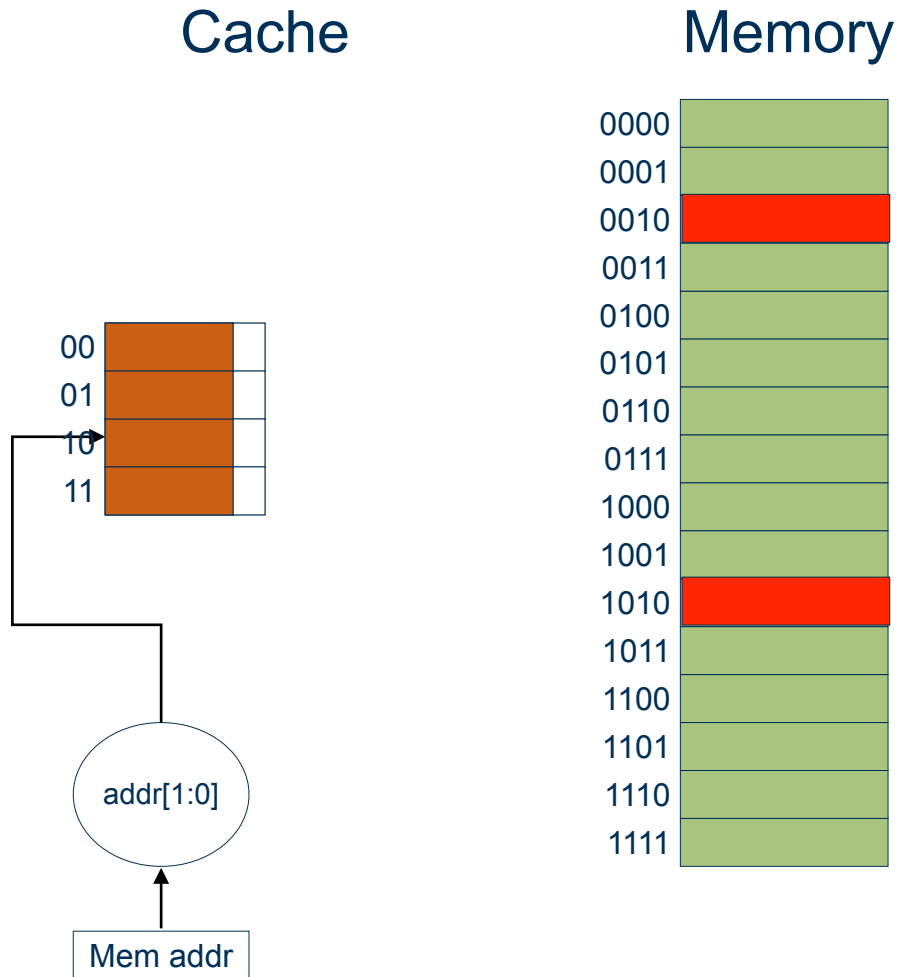
- Direct-Mapped Cache
  - CA = ADDR[1], ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?

# Direct-Mapped Cache



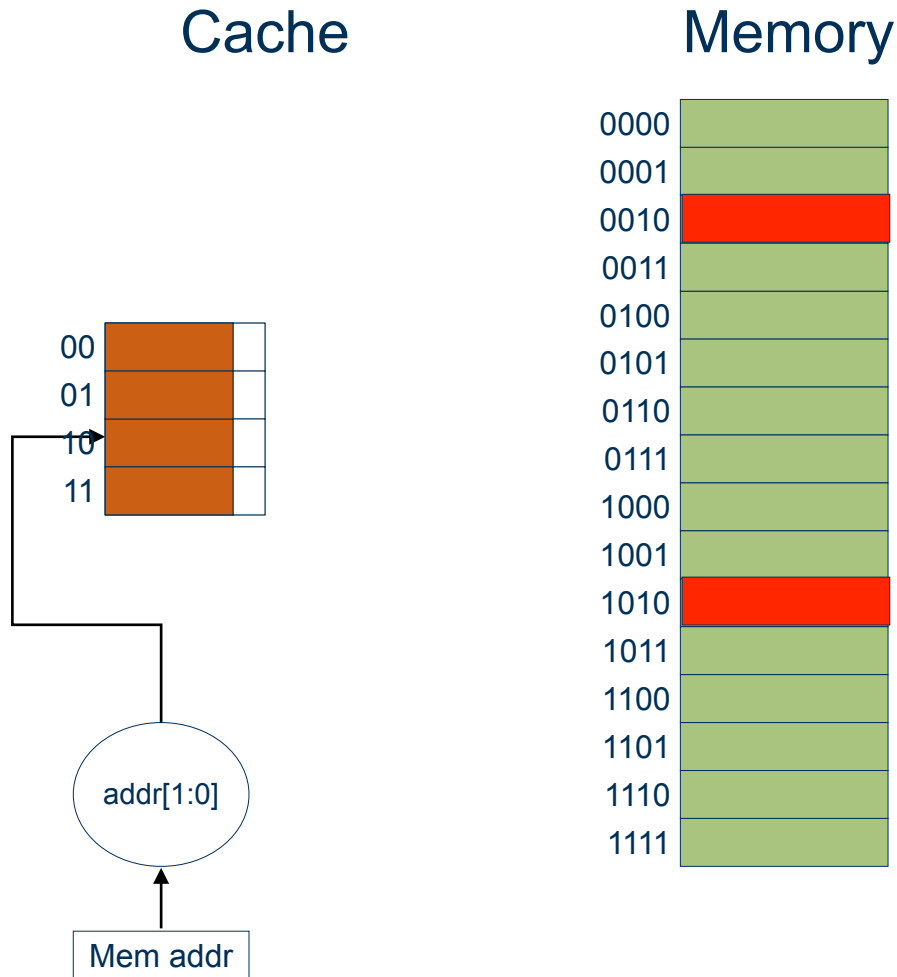
- Direct-Mapped Cache
  - CA = ADDR[1], ADDR[0]
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that purpose

# Direct-Mapped Cache



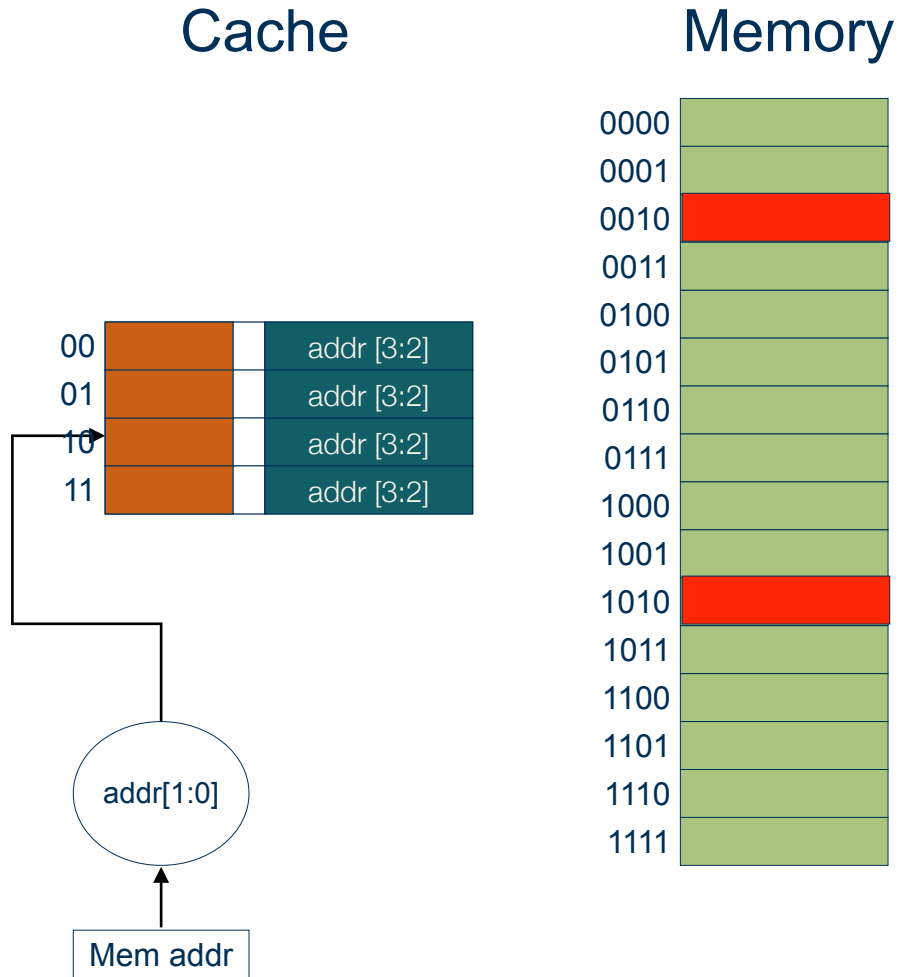
- Direct-Mapped Cache
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that purpose
  - What should the tag field be?

# Direct-Mapped Cache



- **Direct-Mapped Cache**
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- **Multiple addresses can be mapped to the same location**
  - E.g., 0010 and 1010
- **How do we differentiate between different memory locations that are mapped to the same cache location?**
  - Add a tag field for that purpose
  - What should the tag field be?
  - $ADDR[3]$  and  $ADDR[2]$  in this particular example

# Direct-Mapped Cache



- **Direct-Mapped Cache**

- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

- **Multiple addresses can be mapped to the same location**

- E.g., 0010 and 1010

- **How do we differentiate between different memory locations that are mapped to the same cache location?**

- Add a tag field for that purpose
- What should the tag field be?
- $ADDR[3]$  and  $ADDR[2]$  in this particular example

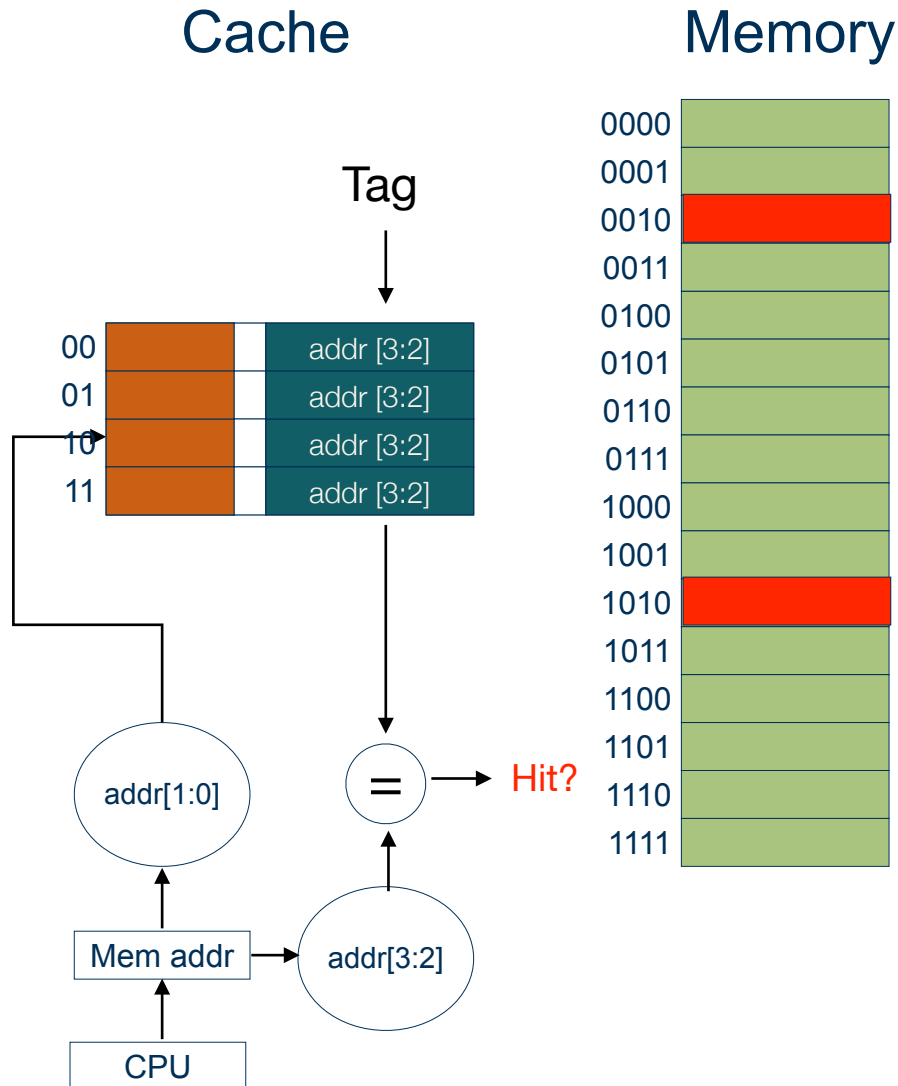
# Direct-Mapped Cache



- Direct-Mapped Cache
  - $CA = ADDR[1], ADDR[0]$
  - Always use the lower order address bits
- Multiple addresses can be mapped to the same location
  - E.g., 0010 and 1010
- How do we differentiate between different memory locations that are mapped to the same cache location?
  - Add a tag field for that purpose
  - What should the tag field be?
  - $ADDR[3]$  and  $ADDR[2]$  in this particular example



# Direct-Mapped Cache



- **Direct-Mapped Cache**

- $CA = ADDR[1], ADDR[0]$
- Always use the lower order address bits

- **Multiple addresses can be mapped to the same location**

- E.g., 0010 and 1010

- **How do we differentiate between different memory locations that are mapped to the same cache location?**

- Add a tag field for that purpose
- What should the tag field be?
- $ADDR[3]$  and  $ADDR[2]$  in this particular example